

0: Fundamentals of Web apps

0a: General info

This course is an introduction to modern web development with JavaScript. The main focus is on single-page applications implemented with React and supporting them with RESTful and GraphQL web services implemented with Node.js. The course also has parts on TypeScript, React Native, and Continuous integration.

Other topics include debugging applications, container technology, configuration, managing runtime environments, and databases.

The course is totally free of charge. You can get a certificate and even the University of Helsinki ECTS (European Credit Transfer and Accumulation System) credits for free.

Prerequisites

Participants are expected to have good programming skills, basic knowledge of web programming and databases, and know the basics of the Git version control system. You are also expected to have perseverance and a capacity for solving problems and seeking information independently.

Previous knowledge of JavaScript or other course topics is not required.

How much programming experience is needed? It is hard to say, but you should be pretty fluent in *your* language. This level of fluency takes usually at least 100-200 hours of practice to develop.

Course material

The course material is meant to be read one part at a time and in order.

The material contains exercises, which are placed so that the preceding material provides enough information for solving each exercise. You can do the exercises as you encounter them in the material, but it can also be beneficial to read all of the material in the part before starting with the exercises.

In many parts of the course, the exercises build one larger application one small piece at a time. Some of the exercise applications are developed through multiple parts.

The course material is based on incrementally expanding example applications, which change from part to part. It's best to follow the code along while making small modifications independently. The code of the example applications for each step of each part can be found on GitHub.

Taking the course

The course contains fourteen parts, the first of which is numbered 0 for consistency with past iterations. One part corresponds loosely to one week (averaging 15-20 hours) of studying, but the speed of completing the course is flexible.

Proceeding from part n to part $n+1$ is not sensible before enough know-how of the topics in part n has been achieved. In pedagogic terms, the course uses [Mastery Learning](#), and you are only intended to

proceed to the next part after doing enough of the exercises of the previous part.

In parts 1-4 you are expected to do at least **all** of the exercises that are **not** marked with an asterisk(*). Exercises marked with an asterisk count towards your final grade, but skipping them does not prevent you from doing the compulsory exercises in the next parts. Parts 5-13 do not have asterisk-marked exercises since there is no similar dependency on previous parts.

The speed of completing the course is flexible.

Exercise completion time statistics can be found via the [submission system](#).

Course channel in Discord and Telegram

You can discuss the course and related topics in our dedicated group on Discord <https://study.cs.helsinki.fi/discord/join/fullstack> and on Telegram: <https://t.me/fullstackcourse>. Discord has fullstack_general and part-specific (channel names with fullstack prefix) channels for course-related discussion. Note that Discord's *chat channel is not suitable for course-related discussions*. Please join the conversation!

How to get help in Discord/Telegram

When you ask for help for a problem in the Discord/Telegram group your question should be as informative and precise as possible. If your question looks like this

Adding a new person does not work, could you help me with that?

it is quite likely that nobody will respond. The bug can be *anywhere*.

A better question could be

- In exercise 2.15 when I try to add a new person to the app, the server responds with a 403, despite the request looking ok to me.

The code looks like this

```
// the relevant part of code is pasted here  
// code should contain several console.log statements for helping the de  
bugging
```

The following gets printed to the console

```
// data printed to console
```

The network tab looks like the following*

[screenshot from the network console]

All the code can be found here (a link to GitHub)

Parts and completion

Full Stack studies consist of the core course and multiple extensions. You can complete the studies in the extent of 5 to 14 credits.

Parts 0-5 (core course) - Full Stack Web Development (5 cr, CSM141081)

The number of credits and the grade for the course are based on the total number of submitted exercises for parts 0-7 (including exercises marked with an asterisk).

Credits and grades are calculated as follows:

exercises	credits	grade
-----------	---------	-------

138	7	5
127	6	5
116	5	5
105	5	4
94	5	3
83	5	2
72	5	1

Once you have completed enough exercises for a passing grade, you can download the course certificate from the [submission system](#).

If you wish to receive university credits, you must complete the course exam. The exam does not count toward your final grade, but you must pass it. More information about the exam [here](#).

You can only take the exam after submitting enough exercises for five credits. It is not wise in practice to take the exam immediately after submitting the critical number of exercises. The exam is the same for 5-14 credits and does not count toward your grade.

You do not need to attend the course exam or register for the Open University course to obtain the course certificate.

Part 6 - Full Stack Web Development, extension 1 (1 cr, CSM141082)

By submitting at least 127 of the exercises for parts 0-7 while working on the core course, you can receive an additional credit through this extension.

- Submit at least 127 exercises for parts 0-7.
- [Enroll in part 6 through the Open University](#).
- [Request credits for part 6](#).

Part 7 - Full Stack Web Development, extension 2 (1 cr, CSM141083)

By submitting at least 138 of the exercises for parts 0-7 while working on the core course, you can receive an additional credit through this extension.

- Submit at least 138 exercises for parts 0-7.
- [Enroll in part 7 through the Open University](#).
- [Request credits for part 7](#).

Part 8 - Full Stack Web Development: GraphQL (1 cr, CSM14113)

By submitting at least 22/26 of the exercises for part 8 of the course, GraphQL, you can get one additional credit. Part 8 can be done any time after part 5 because its contents are independent of parts 6 and 7.

- Submit at least 22/26 exercises for part 8.
- [Enroll in part 8 through the Open University](#).
- [Request credits for part 8](#).

Part 9 - Full Stack Web Development: TypeScript (1 cr, CSM14110)

By submitting at least 24/29 of the exercises for part 9 of the course, TypeScript, you can get one additional credit. It is recommended that you complete parts 0-7 before taking part 9.

- Submit at least 24/29 exercises for part 9.
- [Enroll in part 9 through the Open University](#).
- [Request credits for part 9](#).

Part 10 - Full Stack Web Development: React Native (2 cr, CSM14111)

By submitting 25 exercises for part 10 of the course on React Native, you can earn two additional credits. More information about this part's prerequisites, exercise submission, and credits can be found in [part 10](#).

- Submit at least 25 exercises for part 10.
- [Enroll in part 10 through the Open University](#).
- [Request credits for part 10](#).

Part 11 - Full Stack Web Development: Continuous Integration / Continuous Delivery (1 cr, CSM14112)

By submitting all exercises for part 11 of the course on Continuous Integration / Continuous Delivery, you can earn one additional credit. More information about this part's prerequisites and exercise submission can be found in [part 11](#).

- Submit all exercises for part 11.
- [Enroll in part 11 through the Open University](#).
- [Request credits for part 11](#).

Part 12 - Full Stack Web Development: Containers (1 cr, CSM141084)

By submitting all exercises for part 12 of the course on Container technology, you can earn one additional credit. More information about this part's prerequisites and exercise submission can be found in [part 12](#).

- Submit all exercises for part 12.
- [Enroll in part 12 through the Open University](#).

- Request credits for part 12.

Part 13 - Full Stack Web Development: Relational databases (1 cr, CSM14114)

By submitting all exercises for part 13 of the course on Relational databases, you can earn one additional credit. More information about this part's prerequisites and exercise submission can be found in [part 13](#).

- Submit all exercises for part 13.
- [Enroll in part 13 through the Open University](#).
- Request credits for part 13.

Studying the course in a nutshell

How to study the course – instructions in a nutshell: 5 cr core course CSM141081

- Do the exercises. The exercises are submitted through GitHub and marking them as done on the [submission system](#).
 - [The course certificate](#) will be available in the submission system.
- If you want to get University of Helsinki credits
 - Enroll in the course. You will get the enrollment link through the submission system once you have completed enough exercises. Read more [here](#)
 - Save your student number. After course enrollment, save your University of Helsinki student ID number in the submission system.
 - Do the online exam in the submission system. Read more [here](#)
 - Mark the course completed in the submission system. Read more [here](#)

Please note that if you do the "base course" with 6 or 7 credits, you need separate registrations for the extra credits, see [Parts and completion](#) for more.

How to study the course – instructions in a nutshell: other course parts

- Do the exercises. The exercises are submitted through GitHub and marking them is done on the submission system. Note that parts 8-13 have a separate instance in the submission system
 - [The course certificate](#) will be available in the submission system
- If you want to get University of Helsinki credits
 - Enroll in the course. You can enroll in each part through the link in the course material ([Parts and completion](#)). Enroll in each part separately.
- Mark the part completed in the submission system. Read more [here](#)

Submitting exercises

The exercises are submitted through GitHub and marked as done on the "my submissions" tab of the [submission application](#).

If you are submitting exercises from different parts to the same repository, use an appropriate system for naming your directories. You can of course create a new repository for each part. If you are using a private repository, add *mluukkai* as a collaborator.

Exercises are submitted **one part at a time**. You will mark the number of exercises you have completed from that module. Once you have submitted exercises for a part, you can no longer submit any more exercises for that part.

A system for detecting plagiarism is used to check exercises submitted to GitHub. If code is found from model answers or multiple students hand in the same code, the situation is handled according to the [policy on plagiarism](#) of the University of Helsinki.

Many of the exercises build a larger application bit by bit. In these cases, submitting only the completed application is enough. You can make a commit after each exercise, but that is not compulsory.

The course exam

For the official university credits, you need to pass the course exam that covers parts 1-5 of the course

- If you fail the exam, it can be done again after one week
- You may continue with submissions after the exam

The exam is done in the exercise submission system. Follow the instructions below to complete the exam.

- Enroll in the course through Open University.
- You will get the enrollment link through the [submission system](#) once you have completed enough exercises.

You are entitled to grade 3 and 5 credits based on the exercises you have submitted

Certificate  

In order to get the university credits

- Enroll to the Open University course [here](#) 
- Fill in your student number and name [here](#) if you want to get the University of Helsinki credits
- Do the exam in [Here](#)

After the course enrollment, save your University of Helsinki student number in the [submission system](#):

C studies.cs.helsinki.fi/stats/myinfo

course stats	Matti Luukkainen - mluukkai	logout
--------------	-----------------------------	--------

Your information

Name: Matti Luukkainen

Student Number: [REDACTED]

Name	Matti Luukkainen
Student Number	[REDACTED]

Save

See [this](#) for information on how to find your student number.

After these steps, you can do the course exam in the submission system:

You are entitled to grade 3 and 5 credits based on the exercises you have submitted

Certificate

University of Helsinki credits

If you wan to get the university credits, you should do the exam

Start the exam now

You will have 120 minutes to complete the exam. If all goes well, you should see the following confirmation:

Full Stack Open Exam

Exam has ended 15:40:50 July 24th 2022

You got 23.0/26.0 points. You passed the exam.

For the university credits, go back to [my submissions](#)

If you fail, you must wait for one week for trying the exam again.

If you passed the exam, and you are not going to complete more exercises, you can go back to the "my submissions" tab and ask for the credits:

You are entitled to grade 3 and 5 credits based on the exercises you have submitted

Certificate  

University of Helsinki credits

Completion language

English

The language you wish to associate with your course completion information

I have registered to the university course according to the information given in the course page

I have completed the course (and will not do more exercises) and want to get university credits registered.



Remember to press the big blue button to ask for the credits to be registered.

Note that you have to press the button twice:

I have registered to the university course according to the information given in the course page

Press again to confirm. Make sure that everything is ready and submitted

When pressed twice you should see the following text

University credit registration in progress...

Note: if you have already done the course exam in Moodle contact matti.luukkainen@helsinki.fi or @mluukkai in Discord.

How to get your credits

If you want to receive University of Helsinki credits, save your **University of Helsinki** student number to [the exercise submission system](#)

C studies.cs.helsinki.fi/stats/myinfo

course stats	Matti Luukkainen - mluukkai	logout
--------------	-----------------------------	--------

Your information

Name: Matti Luukkainen

Student Number: [REDACTED]

Name	Matti Luukkainen
Student Number	[REDACTED]

Save

If you are not a student at the University of Helsinki, you can get a student number by registering for the course through [Open University](#), see [this](#) for more information.

You will receive your credits after you have submitted enough exercises for a passing grade, passed the exam and then letting us know through the exercise submission system that you have completed the course:

You are entitled to grade 3 and 5 credits based on the exercises you have submitted

Certificate

University of Helsinki credits

Completion language

English

The language you wish to associate with your course completion information

I have registered to the university course according to the information given in the course page

I have completed the course (and will not do more exercises) and want to get university credits registered.



Remember to press the big blue button to ask for the credits to be registered.

Note that you have to press the button twice:

I have registered to the university course according to the information given in the course page

Press again to confirm. Make sure that everything is ready and submitted

When pressed twice you should see the following text

University credit registration in progress...

Please note that to get university credits you need a registration for each completed part. Please see [more information about registration](#).

You can view your grade in the University of Helsinki Sisu and [Opintopolku](#) approximately four weeks after notifying us.

When the registration is done, the following text appears in the submission system

University credits registered, see the course page for how to get a transcript if you need one

Where do I get my University of Helsinki Student number

When and if you enroll in a course for the first time through the Open University, a University of Helsinki student number will be automatically generated. ***Please make sure you have enrolled in the course before you try to find out what your student number is.***

Note also that **you do not need to enroll** in Open University to get the course certificate!

You can find out what your student number is through one of the options below:

A) Sisu

If you have a University of Helsinki user account, you can find your student number from your profile in the University of Helsinki's study information system Sisu:

- Log in to Sisu with your University of Helsinki username and password.
- Select: My profile
- Select: Personal information

B) Enrollment confirmation email

After course enrollment, you will receive a confirmation email to an email address you have entered on the enrollment form. This message either directly has your student number on it or includes a link that takes you to a page displaying your University of Helsinki student number.

C) Contact Student Services

If you have trouble finding your student number through the means listed above, you can send an email to the University of Helsinki Student Services. **Make sure you have enrolled in the course through the Open University before sending the email!**

In your email, include the following information

- the name of the course that you have enrolled in,
- your name
- your date of birth.

Student Services email address: avoin-student@helsinki.fi

One more reminder: make sure you have enrolled in the course through the Open University before sending the email

Course certificate

Even if you do not register to Open University for the exam and the credits, you can still download the course certificate from the "My submissions" tab in the [submission system](#) once you have completed enough exercises for a passing grade.

There is one certificate for the base parts (0-7) of the course and after that a separate certificate for each course part.

Request a transcript of studies

You can request a verified transcript *after your university credits have been registered*. To request an official transcript, please contact avoin-student@helsinki.fi.

- When requesting an official transcript, remember to mention
 - the name of the course,
 - your full name,
 - your date of birth
 - the language of the transcript.

The transcript will be delivered to you electronically through email. Present this document at your institution to have the credits included in your degree. The decision to include the credits will be made by your home institution.

No more yearly versions

There are no more "yearly versions" of the course, the course is open all the time. Each part is updated once or twice per year. Updates are mostly minor: library versions are updated and text clarity is improved. However, there might also be some bigger changes.

Despite changes *all the submitted exercises remain valid*, and the course can be continued without being bothered about updates. Also, the policy for getting certificates, university credits etc. shall remain the same no matter what happens.

Recent major changes

- Part 10 (March 21, 2024): Create React app replaced with Vite
- Parts 3-5 (February-March 2024): Libraries used for testing changed
- Part 10 (February 26, 2024): Library versions updated
- Part 11 (16th January 2024): Example project dependencies updated
- Parts 1-2, 5-9 (11th September - 5th October): Create React app replaced with Vite

Expanding on a previously completed course

If you have already taken the course either as a MOOC or as a university course, you can now expand on your course.

Expanding on Full stack open

You can just pick up where you left off! If you wish to resubmit a whole part, please contact the course personnel via [email](#) or Discord *mluukkai*, with your GitHub username and which parts you would like to have deleted from your submissions.

Expanding on the University of Helsinki instance of this course

That is also possible, just contact the course personnel via [email](#) or Discord *mluukkai*.

Full stack project

A full stack project worth 5, 7 or 10 credits will be available through Open University.

For the project, an application is implemented in React and/or Node, though implementing a mobile application in React Native is also possible.

The number of credits is based on hours of work done. One credit is approximately 17.5 hours of work. The work is graded pass/fail.

It is possible to complete the project as a pair or a group.

See [more information on the project](#).

Interview promise

Our collaborators, [Houston Inc.](#), [Terveystalo](#) and [Smartly.io](#), have given the *promise of a job interview* for everyone who completes the course and the project work with maximum credits (14 + 10).

This means that the student can, if they so choose to, sign up for a job interview with a collaborator who has given the promise. The teacher of the course, Matti Luukainen, will send instructions to the student after the courses have been completed with maximum credits.

You need to be a resident of Finland to participate in these interviews.

Before you start

Using the [Chrome browser](#) is recommended for this course because it provides the best tools for web development. Another alternative is the [Developer Edition of Firefox](#), which provides the same range of features.

The course exercises are submitted to GitHub, so Git must be installed and you should know how to use it. For instructions, see [Git and GitHub tutorial for beginners](#).

Install a sensible text editor that supports web development. [Visual Studio Code](#) is highly recommended.

Don't code with nano, Notepad or Gedit. NetBeans isn't very good for web development either. It is also rather heavy in comparison to [Visual Studio Code](#).

Also, install [Node.js](#). The material has been done with version 18.13.0, so don't install any version older than that. See [Node.js installation instructions](#).

Node package manager [npm](#) will be automatically installed with Node.js. We will be actively using npm throughout the course. Node also comes with [npx](#), which we'll need a few times.

Typos in the material

If you find a typo in the material, or something has been expressed unclearly or is simply bad grammar, submit a *pull request* to the course material in the [repository](#). For example, the markdown source code of this page can be found in the repository at <https://github.com/fullstack-hy2020/fullstack-hy2020.github.io/edit/source/src/content/0/en/part0a.md>

At the bottom of each part of the material is a link to *propose changes to the material*. You can edit the source code of the page by clicking on the link.

There are also lots of links in the material for many kinds of background material. If you notice that a link is broken (that happens too often...), propose a change or ping us in Discord if you do not find a replacement for the broken link.

0b: Fundamentals of Web apps

Before we start programming, we will go through some principles of web development by examining an example application at <https://studies.cs.helsinki.fi/exampleapp>.

The application exists only to demonstrate some basic concepts of the course, and is, by no means, an example of *how* a modern web application should be made. On the contrary, it demonstrates some old techniques of web development, which could even be considered *bad practices* nowadays.

Code will conform to contemporary best practices from [part 1](#) onwards.

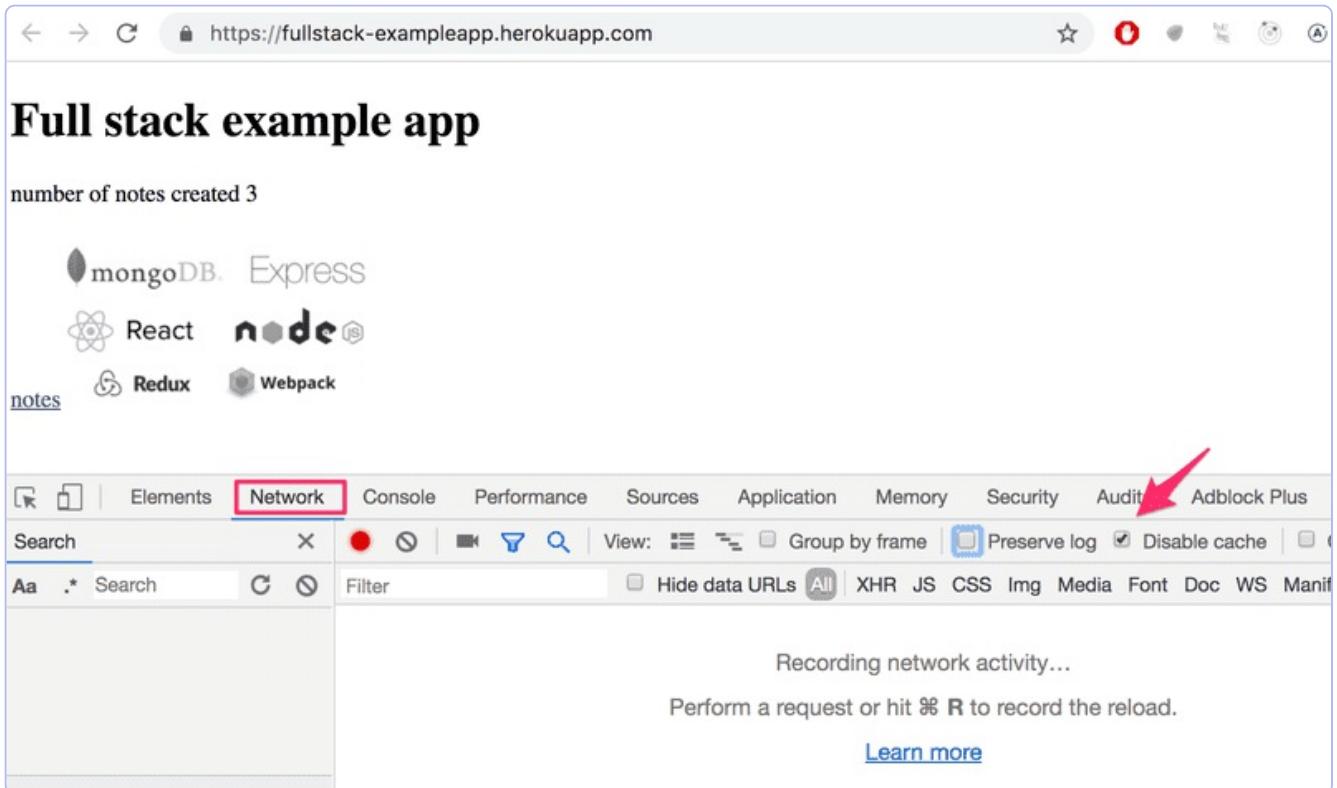
Open the [example application](#) in your browser. Sometimes this takes a while.

The course material is done with the Chrome browser.

The 1st rule of web development: Always keep the Developer Console open on your web browser. On macOS, open the console by pressing *fn-F12* or *option-cmd-i* simultaneously. On Windows or Linux, open the console by pressing *Fn-F12* or *ctrl-shift-i* simultaneously. The console can also be opened via the [context menu](#).

Remember to *always* keep the Developer Console open when developing web applications.

The console looks like this:



Make sure that the *Network* tab is open, and check the *Disable cache* option as shown. *Preserve log* can also be useful (it saves the logs printed by the application when the page is reloaded) as well as "Hide extension URLs"(hides requests of any extensions installed in the browser, not shown in the picture above).

NB: The most important tab is the *Console* tab. However, in this introduction, we will be using the *Network* tab quite a bit.

HTTP GET

The server and the web browser communicate with each other using the [HTTP](#) protocol. The *Network* tab shows how the browser and the server communicate.

When you reload the page (To refresh a webpage, on windows, press the *Fn-F5* keys. On macOS, press *command-R*. Or press the *⟳* symbol on your browser), the console will show that two events have happened:

- The browser has fetched the contents of the page studies.cs.helsinki.fi/exampleapp from the server
- And has downloaded the image *kuva.png*

2 requests | 88.1 KB transferred | 87.6 KB resources | Finish: 888 ms | DOMContentLoaded: 707 ms | Load: 903 ms

On a small screen, you might have to widen the console window to see these.

Clicking the first event reveals more information on what's happening:

2 requests | 88.1 KB trans...

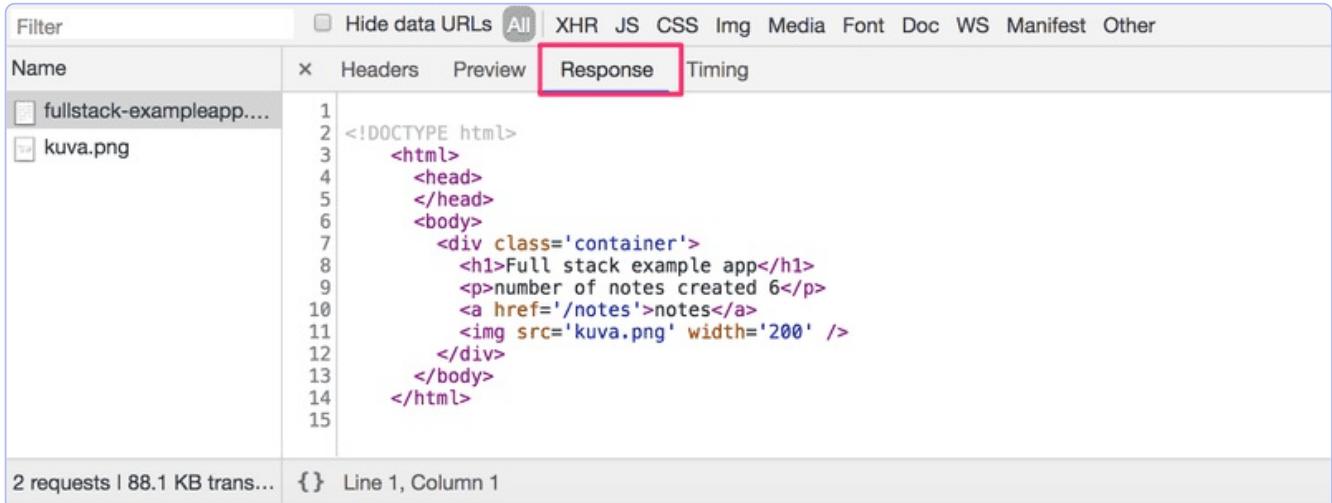
The upper part, *General*, shows that the browser requested the address <https://studies.cs.helsinki.fi/exampleapp> (though the address has changed slightly since this picture was taken) using the **GET** method, and that the request was successful, because the server response had the **Status code** 200.

The request and the server response have several **headers**:

2 requests | 88.1 KB trans... ► Request Headers (9)

The *Response headers* on top tell us e.g. the size of the response in bytes and the exact time of the response. An important header [Content-Type](#) tells us that the response is a text file in [utf-8](#) format and the contents of which have been formatted with HTML. This way the browser knows the response to be a regular [HTML](#) page and to render it to the browser 'like a web page'.

The *Response* tab shows the response data, a regular HTML page. The *body* section determines the structure of the page rendered to the screen:



```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     </head>
5   <body>
6     <div class='container'>
7       <h1>Full stack example app</h1>
8       <p>number of notes created 6</p>
9       <a href='/notes'>notes</a>
10      <img src='kuva.png' width='200' />
11    </div>
12  </body>
13 </html>
14
15

```

2 requests | 88.1 KB trans...

The page contains a [div](#) element, which in turn contains a heading, a link to the page *notes*, and an [img](#) tag, and displays the number of notes created.

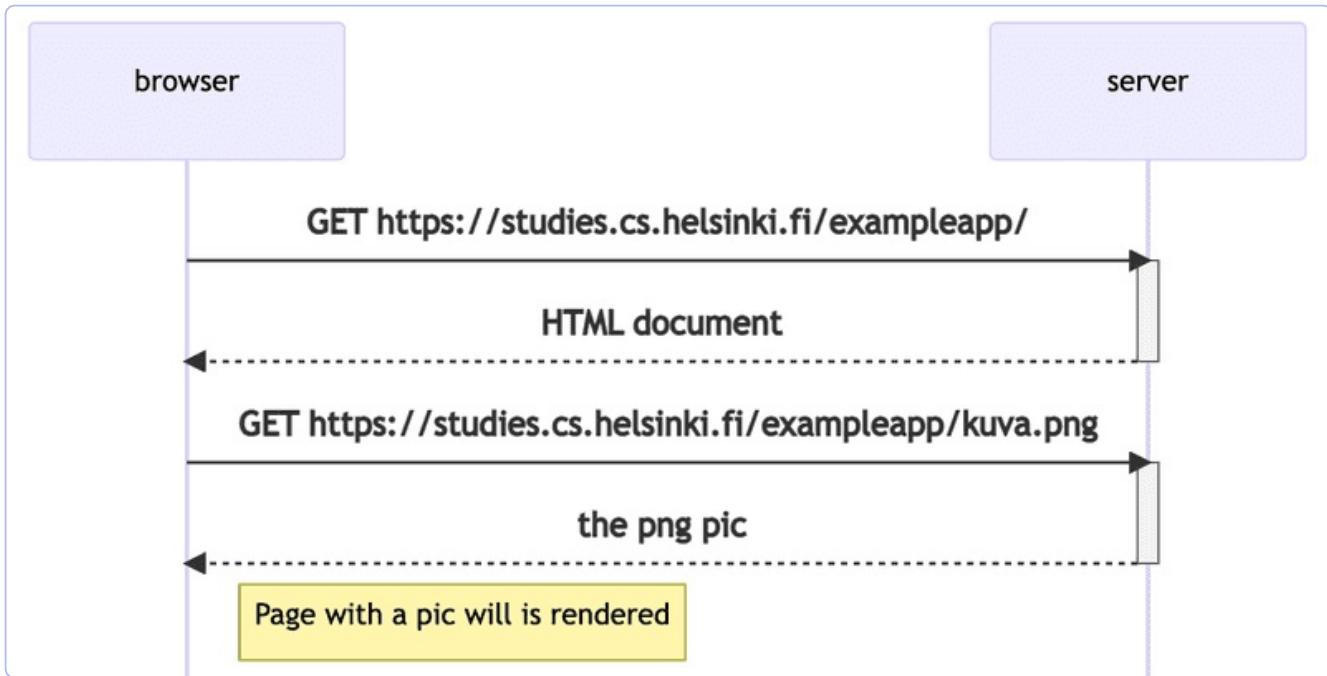
Because of the [img](#) tag, the browser does a second *HTTP request* to fetch the image *kuva.png* from the server. The details of the request are as follows:

Name	Headers	Preview	Response	Timing
fullstack-exampleapp....				
kuva.png	General <p>Request URL: https://fullstack-exampleapp.herokuapp.com/kuva.png</p> <p>Request Method: GET</p> <p>Status Code: 200 OK</p> <p>Remote Address: 52.21.245.216:443</p> <p>Referrer Policy: no-referrer-when-downgrade</p>			
	Response Headers <p>Accept-Ranges: bytes</p> <p>Cache-Control: public, max-age=0</p> <p>Connection: keep-alive</p> <p>Content-Length: 89350</p> <p>Content-Type: image/png</p> <p>Date: Fri, 24 May 2019 12:40:55 GMT</p>			

2 requests | 88.1 KB trans...

The request was made to the address <https://studies.cs.helsinki.fi/exampleapp/kuva.png> and its type is HTTP GET. The response headers tell us that the response size is 89350 bytes, and its [Content-type](#) is *image/png*, so it is a png image. The browser uses this information to render the image correctly to the screen.

The chain of events caused by opening the page <https://studies.cs.helsinki.fi/exampleapp> on a browser forms the following [sequence diagram](#):



The sequence diagram visualizes how the browser and server are communicating over time. The time flows in the diagram from top to bottom, so the diagram starts with the first request that the browser sends to server, followed by the response.

First, the browser sends an HTTP GET request to the server to fetch the HTML code of the page. The *img* tag in the HTML prompts the browser to fetch the image *kuva.png*. The browser renders the HTML page and the image to the screen.

Even though it is difficult to notice, the HTML page begins to render before the image has been fetched from the server.

Traditional web applications

The homepage of the example application works like a *traditional web application*. When entering the page, the browser fetches the HTML document detailing the structure and the textual content of the page from the server.

The server has formed this document somehow. The document can be a *static* text file saved into the server's directory. The server can also form the HTML documents *dynamically* according to the application's code, using, for example, data from a database. The HTML code of the example application has been formed dynamically because it contains information on the number of created notes.

The HTML code of the homepage is formed dynamically on the server as follows:

```

const getFrontPageHtml = noteCount => [
  return `
    <!DOCTYPE html>
    <html>
      <head>
      </head>
      <body>
        <div class='container'>
          <h1>Full stack example app</h1>

```

```

<p>number of notes created ${noteCount}</p>
<a href='/notes'>notes</a>
<img src='kuva.png' width='200' />
</div>
</body>
</html>
`

}

app.get('/', (req, res) => {
  const page = getFrontPageHtml(notes.length)
  res.send(page)
})

```

You don't have to understand the code just yet.

The content of the HTML page has been saved as a template string or a string that allows for evaluating, for example, variables, like *noteCount*, in the midst of it. The dynamically changing part of the homepage, the number of saved notes (in the code *noteCount*), is replaced by the current number of notes (in the code *notes.length*) in the template string.

Writing HTML amid the code is of course not smart, but for old-school PHP programmers, it was a normal practice.

In traditional web applications, the browser is "dumb". It only fetches HTML data from the server, and all application logic is on the server. A server can be created using [Java Spring](#) , [Python Flask](#) or [Ruby on Rails](#) to name just a few examples.

The example uses [Express](#) library with Node.js. This course will use Node.js and Express to create web servers.

Running application logic in the browser

Keep the Developer Console open. Empty the console by clicking the  symbol, or by typing clear() in the console. Now when you go to the [notes](#) page, the browser does 4 HTTP requests:

Name	Status	Type	Initiator	Size	Time
notes	200	document	Other	644 B	1
main.css	200	stylesheet	notes	400 B	1
main.js	200	script	notes	906 B	1
data.json	200	xhr	main.js:23	660 B	1

All of the requests have *different* types. The first request's type is *document*. It is the HTML code of the page, and it looks as follows:

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <link rel="stylesheet" type="text/css" href="/main.css" />
5     <script type="text/javascript" src="main.js"></script>
6   </head>
7   <body>
8     <div class='container'>
9       <h1>Notes</h1>
10      <div id='notes'>
11        </div>
12      <form action='/new_note' method='POST'>
13        <input type="text" name="note"><br>
14        <input type="submit" value="Save">
15      </form>
16    </div>
17  </body>
18</html>
19
20

```

When we compare the page shown on the browser and the HTML code returned by the server, we notice that the code does not contain the list of notes. The `head` section of the HTML contains a `script` tag, which causes the browser to fetch a JavaScript file called *main.js*.

The JavaScript code looks as follows:

```

var xhttp = new XMLHttpRequest()

xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    const data = JSON.parse(this.responseText)
  }
}

```

```

console.log(data)

var ul = document.createElement('ul')
ul.setAttribute('class', 'notes')

data.forEach(function(note) {
  var li = document.createElement('li')

  ul.appendChild(li)
  li.appendChild(document.createTextNode(note.content))
})

document.getElementById('notes').appendChild(ul)
}

}

xhttp.open('GET', '/data.json', true)
xhttp.send()

```

The details of the code are not important right now, but some code has been included to spice up the images and the text. We will properly start coding in [part 1](#). The sample code in this part is actually not relevant at all to the coding techniques of this course.

Some might wonder why xhttp-object is used instead of the modern fetch. This is due to not wanting to go into promises at all yet, and the code having a secondary role in this part. We will return to modern ways to make requests to the server in [part 2](#).

Immediately after fetching the `script` tag, the browser begins to execute the code.

The last two lines instruct the browser to do an HTTP GET request to the server's address `/data.json`:

```

xhttp.open('GET', '/data.json', true)
xhttp.send()

```

This is the bottom-most request shown on the Network tab.

We can try going to the address <https://studies.cs.helsinki.fi/exampleapp/data.json> straight from the browser:



There we find the notes in [JSON](#) "raw data". By default, Chromium-based browsers are not too good at displaying JSON data. Plugins can be used to handle the formatting. Install, for example, [JSONVue](#) on Chrome, and reload the page. The data is now nicely formatted:

```
[  
- {  
  content: "HTML is easy",  
  date: "2019-05-23T17:30:31.098Z"  
},  
- {  
  content: "Browser can execute only Javascript",  
  date: "2019-05-23T18:39:34.091Z"  
},  
- {  
  content: "Most important methods of HTTP-protocol are GET and POST",  
  date: "2019-05-23T19:20:14.298Z"  
},  
- {  
  content: "kolmas",  
  date: "2019-05-24T12:37:59.879Z"  
},
```

So, the JavaScript code of the notes page above downloads the JSON data containing the notes, and forms a bullet-point list from the note contents:

This is done by the following code:

```
const data = JSON.parse(this.responseText)  
console.log(data)  
  
var ul = document.createElement('ul')  
ul.setAttribute('class', 'notes')  
  
data.forEach(function(note) {  
  var li = document.createElement('li')  
  
  ul.appendChild(li)  
  li.appendChild(document.createTextNode(note.content))  
})  
  
document.getElementById('notes').appendChild(ul)
```

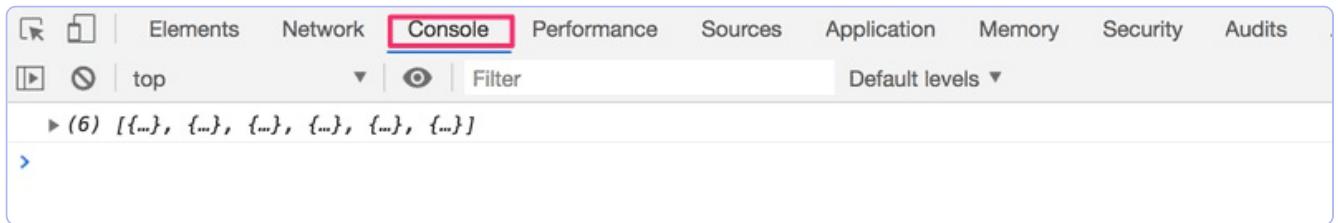
The code first creates an unordered list with a `ul`-tag...

```
var ul = document.createElement('ul')  
ul.setAttribute('class', 'notes')
```

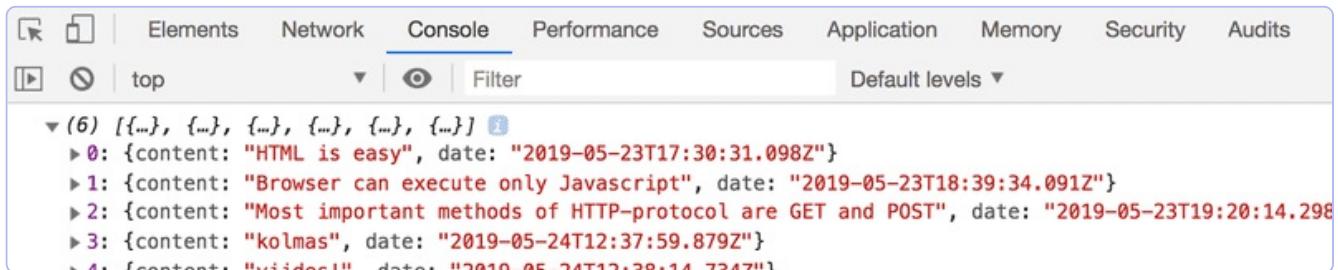
...and then adds one `li`-tag for each note. Only the `content` field of each note becomes the contents of the `li`-tag. The timestamps found in the raw data are not used for anything here.

```
data.forEach(function(note) {  
  var li = document.createElement('li')  
  
  ul.appendChild(li)  
  li.appendChild(document.createTextNode(note.content))  
})
```

Now open the *Console* tab on your Developer Console:



By clicking the little triangle at the beginning of the line, you can expand the text on the console.



This output on the console is caused by the *console.log* command in the code:

```
const data = JSON.parse(this.responseText)
console.log(data)
```

So, after receiving data from the server, the code prints it to the console.

The *Console* tab and the *console.log* command will become very familiar to you during the course.

Event handlers and Callback functions

The structure of this code is a bit odd:

```
var xhttp = new XMLHttpRequest()

xhttp.onreadystatechange = function() {
    // code that takes care of the server response
}

xhttp.open('GET', '/data.json', true)
xhttp.send()
```

The request to the server is sent on the last line, but the code to handle the response can be found further up. What's going on?

```
xhttp.onreadystatechange = function () {
```

On this line, an *event handler* for the event *onreadystatechange* is defined for the *xhttp* object doing the request. When the state of the object changes, the browser calls the event handler function. The function code checks that the *readyState* equals 4 (which depicts the situation *The operation is complete*) and that the HTTP status code of the response is 200.

```

xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    // code that takes care of the server response
  }
}

```

The mechanism of invoking event handlers is very common in JavaScript. Event handler functions are called **callback** functions. The application code does not invoke the functions itself, but the runtime environment - the browser, invokes the function at an appropriate time when the *event* has occurred.

Document Object Model or DOM

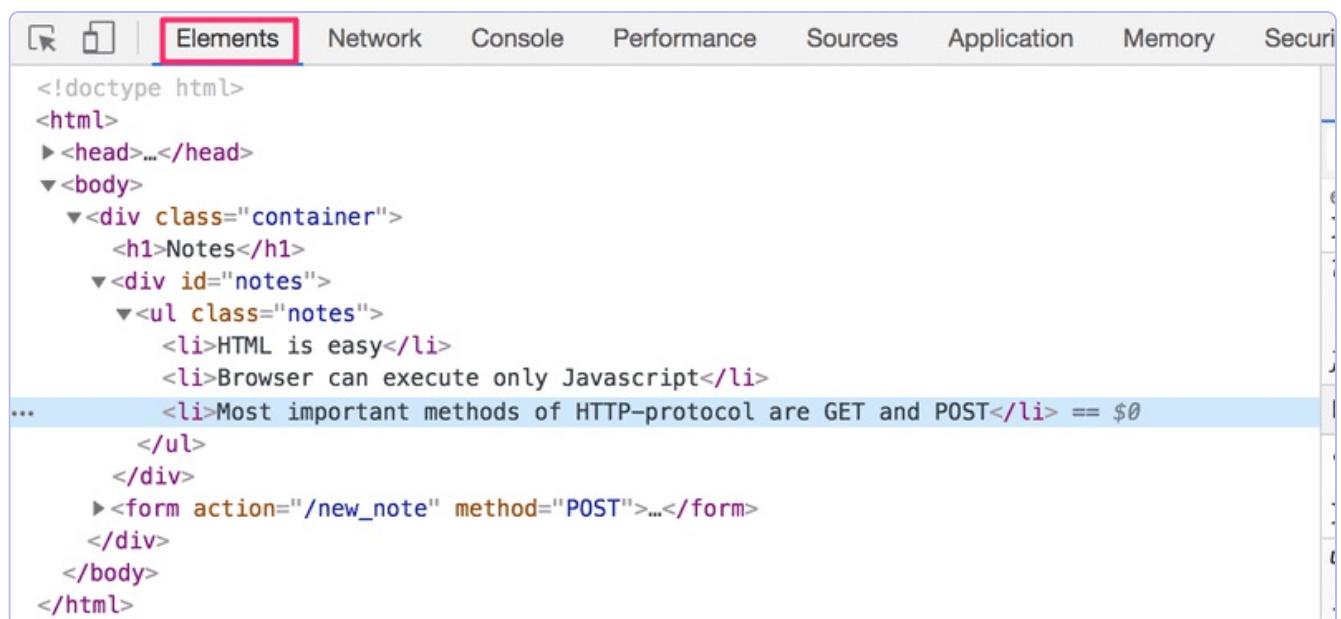
We can think of HTML pages as implicit tree structures.

```

html
  head
    link
    script
  body
    div
      h1
      div
        ul
          li
          li
          li
    form
      input
      input

```

The same treelike structure can be seen on the console tab *Elements*.



The functioning of the browser is based on the idea of depicting HTML elements as a tree.

Document Object Model, or **DOM**, is an Application Programming Interface (**API**) that enables programmatic modification of the *element trees* corresponding to web pages.

The JavaScript code introduced in the previous chapter used the DOM-API to add a list of notes to the page.

The following code creates a new node to the variable *ul*, and adds some child nodes to it:

```
var ul = document.createElement('ul')

data.forEach(function(note) {
  var li = document.createElement('li')

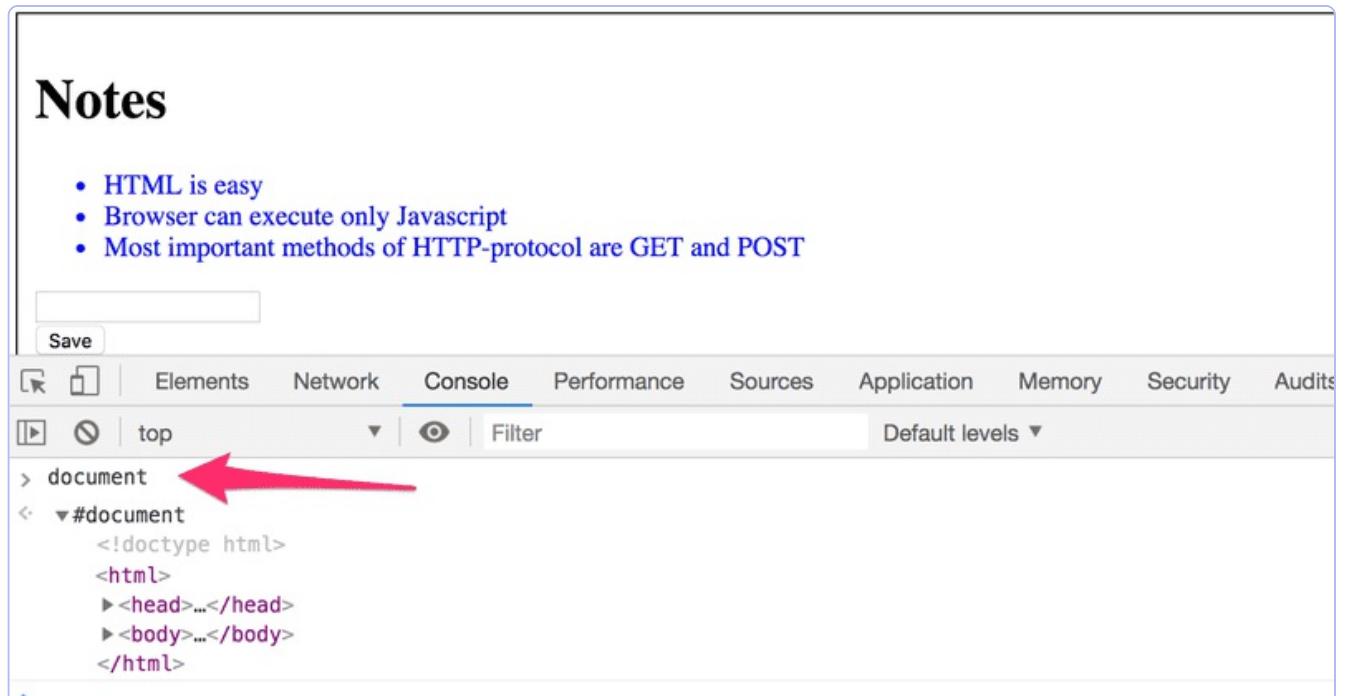
  ul.appendChild(li)
  li.appendChild(document.createTextNode(note.content))
})
```

Finally, the tree branch of the *ul* variable is connected to its proper place in the HTML tree of the whole page:

```
document.getElementById('notes').appendChild(ul)
```

Manipulating the document object from console

The topmost node of the DOM tree of an HTML document is called the *document* object. We can perform various operations on a webpage using the DOM-API. You can access the *document* object by typing *document* into the Console tab:



Let's add a new note to the page from the console.

First, we'll get the list of notes from the page. The list is in the first ul-element of the page:

```
list = document.getElementsByTagName('ul')[0]
```

Then create a new li-element and add some text content to it:

```
newElement = document.createElement('li')
newElement.textContent = 'Page manipulation from console is easy'
```

And add the new li-element to the list:

```
list.appendChild(newElement)
```

Notes

- HTML is easy
- Browser can execute only Javascript
- Most important methods of HTTP-protocol are GET and POST
- Page manipulation from console is easy

Save

Elements Network **Console** Performance Sources Application Memory Security Audits

top Filter Default levels ▾

```
> list = document.getElementsByTagName('ul')[0]
<- ><ul class="notes">..</ul>
> newElement = document.createElement('li')
<- <li></li>
> newElement.textContent = 'Page manipulation from console is easy'
<- "Page manipulation from console is easy"
> list.appendChild(newElement)
<- <li>Page manipulation from console is easy</li>
`-
```

Even though the page updates on your browser, the changes are not permanent. If the page is reloaded, the new note will disappear, because the changes were not pushed to the server. The JavaScript code the browser fetches will always create the list of notes based on JSON data from the address <https://studies.cs.helsinki.fi/exampleapp/data.json>.

CSS

The `head` element of the HTML code of the Notes page contains a `link` tag, which determines that the browser must fetch a `CSS` style sheet from the address `main.css`.

Cascading Style Sheets, or CSS, is a style sheet language used to determine the appearance of web pages.

The fetched CSS file looks as follows:

```
.container {
  padding: 10px;
  border: 1px solid;
```

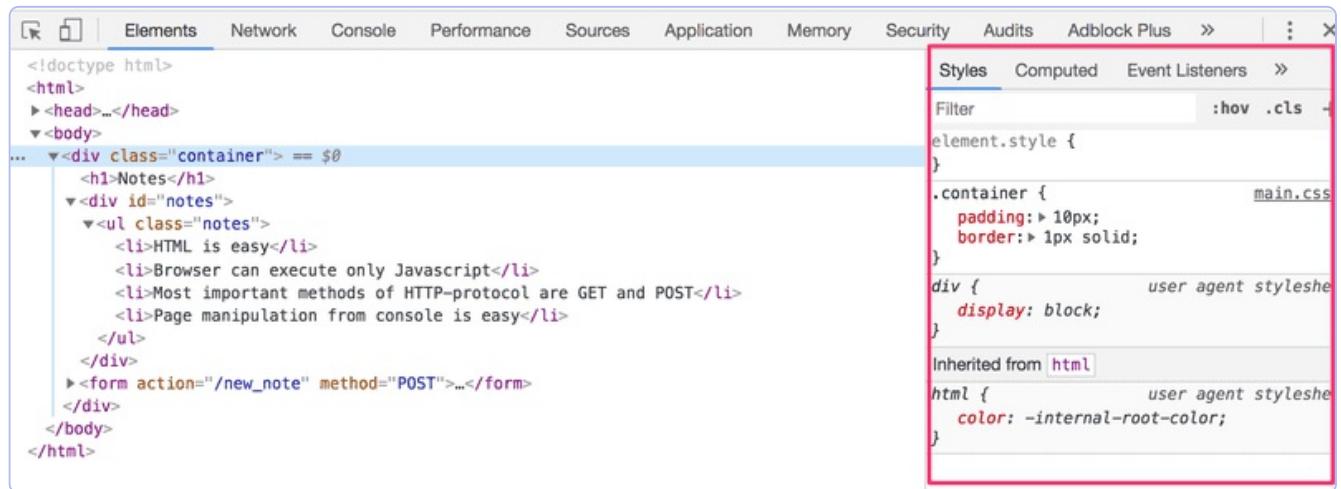
```
.notes {  
    color: blue;  
}
```

The file defines two **class selectors**. These are used to select certain parts of the page and to define styling rules to style them.

A class selector definition always starts with a period and contains the name of the class.

Classes are **attributes**, which can be added to HTML elements.

CSS attributes can be examined on the *elements* tab of the console:



The outermost `div` element has the class *container*. The `ul` element containing the list of notes has the class *notes*.

The CSS rule defines that elements with the *container* class will be outlined with a one-pixel wide **border**. It also sets 10-pixel **padding** on the element. This adds some empty space between the element's content and the border.

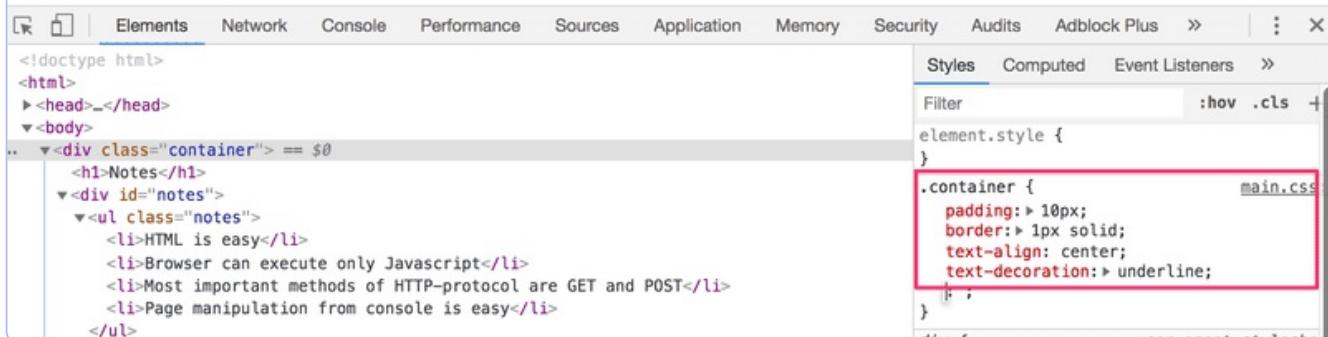
The second CSS rule sets the text color of the *notes* class as blue.

HTML elements can also have other attributes apart from classes. The `div` element containing the notes has an **id** attribute. JavaScript code uses the id to find the element.

The *Elements* tab of the console can be used to change the styles of the elements.

Notes

- HTML is easy
- Browser can execute only Javascript
- Most important methods of HTTP-protocol are GET and POST
- Page manipulation from console is easy



The screenshot shows the Chrome DevTools Elements tab. On the left, the HTML structure of a notes page is displayed, including an

Notes

 heading and a

 list with four items. On the right, the Styles panel shows the CSS rule for the container class defined in the main.css file.

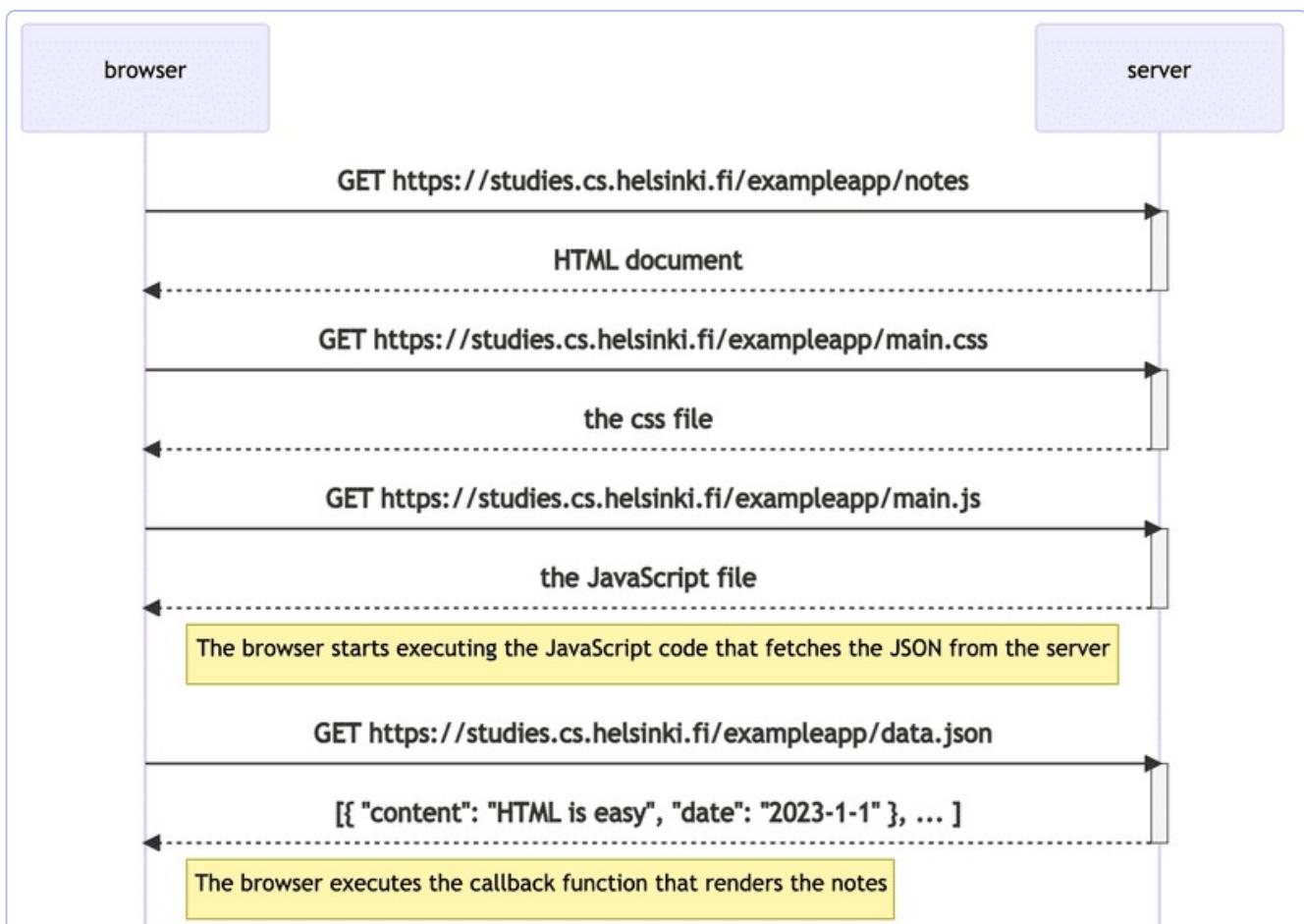
```
<!doctype html>
<html>
  <head></head>
  <body>
    <div class="container"> == $0
      <h1>Notes</h1>
      <div id="notes">
        <ul class="notes">
          <li>HTML is easy</li>
          <li>Browser can execute only Javascript</li>
          <li>Most important methods of HTTP-protocol are GET and POST</li>
          <li>Page manipulation from console is easy</li>
        </ul>
      </div>
    </div>
  </body>
</html>
```

```
.container {
  padding: 10px;
  border: 1px solid black;
  text-align: center;
  text-decoration: underline;
}
```

Changes made on the console will not be permanent. If you want to make lasting changes, they must be saved to the CSS style sheet on the server.

Loading a page containing JavaScript - review

Let's review what happens when the page <https://studies.cs.helsinki.fi/exampleapp/notes> is opened on the browser.



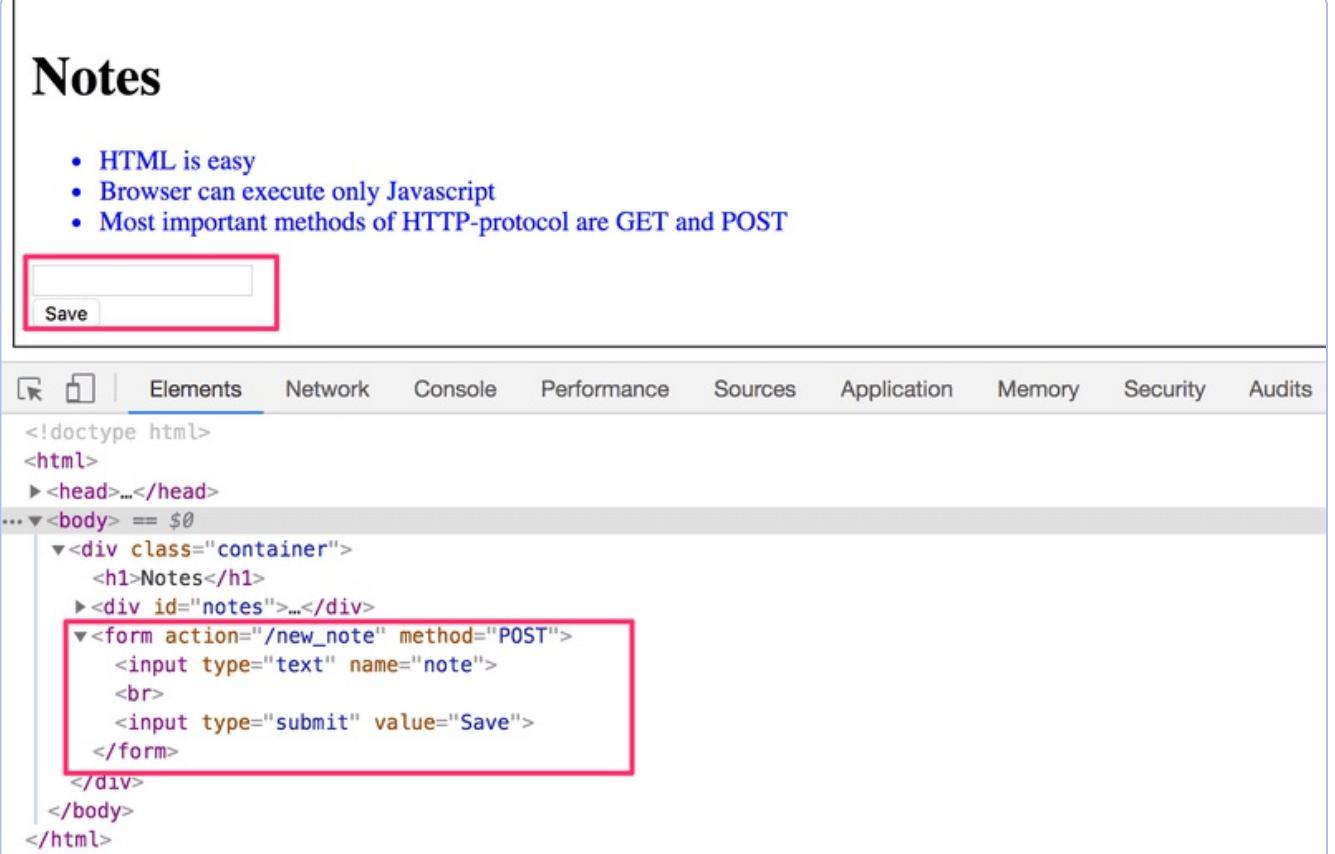
- The browser fetches the HTML code defining the content and the structure of the page from the server using an HTTP GET request.

- Links in the HTML code cause the browser to also fetch the CSS style sheet *main.css*...
- ...and the JavaScript code file *main.js*
- The browser executes the JavaScript code. The code makes an HTTP GET request to the address <https://studies.cs.helsinki.fi/exampleapp/data.json>, which returns the notes as JSON data.
- When the data has been fetched, the browser executes an *event handler*, which renders the notes to the page using the DOM-API.

Forms and HTTP POST

Next, let's examine how adding a new note is done.

The Notes page contains a [form element](#).



The screenshot shows the Chrome DevTools Elements tab with the HTML source code of the 'Notes' page. A red box highlights the `<form>` element, and another red box highlights the `<input type="submit" value="Save">` button. The HTML code is as follows:

```

<!doctype html>
<html>
  <head>...</head>
  <body> == $0
    <div class="container">
      <h1>Notes</h1>
      <div id="notes">...</div>
      <form action="/new_note" method="POST">
        <input type="text" name="note">
        <br>
        <input type="submit" value="Save">
      </form>
    </div>
  </body>
</html>

```

When the button on the form is clicked, the browser will send the user input to the server. Let's open the *Network* tab and see what submitting the form looks like:

- form data is sent with HTTP POST

Name	Status	Type	Initiator	Size
new_note	302	text/html	Other	2
notes	200	document	new_note	64
main.css	200	stylesheet	notes	40
main.js	200	script	notes	90
data.json	200	xhr	main.js:23	51

Surprisingly, submitting the form causes no fewer than *five* HTTP requests. The first one is the form submit event. Let's zoom into it:

Name	Headers	Preview	Response	Timing
new_note			<p>Request URL: https://fullstack-exampleapp.herokuapp.com/new_note</p> <p>Request Method: POST</p> <p>Status Code: 302 Found</p> <p>Remote Address: 52.6.165.91:443</p> <p>Referrer Policy: no-referrer-when-downgrade</p> <p>Response Headers</p> <p>Connection: keep-alive</p> <p>Content-Length: 56</p> <p>Content-Type: text/html; charset=utf-8</p> <p>Date: Sat, 25 May 2019 09:05:12 GMT</p> <p>Location: /notes</p>	

It is an [HTTP POST](#) request to the server address *new_note*. The server responds with HTTP status code 302. This is a [URL redirect](#), with which the server asks the browser to perform a new HTTP GET request to the address defined in the header's *Location* - the address *notes*.

So, the browser reloads the Notes page. The reload causes three more HTTP requests: fetching the style sheet (main.css), the JavaScript code (main.js), and the raw data of the notes (data.json).

The network tab also shows the data submitted with the form:

The Form Data dropdown is within the new Payload tab, located to the right of the Headers tab.

Filter Hide data URLs **All** XHR JS CSS Img Media Font Doc WS Manifest Other

Name	Headers	Preview	Response	Timing
<input type="checkbox"/> new_note				
<input type="checkbox"/> notes				
<input type="checkbox"/> main.css				
<input type="checkbox"/> main.js				
<input type="checkbox"/> data.json				

General

Request URL: https://fullstack-exampleapp.herokuapp.com/new_note
 Request Method: POST
 Status Code: 302 Found
 Remote Address: 52.6.165.91:443
 Referrer Policy: no-referrer-when-downgrade

▶ Response Headers (9)

▶ Request Headers (13)

▼ Form Data [view source](#) [view URL encoded](#)

note: form data is sent with HTTP POST

The Form tag has attributes *action* and *method*, which define that submitting the form is done as an HTTP POST request to the address *new_note*.

Elements Network Console Performance Sources Application Memory Security

```
<!doctype html>
<html>
  <head>...</head>
  ... <body> == $0
    <div class="container">
      <h1>Notes</h1>
      <div id="notes">...</div>
      <form action="/new_note" method="POST">
        <input type="text" name="note">
        <br>
        <input type="submit" value="Save">
      </form>
    </div>
  </body>
</html>
```

The code on the server responsible for the POST request is quite simple (NB: this code is on the server, and not on the JavaScript code fetched by the browser):

```
app.post('/new_note', (req, res) => {
  notes.push({
    content: req.body.note,
    date: new Date(),
  })

  return res.redirect('/notes')
})
```

Data is sent as the **body** of the POST request.

The server can access the data by accessing the *req.body* field of the request object *req*.

The server creates a new note object, and adds it to an array called *notes*.

```
notes.push({
```

```
    content: req.body.note,
    date: new Date(),
  })
}
```

Each note object has two fields: *content* containing the actual content of the note, and *date* containing the date and time the note was created.

The server does not save new notes to a database, so new notes disappear when the server is restarted.

AJAX

The Notes page of the application follows an early-nineties style of web development and uses "Ajax". As such, it's on the crest of the wave of early 2000s web technology.

[AJAX](#) (Asynchronous JavaScript and XML) is a term introduced in February 2005 on the back of advancements in browser technology to describe a new revolutionary approach that enabled the fetching of content to web pages using JavaScript included within the HTML, without the need to rerender the page.

Before the AJAX era, all web pages worked like the [traditional web application](#) we saw earlier in this chapter. All of the data shown on the page was fetched with the HTML code generated by the server.

The Notes page uses AJAX to fetch the notes data. Submitting the form still uses the traditional mechanism of submitting web forms.

The application URLs reflect the old, carefree times. JSON data is fetched from the URL <https://studies.cs.helsinki.fi/exampleapp/data.json> and new notes are sent to the URL https://studies.cs.helsinki.fi/exampleapp/new_note. Nowadays URLs like these would not be considered acceptable, as they don't follow the generally acknowledged conventions of [RESTful](#) APIs, which we'll look into more in [part 3](#).

The thing termed AJAX is now so commonplace that it's taken for granted. The term has faded into oblivion, and the new generation has not even heard of it.

Single page app

In our example app, the home page works like a traditional webpage: All of the logic is on the server, and the browser only renders the HTML as instructed.

The Notes page gives some of the responsibility, generating the HTML code for existing notes, to the browser. The browser tackles this task by executing the JavaScript code it fetched from the server. The code fetches the notes from the server as JSON data and adds HTML elements for displaying the notes to the page using the [DOM-API](#).

In recent years, the [Single-page application](#) (SPA) style of creating web applications has emerged. SPA-style websites don't fetch all of their pages separately from the server like our sample application does, but instead comprise only one HTML page fetched from the server, the contents of which are manipulated with JavaScript that executes in the browser.

The Notes page of our application bears some resemblance to SPA-style apps, but it's not quite there yet. Even though the logic for rendering the notes is run on the browser, the page still uses the traditional way of adding new notes. The data is sent to the server via the form's submit, and the

server instructs the browser to reload the Notes page with a *redirect*.

A single-page app version of our example application can be found at <https://studies.cs.helsinki.fi/exampleapp/spa>. At first glance, the application looks exactly the same as the previous one. The HTML code is almost identical, but the JavaScript file is different (*spa.js*) and there is a small change in how the form-tag is defined:

The screenshot shows a browser window with the URL <https://fullstack-exampleapp.herokuapp.com/spa>. The title of the page is "Notes -- single page app". Below the title is a bulleted list of four items. The browser's developer tools are open, specifically the "Elements" tab. The DOM tree is displayed, showing the HTML structure. A red box highlights the `<form>` element and its contents. The highlighted code is as follows:

```
<!doctype html>
<html>
  <head>...</head>
  <body>
    <div class="container">
      <h1>Notes -- single page app</h1>
      ... <div id="notes"> == $0
        <ul class="notes">
          <li>HTML is easy</li>
          <li>Browser can execute only Javascript</li>
          <li>Most important methods of HTTP-protocol are GET and POST</li>
          <li>form data is sent with HTTP POST</li>
        </ul>
      </div>
      <div id="notes_form">
        <form id="notes_form">
          <input type="text" name="note">
          <br>
          <input type="submit" value="Save">
        </form>
      </div>
    </div>
  </body>
</html>
```

The form has no *action* or *method* attributes to define how and where to send the input data.

Open the *Network* tab and empty it. When you now create a new note, you'll notice that the browser sends only one request to the server.

Notes -- single page app

- HTML is easy
- Browser can execute only Javascript
- Most important methods of HTTP-protocol are GET and POST
- form data is sent with HTTP POST
- single page app does not reload the whole page

The screenshot shows the Network tab in the Chrome DevTools. A single request is listed: "new_note_spa". The "General" section shows the following details:

- Request URL: https://fullstack-exampleapp.herokuapp.com/new_note_spa
- Request Method: POST
- Status Code: 201 Created
- Remote Address: 35.175.30.164:443
- Referrer Policy: no-referrer-when-downgrade

Below these details are sections for "Response Headers (8)", "Request Headers (12)", and "Request Payload". The "Request Payload" section contains the JSON data sent in the POST request:

```
{content: "single page app does not reload the whole page", date: "2019-05-25T09:15:59.905Z"}  
content: "single page app does not reload the whole page"  
date: "2019-05-25T09:15:59.905Z"
```

At the bottom left, it says "1 requests | 275 B transfer...".

The POST request to the address *new_note_spa* contains the new note as JSON data containing both the content of the note (*content*) and the timestamp (*date*):

```
content: "single page app does not reload the whole page",  
date: "2019-05-25T15:15:59.905Z"
```

The *Content-Type* header of the request tells the server that the included data is represented in JSON format.

The screenshot shows the Request Headers section in the Chrome DevTools. The "Content-type" header is highlighted with a red box:

```
Accept: */*  
Accept-Encoding: gzip, deflate, br  
Accept-Language: en-US,en;q=0.9,fi;q=0.8,la;q=0.7  
Cache-Control: no-cache  
Connection: keep-alive  
Content-Length: 94  
Content-type: application/json  
Host: fullstack-exampleapp.herokuapp.com  
Origin: https://fullstack-exampleapp.herokuapp.com  
Pragma: no-cache
```

At the bottom left, it says "1 requests | 275 B transfer...".

Without this header, the server would not know how to correctly parse the data.

The server responds with status code **201 created**. This time the server does not ask for a redirect, the browser stays on the same page, and it sends no further HTTP requests.

The SPA version of the app does not traditionally send the form data, but instead uses the JavaScript code it fetched from the server. We'll look into this code a bit, even though understanding all the details of it is not important just yet.

```
var form = document.getElementById('notes_form')
form.onsubmit = function(e) {
    e.preventDefault()

    var note = {
        content: e.target.elements[0].value,
        date: new Date(),
    }

    notes.push(note)
    e.target.elements[0].value = ''
    redrawNotes()
    sendToServer(note)
}
```

The command `document.getElementById('notes_form')` instructs the code to fetch the form element from the page and to register an *event handler* to handle the form's submit event. The event handler immediately calls the method `e.preventDefault()` to prevent the default handling of form's submit. The default method would send the data to the server and cause a new GET request, which we don't want to happen.

Then the event handler creates a new note, adds it to the notes list with the command `notes.push(note)`, rerenders the note list on the page and sends the new note to the server.

The code for sending the note to the server is as follows:

```
var sendToServer = function(note) {
    var xhttpForPost = new XMLHttpRequest()
    // ...

    xhttpForPost.open('POST', '/new_note_spa', true)
    xhttpForPost.setRequestHeader('Content-type', 'application/json')
    xhttpForPost.send(JSON.stringify(note))
}
```

The code determines that the data is to be sent with an HTTP POST request and the data type is to be JSON. The data type is determined with a *Content-type* header. Then the data is sent as JSON string.

The application code is available at https://github.com/mluukkai/example_app. It's worth remembering that the application is only meant to demonstrate the concepts of the course. The code follows a poor style of development in some measures, and should not be used as an example when creating your applications. The same is true for the URLs used. The URL `new_note_spa` that new notes are sent to, does not adhere to current best practices.

JavaScript-libraries

The sample app is done with so-called [vanilla JavaScript](#), using only the DOM-API and JavaScript to

manipulate the structure of the pages.

Instead of using JavaScript and the DOM-API only, different libraries containing tools that are easier to work with compared to the DOM-API are often used to manipulate pages. One of these libraries is the ever-so-popular [jQuery](#).

jQuery was developed back when web applications mainly followed the traditional style of the server generating HTML pages, the functionality of which was enhanced on the browser side using JavaScript written with jQuery. One of the reasons for the success of jQuery was its so-called cross-browser compatibility. The library worked regardless of the browser or the company that made it, so there was no need for browser-specific solutions. Nowadays using jQuery is not as justified given the advancement of JavaScript, and the most popular browsers generally support basic functionalities well.

The rise of the single-page app brought several more "modern" ways of web development than jQuery. The favorite of the first wave of developers was [BackboneJS](#). After its [launch](#) in 2012, Google's [AngularJS](#) quickly became almost the de facto standard of modern web development.

However, the popularity of Angular plummeted in October 2014 after the [Angular team announced that support for version 1 will end](#), and that Angular 2 will not be backwards compatible with the first version. Angular 2 and the newer versions have not received too warm of a welcome.

Currently, the most popular tool for implementing the browser-side logic of web applications is Facebook's [React](#) library. During this course, we will get familiar with React and the [Redux](#) library, which are frequently used together.

The status of React seems strong, but the world of JavaScript is ever-changing. For example, recently a newcomer - [VueJS](#) - has been capturing some interest.

Full-stack web development

What does the name of the course, *Full stack web development*, mean? Full stack is a buzzword that everyone talks about, but no one knows what it means. Or at least, there is no agreed-upon definition for the term.

Practically all web applications have (at least) two "layers": the browser, being closer to the end-user, is the top layer, and the server the bottom one. There is often also a database layer below the server. We can therefore think of the *architecture* of a web application as a *stack* of layers.

Often, we also talk about the [frontend and the backend](#). The browser is the frontend, and the JavaScript that runs on the browser is the frontend code. The server on the other hand is the backend.

In the context of this course, full-stack web development means that we focus on all parts of the application: the frontend, the backend, and the database. Sometimes the software on the server and its operating system are seen as parts of the stack, but we won't go into those.

We will code the backend with JavaScript, using the [Node.js](#) runtime environment. Using the same programming language on multiple layers of the stack gives full-stack web development a whole new dimension. However, it's not a requirement of full-stack web development to use the same programming language (JavaScript) for all layers of the stack.

It used to be more common for developers to specialize in one layer of the stack, for example, the backend. Technologies on the backend and the frontend were quite different. With the Full stack trend, it has become common for developers to be proficient in all layers of the application and the

database. Oftentimes, full-stack developers must also have enough configuration and administration skills to operate their applications, for example, in the cloud.

JavaScript fatigue

Full-stack web development is challenging in many ways. Things are happening in many places at once, and debugging is quite a bit harder than with regular desktop applications. JavaScript does not always work as you'd expect it to (compared to many other languages), and the asynchronous way its runtime environments work causes all sorts of challenges. Communicating on the web requires knowledge of the HTTP protocol. One must also handle databases and server administration and configuration. It would also be good to know enough CSS to make applications at least somewhat presentable.

The world of JavaScript develops fast, which brings its own set of challenges. Tools, libraries and the language itself are under constant development. Some are starting to get tired of the constant change, and have coined a term for it: *JavaScript fatigue*. See [How to Manage JavaScript Fatigue on auth0](#) or [JavaScript fatigue on Medium](#).

You will suffer from JavaScript fatigue yourself during this course. Fortunately for us, there are a few ways to smooth the learning curve, and we can start with coding instead of configuration. We can't avoid configuration completely, but we can merrily push ahead in the next few weeks while avoiding the worst of configuration hells.

Exercises 0.1.-0.6.

The exercises are submitted via GitHub, and by marking the exercises as done in the "my submissions" tab of the [submission system](#).

You can submit all of the exercises into the same repository, or use multiple different repositories. If you submit exercises from different parts into the same repository, name your directories well. If you use a private repository to submit the exercises, add *m luukkai* as a collaborator to it.

One good way to name the directories in your submission repository is as follows:

```
part0
part1
  courseinfo
  unicafe
  anecdotes
part2
  courseinfo
  phonebook
  countries
```

So, each part has its own directory, which contains a directory for each exercise set (like the *unicafe* exercises in part 1).

The exercises are submitted **one part at a time**. When you have submitted the exercises for a part, you can no longer submit any missed exercises for that part.

0.1: HTML

Review the basics of HTML by reading this tutorial from Mozilla: [HTML tutorial](#).

This exercise is not submitted to GitHub, it's enough to just read the tutorial

0.2: CSS

Review the basics of CSS by reading this tutorial from Mozilla: [CSS tutorial](#).

This exercise is not submitted to GitHub, it's enough to just read the tutorial

0.3: HTML forms

Learn about the basics of HTML forms by reading Mozilla's tutorial [Your first form](#).

This exercise is not submitted to GitHub, it's enough to just read the tutorial

0.4: New note diagram

In the section [Loading a page containing JavaScript - review](#), the chain of events caused by opening the page <https://studies.cs.helsinki.fi/exampleapp/notes> is depicted as a [sequence diagram](#)

The diagram was made as a GitHub Markdown-file using the [Mermaid](#)-syntax, as follows:

```
sequenceDiagram
    participant browser
    participant server

    browser->>server: GET https://studies.cs.helsinki.fi/exampleapp/notes
    activate server
    server-->>browser: HTML document
    deactivate server

    browser->>server: GET https://studies.cs.helsinki.fi/exampleapp/main.css
    activate server
    server-->>browser: the css file
    deactivate server

    browser->>server: GET https://studies.cs.helsinki.fi/exampleapp/main.js
    activate server
    server-->>browser: the JavaScript file
    deactivate server

    Note right of browser: The browser starts executing the JavaScript code that
    fetches the JSON from the server

    browser->>server: GET https://studies.cs.helsinki.fi/exampleapp/data.json
    activate server
    server-->>browser: [{ "content": "HTML is easy", "date": "2023-1-1" }, ... ]
    deactivate server

    Note right of browser: The browser executes the callback function that ren-
    ds the notes
```

Create a similar diagram depicting the situation where the user creates a new note on the page <https://studies.cs.helsinki.fi/exampleapp/notes> by writing something into the text field and clicking the *Save* button.

If necessary, show operations on the browser or on the server as comments on the diagram.

The diagram does not have to be a sequence diagram. Any sensible way of presenting the events is fine.

All necessary information for doing this, and the next two exercises, can be found in the text of [this part](#). The idea of these exercises is to read the text once more and to think through what is going on there. Reading the application [code](#) is not necessary, but it is of course possible.

You can do the diagrams with any program, but perhaps the easiest and the best way to do diagrams is the [Mermaid](#) syntax that is now implemented in [GitHub](#) Markdown pages!

0.5: Single page app diagram

Create a diagram depicting the situation where the user goes to the [single-page app](#) version of the notes app at <https://studies.cs.helsinki.fi/exampleapp/spa>.

0.6: New note in Single page app diagram

Create a diagram depicting the situation where the user creates a new note using the single-page version of the app.

This was the last exercise, and it's time to push your answers to GitHub and mark the exercises as done in the [submission system](#).

1: Introduction to React

1a: Introduction to React

We will now start getting familiar with probably the most important topic of this course, namely the [React](#) library. Let's start by making a simple React application as well as getting to know the core concepts of React.

The easiest way to get started by far is by using a tool called [Vite](#).

Let's create an application called *part1*, navigate to its directory and install the libraries:

```
# npm 6.x (outdated, but still used by some):
npm create vite@latest part1 --template react

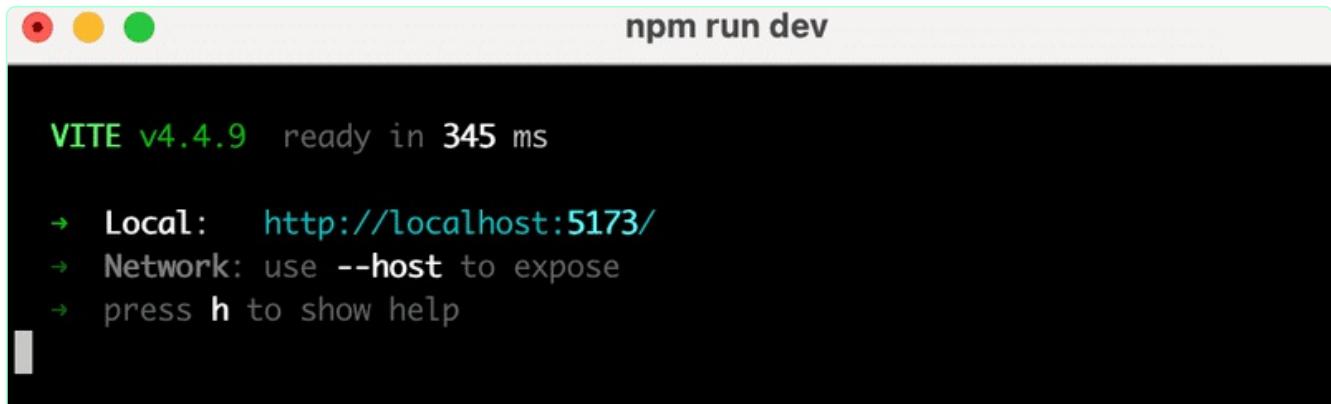
# npm 7+, extra double-dash is needed:
npm create vite@latest part1 -- --template react
```

```
cd part1
npm install
```

The application is started as follows

```
npm run dev
```

The console says that the application has started on localhost port 5173, i.e. the address <http://localhost:5173/>:



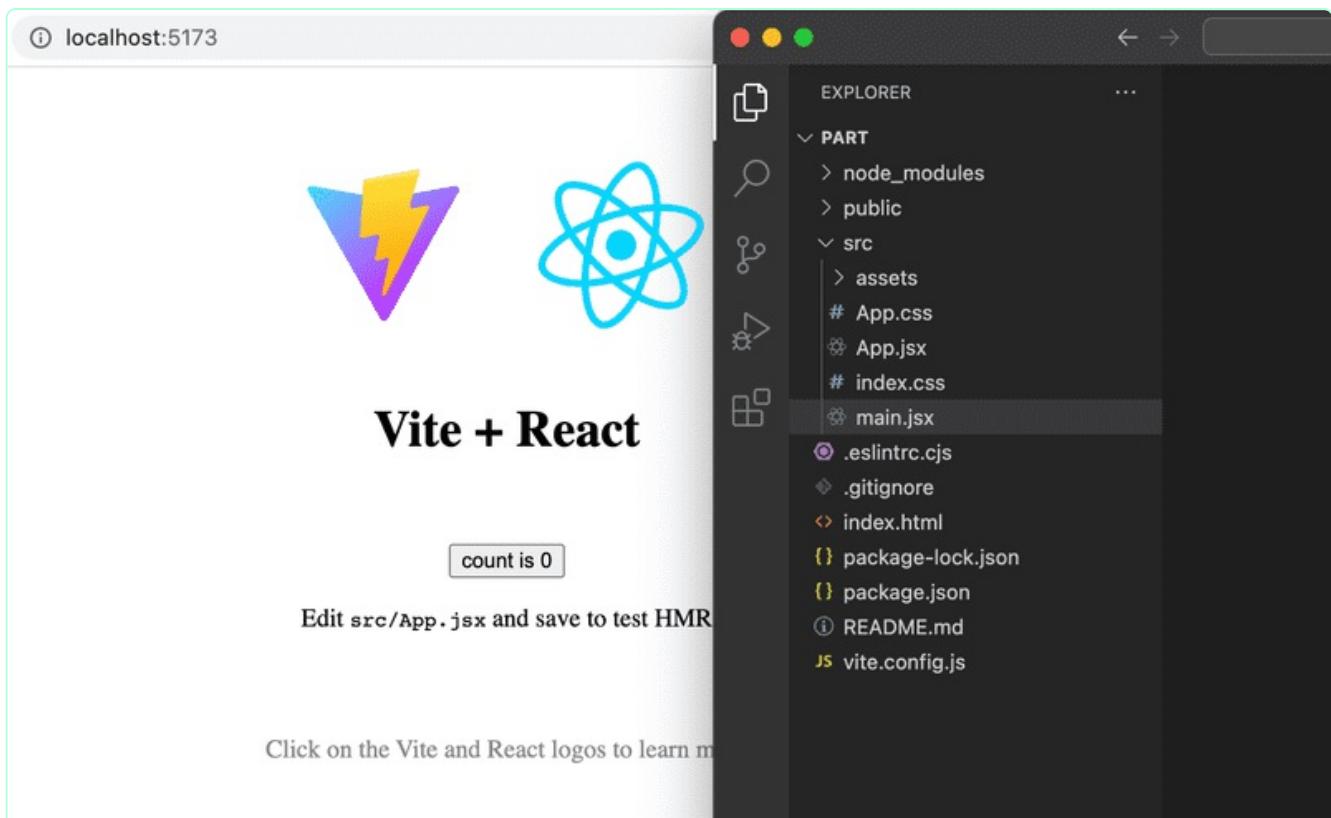
```
npm run dev

VITE v4.4.9 ready in 345 ms

→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h to show help
```

Vite starts the application [by default](#) on port 5173. If it is not free, Vite uses the next free port number.

Open the browser and a text editor so that you can view the code as well as the webpage at the same time on the screen:



The code of the application resides in the `src` folder. Let's simplify the default code such that the contents of the file `main.jsx` looks like this:

```
import ReactDOM from 'react-dom/client'
```

```
import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

and file *App.jsx* looks like this

```
const App = () => {
  return (
    <div>
      <p>Hello world</p>
    </div>
  )
}

export default App
```

The files *App.css* and *index.css*, and the directory *assets* may be deleted as they are not needed in our application right now.

create-react-app

Instead of Vite you can also use the older generation tool [create-react-app](#) in the course to set up the applications. The most visible difference to Vite is the name of the application startup file, which is *index.js*.

The way to start the application is also different in CRA, it is started with a command

```
npm start
```

in contrast to Vite's

```
npm run dev
```

Component

The file *App.jsx* now defines a [React component](#) with the name *App*. The command on the final line of file *main.jsx*

```
ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

renders its contents into the *div*-element, defined in the file *index.html*, having the *id* value 'root'.

By default, the file *index.html* doesn't contain any HTML markup that is visible to us in the browser:

```
<!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>
</head>
<body>
  <div id="root"></div>
  <script type="module" src="/src/main.jsx"></script>
</body>
</html>
```

You can try adding there some HTML to the file. However, when using React, all content that needs to be rendered is usually defined as React components.

Let's take a closer look at the code defining the component:

```
const App = () => (
  <div>
    <p>Hello world</p>
  </div>
)
```

As you probably guessed, the component will be rendered as a *div*-tag, which wraps a *p*-tag containing the text *Hello world*.

Technically the component is defined as a JavaScript function. The following is a function (which does not receive any parameters):

```
() => (
  <div>
    <p>Hello world</p>
  </div>
)
```

The function is then assigned to a constant variable *App*:

```
const App = ...
```

There are a few ways to define functions in JavaScript. Here we will use [arrow functions](#), which are described in a newer version of JavaScript known as [ECMAScript 6](#), also called ES6.

Because the function consists of only a single expression we have used a shorthand, which represents this piece of code:

```
const App = () => {
  return (
    <div>
      <p>Hello world</p>
    </div>
  )
}
```

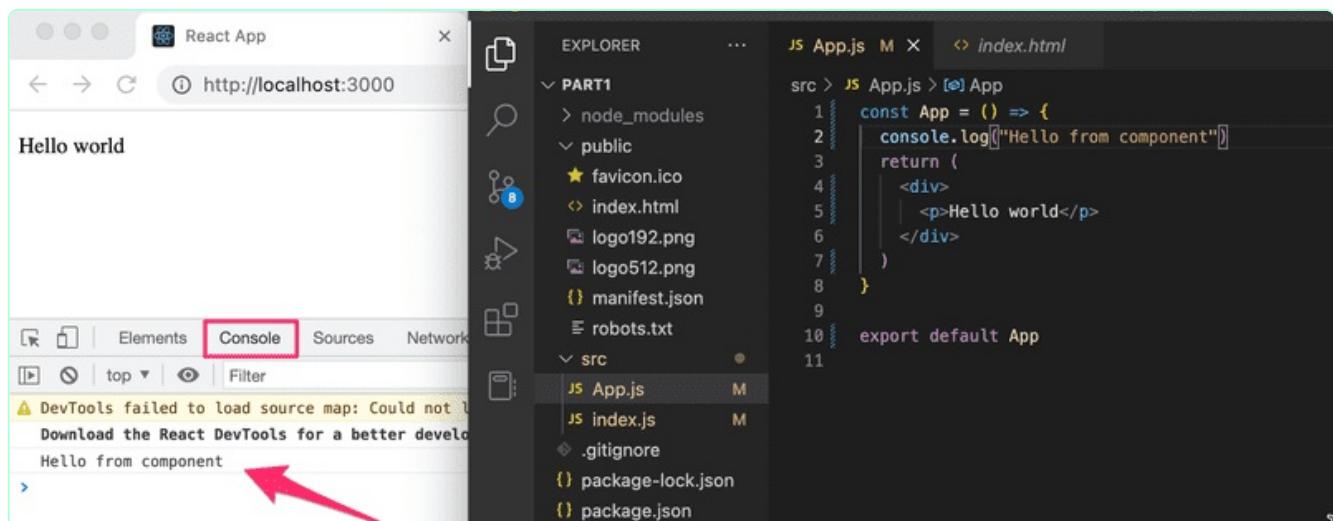
In other words, the function returns the value of the expression.

The function defining the component may contain any kind of JavaScript code. Modify your component to be as follows:

```
const App = () => [
  console.log('Hello from component')
  return (
    <div>
      <p>Hello world</p>
    </div>
  )
}

export default App
```

and observe what happens in the browser console



The first rule of frontend web development:

keep the console open all the time

Let us repeat this together: *I promise to keep the console open all the time* during this course, and for the rest of my life when I'm doing web development.

It is also possible to render dynamic content inside of a component.

Modify the component as follows:

```
const App = () => [
  const now = new Date()
  const a = 10
  const b = 20
  console.log(now, a+b)
```

```

    return [
      <div>
        <p>Hello world, it is {now.toString()}</p>
        <p>
          {a} plus {b} is {a + b}
        </p>
      </div>
    ]
  }

```

Any JavaScript code within the curly braces is evaluated and the result of this evaluation is embedded into the defined place in the HTML produced by the component.

Note that you should not remove the line at the bottom of the component

```
export default App
```

The export is not shown in most of the examples of the course material. Without the export the component and the whole app breaks down.

Did you remember your promise to keep the console open? What was printed out there?

JSX

It seems like React components are returning HTML markup. However, this is not the case. The layout of React components is mostly written using [JSX](#). Although JSX looks like HTML, we are dealing with a way to write JavaScript. Under the hood, JSX returned by React components is compiled into JavaScript.

After compiling, our application looks like this:

```

const App = () => {
  const now = new Date()
  const a = 10
  const b = 20
  return React.createElement(
    'div',
    null,
    React.createElement(
      'p', null, 'Hello world, it is ', now.toString()
    ),
    React.createElement(
      'p', null, a, ' plus ', b, ' is ', a + b
    )
  )
}

```

The compilation is handled by [Babel](#). Projects created with `create-react-app` or `vite` are configured to compile automatically. We will learn more about this topic in [part 7](#) of this course.

It is also possible to write React as "pure JavaScript" without using JSX. Although, nobody with a sound mind would do so.

In practice, JSX is much like HTML with the distinction that with JSX you can easily embed dynamic content by writing appropriate JavaScript within curly braces. The idea of JSX is quite similar to many templating languages, such as Thymeleaf used along with Java Spring, which are used on servers.

JSX is "[XML-like](#)", which means that every tag needs to be closed. For example, a newline is an empty element, which in HTML can be written as follows:

```
<br>
```

but when writing JSX, the tag needs to be closed:

```
<br />
```

Multiple components

Let's modify the file *App.jsx* as follows:

```
const Hello = () => {  return [    <div>      <p>Hello world</p>    </div>  ]}  
const App = () => [  
  return [  
    <div>  
      <h1>Greetings</h1>  
      <Hello />    </div>  
  ]  
]
```

We have defined a new component *Hello* and used it inside the component *App*. Naturally, a component can be used multiple times:

```
const App = () => [  
  return [  
    <div>  
      <h1>Greetings</h1>  
      <Hello />  
      <Hello />      <Hello />    </div>  
  ]  
]
```

NB: *export* at the bottom is left out in these *examples*, now and in the future. It is still needed for the code to work

Writing components with React is easy, and by combining components, even a more complex application can be kept fairly maintainable. Indeed, a core philosophy of React is composing applications from many specialized reusable components.

Another strong convention is the idea of a *root component* called *App* at the top of the component tree of the application. Nevertheless, as we will learn in [part 6](#), there are situations where the

component *App* is not exactly the root, but is wrapped within an appropriate utility component.

props: passing data to components

It is possible to pass data to components using so-called **props**.

Let's modify the component *Hello* as follows:

```
const Hello = (props) => {
  return (
    <div>
      <p>Hello {props.name}</p>
    </div>
  )
}
```

Now the function defining the component has a parameter *props*. As an argument, the parameter receives an object, which has fields corresponding to all the "props" the user of the component defines.

The props are defined as follows:

```
const App = () => {
  return (
    <div>
      <h1>Greetings</h1>
      <Hello name='George' />      <Hello name='Daisy' />
    </div>
  )
}
```

There can be an arbitrary number of props and their values can be "hard-coded" strings or the results of JavaScript expressions. If the value of the prop is achieved using JavaScript it must be wrapped with curly braces.

Let's modify the code so that the component *Hello* uses two props:

```
const Hello = (props) => {
  console.log(props)
  return (
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>
    </div>
  )
}

const App = () => {
  const name = 'Peter'  const age = 10
  return (
    <div>
      <h1>Greetings</h1>
      <Hello name='Maya' age={26 + 10} />      <Hello name={name} age={age} />
    </div>
  )
}
```

The props sent by the component `App` are the values of the variables, the result of the evaluation of the sum expression and a regular string.

Component `Hello` also logs the value of the object `props` to the console.

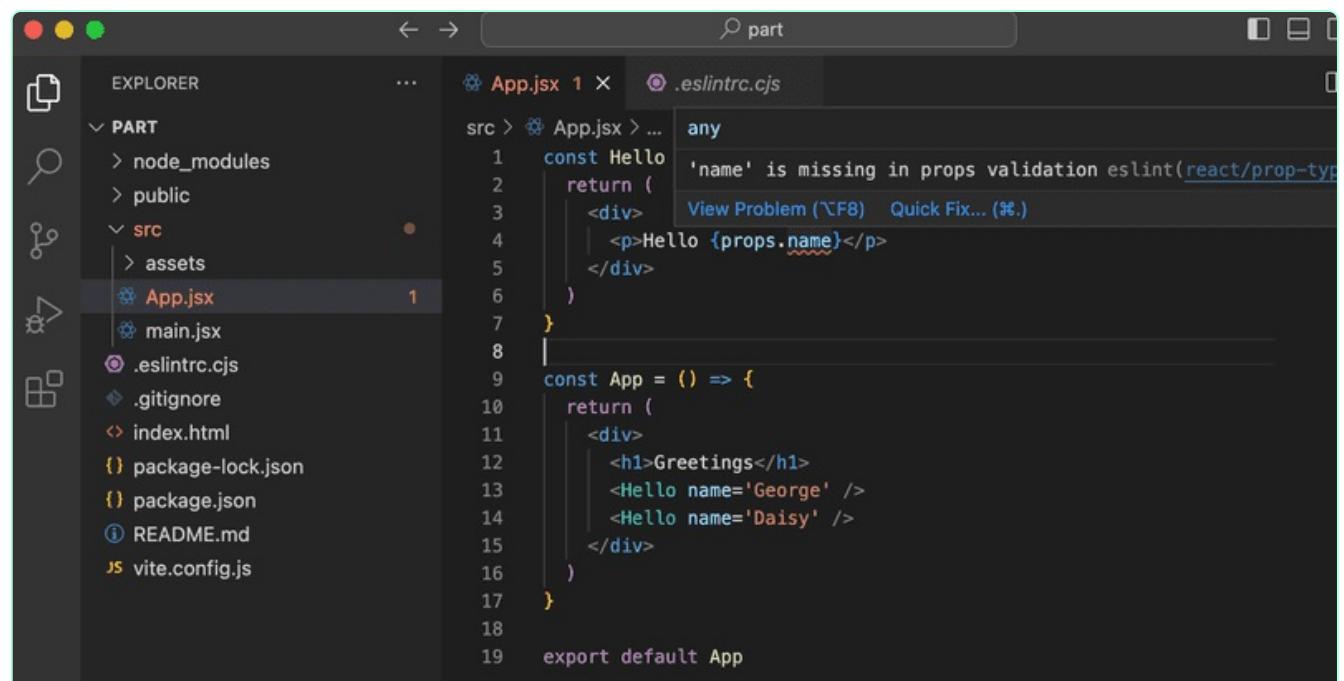
I really hope your console was open. If it was not, remember what you promised:

I promise to keep the console open all the time during this course, and for the rest of my life when I'm doing web development

Software development is hard. It gets even harder if one is not using all the possible available tools such as the web-console and debug printing with `console.log`. Professionals use both *all the time* and there is no single reason why a beginner should not adopt the use of these wonderful helper methods that will make his life so much easier.

Possible error message

Depending on the editor you are using, you may receive the following error message at this point:



A screenshot of a code editor (VS Code) showing the file `App.jsx` in the center pane. The left sidebar shows a project structure with files like `node_modules`, `public`, and `src` containing `assets`, `App.jsx`, `main.jsx`, and `.eslintrc.cjs`. The right sidebar shows the status bar with the text "part". In the center pane, the code for `App.jsx` is displayed:

```
src > App.jsx > ...
1 const Hello
2   return (
3     <div>
4       <p>Hello {props.name}</p>
5     </div>
6   )
7
8   const App = () => {
9     return (
10       <div>
11         <h1>Greetings</h1>
12         <Hello name='George' />
13         <Hello name='Daisy' />
14       </div>
15     )
16   }
17
18   export default App
```

An ESLint warning is shown in the gutter of line 1, indicating that the prop `name` is missing in the `Hello` component. The warning text is: "'name' is missing in props validation eslint(react/prop-types)". There is a "View Problem" link and a "Quick Fix..." button.

It's not an actual error, but a warning caused by the [ESLint](#) tool. You can silence the warning `react/prop-types` by adding to the file `.eslintrc.cjs` the next line

```
module.exports = [
  root: true,
  env: { browser: true, es2020: true },
  extends: [
    'eslint:recommended',
    'plugin:react/recommended',
```

```

    'plugin:react/jsx-runtime',
    'plugin:react-hooks/recommended',
  ],
  ignorePatterns: ['dist', '.eslintrc.cjs'],
  parserOptions: { ecmaVersion: 'latest', sourceType: 'module' },
  settings: { react: { version: '18.2' } },
  plugins: ['react-refresh'],
  rules: [
    'react-refresh/only-export-components': [
      'warn',
      { allowConstantExport: true },
    ],
    'react/prop-types': 0
  ]
}

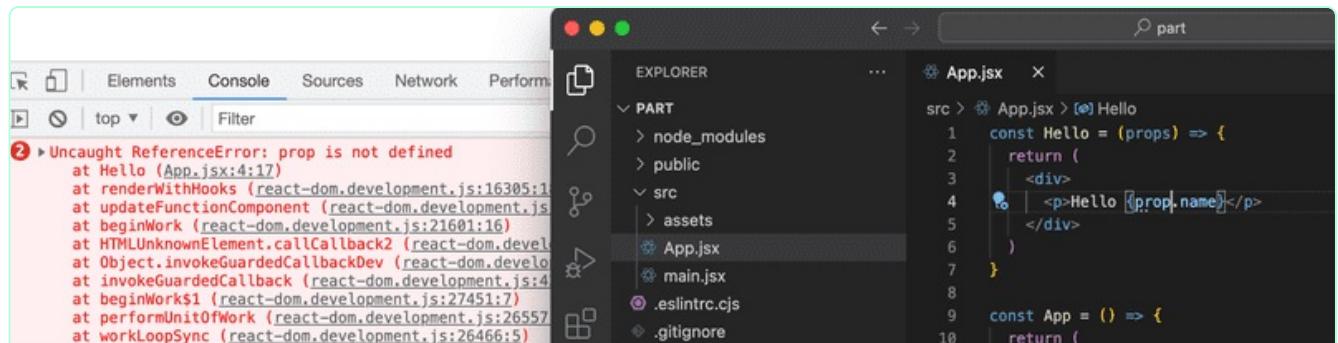
```

We will get to know ESLint in more detail [in part 3](#).

Some notes

React has been configured to generate quite clear error messages. Despite this, you should, at least in the beginning, advance in **very small steps** and make sure that every change works as desired.

The console should always be open. If the browser reports errors, it is not advisable to continue writing more code, hoping for miracles. You should instead try to understand the cause of the error and, for example, go back to the previous working state:



As we already mentioned, when programming with React, it is possible and worthwhile to write `console.log()` commands (which print to the console) within your code.

Also, keep in mind that **First letter of React component names must be capitalized**. If you try defining a component as follows:

```

const footer = () => {
  return (
    <div>
      greeting app created by <a href='https://github.com/mluukkai'>mluukkai</a>
    </div>
  )
}

```

and use it like this

```

const App = () => {
  return [
    <div>
      <h1>Greetings</h1>
      <Hello name='Maya' age={26 + 10} />
      <Footer />    </div>
    ]
}

```

the page is not going to display the content defined within the footer component, and instead React only creates an empty `footer` element, i.e. the built-in HTML element instead of the custom React element of the same name. If you change the first letter of the component name to a capital letter, then React creates a *div*-element defined in the Footer component, which is rendered on the page.

Note that the content of a React component (usually) needs to contain **one root element**. If we, for example, try to define the component *App* without the outermost *div*-element:

```

const App = () => [
  return [
    <h1>Greetings</h1>
    <Hello name='Maya' age={26 + 10} />
    <Footer />
  ]
]

```

the result is an error message.

```

[plugin:vite:react-babel] /Users/mluukkai/opetus/hy-fs/vite/part1/part/src/App.jsx
Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX
fragment <>...</>? (13:6)
  16 |   )
  17 |
  18 |   <h1>Greetings</h1>
  19 |   <Hello name='George' />
  20 |   ^
  21 |   <Hello name='Daisy' />
  22 |
  23 |
  24 |
  25 |
  26 |
  27 |
  28 |
  29 |
  30 |
  31 |
  32 |
  33 |
  34 |
  35 |
  36 |
  37 |
  38 |
  39 |
  40 |
  41 |
  42 |
  43 |
  44 |
  45 |
  46 |
  47 |
  48 |
  49 |
  50 |
  51 |
  52 |
  53 |
  54 |
  55 |
  56 |
  57 |
  58 |
  59 |
  60 |
  61 |
  62 |
  63 |
  64 |
  65 |
  66 |
  67 |
  68 |
  69 |
  70 |
  71 |
  72 |
  73 |
  74 |
  75 |
  76 |
  77 |
  78 |
  79 |
  80 |
  81 |
  82 |
  83 |
  84 |
  85 |
  86 |
  87 |
  88 |
  89 |
  90 |
  91 |
  92 |
  93 |
  94 |
  95 |
  96 |
  97 |
  98 |
  99 |
  100 |
  101 |
  102 |
  103 |
  104 |
  105 |
  106 |
  107 |
  108 |
  109 |
  110 |
  111 |
  112 |
  113 |
  114 |
  115 |
  116 |
  117 |
  118 |
  119 |
  120 |
  121 |
  122 |
  123 |
  124 |
  125 |
  126 |
  127 |
  128 |
  129 |
  130 |
  131 |
  132 |
  133 |
  134 |
  135 |
  136 |
  137 |
  138 |
  139 |
  140 |
  141 |
  142 |
  143 |
  144 |
  145 |
  146 |
  147 |
  148 |
  149 |
  150 |
  151 |
  152 |
  153 |
  154 |
  155 |
  156 |
  157 |
  158 |
  159 |
  160 |
  161 |
  162 |
  163 |
  164 |
  165 |
  166 |
  167 |
  168 |
  169 |
  170 |
  171 |
  172 |
  173 |
  174 |
  175 |
  176 |
  177 |
  178 |
  179 |
  180 |
  181 |
  182 |
  183 |
  184 |
  185 |
  186 |
  187 |
  188 |
  189 |
  190 |
  191 |
  192 |
  193 |
  194 |
  195 |
  196 |
  197 |
  198 |
  199 |
  200 |
  201 |
  202 |
  203 |
  204 |
  205 |
  206 |
  207 |
  208 |
  209 |
  210 |
  211 |
  212 |
  213 |
  214 |
  215 |
  216 |
  217 |
  218 |
  219 |
  220 |
  221 |
  222 |
  223 |
  224 |
  225 |
  226 |
  227 |
  228 |
  229 |
  230 |
  231 |
  232 |
  233 |
  234 |
  235 |
  236 |
  237 |
  238 |
  239 |
  240 |
  241 |
  242 |
  243 |
  244 |
  245 |
  246 |
  247 |
  248 |
  249 |
  250 |
  251 |
  252 |
  253 |
  254 |
  255 |
  256 |
  257 |
  258 |
  259 |
  260 |
  261 |
  262 |
  263 |
  264 |
  265 |
  266 |
  267 |
  268 |
  269 |
  270 |
  271 |
  272 |
  273 |
  274 |
  275 |
  276 |
  277 |
  278 |
  279 |
  280 |
  281 |
  282 |
  283 |
  284 |
  285 |
  286 |
  287 |
  288 |
  289 |
  290 |
  291 |
  292 |
  293 |
  294 |
  295 |
  296 |
  297 |
  298 |
  299 |
  300 |
  301 |
  302 |
  303 |
  304 |
  305 |
  306 |
  307 |
  308 |
  309 |
  310 |
  311 |
  312 |
  313 |
  314 |
  315 |
  316 |
  317 |
  318 |
  319 |
  320 |
  321 |
  322 |
  323 |
  324 |
  325 |
  326 |
  327 |
  328 |
  329 |
  330 |
  331 |
  332 |
  333 |
  334 |
  335 |
  336 |
  337 |
  338 |
  339 |
  340 |
  341 |
  342 |
  343 |
  344 |
  345 |
  346 |
  347 |
  348 |
  349 |
  350 |
  351 |
  352 |
  353 |
  354 |
  355 |
  356 |
  357 |
  358 |
  359 |
  360 |
  361 |
  362 |
  363 |
  364 |
  365 |
  366 |
  367 |
  368 |
  369 |
  370 |
  371 |
  372 |
  373 |
  374 |
  375 |
  376 |
  377 |
  378 |
  379 |
  380 |
  381 |
  382 |
  383 |
  384 |
  385 |
  386 |
  387 |
  388 |
  389 |
  390 |
  391 |
  392 |
  393 |
  394 |
  395 |
  396 |
  397 |
  398 |
  399 |
  400 |
  401 |
  402 |
  403 |
  404 |
  405 |
  406 |
  407 |
  408 |
  409 |
  410 |
  411 |
  412 |
  413 |
  414 |
  415 |
  416 |
  417 |
  418 |
  419 |
  420 |
  421 |
  422 |
  423 |
  424 |
  425 |
  426 |
  427 |
  428 |
  429 |
  430 |
  431 |
  432 |
  433 |
  434 |
  435 |
  436 |
  437 |
  438 |
  439 |
  440 |
  441 |
  442 |
  443 |
  444 |
  445 |
  446 |
  447 |
  448 |
  449 |
  450 |
  451 |
  452 |
  453 |
  454 |
  455 |
  456 |
  457 |
  458 |
  459 |
  460 |
  461 |
  462 |
  463 |
  464 |
  465 |
  466 |
  467 |
  468 |
  469 |
  470 |
  471 |
  472 |
  473 |
  474 |
  475 |
  476 |
  477 |
  478 |
  479 |
  480 |
  481 |
  482 |
  483 |
  484 |
  485 |
  486 |
  487 |
  488 |
  489 |
  490 |
  491 |
  492 |
  493 |
  494 |
  495 |
  496 |
  497 |
  498 |
  499 |
  500 |
  501 |
  502 |
  503 |
  504 |
  505 |
  506 |
  507 |
  508 |
  509 |
  510 |
  511 |
  512 |
  513 |
  514 |
  515 |
  516 |
  517 |
  518 |
  519 |
  520 |
  521 |
  522 |
  523 |
  524 |
  525 |
  526 |
  527 |
  528 |
  529 |
  530 |
  531 |
  532 |
  533 |
  534 |
  535 |
  536 |
  537 |
  538 |
  539 |
  540 |
  541 |
  542 |
  543 |
  544 |
  545 |
  546 |
  547 |
  548 |
  549 |
  550 |
  551 |
  552 |
  553 |
  554 |
  555 |
  556 |
  557 |
  558 |
  559 |
  560 |
  561 |
  562 |
  563 |
  564 |
  565 |
  566 |
  567 |
  568 |
  569 |
  570 |
  571 |
  572 |
  573 |
  574 |
  575 |
  576 |
  577 |
  578 |
  579 |
  580 |
  581 |
  582 |
  583 |
  584 |
  585 |
  586 |
  587 |
  588 |
  589 |
  590 |
  591 |
  592 |
  593 |
  594 |
  595 |
  596 |
  597 |
  598 |
  599 |
  600 |
  601 |
  602 |
  603 |
  604 |
  605 |
  606 |
  607 |
  608 |
  609 |
  610 |
  611 |
  612 |
  613 |
  614 |
  615 |
  616 |
  617 |
  618 |
  619 |
  620 |
  621 |
  622 |
  623 |
  624 |
  625 |
  626 |
  627 |
  628 |
  629 |
  630 |
  631 |
  632 |
  633 |
  634 |
  635 |
  636 |
  637 |
  638 |
  639 |
  640 |
  641 |
  642 |
  643 |
  644 |
  645 |
  646 |
  647 |
  648 |
  649 |
  650 |
  651 |
  652 |
  653 |
  654 |
  655 |
  656 |
  657 |
  658 |
  659 |
  660 |
  661 |
  662 |
  663 |
  664 |
  665 |
  666 |
  667 |
  668 |
  669 |
  670 |
  671 |
  672 |
  673 |
  674 |
  675 |
  676 |
  677 |
  678 |
  679 |
  680 |
  681 |
  682 |
  683 |
  684 |
  685 |
  686 |
  687 |
  688 |
  689 |
  690 |
  691 |
  692 |
  693 |
  694 |
  695 |
  696 |
  697 |
  698 |
  699 |
  700 |
  701 |
  702 |
  703 |
  704 |
  705 |
  706 |
  707 |
  708 |
  709 |
  710 |
  711 |
  712 |
  713 |
  714 |
  715 |
  716 |
  717 |
  718 |
  719 |
  720 |
  721 |
  722 |
  723 |
  724 |
  725 |
  726 |
  727 |
  728 |
  729 |
  730 |
  731 |
  732 |
  733 |
  734 |
  735 |
  736 |
  737 |
  738 |
  739 |
  740 |
  741 |
  742 |
  743 |
  744 |
  745 |
  746 |
  747 |
  748 |
  749 |
  750 |
  751 |
  752 |
  753 |
  754 |
  755 |
  756 |
  757 |
  758 |
  759 |
  760 |
  761 |
  762 |
  763 |
  764 |
  765 |
  766 |
  767 |
  768 |
  769 |
  770 |
  771 |
  772 |
  773 |
  774 |
  775 |
  776 |
  777 |
  778 |
  779 |
  780 |
  781 |
  782 |
  783 |
  784 |
  785 |
  786 |
  787 |
  788 |
  789 |
  790 |
  791 |
  792 |
  793 |
  794 |
  795 |
  796 |
  797 |
  798 |
  799 |
  800 |
  801 |
  802 |
  803 |
  804 |
  805 |
  806 |
  807 |
  808 |
  809 |
  810 |
  811 |
  812 |
  813 |
  814 |
  815 |
  816 |
  817 |
  818 |
  819 |
  820 |
  821 |
  822 |
  823 |
  824 |
  825 |
  826 |
  827 |
  828 |
  829 |
  830 |
  831 |
  832 |
  833 |
  834 |
  835 |
  836 |
  837 |
  838 |
  839 |
  840 |
  841 |
  842 |
  843 |
  844 |
  845 |
  846 |
  847 |
  848 |
  849 |
  850 |
  851 |
  852 |
  853 |
  854 |
  855 |
  856 |
  857 |
  858 |
  859 |
  860 |
  861 |
  862 |
  863 |
  864 |
  865 |
  866 |
  867 |
  868 |
  869 |
  870 |
  871 |
  872 |
  873 |
  874 |
  875 |
  876 |
  877 |
  878 |
  879 |
  880 |
  881 |
  882 |
  883 |
  884 |
  885 |
  886 |
  887 |
  888 |
  889 |
  890 |
  891 |
  892 |
  893 |
  894 |
  895 |
  896 |
  897 |
  898 |
  899 |
  900 |
  901 |
  902 |
  903 |
  904 |
  905 |
  906 |
  907 |
  908 |
  909 |
  910 |
  911 |
  912 |
  913 |
  914 |
  915 |
  916 |
  917 |
  918 |
  919 |
  920 |
  921 |
  922 |
  923 |
  924 |
  925 |
  926 |
  927 |
  928 |
  929 |
  930 |
  931 |
  932 |
  933 |
  934 |
  935 |
  936 |
  937 |
  938 |
  939 |
  940 |
  941 |
  942 |
  943 |
  944 |
  945 |
  946 |
  947 |
  948 |
  949 |
  950 |
  951 |
  952 |
  953 |
  954 |
  955 |
  956 |
  957 |
  958 |
  959 |
  960 |
  961 |
  962 |
  963 |
  964 |
  965 |
  966 |
  967 |
  968 |
  969 |
  970 |
  971 |
  972 |
  973 |
  974 |
  975 |
  976 |
  977 |
  978 |
  979 |
  980 |
  981 |
  982 |
  983 |
  984 |
  985 |
  986 |
  987 |
  988 |
  989 |
  990 |
  991 |
  992 |
  993 |
  994 |
  995 |
  996 |
  997 |
  998 |
  999 |
  1000 |
  1001 |
  1002 |
  1003 |
  1004 |
  1005 |
  1006 |
  1007 |
  1008 |
  1009 |
  1010 |
  1011 |
  1012 |
  1013 |
  1014 |
  1015 |
  1016 |
  1017 |
  1018 |
  1019 |
  1020 |
  1021 |
  1022 |
  1023 |
  1024 |
  1025 |
  1026 |
  1027 |
  1028 |
  1029 |
  1030 |
  1031 |
  1032 |
  1033 |
  1034 |
  1035 |
  1036 |
  1037 |
  1038 |
  1039 |
  1040 |
  1041 |
  1042 |
  1043 |
  1044 |
  1045 |
  1046 |
  1047 |
  1048 |
  1049 |
  1050 |
  1051 |
  1052 |
  1053 |
  1054 |
  1055 |
  1056 |
  1057 |
  1058 |
  1059 |
  1060 |
  1061 |
  1062 |
  1063 |
  1064 |
  1065 |
  1066 |
  1067 |
  1068 |
  1069 |
  1070 |
  1071 |
  1072 |
  1073 |
  1074 |
  1075 |
  1076 |
  1077 |
  1078 |
  1079 |
  1080 |
  1081 |
  1082 |
  1083 |
  1084 |
  1085 |
  1086 |
  1087 |
  1088 |
  1089 |
  1090 |
  1091 |
  1092 |
  1093 |
  1094 |
  1095 |
  1096 |
  1097 |
  1098 |
  1099 |
  1100 |
  1101 |
  1102 |
  1103 |
  1104 |
  1105 |
  1106 |
  1107 |
  1108 |
  1109 |
  1110 |
  1111 |
  1112 |
  1113 |
  1114 |
  1115 |
  1116 |
  1117 |
  1118 |
  1119 |
  1120 |
  1121 |
  1122 |
  1123 |
  1124 |
  1125 |
  1126 |
  1127 |
  1128 |
  1129 |
  1130 |
  1131 |
  1132 |
  1133 |
  1134 |
  1135 |
  1136 |
  1137 |
  1138 |
  1139 |
  1140 |
  1141 |
  1142 |
  1143 |
  1144 |
  1145 |
  1146 |
  1147 |
  1148 |
  1149 |
  1150 |
  1151 |
  1152 |
  1153 |
  1154 |
  1155 |
  1156 |
  1157 |
  1158 |
  1159 |
  1160 |
  1161 |
  1162 |
  1163 |
  1164 |
  1165 |
  1166 |
  1167 |
  1168 |
  1169 |
  1170 |
  1171 |
  1172 |
  1173 |
  1174 |
  1175 |
  1176 |
  1177 |
  1178 |
  1179 |
  1180 |
  1181 |
  1182 |
  1183 |
  1184 |
  1185 |
  1186 |
  1187 |
  1188 |
  1189 |
  1190 |
  1191 |
  1192 |
  1193 |
  1194 |
  1195 |
  1196 |
  1197 |
  1198 |
  1199 |
  1200 |
  1201 |
  1202 |
  1203 |
  1204 |
  1205 |
  1206 |
  1207 |
  1208 |
  1209 |
  1210 |
  1211 |
  1212 |
  1213 |
  1214 |
  1215 |
  1216 |
  1217 |
  1218 |
  1219 |
  1220 |
  1221 |
  1222 |
  1223 |
  1224 |
  1225 |
  1226 |
  1227 |
  1228 |
  1229 |
  1230 |
  1231 |
  1232 |
  1233 |
  1234 |
  1235 |
  1236 |
  1237 |
  1238 |
  1239 |
  1240 |
  1241 |
  1242 |
  1243 |
  1244 |
  1245 |
  1246 |
  1247 |
  1248 |
  1249 |
  1250 |
  1251 |
  1252 |
  1253 |
  1254 |
  1255 |
  1256 |
  1257 |
  1258 |
  1259 |
  1260 |
  1261 |
  1262 |
  1263 |
  1264 |
  1265 |
  1266 |
  1267 |
  1268 |
  1269 |
  1270 |
  1271 |
  1272 |
  1273 |
  1274 |
  1275 |
  1276 |
  1277 |
  1278 |
  1279 |
  1280 |
  1281 |
  1282 |
  1283 |
  1284 |
  1285 |
  1286 |
  1287 |
  1288 |
  1289 |
  1290 |
  1291 |
  1292 |
  1293 |
  1294 |
  1295 |
  1296 |
  1297 |
  1298 |
  1299 |
  1300 |
  1301 |
  1302 |
  1303 |
  1304 |
  1305 |
  1306 |
  1307 |
  1308 |
  1309 |
  1310 |
  1311 |
  1312 |
  1313 |
  1314 |
  1315 |
  1316 |
  1317 |
  1318 |
  1319 |
  1320 |
  1321 |
  1322 |
  1323 |
  1324 |
  1325 |
  1326 |
  1327 |
  1328 |
  1329 |
  1330 |
  1331 |
  1332 |
  1333 |
  1334 |
  1335 |
  1336 |
  1337 |
  1338 |
  1339 |
  1340 |
  1341 |
  1342 |
  1343 |
  1344 |
  1345 |
  1346 |
  1347 |
  1348 |
  1349 |
  1350 |
  1351 |
  1352 |
  1353 |
  1354 |
  1355 |
  1356 |
  1357 |
  1358 |
  1359 |
  1360 |
  1361 |
  1362 |
  1363 |
  1364 |
  1365 |
  1366 |
  1367 |
  1368 |
  1369 |
  1370 |
  1371 |
  1372 |
  1373 |
  1374 |
  1375 |
  1376 |
  1377 |
  1378 |
  1379 |
  1380 |
  1381 |
  1382 |
  1383 |
  1384 |
  1385 |
  1386 |
  1387 |
  1388 |
  1389 |
  1390 |
  1391 |
  1392 |
  1393 |
  1394 |
  1395 |
  1396 |
  1397 |
  1398 |
  1399 |
  1400 |
  1401 |
  1402 |
  1403 |
  1404 |
  1405 |
  1406 |
  1407 |
  1408 |
  1409 |
  1410 |
  1411 |
  1412 |
  1413 |
  1414 |
  1415 |
  1416 |
  1417 |
  1418 |
  1419 |
  1420 |
  1421 |
  1422 |
  1423 |
  1424 |
  1425 |
  1426 |
  1427 |
  1428 |
  1429 |
  1430 |
  1431 |
  1432 |
  1433 |
  1434 |
  1435 |
  1436 |
  1437 |
  1438 |
  1439 |
  1440 |
  1441 |
  1442 |
  1443 |
  1444 |
  1445 |
  1446 |
  1447 |
  1448 |
  1449 |
  1450 |
  1451 |
  1452 |
  1453 |
  1454 |
  1455 |
  1456 |
  1457 |
  1458 |
  1459 |
  1460 |
  1461 |
  1462 |
  1463 |
  1464 |
  1465 |
  1466 |
  1467 |
  1468 |
  1469 |
  1470 |
  1471 |
  1472 |
  1473 |
  1474 |
  1475 |
  1476 |
  1477 |
  1478 |
  1479 |
  1480 |
  1481 |
  1482 |
  1483 |
  1484 |
  1485 |
  1486 |
  1487 |
  1488 |
  1489 |
  1490 |
  1491 |
  1492 |
  1493 |
  1494 |
  1495 |
  1496 |
  1497 |
  1498 |
  1499 |
  1500 |
  1501 |
  1502 |
  1503 |
  1504 |
  1505 |
  1506 |
  1507 |
  1508 |
  1509 |
  1510 |
  1511 |
  1512 |
  1513 |
  1514 |
  1515 |
  1516 |
  1517 |
  1518 |
  1519 |
  1520 |
  1521 |
  1522 |
  1523 |
  1524 |
  1525 |
  1526 |
  1527 |
  1528 |
  1529 |
  1530 |
  1531 |
  1532 |
  1533 |
  1534 |
  1535 |
  1536 |
  1537 |
  1538 |
  1539 |
  1540 |
  1541 |
  1542 |
  1543 |
  1544 |
  1545 |
  1546 |
  1547 |
  1548 |
  1549 |
  1550 |
  1551 |
  1552 |
  1553 |
  1554 |
  1555 |
  1556 |
  1557 |
  1558 |
  1559 |
  1560 |
  1561 |
  1562 |
  1563 |
  1564 |
  1565 |
  1566 |
  1567 |
  1568 |
  1569 |
  1570 |
  1571 |
  1572 |
  1573 |
  1574 |
  1575 |
  1576 |
  1577 |
  1578 |
  1579 |
  1580 |
  1581 |
  1582 |
  1583 |
  1584 |
  1585 |
  1586 |
  1587 |
  1588 |
  1589 |
  1590 |
  1591 |
  1592 |
  1593 |
  1594 |
  1595 |
  1596 |
  1597 |
  1598 |
  1599 |
  1600 |
  1601 |
  1602 |
  1603 |
  1604 |
  1605 |
  1606 |
  1607 |
  1608 |
  1609 |
  1610 |
  1611 |
  1612 |
  1613 |
  1614 |
  1615 |
  1616 |
  1617 |
  1618 |
  1619 |
  1620 |
  1621 |
  1622 |
  1623 |
  1624 |
  1625 |
  1626 |
  1627 |
  1628 |
  1629 |
  1630 |
  1631 |
  1632 |
  1633 |
  1634 |
  1635 |
  1636 |
  1637 |
  1638 |
  1639 |
  1640 |
  1641 |
  1642 |
  1643 |
  1644 |
  1645 |
  1646 |
  1647 |
  1648 |
  1649 |
  1650 |
  1651 |
  1652 |
  1653 |
  1654 |
  1655 |
  1656 |
  1657 |
  1658 |
  1659 |
  1660 |
  1661 |
  1662 |
  1663 |
  1664 |
  1665 |
  1666 |
  1667 |
  1668 |
  1669 |
  1670 |
  1671 |
  1672 |
  1673 |
  1674 |
  1675 |
  1676 |
  1677 |
  1678 |
  1679 |
  1680 |
  1681
```

However, when defining the root component of the application this is not a particularly wise thing to do, and it makes the code look a bit ugly.

Because the root element is stipulated, we have "extra" div elements in the DOM tree. This can be avoided by using [fragments](#), i.e. by wrapping the elements to be returned by the component with an empty element:

```
const App = () => {
  const name = 'Peter'
  const age = 10

  return [
    <>
    <h1>Greetings</h1>
    <Hello name='Maya' age={26 + 10} />
    <Hello name={name} age={age} />
    <Footer />
  </>
]
```

It now compiles successfully, and the DOM generated by React no longer contains the extra div element.

Do not render objects

Consider an application that prints the names and ages of our friends on the screen:

```
const App = () => {
  const friends = [
    { name: 'Peter', age: 4 },
    { name: 'Maya', age: 10 },
  ]

  return [
    <div>
      <p>{friends[0]}</p>
      <p>{friends[1]}</p>
    </div>
  ]
}

export default App
```

However, nothing appears on the screen. I've been trying to find a problem in the code for 15 minutes, but I can't figure out where the problem could be.

I finally remember the promise we made

I promise to keep the console open all the time during this course, and for the rest of my life

when I'm doing web development

The console screams in red:

```
✖ Uncaught Error: Objects are not valid as a React child (found: object with keys {name, age}). If you meant to render a collection of children, use an array instead.
    at throwOnInvalidObjectType (react-dom.development.js:14887:1)
    at reconcileChildFibers (react-dom.development.js:15828:1)
    at reconcileChildren (react-dom.development.js:19167:1)
    at updateHostComponent (react-dom.development.js:19924:1)
    at beginWork (react-dom.development.js:21618:1)
    at HTMLUnknownElement.callCallback (react-dom.development.js:4164:1)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:1)
    at invokeGuardedCallback (react-dom.development.js:4277:1)
    at beginWork$1 (react-dom.development.js:27451:1)
    at performUnitOfWork (react-dom.development.js:26557:1)
```

The core of the problem is *Objects are not valid as a React child*, i.e. the application tries to render *objects* and it fails again.

The code tries to render the information of one friend as follows

```
<p>{friends[0]}</p>
```

and this causes a problem because the item to be rendered in the braces is an object.

```
{ name: 'Peter', age: 4 }
```

In React, the individual things rendered in braces must be primitive values, such as numbers or strings.

The fix is as follows

```
const App = () => {
  const friends = [
    { name: 'Peter', age: 4 },
    { name: 'Maya', age: 10 },
  ]

  return (
    <div>
      <p>{friends[0].name} {friends[0].age}</p>
      <p>{friends[1].name} {friends[1].age}</p>
    </div>
  )
}

export default App
```

So now the friend's name is rendered separately inside the curly braces

```
{friends[0].name}
```

and age

```
  {{friends[0].age}}
```

After correcting the error, you should clear the console error messages by pressing and then reload the page content and make sure that no error messages are displayed.

A small additional note to the previous one. React also allows arrays to be rendered *if* the array contains values that are eligible for rendering (such as numbers or strings). So the following program would work, although the result might not be what we want:

```
const App = () => {
  const friends = [ 'Peter', 'Maya' ]

  return [
    <div>
      <p>{{friends}}</p>
    </div>
  ]
}
```

In this part, it is not even worth trying to use the direct rendering of the tables, we will come back to it in the next part.

Exercises 1.1.-1.2.

The exercises are submitted via GitHub, and by marking the exercises as done in the "my submissions" tab of the [submission application](#).

The exercises are submitted **one part at a time**. When you have submitted the exercises for a part of the course you can no longer submit undone exercises for the same part.

Note that in this part, there are [more exercises](#) besides those found below. *Do not submit your work* until you have completed all of the exercises you want to submit for the part.

You may submit all the exercises of this course into the same repository, or use multiple repositories. If you submit exercises of different parts into the same repository, please use a sensible naming scheme for the directories.

One very functional file structure for the submission repository is as follows:

```
part0
part1
  courseinfo
  unicafe
  anecdotes
part2
  phonebook
  countries
```

See this [example submission repository](#)!

For each part of the course, there is a directory, which further branches into directories containing a

series of exercises, like "unicafe" for part 1.

Most of the exercises of the course build a larger application, eg. courseinfo, unicafe and anecdotes in this part, bit by bit. It is enough to submit the completed application. You can make a commit after each exercise, but that is not compulsory. For example the course info app is built in exercises 1.1.-1.5. It is just the end result after 1.5 that you need to submit!

For each web application for a series of exercises, it is recommended to submit all files relating to that application, except for the directory `node_modules`.

1.1: Course Information, step 1

The application that we will start working on in this exercise will be further developed in a few of the following exercises. In this and other upcoming exercise sets in this course, it is enough to only submit the final state of the application. If desired, you may also create a commit for each exercise of the series, but this is entirely optional.

Use Vite to initialize a new application. Modify `main.jsx` to match the following

```
import ReactDOM from 'react-dom/client'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

and `App.jsx` to match the following

```
const App = () => {
  const course = 'Half Stack application development'
  const part1 = 'Fundamentals of React'
  const exercises1 = 10
  const part2 = 'Using props to pass data'
  const exercises2 = 7
  const part3 = 'State of a component'
  const exercises3 = 14

  return (
    <div>
      <h1>{course}</h1>
      <p>
        {part1} {exercises1}
      </p>
      <p>
        {part2} {exercises2}
      </p>
      <p>
        {part3} {exercises3}
      </p>
      <p>Number of exercises {exercises1 + exercises2 + exercises3}</p>
    </div>
  )
}
```

```
export default App
```

and remove the extra files `App.css` and `index.css`, also remove the directory assets.

Unfortunately, the entire application is in the same component. Refactor the code so that it consists of three new components: *Header*, *Content*, and *Total*. All data still resides in the *App* component, which passes the necessary data to each component using *props*. *Header* takes care of rendering the name of the course, *Content* renders the parts and their number of exercises and *Total* renders the total number of exercises.

Define the new components in the file `App.jsx`.

The *App* component's body will approximately be as follows:

```
const App = () => [
  // const-definitions

  return (
    <div>
      <Header course={course} />
      <Content ... />
      <Total ... />
    </div>
  )
]
```

WARNING Don't try to program all the components concurrently, because that will almost certainly break down the whole app. Proceed in small steps, first make e.g. the component *Header* and only when it works for sure, you could proceed to the next component.

Careful, small-step progress may seem slow, but it is actually *by far the fastest* way to progress. Famous software developer Robert "Uncle Bob" Martin has stated

"The only way to go fast, is to go well"

that is, according to Martin, careful progress with small steps is even the only way to be fast.

1.2: Course Information, step 2

Refactor the *Content* component so that it does not render any names of parts or their number of exercises by itself. Instead, it only renders three *Part* components of which each renders the name and number of exercises of one part.

```
const Content = ... {
  return [
    <div>
      <Part .../>
    </div>
  ]
}
```

```
<Part .../>
<Part .../>
</div>
}
}
```

Our application passes on information in quite a primitive way at the moment, since it is based on individual variables. We shall fix that in [part 2](#), but before that, let's go to part1b to learn about JavaScript.

1b: JavaScript

During the course, we have a goal and a need to learn a sufficient amount of JavaScript in addition to web development.

JavaScript has advanced rapidly in the last few years and in this course, we use features from the newer versions. The official name of the JavaScript standard is [ECMAScript](#). At this moment, the latest version is the one released in June of 2023 with the name [ECMAScript®2023](#), otherwise known as ES14.

Browsers do not yet support all of JavaScript's newest features. Due to this fact, a lot of code run in browsers has been *transpiled* from a newer version of JavaScript to an older, more compatible version.

Today, the most popular way to do transpiling is by using [Babel](#). Transpilation is automatically configured in React applications created with vite. We will take a closer look at the configuration of the transpilation in [part 7](#) of this course.

[Node.js](#) is a JavaScript runtime environment based on Google's [Chrome V8](#) JavaScript engine and works practically anywhere - from servers to mobile phones. Let's practice writing some JavaScript using Node. The latest versions of Node already understand the latest versions of JavaScript, so the code does not need to be transpiled.

The code is written into files ending with `.js` that are run by issuing the command `node name_of_file.js`

It is also possible to write JavaScript code into the Node.js console, which is opened by typing `node` in the command line, as well as into the browser's developer tool console. [The newest revisions of Chrome handle the newer features of JavaScript pretty well](#) without transpiling the code.

Alternatively, you can use a tool like [JS Bin](#).

JavaScript is sort of reminiscent, both in name and syntax, to Java. But when it comes to the core mechanism of the language they could not be more different. Coming from a Java background, the behavior of JavaScript can seem a bit alien, especially if one does not make the effort to look up its features.

In certain circles, it has also been popular to attempt "simulating" Java features and design patterns in JavaScript. We do not recommend doing this as the languages and respective ecosystems are ultimately very different.

Variables

In JavaScript there are a few ways to go about defining variables:

```

const x = 1
let y = 5

console.log(x, y)    // 1, 5 are printed
y += 10
console.log(x, y)    // 1, 15 are printed
y = 'sometext'
console.log(x, y)    // 1, sometext are printed
x = 4                // causes an error

```

`const` does not define a variable but a *constant* for which the value can no longer be changed. On the other hand, `let` defines a normal variable.

In the example above, we also see that the variable's data type can change during execution. At the start, `y` stores an integer; at the end, it stores a string.

It is also possible to define variables in JavaScript using the keyword `var`. `var` was, for a long time, the only way to define variables. `const` and `let` were only recently added in version ES6. In specific situations, `var` works in a different way compared to variable definitions in most languages - see [JavaScript Variables - Should You Use let, var or const? on Medium](#) or [Keyword: var vs. let on JS Tips](#) for more information. During this course the use of `var` is ill-advised and you should stick with using `const` and `let`! You can find more on this topic on YouTube - e.g. [var, let and const - ES6 JavaScript Features](#)

Arrays

An `array` and a couple of examples of its use:

```

const t = [1, -1, 3]

t.push(5)

console.log(t.length) // 4 is printed
console.log(t[1])    // -1 is printed

t.forEach(value => {
  console.log(value) // numbers 1, -1, 3, 5 are printed, each on its own line
})

```

Notable in this example is the fact that the contents of the array can be modified even though it is defined as a `const`. Because the array is an object, the variable always points to the same object. However, the content of the array changes as new items are added to it.

One way of iterating through the items of the array is using `forEach` as seen in the example. `forEach` receives a *function* defined using the arrow syntax as a parameter.

```

value => [
  console.log(value)
]

```

`forEach` calls the function *for each of the items in the array*, always passing the individual item as an

argument. The function as the argument of `forEach` may also receive [other arguments](#).

In the previous example, a new item was added to the array using the method `push`. When using React, techniques from functional programming are often used. One characteristic of the functional programming paradigm is the use of [immutable](#) data structures. In React code, it is preferable to use the method `concat`, which creates a new array with the added item. This ensures the original array remains unchanged.

```
const t = [1, -1, 3]

const t2 = t.concat(5) // creates new array

console.log(t) // [1, -1, 3] is printed
console.log(t2) // [1, -1, 3, 5] is printed
```

The method call `t.concat(5)` does not add a new item to the old array but returns a new array which, besides containing the items of the old array, also contains the new item.

There are plenty of useful methods defined for arrays. Let's look at a short example of using the `map` method.

```
const t = [1, 2, 3]

const m1 = t.map(value => value * 2)
console.log(m1) // [2, 4, 6] is printed
```

Based on the old array, `map` creates a *new array*, for which the function given as a parameter is used to create the items. In the case of this example, the original value is multiplied by two.

`Map` can also transform the array into something completely different:

```
const m2 = t.map(value => '<li>' + value + '</li>')
console.log(m2)
// [<li>1</li>, <li>2</li>, <li>3</li>] is printed
```

Here an array filled with integer values is transformed into an array containing strings of HTML using the `map` method. In [part 2](#) of this course, we will see that `map` is used quite frequently in React.

Individual items of an array are easy to assign to variables with the help of the [destructuring assignment](#).

```
const t = [1, 2, 3, 4, 5]

const [first, second, ...rest] = t

console.log(first, second) // 1, 2 is printed
console.log(rest) // [3, 4, 5] is printed
```

Thanks to the assignment, the variables `first` and `second` will receive the first two integers of the array as their values. The remaining integers are "collected" into an array of their own which is then

assigned to the variable *rest*.

Objects

There are a few different ways of defining objects in JavaScript. One very common method is using [object literals](#), which happens by listing its properties within braces:

```
const object1 = [
  name: 'Arto Hellas',
  age: 35,
  education: 'PhD',
]

const object2 = [
  name: 'Full Stack web application development',
  level: 'intermediate studies',
  size: 5,
]

const object3 = [
  name: {
    first: 'Dan',
    last: 'Abramov',
  },
  grades: [2, 3, 5, 3],
  department: 'Stanford University',
]
```

The values of the properties can be of any type, like integers, strings, arrays, objects...

The properties of an object are referenced by using the "dot" notation, or by using brackets:

```
console.log(object1.name)           // Arto Hellas is printed
const fieldName = 'age'
console.log(object1[fieldName])     // 35 is printed
```

You can also add properties to an object on the fly by either using dot notation or brackets:

```
object1.address = 'Helsinki'
object1['secret number'] = 12341
```

The latter of the additions has to be done by using brackets because when using dot notation, *secret number* is not a valid property name because of the space character.

Naturally, objects in JavaScript can also have methods. However, during this course, we do not need to define any objects with methods of their own. This is why they are only discussed briefly during the course.

Objects can also be defined using so-called constructor functions, which results in a mechanism reminiscent of many other programming languages, e.g. Java's classes. Despite this similarity, JavaScript does not have classes in the same sense as object-oriented programming languages. There

has been, however, the addition of the *class syntax* starting from version ES6, which in some cases helps structure object-oriented classes.

Functions

We have already become familiar with defining arrow functions. The complete process, without cutting corners, of defining an arrow function is as follows:

```
const sum = (p1, p2) => [
  console.log(p1)
  console.log(p2)
  return p1 + p2
]
```

and the function is called as can be expected:

```
const result = sum(1, 5)
console.log(result)
```

If there is just a single parameter, we can exclude the parentheses from the definition:

```
const square = p => [
  console.log(p)
  return p * p
]
```

If the function only contains a single expression then the braces are not needed. In this case, the function only returns the result of its only expression. Now, if we remove console printing, we can further shorten the function definition:

```
const square = p => p * p
```

This form is particularly handy when manipulating arrays - e.g. when using the map method:

```
const t = [1, 2, 3]
const tSquared = t.map(p => p * p)
// tSquared is now [1, 4, 9]
```

The arrow function feature was added to JavaScript only a couple of years ago, with version [ES6](#). Before this, the only way to define functions was by using the keyword *function*.

There are two ways to reference the function; one is giving a name in a [function declaration](#).

```
function product(a, b) {
  return a * b
}

const result = product(2, 6)
```

```
// result is now 12
```

The other way to define the function is by using a [function expression](#). In this case, there is no need to give the function a name and the definition may reside among the rest of the code:

```
const average = function(a, b) {
  return (a + b) / 2
}

const result = average(2, 5)
// result is now 3.5
```

During this course, we will define all functions using the arrow syntax.

Exercises 1.3.-1.5.

We continue building the application that we started working on in the previous exercises. You can write the code into the same project since we are only interested in the final state of the submitted application.

Pro-tip: you may run into issues when it comes to the structure of the *props* that components receive. A good way to make things more clear is by printing the props to the console, e.g. as follows:

```
const Header = (props) => {
  console.log(props)  return <h1>{props.course}</h1>
}
```

If and *when* you encounter an error message

Objects are not valid as a React child

keep in mind the things told [here](#).

1.3: Course Information step 3

Let's move forward to using objects in our application. Modify the variable definitions of the *App* component as follows and also refactor the application so that it still works:

```
const App = () => [
  const course = 'Half Stack application development'
  const part1 = [
    name: 'Fundamentals of React',
    exercises: 10
  ]
  const part2 = [
    name: 'Using props to pass data',
    exercises: 7
]
```

```

    ]
  const part3 = [
    name: 'State of a component',
    exercises: 14
  ]

  return [
    <div>
      ...
    </div>
  ]
}

```

1.4: Course Information step 4

Place the objects into an array. Modify the variable definitions of *App* into the following form and modify the other parts of the application accordingly:

```

const App = () => {
  const course = 'Half Stack application development'
  const parts = [
    [
      name: 'Fundamentals of React',
      exercises: 10
    ],
    [
      name: 'Using props to pass data',
      exercises: 7
    ],
    [
      name: 'State of a component',
      exercises: 14
    ]
  ]

  return [
    <div>
      ...
    </div>
  ]
}

```

NB at this point you *can assume that there are always three items*, so there is no need to go through the arrays using loops. We will come back to the topic of rendering components based on items in arrays with a more thorough exploration in the [next part of the course](#).

However, do not pass different objects as separate props from the *App* component to the components *Content* and *Total*. Instead, pass them directly as an array:

```

const App = () => [
  // const definitions
]

```

```

        return [
          <div>
            <Header course={course} />
            <Content parts={parts} />
            <Total parts={parts} />
          </div>
        ]
      }
    }
  }
}

```

1.5: Course Information step 5

Let's take the changes one step further. Change the course and its parts into a single JavaScript object. Fix everything that breaks.

```

const App = () => {
  const course = {
    name: 'Half Stack application development',
    parts: [
      {
        name: 'Fundamentals of React',
        exercises: 10
      },
      {
        name: 'Using props to pass data',
        exercises: 7
      },
      {
        name: 'State of a component',
        exercises: 14
      }
    ]
  }

  return [
    <div>
      ...
    </div>
  ]
}

```

Object methods and "this"

Because this course uses a version of React containing React Hooks we do not need to define objects with methods. **The contents of this chapter are not relevant to the course** but are certainly in many ways good to know. In particular, when using older versions of React one must understand the topics of this chapter.

Arrow functions and functions defined using the *function* keyword vary substantially when it comes to how they behave with respect to the keyword **this**, which refers to the object itself.

We can assign methods to an object by defining properties that are functions:

```
const arto = {
  name: 'Arto Hellas',
  age: 35,
  education: 'PhD',
  greet: function() { console.log('hello, my name is ' + this.name) },
}

arto.greet() // "hello, my name is Arto Hellas" gets printed
```

Methods can be assigned to objects even after the creation of the object:

```
const arto = {
  name: 'Arto Hellas',
  age: 35,
  education: 'PhD',
  greet: function() {
    console.log('hello, my name is ' + this.name)
  },
}

arto.growOlder = function() { this.age += 1}
console.log(arto.age) // 35 is printed
arto.growOlder()
console.log(arto.age) // 36 is printed
```

Let's slightly modify the object:

```
const arto = {
  name: 'Arto Hellas',
  age: 35,
  education: 'PhD',
  greet: function() {
    console.log('hello, my name is ' + this.name)
  },
  doAddition: function(a, b) { console.log(a + b) }
}

arto.doAddition(1, 4) // 5 is printed

const referenceToAddition = arto.doAddition
referenceToAddition(10, 15) // 25 is printed
```

Now the object has the method *doAddition* which calculates the sum of numbers given to it as parameters. The method is called in the usual way, using the object *arto.doAddition(1, 4)* or by storing a *method reference* in a variable and calling the method through the variable: *referenceToAddition(10, 15)*.

If we try to do the same with the method *greet* we run into an issue:

```
arto.greet() // "hello, my name is Arto Hellas" gets printed
```

```
const referenceToGreet = arto.greet
referenceToGreet() // prints "hello, my name is undefined"
```

When calling the method through a reference, the method loses knowledge of what the original *this* was. Contrary to other languages, in JavaScript the value of *this* is defined based on *how the method is called*. When calling the method through a reference, the value of *this* becomes the so-called [global object](#) and the end result is often not what the software developer had originally intended.

Losing track of *this* when writing JavaScript code brings forth a few potential issues. Situations often arise where React or Node (or more specifically the JavaScript engine of the web browser) needs to call some method in an object that the developer has defined. However, in this course, we avoid these issues by using "this-less" JavaScript.

One situation leading to the "disappearance" of *this* arises when we set a timeout to call the *greet* function on the *arto* object, using the [setTimeout](#) function.

```
const arto = {
  name: 'Arto Hellas',
  greet: function() {
    console.log('hello, my name is ' + this.name)
  },
}

setTimeout(arto.greet, 1000)
```

As mentioned, the value of *this* in JavaScript is defined based on how the method is being called. When [setTimeout](#) is calling the method, it is the JavaScript engine that actually calls the method and, at that point, *this* refers to the global object.

There are several mechanisms by which the original *this* can be preserved. One of these is using a method called [bind](#):

```
setTimeout(arto.greet.bind(arto), 1000)
```

Calling *arto.greet.bind(arto)* creates a new function where *this* is bound to point to Arto, independent of where and how the method is being called.

Using [arrow functions](#) it is possible to solve some of the problems related to *this*. They should not, however, be used as methods for objects because then *this* does not work at all. We will come back later to the behavior of *this* in relation to arrow functions.

If you want to gain a better understanding of how *this* works in JavaScript, the Internet is full of material about the topic, e.g. the screencast series [Understand JavaScript's this Keyword in Depth](#) by [egghead.io](#) is highly recommended!

Classes

As mentioned previously, there is no class mechanism in JavaScript like the ones in object-oriented programming languages. There are, however, features to make "simulating" object-oriented [classes](#) possible.

Let's take a quick look at the *class syntax* that was introduced into JavaScript with ES6, which substantially simplifies the definition of classes (or class-like things) in JavaScript.

In the following example we define a "class" called Person and two Person objects:

```
class Person {
  constructor(name, age) {
    this.name = name
    this.age = age
  }
  greet() {
    console.log('hello, my name is ' + this.name)
  }
}

const adam = new Person('Adam Ondra', 29)
adam.greet()

const janja = new Person('Janja Garnbret', 23)
janja.greet()
```

When it comes to syntax, the classes and the objects created from them are very reminiscent of Java classes and objects. Their behavior is also quite similar to Java objects. At the core, they are still objects based on JavaScript's [prototypal inheritance](#). The type of both objects is actually *Object*, since JavaScript essentially only defines the types [Boolean](#), [Null](#), [Undefined](#), [Number](#), [String](#), [Symbol](#), [BigInt](#), and [Object](#).

The introduction of the class syntax was a controversial addition. Check out [Not Awesome: ES6 Classes or Is “Class” In ES6 The New “Bad” Part?](#) on Medium for more details.

The ES6 class syntax is used a lot in "old" React and also in Node.js, hence an understanding of it is beneficial even in this course. However, since we are using the new [Hooks](#) feature of React throughout this course, we have no concrete use for JavaScript's class syntax.

JavaScript materials

There exist both good and poor guides for JavaScript on the Internet. Most of the links on this page relating to JavaScript features reference [Mozilla's JavaScript Guide](#).

It is highly recommended to immediately read [A re-introduction to JavaScript \(JS tutorial\)](#) on Mozilla's website.

If you wish to get to know JavaScript deeply there is a great free book series on the Internet called [You-Dont-Know-JS](#).

Another great resource for learning JavaScript is [javascript.info](#).

The free and highly engaging book [Eloquent JavaScript](#) takes you from the basics to interesting stuff quickly. It is a mixture of theory projects and exercises and covers general programming theory as well as the JavaScript language.

[Namaste 🙏 JavaScript](#) is another great and highly recommended free JavaScript tutorial in order to understand how JS works under the hood. Namaste JavaScript is a pure in-depth JavaScript course

released for free on YouTube. It will cover the core concepts of JavaScript in detail and everything about how JS works behind the scenes inside the JavaScript engine.

[egghead.io](#) has plenty of quality screencasts on JavaScript, React, and other interesting topics. Unfortunately, some of the material is behind a paywall.

1c: Component state, event handlers

Let's go back to working with React.

We start with a new example:

```
const Hello = (props) => [
  return [
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>
    </div>
  ],
]

const App = () => [
  const name = 'Peter'
  const age = 10

  return [
    <div>
      <h1>Greetings</h1>
      <Hello name="Maya" age={26 + 10} />
      <Hello name={name} age={age} />
    </div>
  ],
]
```

Component helper functions

Let's expand our *Hello* component so that it guesses the year of birth of the person being greeted:

```
const Hello = (props) => [
  const bornYear = () => {
    const yearNow = new Date().getFullYear()
    return yearNow - props.age
  }
  return [
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>
      <p>So you were probably born in {bornYear()}</p>
    </div>
  ],
]
```

The logic for guessing the year of birth is separated into a function of its own that is called when the component is rendered.

The person's age does not have to be passed as a parameter to the function, since it can directly access all props that are passed to the component.

If we examine our current code closely, we'll notice that the helper function is defined inside of another function that defines the behavior of our component. In Java programming, defining a function inside another one is complex and cumbersome, so not all that common. In JavaScript, however, defining functions within functions is a commonly-used technique.

Destructuring

Before we move forward, we will take a look at a small but useful feature of the JavaScript language that was added in the ES6 specification, that allows us to **destruct** values from objects and arrays upon assignment.

In our previous code, we had to reference the data passed to our component as `props.name` and `props.age`. Of these two expressions, we had to repeat `props.age` twice in our code.

Since `props` is an object

```
props = {
  name: 'Arto Hellas',
  age: 35,
}
```

we can streamline our component by assigning the values of the properties directly into two variables `name` and `age` which we can then use in our code:

```
const Hello = (props) => [
  const name = props.name const age = props.age
  const bornYear = () => new Date().getFullYear() - age
  return [
    <div>
      <p>Hello {name}, you are {age} years old</p>      <p>So you were probably
      born in {bornYear()}</p>
    </div>
  ]
]
```

Note that we've also utilized the more compact syntax for arrow functions when defining the `bornYear` function. As mentioned earlier, if an arrow function consists of a single expression, then the function body does not need to be written inside of curly braces. In this more compact form, the function simply returns the result of the single expression.

To recap, the two function definitions shown below are equivalent:

```
const bornYear = () => new Date().getFullYear() - age
const bornYear = () => [
```

```
    return new Date().getFullYear() - age
}
```

Destructuring makes the assignment of variables even easier since we can use it to extract and gather the values of an object's properties into separate variables:

```
const Hello = (props) => [
  const { name, age } = props  const bornYear = () => new Date().getFullYear() - age

  return [
    <div>
      <p>Hello {name}, you are {age} years old</p>
      <p>So you were probably born in {bornYear()}</p>
    </div>
  ]
]
```

When the object that we are destructuring has the values

```
props = {
  name: 'Arto Hellas',
  age: 35,
}
```

the expression `const { name, age } = props` assigns the values 'Arto Hellas' to `name` and 35 to `age`.

We can take destructuring a step further:

```
const Hello = ({ name, age }) => [
  const bornYear = () => new Date().getFullYear() - age

  return [
    <div>
      <p>
        Hello {name}, you are {age} years old
      </p>
      <p>So you were probably born in {bornYear()}</p>
    </div>
  ]
]
```

The props that are passed to the component are now directly destructured into the variables, `name` and `age`.

This means that instead of assigning the entire props object into a variable called `props` and then assigning its properties to the variables `name` and `age`

```
const Hello = (props) => [
```

```
const { name, age } = props
```

we assign the values of the properties directly to variables by destructuring the props object that is passed to the component function as a parameter:

```
const Hello = ({ name, age }) => {
```

Page re-rendering

So far all of our applications have been such that their appearance remains the same after the initial rendering. What if we wanted to create a counter where the value increased as a function of time or at the click of a button?

Let's start with the following. File *App.jsx* becomes:

```
const App = (props) => {
  const {counter} = props
  return (
    <div>{counter}</div>
  )
}

export default App
```

And file *main.jsx* becomes:

```
import ReactDOM from 'react-dom/client'

import App from './App'

let counter = 1

ReactDOM.createRoot(document.getElementById('root')).render(
  <App counter={counter} />
)
```

The App component is given the value of the counter via the *counter* prop. This component renders the value to the screen. What happens when the value of *counter* changes? Even if we were to add the following

```
counter += 1
```

the component won't re-render. We can get the component to re-render by calling the *render* method a second time, e.g. in the following way:

```
let counter = 1

const refresh = () => {
  ReactDOM.createRoot(document.getElementById('root')).render(
```

```

        <App counter={counter} />
    }
}

refresh()
counter += 1
refresh()
counter += 1
refresh()

```

The re-rendering command has been wrapped inside of the `refresh` function to cut down on the amount of copy-pasted code.

Now the component *renders three times*, first with the value 1, then 2, and finally 3. However, values 1 and 2 are displayed on the screen for such a short amount of time that they can't be noticed.

We can implement slightly more interesting functionality by re-rendering and incrementing the counter every second by using [setInterval](#):

```

setInterval(() => {
    refresh()
    counter += 1
}, 1000)

```

Making repeated calls to the `render` method is not the recommended way to re-render components. Next, we'll introduce a better way of accomplishing this effect.

Stateful component

All of our components up till now have been simple in the sense that they have not contained any state that could change during the lifecycle of the component.

Next, let's add state to our application's `App` component with the help of React's [state hook](#).

We will change the application as follows. `main.jsx` goes back to:

```

import ReactDOM from 'react-dom/client'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)

```

and `App.jsx` changes to the following:

```

import { useState } from 'react'
const App = () => {
    const [ counter, setCounter ] = useState(0)
    setTimeout(() => setCounter(counter + 1), 1000)
    return (
        <div>{counter}</div>
    )
}

```

```
}

export default App
```

In the first row, the file imports the *useState* function:

```
import { useState } from 'react'
```

The function body that defines the component begins with the function call:

```
const [ counter, setCounter ] = useState(0)
```

The function call adds *state* to the component and renders it initialized with the value zero. The function returns an array that contains two items. We assign the items to the variables *counter* and *setCounter* by using the destructuring assignment syntax shown earlier.

The *counter* variable is assigned the initial value of *state*, which is zero. The variable *setCounter* is assigned a function that will be used to *modify the state*.

The application calls the *setTimeout* function and passes it two parameters: a function to increment the counter state and a timeout of one second:

```
setTimeout(
  () => setCounter(counter + 1),
  1000
)
```

The function passed as the first parameter to the *setTimeout* function is invoked one second after calling the *setTimeout* function

```
() => setCounter(counter + 1)
```

When the state modifying function *setCounter* is called, *React re-renders the component* which means that the function body of the component function gets re-executed:

```
() => {
  const [ counter, setCounter ] = useState(0)

  setTimeout(
    () => setCounter(counter + 1),
    1000
  )

  return (
    <div>{counter}</div>
  )
}
```

The second time the component function is executed it calls the *useState* function and returns the

new value of the state: 1. Executing the function body again also makes a new function call to `setTimeout`, which executes the one-second timeout and increments the `counter` state again. Because the value of the `counter` variable is 1, incrementing the value by 1 is essentially the same as an expression setting the value of `counter` to 2.

```
{() => setCounter(2)}
```

Meanwhile, the old value of `counter` - "1" - is rendered to the screen.

Every time the `setCounter` modifies the state it causes the component to re-render. The value of the state will be incremented again after one second, and this will continue to repeat for as long as the application is running.

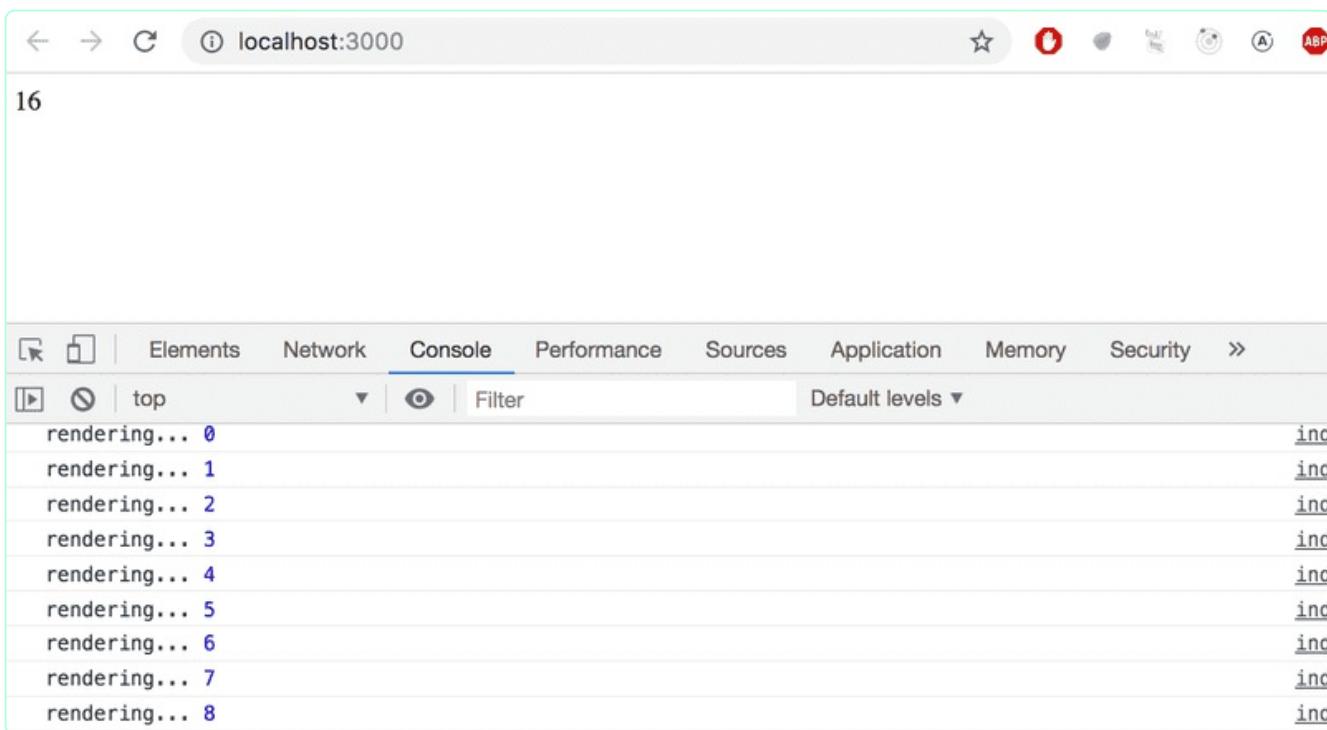
If the component doesn't render when you think it should, or if it renders at the "wrong time", you can debug the application by logging the values of the component's variables to the console. If we make the following additions to our code:

```
const App = () => [
  const [ counter, setCounter ] = useState(0)

  setTimeout(
    () => setCounter(counter + 1),
    1000
  )

  console.log('rendering...', counter)
  return (
    <div>{counter}</div>
  )
]
```

It's easy to follow and track the calls made to the `App` component's render function:



Was your browser console open? If it wasn't, then promise that this was the last time you need to be reminded about it.

Event handling

We have already mentioned the *event handlers* that are registered to be called when specific events occur a few times in [part 0](#). A user's interaction with the different elements of a web page can cause a collection of various kinds of events to be triggered.

Let's change the application so that increasing the counter happens when a user clicks a button, which is implemented with the `button` element.

Button elements support so-called `mouse events`, of which `click` is the most common event. The `click` event on a button can also be triggered with the keyboard or a touch screen despite the name *mouse event*.

In React, [registering an event handler function](#) to the `click` event happens like this:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  const handleClick = () => {    console.log('clicked')  }
  return [
    <div>
      <div>{counter}</div>
      <button onClick={handleClick}>      plus      </button>      </div>
    ]
}
```

We set the value of the button's `onClick` attribute to be a reference to the `handleClick` function defined in the code.

Now every click of the *plus* button causes the *handleClick* function to be called, meaning that every click event will log a *clicked* message to the browser console.

The event handler function can also be defined directly in the value assignment of the *onClick*-attribute:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  return [
    <div>
      <div>{counter}</div>
      <button onClick={() => console.log('clicked')}> plus
      </button>
    </div>
  ]
}
```

By changing the event handler to the following form

```
<button onClick={() => setCounter(counter + 1)}>
  plus
</button>
```

we achieve the desired behavior, meaning that the value of *counter* is increased by one *and* the component gets re-rendered.

Let's also add a button for resetting the counter:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  return [
    <div>
      <div>{counter}</div>
      <button onClick={() => setCounter(counter + 1)}>
        plus
      </button>
      <button onClick={() => setCounter(0)}> zero </button>
    </div>
  ]
}
```

Our application is now ready!

An event handler is a function

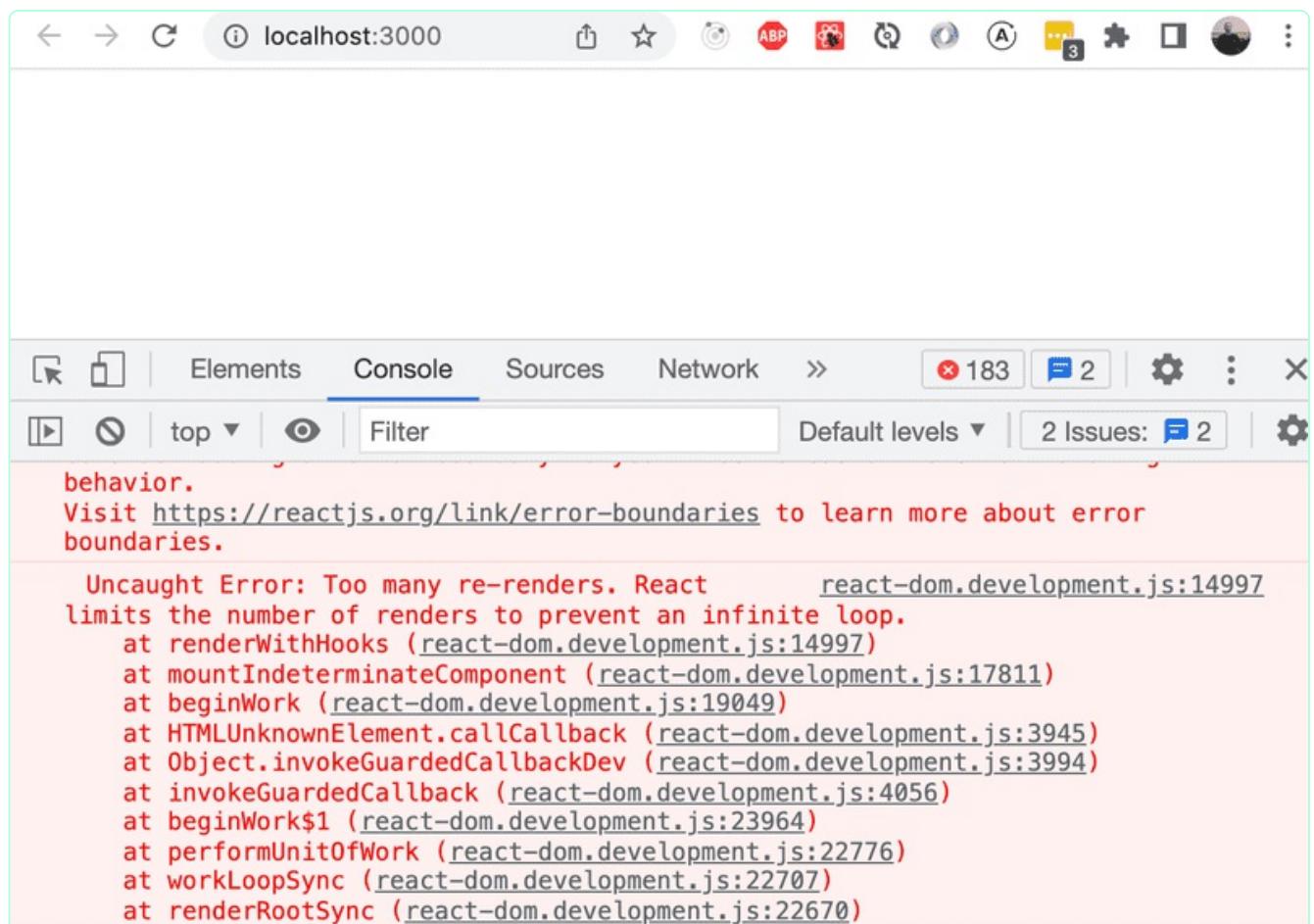
We define the event handlers for our buttons where we declare their *onClick* attributes:

```
<button onClick={() => setCounter(counter + 1)}>  
  plus  
</button>
```

What if we tried to define the event handlers in a simpler form?

```
<button onClick=[setCounter(counter + 1)]>  
  plus  
</button>
```

This would completely break our application:



What's going on? An event handler is supposed to be either a *function* or a *function reference*, and when we write:

```
<button onClick=[setCounter(counter + 1)]>
```

the event handler is actually a *function call*. In many situations this is ok, but not in this particular situation. In the beginning, the value of the *counter* variable is 0. When React renders the component for the first time, it executes the function call *setCounter(0+1)*, and changes the value of the component's state to 1. This will cause the component to be re-rendered, React will execute the *setCounter* function call again, and the state will change leading to another re-render...

Let's define the event handlers like we did before:

```
<button onClick={() => setCounter(counter + 1)}>  
  plus  
</button>
```

Now the button's attribute which defines what happens when the button is clicked - `onClick` - has the value `() => setCounter(counter + 1)`. The `setCounter` function is called only when a user clicks the button.

Usually defining event handlers within JSX-templates is not a good idea. Here it's ok, because our event handlers are so simple.

Let's separate the event handlers into separate functions anyway:

```
const App = () => {  
  const [ counter, setCounter ] = useState(0)  
  
  const increaseByOne = () => setCounter(counter + 1)      const setToZero = () =>  
  setCounter(0)  
  return (  
    <div>  
      <div>{counter}</div>  
      <button onClick={increaseByOne}>          plus  
      </button>  
      <button onClick={setToZero}>          zero  
      </button>  
    </div>  
  )  
}
```

Here, the event handlers have been defined correctly. The value of the `onClick` attribute is a variable containing a reference to a function:

```
<button onClick={increaseByOne}>  
  plus  
</button>
```

Passing state - to child components

It's recommended to write React components that are small and reusable across the application and even across projects. Let's refactor our application so that it's composed of three smaller components, one component for displaying the counter and two components for buttons.

Let's first implement a `Display` component that's responsible for displaying the value of the counter.

One best practice in React is to [lift the state up](#) in the component hierarchy. The documentation says:

Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor.

So let's place the application's state in the *App* component and pass it down to the *Display* component through *props*:

```
const Display = (props) => {
  return [
    <div>{props.counter}</div>
  ]
}
```

Using the component is straightforward, as we only need to pass the state of the *counter* to it:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  const increaseByOne = () => setCounter(counter + 1)
  const setToZero = () => setCounter(0)

  return [
    <div>
      <Display counter={counter}>/>           <button onClick={increaseByOne}>
        plus
      </button>
      <button onClick={setToZero}>
        zero
      </button>
    </div>
  ]
}
```

Everything still works. When the buttons are clicked and the *App* gets re-rendered, all of its children including the *Display* component are also re-rendered.

Next, let's make a *Button* component for the buttons of our application. We have to pass the event handler as well as the title of the button through the component's *props*:

```
const Button = (props) => {
  return [
    <button onClick={props.onClick}>
      {props.text}
    </button>
  ]
}
```

Our *App* component now looks like this:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)
```

```

const increaseByOne = () => setCounter(counter + 1)
const decreaseByOne = () => setCounter(counter - 1)  const setToZero = () => s
etCounter(0)

return [
  <div>
    <Display counter={counter}>/>
    <Button onClick={increaseByOne} text='plus' /> <Bu
tton onClick={setToZero} text='zero' /> <Button
onClick={decreaseByOne} text='minus' />
  </div>
]

```

Since we now have an easily reusable *Button* component, we've also implemented new functionality into our application by adding a button that can be used to decrement the counter.

The event handler is passed to the *Button* component through the *onClick* prop. The name of the prop itself is not that significant, but our naming choice wasn't completely random.

React's own official [tutorial](#) suggests: "In React, it's conventional to use *onSomething* names for props which take functions which handle events and *handleSomething* for the actual function definitions which handle those events."

Changes in state cause re-rendering

Let's go over the main principles of how an application works once more.

When the application starts, the code in *App* is executed. This code uses a [useState](#) hook to create the application state, setting an initial value of the variable *counter*. This component contains the *Display* component - which displays the counter's value, 0 - and three *Button* components. The buttons all have event handlers, which are used to change the state of the counter.

When one of the buttons is clicked, the event handler is executed. The event handler changes the state of the *App* component with the *setCounter* function. **Calling a function that changes the state causes the component to re-render.**

So, if a user clicks the *plus* button, the button's event handler changes the value of *counter* to 1, and the *App* component is re-rendered. This causes its subcomponents *Display* and *Button* to also be re-rendered. *Display* receives the new value of the counter, 1, as props. The *Button* components receive event handlers which can be used to change the state of the counter.

To be sure to understand how the program works, let us add some *console.log* statements to it

```

const App = () => [
  const [counter, setCounter] = useState(0)
  console.log('rendering with counter value', counter)
  const increaseByOne = () => [
    console.log('increasing, value before', counter)      setCounter(counter + 1)
  ]

  const decreaseByOne = () => [
    console.log('decreasing, value before', counter)      setCounter(counter - 1)
  ]
]

```

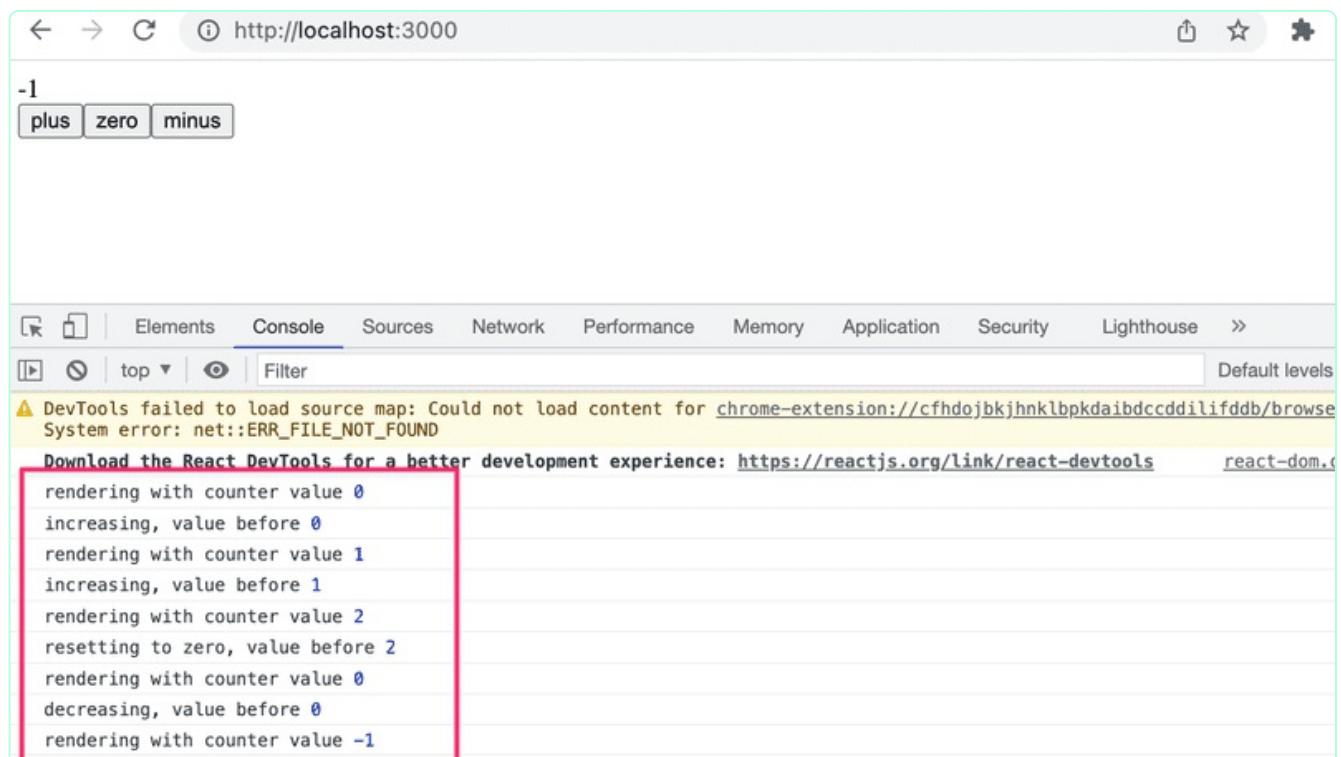
```

const setToZero = () => {
  console.log('resetting to zero, value before', counter)    setCounter(0)
}

return [
  <div>
    <Display counter={counter} />
    <Button onClick={increaseByOne} text="plus" />
    <Button onClick={setToZero} text="zero" />
    <Button onClick={decreaseByOne} text="minus" />
  </div>
]
}

```

Let us now see what gets rendered to the console when the buttons plus, zero and minus are pressed:



Do not ever try to guess what your code does. It is just better to use `console.log` and *see with your own eyes* what it does.

Refactoring the components

The component displaying the value of the counter is as follows:

```

const Display = (props) => {
  return [
    <div>{props.counter}</div>
  ]
}

```

The component only uses the *counter* field of its *props*. This means we can simplify the component by using **destructuring**, like so:

```
const Display = ({ counter }) => {
  return (
    <div>{counter}</div>
  )
}
```

The function defining the component contains only the return statement, so we can define the function using the more compact form of arrow functions:

```
const Display = ({ counter }) => <div>{counter}</div>
```

We can simplify the Button component as well.

```
const Button = (props) => {
  return (
    <button onClick={props.onClick}>
      {props.text}
    </button>
  )
}
```

We can use destructuring to get only the required fields from *props*, and use the more compact form of arrow functions:

NB: While building your own components, you can name their event handler props anyway you like, for this you can refer to the react's documentation on [Naming event handler props](#). It goes as following:

By convention, event handler props should start with `on`, followed by a capital letter. For example, the Button component's `onClick` prop could have been called `onSmash`:

```
const Button = ({ onClick, text }) => (
  <button onClick={onClick}>
    {text}
  </button>
)
```

could also be called as following:

```
const Button = ({ onSmash, text }) => (
  <button onClick={onSmash}>
    {text}
  </button>
)
```

We can simplify the Button component once more by declaring the return statement in just one line:

```
const Button = ({ onSmash, text }) => <button onClick={onSmash}>{text}</button>
```

NB: However, be careful to not oversimplify your components, as this makes adding complexity a more tedious task down the road.

1d: A more complex state, debugging React apps

Complex state

In our previous example, the application state was simple as it was comprised of a single integer. What if our application requires a more complex state?

In most cases, the easiest and best way to accomplish this is by using the `useState` function multiple times to create separate "pieces" of state.

In the following code we create two pieces of state for the application named `left` and `right` that both get the initial value of 0:

```
const App = () => {
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)

  return (
    <div>
      {left}
      <button onClick={() => setLeft(left + 1)}>
        left
      </button>
      <button onClick={() => setRight(right + 1)}>
        right
      </button>
      {right}
    </div>
  )
}
```

The component gets access to the functions `setLeft` and `setRight` that it can use to update the two pieces of state.

The component's state or a piece of its state can be of any type. We could implement the same functionality by saving the click count of both the `left` and `right` buttons into a single object:

```
{
  left: 0,
  right: 0
}
```

In this case, the application would look like this:

```
const App = () => {
  const [clicks, setClicks] = useState({
    left: 0, right: 0
  })

  const handleLeftClick = () => {
    const newClicks = {
      left: clicks.left + 1,
      right: clicks.right
    }
    setClicks(newClicks)
  }

  const handleRightClick = () => {
    const newClicks = {
      left: clicks.left,
      right: clicks.right + 1
    }
    setClicks(newClicks)
  }

  return (
    <div>
      {clicks.left}
      <button onClick={handleLeftClick}>left</button>
      <button onClick={handleRightClick}>right</button>
      {clicks.right}
    </div>
  )
}
```

Now the component only has a single piece of state and the event handlers have to take care of changing the *entire application state*.

The event handler looks a bit messy. When the left button is clicked, the following function is called:

```
const handleLeftClick = () => {
  const newClicks = {
    left: clicks.left + 1,
    right: clicks.right
  }
  setClicks(newClicks)
}
```

The following object is set as the new state of the application:

```
{
  left: clicks.left + 1,
```

```
    right: clicks.right  
  }  
}
```

The new value of the `left` property is now the same as the value of `left + 1` from the previous state, and the value of the `right` property is the same as the value of the `right` property from the previous state.

We can define the new state object a bit more neatly by using the [object spread](#) syntax that was added to the language specification in the summer of 2018:

```
const handleLeftClick = () => {  
  const newClicks = {  
    ...clicks,  
    left: clicks.left + 1  
  }  
  setClicks(newClicks)  
}  
  
const handleRightClick = () => {  
  const newClicks = {  
    ...clicks,  
    right: clicks.right + 1  
  }  
  setClicks(newClicks)  
}
```

The syntax may seem a bit strange at first. In practice `{ ...clicks }` creates a new object that has copies of all of the properties of the `clicks` object. When we specify a particular property - e.g. `right` in `{ ...clicks, right: 1 }`, the value of the `right` property in the new object will be 1.

In the example above, this:

```
{ ...clicks, right: clicks.right + 1 }
```

creates a copy of the `clicks` object where the value of the `right` property is increased by one.

Assigning the object to a variable in the event handlers is not necessary and we can simplify the functions to the following form:

```
const handleLeftClick = () =>  
  setClicks({ ...clicks, left: clicks.left + 1 })  
  
const handleRightClick = () =>  
  setClicks({ ...clicks, right: clicks.right + 1 })
```

Some readers might be wondering why we didn't just update the state directly, like this:

```
const handleLeftClick = () => [  
  clicks.left++,  
  setClicks(clicks)
```

The application appears to work. However, *it is forbidden in React to mutate state directly*, since it can result in unexpected side effects. Changing state has to always be done by setting the state to a new object. If properties from the previous state object are not changed, they need to simply be copied, which is done by copying those properties into a new object and setting that as the new state.

Storing all of the state in a single state object is a bad choice for this particular application; there's no apparent benefit and the resulting application is a lot more complex. In this case, storing the click counters into separate pieces of state is a far more suitable choice.

There are situations where it can be beneficial to store a piece of application state in a more complex data structure. [The official React documentation](#) contains some helpful guidance on the topic.

Handling arrays

Let's add a piece of state to our application containing an array *allClicks* that remembers every click that has occurred in the application.

```
const App = () => {
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)
  const [allClicks, setAll] = useState([])
  const handleLeftClick = () => {
    setAll(allClicks.concat('L'))
    setLeft(left + 1)
  }
  const handleRightClick = () => {
    setAll(allClicks.concat('R'))
    setRight(right + 1)
  }
  return (
    <div>
      {left}
      <button onClick={handleLeftClick}>left</button>
      <button onClick={handleRightClick}>right</button>
      {right}
      <p>{allClicks.join(' ')</p>
    </div>
  )
}
```

Every click is stored in a separate piece of state called *allClicks* that is initialized as an empty array:

```
const [allClicks, setAll] = useState([])
```

When the *left* button is clicked, we add the letter *L* to the *allClicks* array:

```
const handleLeftClick = () => {
  setAll(allClicks.concat('L'))
  setLeft(left + 1)
}
```

The piece of state stored in *allClicks* is now set to be an array that contains all of the items of the previous state array plus the letter *L*. Adding the new item to the array is accomplished with the

`concat` method, which does not mutate the existing array but rather returns a *new copy of the array* with the item added to it.

As mentioned previously, it's also possible in JavaScript to add items to an array with the `push` method. If we add the item by pushing it to the `allClicks` array and then updating the state, the application would still appear to work:

```
const handleLeftClick = () => [
  allClicks.push('L')
  setAll(allClicks)
  setLeft(left + 1)
]
```

However, **don't** do this. As mentioned previously, the state of React components, like `allClicks`, must not be mutated directly. Even if mutating state appears to work in some cases, it can lead to problems that are very hard to debug.

Let's take a closer look at how the clicking is rendered to the page:

```
const App = () => [
  // ...

  return [
    <div>
      {left}
      <button onClick={handleLeftClick}>left</button>
      <button onClick={handleRightClick}>right</button>
      {right}
      <p>{allClicks.join(' ')}</p>
    </div>
  ]
]
```

We call the `join` method on the `allClicks` array, that joins all the items into a single string, separated by the string passed as the function parameter, which in our case is an empty space.

Update of the state is asynchronous

Let's expand the application so that it keeps track of the total number of button presses in the state `total`, whose value is always updated when the buttons are pressed:

```
const App = () => [
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)
  const [allClicks, setAll] = useState([])
  const [total, setTotal] = useState(0)
  const handleLeftClick = () => [
    setAll(allClicks.concat('L'))
    setLeft(left + 1)
    setTotal(left + right)
  ]

  const handleRightClick = () => {
```

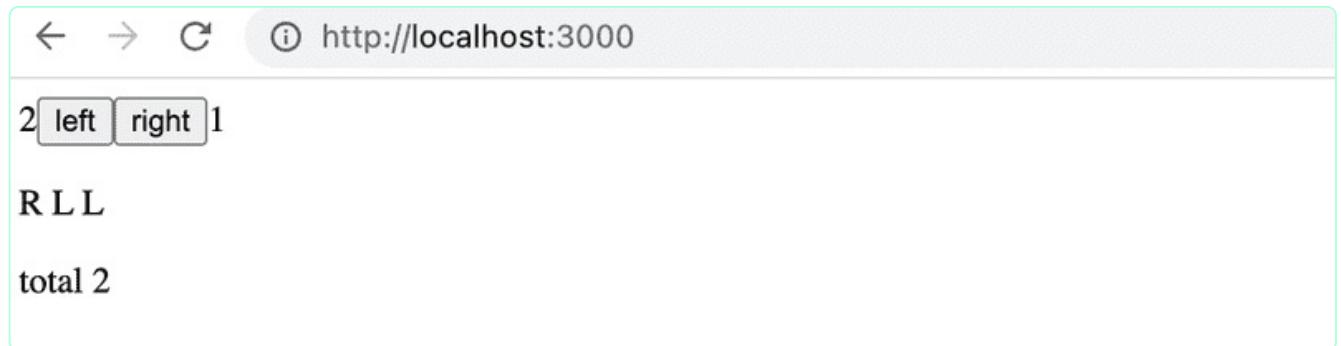
```

    setAll(allClicks.concat('R'))
    setRight(right + 1)
    setTotal(left + right)  }

  return [
    <div>
      {left}
      <button onClick={handleLeftClick}>left</button>
      <button onClick={handleRightClick}>right</button>
      {right}
      <p>{allClicks.join(' ') }</p>
      <p>total {total}</p>    </div>
    )
  ]
}

```

The solution does not quite work:



The total number of button presses is consistently one less than the actual amount of presses, for some reason.

Let us add couple of `console.log` statements to the event handler:

```

const App = () => [
  // ...
  const handleLeftClick = () => {
    setAll(allClicks.concat('L'))
    console.log('left before', left)    setLeft(left + 1)
    console.log('left after', left)     setTotal(left + right)
  }
  // ...
]

```

The console reveals the problem

```
left before 4
left after 4
```

Even though a new value was set for `left` by calling `setLeft(left + 1)`, the old value persists despite the update. As a result, the attempt to count button presses produces a result that is too small:

```
setTotal(left + right)
```

The reason for this is that a state update in React happens **asynchronously**, i.e. not immediately but "at some point" before the component is rendered again.

We can fix the app as follows:

```
const App = () => [
  // ...
  const handleLeftClick = () => [
    setAll(allClicks.concat('L'))
    const updatedLeft = left + 1
    setLeft(updatedLeft)
    setTotal(updatedLeft + right)
  ]
  // ...
]
```

So now the number of button presses is definitely based on the correct number of left button presses.

Conditional rendering

Let's modify our application so that the rendering of the clicking history is handled by a new *History* component:

```
const History = (props) => {
  if (props.allClicks.length === 0) {
    return (
      <div> the app is used by pressing the buttons </div>
    )
  }
  return [
    <div> button press history: {props.allClicks.join(' ')} </div>
  ]
}

const App = () => {
```

```
// ...

return [
  <div>
    {left}
    <button onClick={handleLeftClick}>left</button>
    <button onClick={handleRightClick}>right</button>
    {right}
    <History allClicks={allClicks} />      </div>
  )
]
```

Now the behavior of the component depends on whether or not any buttons have been clicked. If not, meaning that the `allClicks` array is empty, the component renders a `div` element with some instructions instead:

```
<div>the app is used by pressing the buttons</div>
```

And in all other cases, the component renders the clicking history:

```
<div>
  button press history: {props.allClicks.join(' ')}
</div>
```

The `History` component renders completely different React elements depending on the state of the application. This is called *conditional rendering*.

React also offers many other ways of doing [conditional rendering](#). We will take a closer look at this in [part 2](#).

Let's make one last modification to our application by refactoring it to use the `Button` component that we defined earlier on:

```
const History = (props) => {
  if (props.allClicks.length === 0) {
    return [
      <div>
        the app is used by pressing the buttons
      </div>
    ]
  }

  return [
    <div>
      button press history: {props.allClicks.join(' ')}
    </div>
  ]
}

const Button = ({ handleClick, text }) => (  <button onClick={handleClick}>  [
```

```

text} </button>
const App = () => {
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)
  const [allClicks, setAll] = useState([])

  const handleLeftClick = () => {
    setAll(allClicks.concat('L'))
    setLeft(left + 1)
  }

  const handleRightClick = () => {
    setAll(allClicks.concat('R'))
    setRight(right + 1)
  }

  return (
    <div>
      {left}
      <Button handleClick={handleLeftClick} text='left' /> <Button handleClick={handleRightClick} text='right' /> {right}
      <History allClicks={allClicks} />
    </div>
  )
}

```

Old React

In this course, we use the [state hook](#) to add state to our React components, which is part of the newer versions of React and is available from version [16.8.0](#) onwards. Before the addition of hooks, there was no way to add state to functional components. Components that required state had to be defined as [class](#) components, using the JavaScript class syntax.

In this course, we have made the slightly radical decision to use hooks exclusively from day one, to ensure that we are learning the current and future variations of React. Even though functional components are the future of React, it is still important to learn the class syntax, as there are billions of lines of legacy React code that you might end up maintaining someday. The same applies to documentation and examples of React that you may stumble across on the internet.

We will learn more about React class components later on in the course.

Debugging React applications

A large part of a typical developer's time is spent on debugging and reading existing code. Every now and then we do get to write a line or two of new code, but a large part of our time is spent trying to figure out why something is broken or how something works. Good practices and tools for debugging are extremely important for this reason.

Lucky for us, React is an extremely developer-friendly library when it comes to debugging.

Before we move on, let us remind ourselves of one of the most important rules of web development.

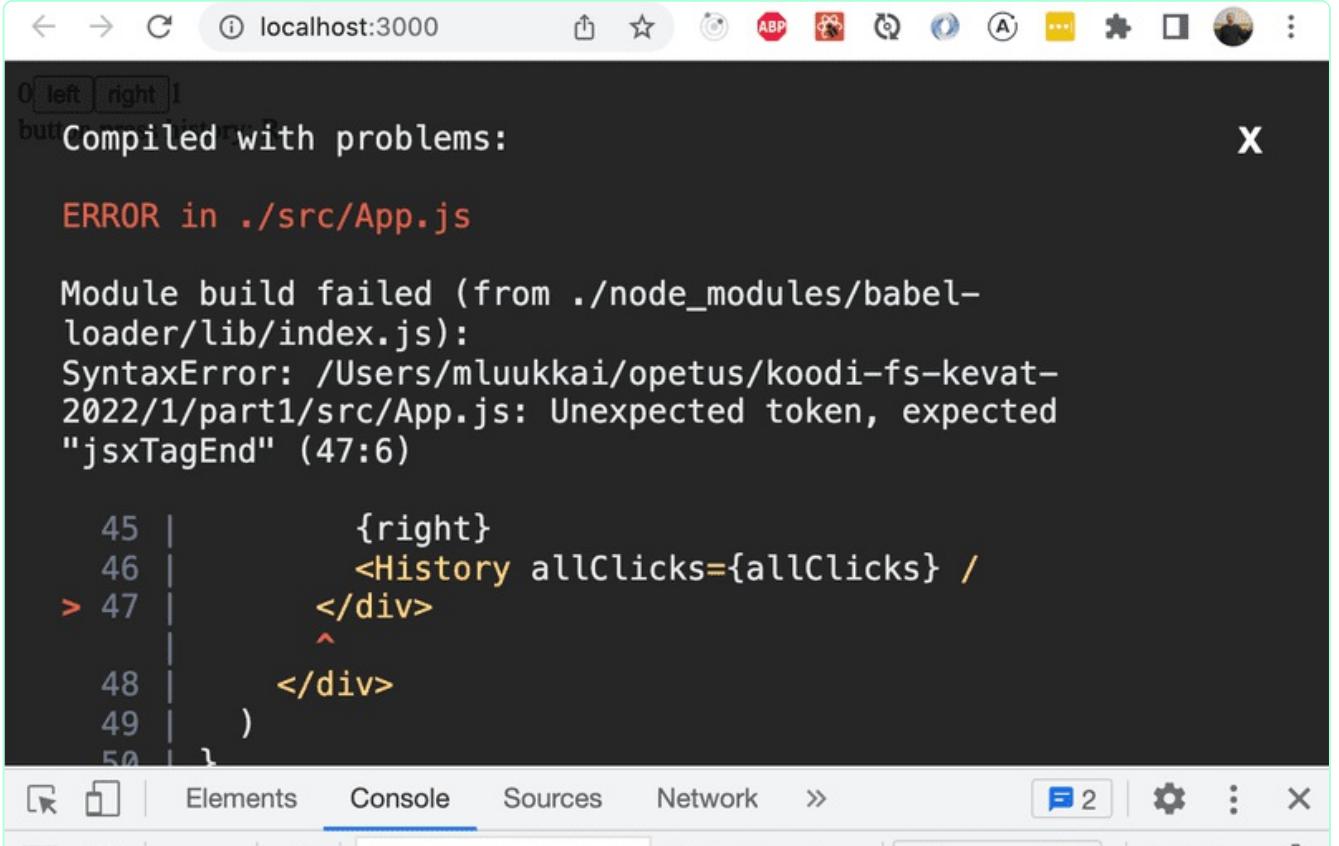
The first rule of web development

Keep the browser's developer console open at all times.

The *Console* tab in particular should always be open, unless there is a specific reason to view another tab.

Keep both your code and the web page open together **at the same time, all the time**.

If and when your code fails to compile and your browser lights up like a Christmas tree:



A screenshot of a browser's developer tools console. The URL bar shows 'localhost:3000'. The console output is as follows:

```
0 [left] right 1
button component
Compiled with problems:
ERROR in ./src/App.js

Module build failed (from ./node_modules/babel-loader/lib/index.js):
SyntaxError: /Users/mluukkai/opetus/koodi-fs-kevat-2022/1/part1/src/App.js: Unexpected token, expected "jsxBalancedEnd" (47:6)

  45 |           {right}
  46 |             <History allClicks={allClicks} /
> 47 |             </div>
    |             ^
  48 |           </div>
  49 |         )
  50 |       
```

The console tab is selected in the bottom navigation bar, along with Elements, Sources, and Network.

don't write more code but rather find and fix the problem **immediately**. There has yet to be a moment in the history of coding where code that fails to compile would miraculously start working after writing large amounts of additional code. I highly doubt that such an event will transpire during this course either.

Old-school, print-based debugging is always a good idea. If the component

```
const Button = ({ handleClick, text }) => (
  <button onClick={handleClick}>
    {text}
  </button>
)
```

is not working as intended, it's useful to start printing its variables out to the console. In order to do this effectively, we must transform our function into the less compact form and receive the entire

props object without destructuring it immediately:

```
const Button = (props) => {
  console.log(props)
  const { handleClick, text } = props
  return (
    <button onClick={handleClick}>
      {text}
    </button>
  )
}
```

This will immediately reveal if, for instance, one of the attributes has been misspelled when using the component.

NB When you use `console.log` for debugging, don't combine *objects* in a Java-like fashion by using the plus operator:

```
console.log('props value is ' + props)
```

If you do that, you will end up with a rather uninformative log message:

```
props value is [object Object]
```

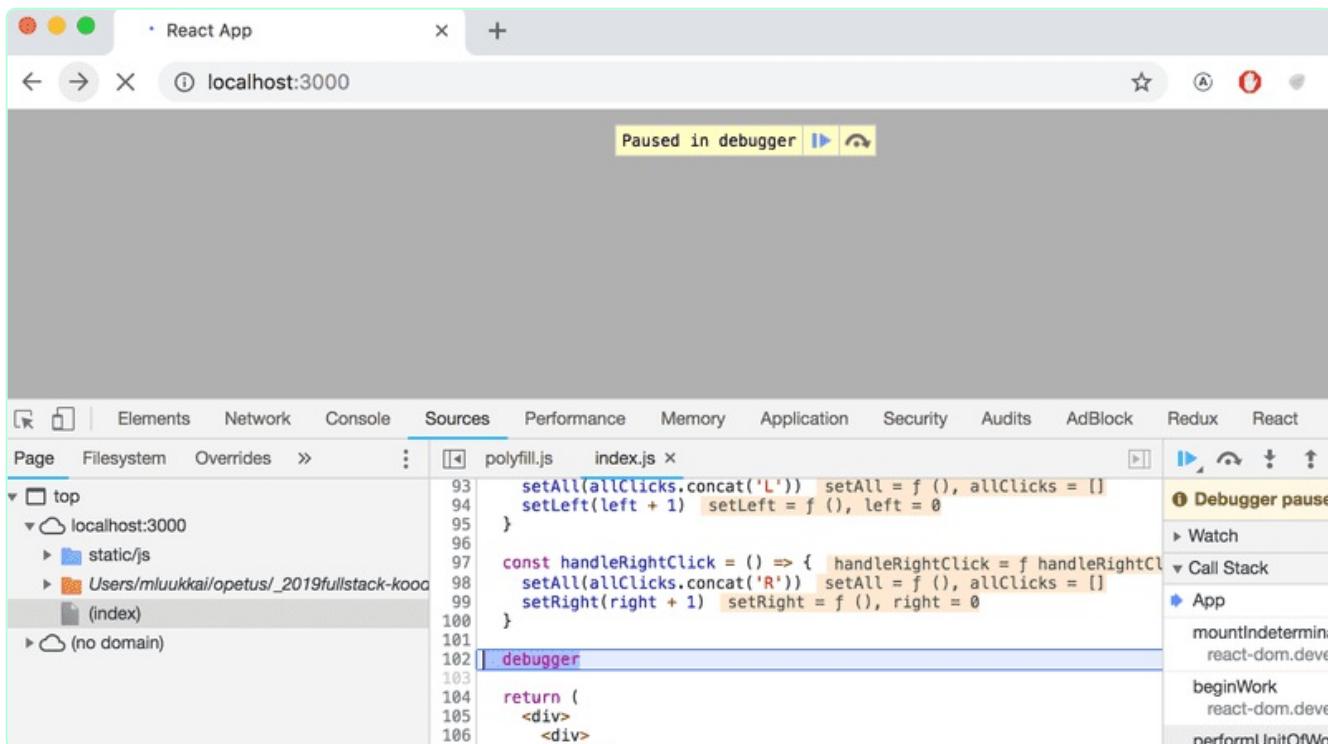
Instead, separate the things you want to log to the console with a comma:

```
console.log('props value is', props)
```

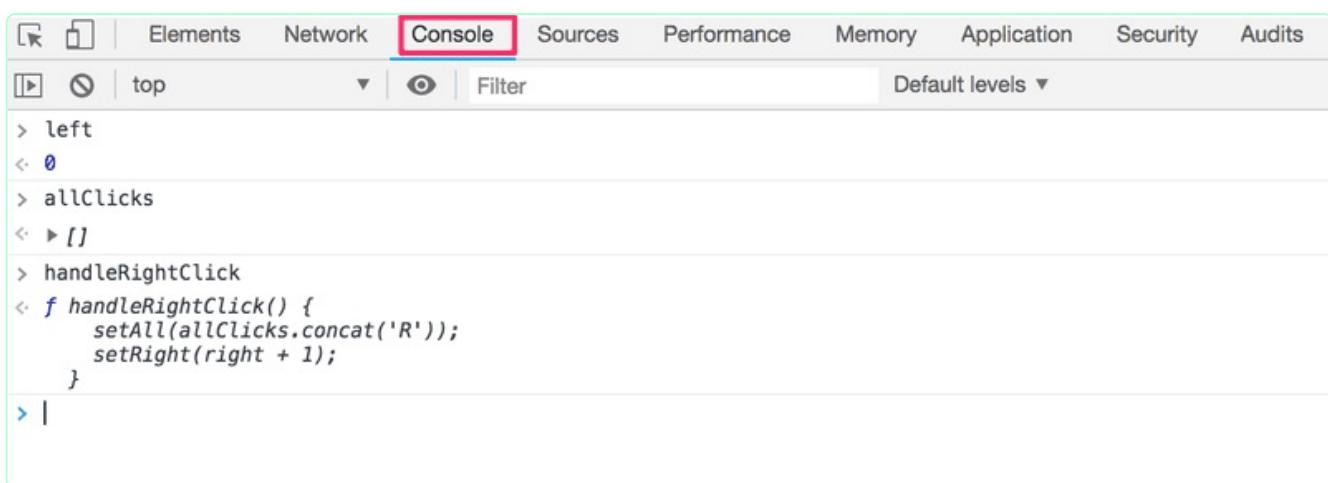
In this way, the separated items will all be available in the browser console for further inspection.

Logging output to the console is by no means the only way of debugging our applications. You can pause the execution of your application code in the Chrome developer console's *debugger*, by writing the command `debugger` anywhere in your code.

The execution will pause once it arrives at a point where the *debugger* command gets executed:



By going to the *Console* tab, it is easy to inspect the current state of variables:



Once the cause of the bug is discovered you can remove the *debugger* command and refresh the page.

The debugger also enables us to execute our code line by line with the controls found on the right-hand side of the *Sources* tab.

You can also access the debugger without the *debugger* command by adding breakpoints in the *Sources* tab. Inspecting the values of the component's variables can be done in the *Scope*-section:

The screenshot shows the Chrome Developer Tools interface with the 'Sources' tab selected. In the left sidebar, there's a tree view of files under 'localhost:3000'. A red arrow points from this sidebar to the code editor area. The code editor shows a snippet of JavaScript with a breakpoint at line 92. The right panel contains the 'Scope' inspector, which shows the state of variables like `this`, `Closure (App)`, and local variables like `left` and `right`.

```

    83  )
84  }
85
86 const App = (props) => {
87   const [left, setLeft] = useState(0)
88   const [right, setRight] = useState(0)
89   const [allClicks, setAll] = useState([])
90
91   const handleLeftClick = () => {
92     setAll(allClicks.concat('L'))
93     setLeft(left + 1)
94   }
95
96   const handleRightClick = () => {
97     setAll(allClicks.concat('R'))
98     setRight(right + 1)
99   }
100
101 } Line 92, Column 1      (source mapped from main.chunk.js)

```

It is highly recommended to add the [React developer tools](#) extension to Chrome. It adds a new *Components* tab to the developer tools. The new developer tools tab can be used to inspect the different React elements in the application, along with their state and props:

The screenshot shows the Chrome Developer Tools interface with the 'Components' tab selected. On the left, there's a tree view of components. A red arrow points to the 'History' section on the right, which displays the click history for the 'App' component. The history shows five events: 'L', 'L', 'R', 'R', and 'L'.

Props
allClicks: ["L", "L", "R", "R", "L"] new prop: ""

rendered by
App

source
src/index.js:119

The *App* component's state is defined like so:

```

const [left, setLeft] = useState(0)
const [right, setRight] = useState(0)
const [allClicks, setAll] = useState([])

```

Dev tools show the state of hooks in the order of their definition:

3 left right 2
button press history: L L R R L

Elements Console Sources Network Performance Memory Application Security Components App

Search (text or /regex/)

App

props
new prop: ""

hooks
State: 3
State: 2
State: ["L", "L", "R", "R", "L"]

source
src/index.js:125

The first *State* contains the value of the *left* state, the next contains the value of the *right* state and the last contains the value of the *allClicks* state.

Rules of Hooks

There are a few limitations and [rules](#) that we have to follow to ensure that our application uses hooks-based state functions correctly.

The *useState* function (as well as the *useEffect* function introduced later on in the course) *must not be called* from inside of a loop, a conditional expression, or any place that is not a function defining a component. This must be done to ensure that the hooks are always called in the same order, and if this isn't the case the application will behave erratically.

To recap, hooks may only be called from the inside of a function body that defines a React component:

```
const App = () => [
  // these are ok
  const [age, setAge] = useState(0)
  const [name, setName] = useState('Juha Tauriainen')

  if (age > 10) {
    // this does not work!
    const [foobar, setFoobar] = useState(null)
  }

  for (let i = 0; i < age; i++) {
    // also this is not good
    const [rightWay, setRightWay] = useState(false)
  }

  const notGood = () => {
    // and this is also illegal
    const [x, setX] = useState(-1000)
  }
]
```

```
}

return [
  //...
]

}
```

Event Handling Revisited

Event handling has proven to be a difficult topic in previous iterations of this course.

For this reason, we will revisit the topic.

Let's assume that we're developing this simple application with the following component *App*:

```
const App = () => {
  const [value, setValue] = useState(10)

  return [
    <div>
      {value}
      <button>reset to zero</button>
    </div>
  ]
}
```

We want the clicking of the button to reset the state stored in the *value* variable.

In order to make the button react to a click event, we have to add an *event handler* to it.

Event handlers must always be a function or a reference to a function. The button will not work if the event handler is set to a variable of any other type.

If we were to define the event handler as a string:

```
<button onClick="crap...">button</button>
```

React would warn us about this in the console:

```
index.js:2178 Warning: Expected `onClick` listener to be a function, instead got
a value of `string` type.
  in button (at index.js:20)
  in div (at index.js:18)
  in App (at index.js:27)
```

The following attempt would also not work:

```
<button onClick={value + 1}>button</button>
```

We have attempted to set the event handler to *value + 1* which simply returns the result of the

operation. React will kindly warn us about this in the console:

```
index.js:2178 Warning: Expected `onClick` listener to be a function, instead got  
a value of `number` type.
```

This attempt would not work either:

```
<button onClick={value = 0}>button</button>
```

The event handler is not a function but a variable assignment, and React will once again issue a warning to the console. This attempt is also flawed in the sense that we must never mutate state directly in React.

What about the following:

```
<button onClick={console.log('clicked the button')}>  
  button  
</button>
```

The message gets printed to the console once when the component is rendered but nothing happens when we click the button. Why does this not work even when our event handler contains a function `console.log`?

The issue here is that our event handler is defined as a *function call* which means that the event handler is assigned the returned value from the function, which in the case of `console.log` is *undefined*.

The `console.log` function call gets executed when the component is rendered and for this reason, it gets printed once to the console.

The following attempt is flawed as well:

```
<button onClick={setValue(0)}>button</button>
```

We have once again tried to set a function call as the event handler. This does not work. This particular attempt also causes another problem. When the component is rendered the function `setValue(0)` gets executed which in turn causes the component to be re-rendered. Re-rendering in turn calls `setValue(0)` again, resulting in an infinite recursion.

Executing a particular function call when the button is clicked can be accomplished like this:

```
<button onClick={() => console.log('clicked the button')}>  
  button  
</button>
```

Now the event handler is a function defined with the arrow function syntax `() => console.log('clicked the button')`. When the component gets rendered, no function gets called and only the reference to the arrow function is set to the event handler. Calling the function happens only once the button is clicked.

We can implement resetting the state in our application with this same technique:

```
<button onClick={() => setValue(0)}>button</button>
```

The event handler is now the function `() => setValue(0)`.

Defining event handlers directly in the attribute of the button is not necessarily the best possible idea.

You will often see event handlers defined in a separate place. In the following version of our application we define a function that then gets assigned to the `handleClick` variable in the body of the component function:

```
const App = () => {
  const [value, setValue] = useState(10)

  const handleClick = () =>
    console.log('clicked the button')

  return (
    <div>
      {value}
      <button onClick={handleClick}>button</button>
    </div>
  )
}
```

The `handleClick` variable is now assigned to a reference to the function. The reference is passed to the button as the `onClick` attribute:

```
<button onClick={handleClick}>button</button>
```

Naturally, our event handler function can be composed of multiple commands. In these cases we use the longer curly brace syntax for arrow functions:

```
const App = () => {
  const [value, setValue] = useState(10)

  const handleClick = () => {
    console.log('clicked the button')
    setValue(0)
  }

  return (
    <div>
      {value}
      <button onClick={handleClick}>button</button>
    </div>
  )
}
```

A function that returns a function

Another way to define an event handler is to use a *function that returns a function*.

You probably won't need to use functions that return functions in any of the exercises in this course. If the topic seems particularly confusing, you may skip over this section for now and return to it later.

Let's make the following changes to our code:

```
const App = () => [
  const [value, setValue] = useState(10)

  const hello = () => {
    const handler = () => console.log('hello world')
    return handler
  }

  return (
    <div>
      {value}
      <button onClick={hello()}>button</button>
    </div>
  )
}
```

The code functions correctly even though it looks complicated.

The event handler is now set to a function call:

```
<button onClick={hello()}>button</button>
```

Earlier on we stated that an event handler may not be a call to a function and that it has to be a function or a reference to a function. Why then does a function call work in this case?

When the component is rendered, the following function gets executed:

```
const hello = () => {
  const handler = () => console.log('hello world')

  return handler
}
```

The *return value* of the function is another function that is assigned to the *handler* variable.

When React renders the line:

```
<button onClick={hello()}>button</button>
```

It assigns the return value of *hello()* to the *onClick* attribute. Essentially the line gets transformed into:

```
<button onClick={() => console.log('hello world')}>
  button
</button>
```

Since the `hello` function returns a function, the event handler is now a function.

What's the point of this concept?

Let's change the code a tiny bit:

```
const App = () => {
  const [value, setValue] = useState(10)

  const hello = (who) => {
    const handler = () => {
      console.log('hello', who)
    }
    return {
      <div>
        {value}
        <button onClick={hello('world')}>button</button>
        <button onClick={hello('react')}>button</button>
        <button onClick={hello('function')}>button</button>
      </div>
    }
  }
}
```

Now the application has three buttons with event handlers defined by the `hello` function that accepts a parameter.

The first button is defined as

```
<button onClick={hello('world')}>button</button>
```

The event handler is created by *executing* the function call `hello('world')`. The function call returns the function:

```
() => {
  console.log('hello', 'world')
}
```

The second button is defined as:

```
<button onClick={hello('react')}>button</button>
```

The function call `hello('react')` that creates the event handler returns:

```
() => {
  console.log('hello', 'react')
}
```

Both buttons get their individualized event handlers.

Functions returning functions can be utilized in defining generic functionality that can be customized with parameters. The `hello` function that creates the event handlers can be thought of as a factory that produces customized event handlers meant for greeting users.

Our current definition is slightly verbose:

```
const hello = (who) => [
  const handler = () => [
    console.log('hello', who)
  ]

  return handler
]
```

Let's eliminate the helper variables and directly return the created function:

```
const hello = (who) => [
  return () => [
    console.log('hello', who)
  ]
]
```

Since our *hello* function is composed of a single return command, we can omit the curly braces and use the more compact syntax for arrow functions:

```
const hello = (who) =>
  () => [
    console.log('hello', who)
  ]

```

Lastly, let's write all of the arrows on the same line:

```
const hello = (who) => () => [
  console.log('hello', who)
]
```

We can use the same trick to define event handlers that set the state of the component to a given value. Let's make the following changes to our code:

```
const App = () => [
  const [value, setValue] = useState(10)

  const setToValue = (newValue) => () => {
    console.log('value now', newValue)
    // print the new value to console
    setValue(newValue)
  }

  return (
    <div>
      {value}
      <button onClick={setToValue(1000)}>thousand</button> <button onClick={setToValue(0)}>reset</button> <button onClick={setToValue(value + 1)}>increment</button>
    </div>
  )
]
```

When the component is rendered, the *thousand* button is created:

```
<button onClick={setToValue(1000)}>thousand</button>
```

The event handler is set to the return value of *setToValue(1000)* which is the following function:

```
() => {
  console.log('value now', 1000)
  setValue(1000)
}
```

The increase button is declared as follows:

```
<button onClick={setToValue(value + 1)}>increment</button>
```

The event handler is created by the function call *setToValue(value + 1)* which receives as its parameter the current value of the state variable *value* increased by one. If the value of *value* was 10, then the created event handler would be the function:

```
() => {
  console.log('value now', 11)
  setValue(11)
}
```

Using functions that return functions is not required to achieve this functionality. Let's return the *setToValue* function which is responsible for updating state into a normal function:

```
const App = () => {
  const [value, setValue] = useState(10)

  const setToValue = (newValue) => {
    console.log('value now', newValue)
    setValue(newValue)
  }

  return (
    <div>
      {value}
      <button onClick={() => setToValue(1000)}>
        thousand
      </button>
      <button onClick={() => setToValue(0)}>
        reset
      </button>
      <button onClick={() => setToValue(value + 1)}>
        increment
      </button>
    </div>
  )
}
```

We can now define the event handler as a function that calls the `setToValue` function with an appropriate parameter. The event handler for resetting the application state would be:

```
<button onClick={() => setToValue(0)}>reset</button>
```

Choosing between the two presented ways of defining your event handlers is mostly a matter of taste.

Passing Event Handlers to Child Components

Let's extract the button into its own component:

```
const Button = (props) => {
  <button onClick={props.handleClick}>
    {props.text}
  </button>
}
```

The component gets the event handler function from the `handleClick` prop, and the text of the button from the `text` prop. Lets use the new component:

```
const App = (props) => [
  // ...
  return [
    <div>
      {value}
      <Button handleClick={() => setToValue(1000)} text="thousand" />      <Button
on handleClick={() => setToValue(0)} text="reset" />      <Button handleClick={() =>
  setToValue(value + 1)} text="increment" />    </div>
    ]
]
```

Using the `Button` component is simple, although we have to make sure that we use the correct attribute names when passing props to the component.

```

import React, {useState} from 'react'
import ReactDOM from 'react-dom'

const Button = (props) => (
  <button onClick={props.handleClick}>{props.text}</button>
)

const App = () => {
  const [value, setValue] = useState(10)

  const setToValue = (newValue) => {
    setValue(newValue)
  }

  return (
    <div>
      {value}
      <Button handleClick={() => setToValue(1000)} text="thousand" />
      <Button handleClick={() => setToValue(0)} text="reset" />
      <Button handleClick={() => setToValue(value + 1)} text="increment" />
    </div>
  )
}

```

Do Not Define Components Within Components

Let's start displaying the value of the application in its *Display* component.

We will change the application by defining a new component inside of the *App* component.

```

// This is the right place to define a component
const Button = (props) => [
  <button onClick={props.handleClick}>
    {props.text}
  </button>
]

const App = () => {
  const [value, setValue] = useState(10)

  const setToValue = newValue => {
    console.log('value now', newValue)
    setValue(newValue)
  }

  // Do not define components inside another component
  const Display = props => <div>{props.value}</div>
  return [
    <div>
      <Display value={value} />
      <Button handleClick={() => setToValue(1000)} text="thousand" />
      <Button handleClick={() => setToValue(0)} text="reset" />
      <Button handleClick={() => setToValue(value + 1)} text="increment" />
    </div>
  ]
}

```

```
}
```

The application still appears to work, but **don't implement components like this!** Never define components inside of other components. The method provides no benefits and leads to many unpleasant problems. The biggest problems are because React treats a component defined inside of another component as a new component in every render. This makes it impossible for React to optimize the component.

Let's instead move the *Display* component function to its correct place, which is outside of the *App* component function:

```
const Display = props => <div>{props.value}</div>

const Button = (props) => [
  <button onClick={props.handleClick}>
    {props.text}
  </button>
]

const App = () => {
  const [value, setValue] = useState(10)

  const setToValue = newValue => {
    console.log('value now', newValue)
    setValue(newValue)
  }

  return (
    <div>
      <Display value={value} />
      <Button handleClick={() => setToValue(1000)} text="thousand" />
      <Button handleClick={() => setToValue(0)} text="reset" />
      <Button handleClick={() => setToValue(value + 1)} text="increment" />
    </div>
  )
}
```

Useful Reading

The internet is full of React-related material. However, we use the new style of React for which a large majority of the material found online is outdated.

You may find the following links useful:

- The [official React documentation](#) is worth checking out at some point, although most of it will become relevant only later on in the course. Also, everything related to class-based components is irrelevant to us;
- Some courses on [Egghead.io](#) like [Start learning React](#) are of high quality, and the recently updated [Beginner's Guide to React](#) is also relatively good; both courses introduce concepts that will also be introduced later on in this course. **NB** The first one uses class components but the latter uses the new functional ones.

Web programmers oath

Programming is hard, that is why I will use all the possible means to make it easier

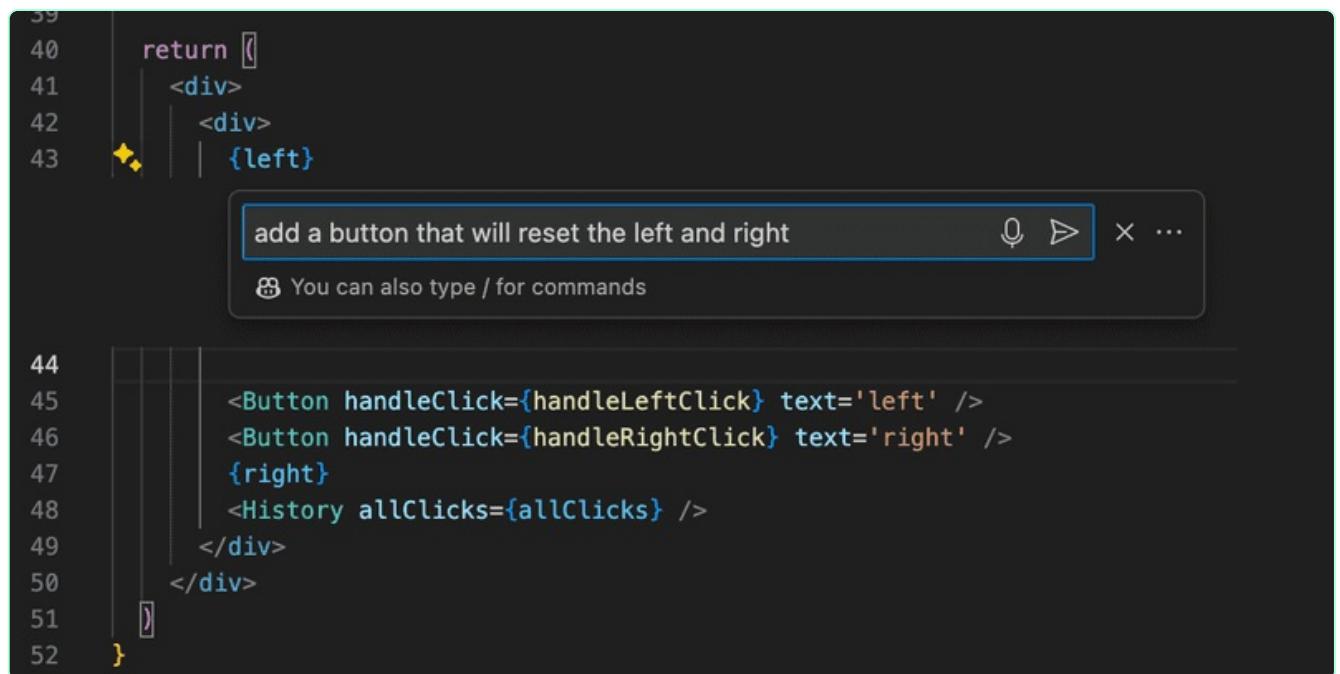
- I will have my browser developer console open all the time
- I progress with small steps
- I will write lots of `console.log` statements to make sure I understand how the code behaves and to help pinpointing problems
- If my code does not work, I will not write more code. Instead I start deleting the 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

Utilization of Large language models

Large language models such as [ChatGPT](#), [Claude](#) and [GitHub Copilot](#) have proven to be very useful in software development.

Personally, I mainly use Copilot, which integrates seamlessly with VS Code thanks to the [plugin](#).

Copilot is useful in a wide variety of scenarios. Copilot can be asked to generate code for an open file by describing the desired functionality in text:



```
39
40     return []
41     <div>
42       <div>
43         {left}
44
45         <Button handleClick={handleLeftClick} text='left' />
46         <Button handleClick={handleRightClick} text='right' />
47         {right}
48         <History allClicks={allClicks} />
49       </div>
50     </div>
51   ]
52 }
```

The screenshot shows a code editor with several lines of JSX code. At line 43, there is a yellow star icon indicating a suggestion. A tooltip box is overlaid on the screen, containing the text "add a button that will reset the left and right" and a "Copilot" logo. Below the tooltip, a message says "You can also type / for commands". The rest of the code continues from line 44, showing closing tags and additional components like a history component.

If the code looks good, Copilot adds it to the file:



```
return (
  <div>
    <div>
      {left}
      <Button handleClick={handleResetClick} text='reset' />
      <Button handleClick={handleLeftClick} text='left' />
      <Button handleClick={handleRightClick} text='right' />
      {right}
    </div>
  </div>
)
```

The screenshot shows the same code editor with the addition of a new button at line 43. The button has the text "reset" and its handleClick prop is set to handleResetClick. The rest of the code remains the same as in the previous screenshot.

In the case of our example, Copilot only created a button, the event handler `handleResetClick` is undefined.

An event handler may also be generated. By writing the first line of the function, Copilot offers the functionality to be generated:

```
◆ const handleResetClick = () => {
  setAll([])
  setLeft(0)
  setRight(0)
}

return (
  <div>
    <div>
      {left}
      <Button handleClick={handleResetClick} text='reset' />
      <Button handleClick={handleLeftClick} text='left' />
      <Button handleClick={handleRightClick} text='right' />
    {right}
    <History allClicks={allClicks} />
  </div>
</div>
```

In Copilot's chat window, it is possible to ask for an explanation of the function of the painted code area:

The screenshot shows the GitHub Copilot interface. On the left is a sidebar with a user profile icon for 'mluukkai', a workspace named '@workspace', and a link to '/explain'. Below this, a message from 'GitHub Copilot' is shown, stating it used '@workspace /explain' and found 1 reference. The main area is a code editor with the following content:

```
src > App.jsx > [4] App
25 const App = () => {
26   const handleLeftClick = () => {
27     setAll(allClicks.concat('L'))
28     setLeft(left + 1)
29   }
30
31   const handleRightClick = () => {
32     setAll(allClicks.concat('R'))
33     setRight(right + 1)
34   }
35
36   const handleResetClick = () => {
37     setAll([])
38     setLeft(0)
39     setRight(0)
40   }
41
42   return (
43     <div>
44       <div>
45         {left}
46         <Button handleClick={handleResetClick} text='reset' />
47         <Button handleClick={handleLeftClick} text='left' />
48         <Button handleClick={handleRightClick} text='right' />
49       {right}
50       <History allClicks={allClicks} />
51     </div>
52   </div>
53 }
54
55
56
57
58
59
60 export default App
```

The code is a React component named 'App'. It contains three event handlers: `handleLeftClick`, `handleRightClick`, and `handleResetClick`. The `handleResetClick` function is currently empty. The component returns a `div` element containing another `div` element. Inside the inner `div`, there are two curly braces {}, which are highlighted in yellow. The explanatory text in the sidebar describes these braces as being replaced by the value of the `left` variable, which is not defined in the provided code snippet but is presumably defined elsewhere in the component. It also describes the `Button` components and their props.

Copilot is also useful in error situations, by copying the error message into Copilot's chat, you will get

an explanation of the problem and a suggested fix:

The screenshot shows a browser developer tools window on the left and a GitHub Copilot chat window on the right.

Developer Tools Console:

```
[plugin:vite:react-babel] /Users/mluukkai/opetus/2024-fs/ai/part1ai/src/App.jsx: Unexpected token
47 |     <div>
  |
  |
42 |     setLeft(0)
43 |     setRight(0)
44 | }
  |
45 |
46 |     return (
  |
  |   at constructor (/Users/mluukkai/opetus/2024-fs/ai/part1ai/src/App.jsx?t=1711019142540 net::ERR_ABORTED 500 (Intended to reload /src/App.jsx. This could be due to syntax errors or incomplete requests.))
```

GitHub Copilot Chat:

CHAT: GITHUB COPILOT

mluukkai [plugin:vite:react-babel] /Users/mluukkai/opetus/2024-fs/ai/part1ai/src/App.jsx: Unexpected token, expected "," (44:2) 47 | /Users/mluukkai/opetus/2024-fs/ai/part1ai/src/App.jsx:44:2 42 | setLeft(0) 43 | setRight(0 44 | }

GitHub Copilot Response:

Used 1 reference >

The error message is indicating that there's a syntax error in your code. It's expecting a comma (",") or a closing parenthesis ("") on line 44, but it found a closing brace ("}") instead.

The issue is with the `setRight(0)` line. It seems like you forgot to close the parenthesis.

Here's the corrected code:

```
const handleResetClick = () => {
  setAll([ ])
  setLeft(0)
  setRight(0)
}
```

This should resolve the syntax error you're encountering.

❖ How can I fix the syntax error in my React code?

Copilot's chat also enables the creation of larger set of functionality



mluukkai

@workspace create a component that makes it possible to log in to the app



GitHub Copilot used @workspace

Used 7 references >

You can create a simple login component using React's useState hook to manage the state of the username and password fields. Here's a basic example:

```
import React, { useState } from 'react';

const Login = () => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const handleLogin = (event) => {
    event.preventDefault();
    // Handle login logic here
    console.log(`Logging in with username: ${username} and password: ${password}`);
  };

  return (
    <div>
      <form onSubmit={handleLogin}>
        <label>
          Username:
          <input type="text" value={username} onChange={(e) => setUsername(e.target.value)} />
        </label>
        <label>
          Password:
          <input type="password" value={password} onChange={(e) => setPassword(e.target.value)} />
        </label>
        <input type="submit" value="Login" />
      </form>
    </div>
  );
}
```

The degree of usefulness of the hints provided by Copilot and other language models varies. Perhaps the biggest problem with language models is [hallucination](#), they sometimes generate completely convincing-looking answers, which, however, are completely wrong. When programming, of course, the hallucinated code is often caught quickly if the code does not work. More problematic situations are those where the code generated by the language model seems to work, but it contains more difficult to detect bugs or e.g. security vulnerabilities.

Another problem in applying language models to software development is that it is difficult for language models to "understand" larger projects, and e.g. to generate functionality that would require changes to several files. Language models are also currently unable to generalize code, i.e. if the code has, for example, existing functions or components that the language model could use with minor changes for the requested functionality, the language model will not bend to this. The result of this can be that the code base deteriorates, as the language models generate a lot of repetition in the code, see more e.g. [here](#).

When using language models, the responsibility always stays with the programmer.

The rapid development of language models puts the student of programming in a challenging position: is it worth and is it even necessary to learn programming in a detailed level, when you can get almost everything ready-made from language models?

At this point, it is worth remembering the old wisdom of [Brian Kerningham](#), the developer of the programming language C:



“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?”

– Brian Kernighan

In other words, since debugging is twice as difficult as programming, it is not worth programming such code that you can only barely understand. How can debugging be even possible in a situation where programming is outsourced to a language model and the software developer does not understand the debugged code at all?

So far, the development of language models and artificial intelligence is still at the stage where they are not self-sufficient, and the most difficult problems are left for humans to solve. Because of this, even novice software developers must learn to program really well just in case. It may be that, despite the development of language models, even more in-depth knowledge is needed. Artificial intelligence does the easy things, but a human is needed to sort out the most complicated messes caused by AI. GitHub Copilot is a very well-named product, it's Copilot, a second pilot who helps the main pilot in an aircraft. The programmer is still the main pilot, the captain and bears the ultimate responsibility.

It may be in your own interest that you turn off Copilot by default when you do this course and rely on it only in a real emergency.

Exercises 1.6.-1.14.

Submit your solutions to the exercises by first pushing your code to GitHub and then marking the completed exercises into the "my submissions" tab of the [submission application](#).

Remember, submit **all** the exercises of one part **in a single submission**. Once you have submitted your solutions for one part, **you cannot submit more exercises to that part anymore**.

Some of the exercises work on the same application. In these cases, it is sufficient to submit just the final version of the application. If you wish, you can make a commit after every finished exercise, but it is not mandatory.

In some situations you may also have to run the command below from the root of the project:

```
rm -rf node_modules/ && npm i
```

If and *when* you encounter an error message

Objects are not valid as a React child

keep in mind the things told [here](#).

1.6: unicafe step 1

Like most companies, the student restaurant of the University of Helsinki [Unicafe](#) collects feedback

from its customers. Your task is to implement a web application for collecting customer feedback. There are only three options for feedback: *good*, *neutral*, and *bad*.

The application must display the total number of collected feedback for each category. Your final application could look like this:

give feedback

good neutral bad

statistics

good 6
neutral 2
bad 1

Note that your application needs to work only during a single browser session. Once you refresh the page, the collected feedback is allowed to disappear.

It is advisable to use the same structure that is used in the material and previous exercise. File *main.jsx* is as follows:

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

You can use the code below as a starting point for the *App.jsx* file:

```
import { useState } from 'react'

const App = () => {
  // save clicks of each button to its own state
  const [good, setGood] = useState(0)
  const [neutral, setNeutral] = useState(0)
  const [bad, setBad] = useState(0)

  return [
    <div>
      code here
    </div>
  ]
}

export default App
```

1.7: unicafe step 2

Expand your application so that it shows more statistics about the gathered feedback: the total number of collected feedback, the average score (good: 1, neutral: 0, bad: -1) and the percentage of positive feedback.



The screenshot shows a web browser window with the URL `localhost:3000`. The page has two main sections: 'give feedback' and 'statistics'. Under 'give feedback', there are three buttons: 'good', 'neutral', and 'bad'. Under 'statistics', the following data is displayed:

Category	Value
good	6
neutral	2
bad	1
all	9
average	0.5555555555555556
positive	66.66666666666667 %

1.8: unicafe step 3

Refactor your application so that displaying the statistics is extracted into its own *Statistics* component. The state of the application should remain in the *App* root component.

Remember that components should not be defined inside other components:

```
// a proper place to define a component
const Statistics = (props) => [
  // ...
]

const App = () => {
  const [good, setGood] = useState(0)
  const [neutral, setNeutral] = useState(0)
  const [bad, setBad] = useState(0)

  // do not define a component within another component
  const Statistics = (props) => [
    // ...
  ]

  return [
    // ...
  ]
}
```

1.9: unicafe step 4

Change your application to display statistics only once feedback has been gathered.

The screenshot shows a web browser window with the URL 'localhost:3000'. The page title is 'give feedback'. Below the title are three buttons labeled 'good', 'neutral', and 'bad'. Underneath these buttons is a section titled 'statistics' which contains the text 'No feedback given'.

1.10: unicafe step 5

Let's continue refactoring the application. Extract the following two components:

- *Button* handles the functionality of each feedback submission button.
- *StatisticLine* for displaying a single statistic, e.g. the average score.

To be clear: the *StatisticLine* component always displays a single statistic, meaning that the application uses multiple components for rendering all of the statistics:

```
const Statistics = (props) => [
  // ...
  return [
    <div>
      <StatisticLine text="good" value={...} />
      <StatisticLine text="neutral" value={...} />
      <StatisticLine text="bad" value={...} />
      // ...
    </div>
  ]
}
```

The application's state should still be kept in the root *App* component.

1.11*: unicafe step 6

Display the statistics in an HTML [table](#), so that your application looks roughly like this:

A screenshot of a web browser window titled "localhost:3000". The main content area has a light blue background. At the top left, there is a heading "give feedback" followed by three buttons: "good", "neutral", and "bad". Below this, the word "statistics" is displayed in bold black font. Under "statistics", there is a table of data:

good	5
neutral	2
bad	1
all	8
average	0.5
positive	62.5 %

Remember to keep your console open at all times. If you see this warning in your console:

A screenshot of the Chrome DevTools Console tab. The title bar shows "Console" is selected. The console output area has a red background and displays the following warning message:

```
in App (at src/index.js:56)
  ⚠ Warning: validateDOMNesting(...): <tr> cannot appear as a child of <table>. Add a <tbody> to your code to match the DOM tree generated by the browser.
    in tr (at src/index.js:5)
    in Statistic (at src/index.js:34)
    in table (at src/index.js:28)
    in div (at src/index.js:26)
    in Statistics (at src/index.js:51)
    in div (at src/index.js:46)
    in App (at src/index.js:56)
```

Then perform the necessary actions to make the warning disappear. Try pasting the error message into a search engine if you get stuck.

Typical source of an error Unchecked runtime.lastError: Could not establish connection. Receiving end does not exist. is from a Chrome extension. Try going to chrome://extensions/ and try disabling them one by one and refreshing React app page; the error should eventually disappear.

Make sure that from now on you don't see any warnings in your console!

1.12*: anecdotes step 1

The world of software engineering is filled with [anecdotes](#) that distill timeless truths from our field into short one-liners.

Expand the following application by adding a button that can be clicked to display a *random* anecdote from the field of software engineering:

```
import { useState } from 'react'

const App = () => [
  const anecdotes = [
    'If it hurts, do it more often.',
    'Adding manpower to a late software project makes it later!',
  ]
```

```

    'The first 90 percent of the code accounts for the first 90 percent of the development time...The remaining 10 percent of the code accounts for the other 90 percent of the development time.',  

    'Any fool can write code that a computer can understand. Good programmers write code that humans can understand.',  

    'Premature optimization is the root of all evil.',  

    'Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.',  

    'Programming without an extremely heavy use of console.log is same as if a doctor would refuse to use x-rays or blood tests when diagnosing patients.',  

    'The only way to go fast, is to go well.'  

]  
  

const [selected, setSelected] = useState(0)  
  

return [  

  <div>  

    {anecdotes[selected]}  

  </div>  

]  

}  
  

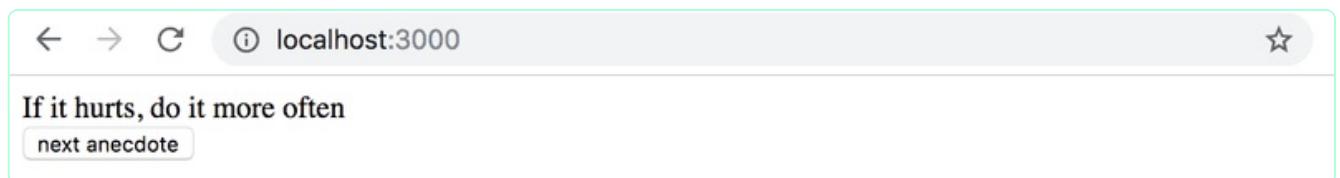
export default App

```

Content of the file `main.jsx` is the same as in previous exercises.

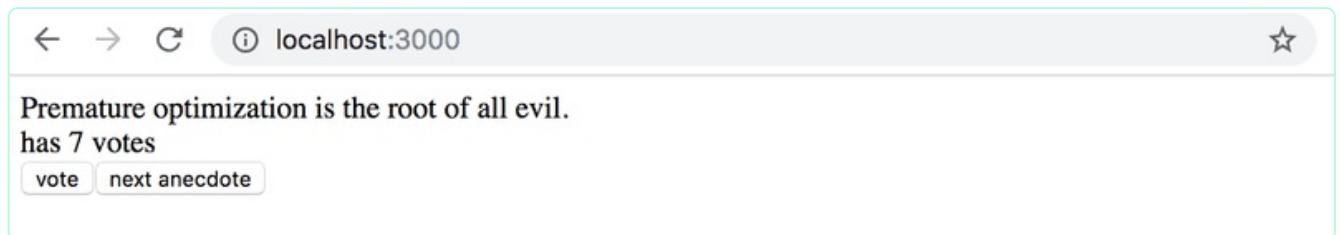
Find out how to generate random numbers in JavaScript, eg. via a search engine or on [Mozilla Developer Network](#). Remember that you can test generating random numbers e.g. straight in the console of your browser.

Your finished application could look something like this:



1.13*: anecdotes step 2

Expand your application so that you can vote for the displayed anecdote.



NB store the votes of each anecdote into an array or object in the component's state. Remember that the correct way of updating state stored in complex data structures like objects and arrays is to make a copy of the state.

You can create a copy of an object like this:

```
const points = { 0: 1, 1: 3, 2: 4, 3: 2 }

const copy = { ...points }
// increment the property 2 value by one
copy[2] += 1
```

OR a copy of an array like this:

```
const points = [1, 4, 6, 3]

const copy = [...points]
// increment the value in position 2 by one
copy[2] += 1
```

Using an array might be the simpler choice in this case. Searching the Internet will provide you with lots of hints on how to [create a zero-filled array of the desired length](#).

1.14*: anecdotes step 3

Now implement the final version of the application that displays the anecdote with the largest number of votes:

The screenshot shows a web browser window with the URL `localhost:3000`. The page contains two main sections:

- Anecdote of the day:** Displays the text "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." It also says "has 0 votes" and has two buttons: "vote" and "next anecdote".
- Anecdote with most votes:** Displays the text "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." It also says "has 7 votes".

If multiple anecdotes are tied for first place it is sufficient to just show one of them.

This was the last exercise for this part of the course and it's time to push your code to GitHub and mark all of your finished exercises to the "my submissions" tab of the [submission application](#).

2: Communicating with server

2a: Rendering a collection, modules

Before starting a new part, let's recap some of the topics that proved difficult last year.

console.log

What's the difference between an experienced JavaScript programmer and a rookie? The experienced one uses `console.log` 10-100 times more.

Paradoxically, this seems to be true even though a rookie programmer would need `console.log` (or any debugging method) more than an experienced one.

When something does not work, don't just guess what's wrong. Instead, log or use some other way of debugging.

NB As explained in part 1, when you use the command `console.log` for debugging, don't concatenate things 'the Java way' with a plus. Instead of writing:

```
console.log('props value is ' + props)
```

separate the things to be printed with a comma:

```
console.log('props value is', props)
```

If you concatenate an object with a string and log it to the console (like in our first example), the result will be pretty useless:

```
props value is [object Object]
```

On the contrary, when you pass objects as distinct arguments separated by commas to `console.log`, like in our second example above, the content of the object is printed to the developer console as strings that are insightful. If necessary, read more about [debugging React applications](#).

Protip: Visual Studio Code snippets

With Visual Studio Code it's easy to create 'snippets', i.e., shortcuts for quickly generating commonly re-used portions of code, much like how 'sout' works in Netbeans.

Instructions for creating snippets can be found [here](#).

Useful, ready-made snippets can also be found as VS Code plugins, in the [marketplace](#).

The most important snippet is the one for the `console.log()` command, for example, `clog`. This can be created like so:

```
{
  "console.log": [
    "prefix": "clog",
    "body": [
      "console.log('$1')",
    ],
    "description": "Log output to console"
  ]
}
```

Debugging your code using `console.log()` is so common that Visual Studio Code has that snippet built

in. To use it, type `log` and hit Tab to autocomplete. More fully featured `console.log()` snippet extensions can be found in the [marketplace](#).

JavaScript Arrays

From here on out, we will be using the functional programming operators of the JavaScript [array](#), such as `find`, `filter`, and `map` - all of the time.

If operating arrays with functional operators feels foreign to you, it is worth watching at least the first three parts of the YouTube video series [Functional Programming in JavaScript](#):

- [Higher-order functions](#)
- [Map](#)
- [Reduce basics](#)

Event Handlers Revisited

Based on last year's course, event handling has proved to be difficult.

It's worth reading the revision chapter at the end of the previous part - [event handlers revisited](#) - if it feels like your own knowledge on the topic needs some brushing up.

Passing event handlers to the child components of the `App` component has raised some questions. A small revision on the topic can be found [here](#).

Rendering Collections

We will now do the 'frontend', or the browser-side application logic, in React for an application that's similar to the example application from [part 0](#)

Let's start with the following (the file `App.jsx`):

```
const App = (props) => {
  const { notes } = props

  return [
    <div>
      <h1>Notes</h1>
      <ul>
        <li>{notes[0].content}</li>
        <li>{notes[1].content}</li>
        <li>{notes[2].content}</li>
      </ul>
    </div>
  ]
}

export default App
```

The file `main.jsx` looks like this:

```
import React from 'react'
```

```

import ReactDOM from 'react-dom/client'

import App from './App'

const 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
  }
]

ReactDOM.createRoot(document.getElementById('root')).render(
  <App notes={notes} />
)

```

Every note contains its textual content, a *boolean* value for marking whether the note has been categorized as important or not, and also a unique *id*.

The example above works due to the fact that there are exactly three notes in the array.

A single note is rendered by accessing the objects in the array by referring to a hard-coded index number:

```
<li>{notes[1].content}</li>
```

This is, of course, not practical. We can improve on this by generating React elements from the array objects using the `map` function.

```
notes.map(note => <li>{note.content}</li>)
```

The result is an array of *li* elements.

```
[<li>HTML is easy</li>,<li>Browser can execute only JavaScript</li>,<li>GET and POST are the most important methods of HTTP protocol</li>]
```

Which can then be placed inside *ul* tags:

```

const App = (props) => [
  const { notes } = props

  return (
    <div>
      <h1>Notes</h1>
      <ul>          {notes.map(note => <li>{note.content}</li>)}      </ul>    </d
iv>
  )
]

```

Because the code generating the `li` tags is JavaScript, it must be wrapped in curly braces in a JSX template just like all other JavaScript code.

We will also make the code more readable by separating the arrow function's declaration across multiple lines:

```

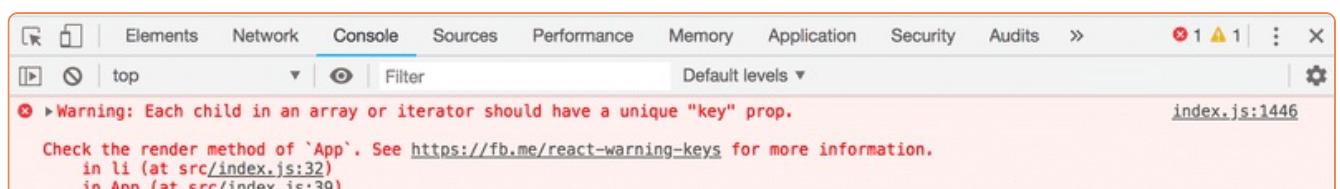
const App = (props) => [
  const { notes } = props

  return (
    <div>
      <h1>Notes</h1>
      <ul>
        {notes.map(note =>
          <li>          {note.content}          </li>        )}
      </ul>
    </div>
  )
]

```

Key-attribute

Even though the application seems to be working, there is a nasty warning in the console:



As the linked [React page](#) in the error message suggests; the list items, i.e. the elements generated by the `map` method, must each have a unique key value: an attribute called `key`.

Let's add the keys:

```

const App = (props) => [
  const { notes } = props

  return (
    <div>

```

```

<h1>Notes</h1>
<ul>
  {notes.map(note =>
    <li key={note.id}>{note.content}</li>
  )}
</ul>
</div>
}
}

```

And the error message disappears.

React uses the key attributes of objects in an array to determine how to update the view generated by a component when the component is re-rendered. More about this is in the [React documentation](#).

Map

Understanding how the array method `map` works is crucial for the rest of the course.

The application contains an array called `notes`:

```

const 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
  }
]

```

Let's pause for a moment and examine how `map` works.

If the following code is added to, let's say, the end of the file:

```

const result = notes.map(note => note.id)
console.log(result)

```

`[1, 2, 3]` will be printed to the console. `map` always creates a new array, the elements of which have been created from the elements of the original array by *mapping*: using the function given as a parameter to the `map` method.

The function is

```
note => note.id
```

Which is an arrow function written in compact form. The full form would be:

```
{note} => {  
  return note.id  
}
```

The function gets a note object as a parameter and *returns* the value of its *id* field.

Changing the command to:

```
const result = notes.map(note => note.content)
```

results in an array containing the contents of the notes.

This is already pretty close to the React code we used:

```
notes.map(note =>  
  <li key={note.id}>  
    {note.content}  
  </li>  
)
```

which generates an *li* tag containing the contents of the note from each note object.

Because the function parameter passed to the *map* method -

```
note => <li key={note.id}>{note.content}</li>
```

- is used to create view elements, the value of the variable must be rendered inside curly braces. Try to see what happens if the braces are removed.

The use of curly braces will cause some pain in the beginning, but you will get used to them soon enough. The visual feedback from React is immediate.

Anti-pattern: Array Indexes as Keys

We could have made the error message on our console disappear by using the array indexes as keys. The indexes can be retrieved by passing a second parameter to the callback function of the *map* method:

```
notes.map((note, i) => ...)
```

When called like this, *i* is assigned the value of the index of the position in the array where the note resides.

As such, one way to define the row generation without getting errors is:

```
<ul>
  {notes.map((note, i) =>
    <li key={i}>
      {note.content}
    </li>
  )}
</ul>
```

This is, however, **not recommended** and can create undesired problems even if it seems to be working just fine.

Read more about this in [this article](#).

Refactoring Modules

Let's tidy the code up a bit. We are only interested in the field *notes* of the props, so let's retrieve that directly using [destructuring](#):

```
const App = ({ notes }) => {
  return (
    <div>
      <h1>Notes</h1>
      <ul>
        {notes.map(note =>
          <li key={note.id}>
            {note.content}
          </li>
        )}
      </ul>
    </div>
  )
}
```

If you have forgotten what destructuring means and how it works, please review the [section on destructuring](#).

We'll separate displaying a single note into its own component *Note*:

```
const Note = ({ note }) => {
  return <li>{note.content}</li>
}

const App = ({ notes }) => {
  return (
    <div>
      <h1>Notes</h1>
      <ul>
        {notes.map(note =>
          <Note key={note.id} note={note} />
        )}
      </ul>
    </div>
  )
}
```

Note that the `key` attribute must now be defined for the `Note` components, and not for the `li` tags like before.

A whole React application can be written in a single file. Although that is, of course, not very practical. Common practice is to declare each component in its own file as an *ES6-module*.

We have been using modules the whole time. The first few lines of the file `main.jsx`:

```
import ReactDOM from "react-dom/client"  
  
import App from "./App"
```

`import` two modules, enabling them to be used in that file. The module `react-dom/client` is placed into the variable `ReactDOM`, and the module that defines the main component of the app is placed into the variable `App`

Let's move our `Note` component into its own module.

In smaller applications, components are usually placed in a directory called `components`, which is in turn placed within the `src` directory. The convention is to name the file after the component.

Now, we'll create a directory called `components` for our application and place a file named `Note.jsx` inside. The contents of the file are as follows:

```
const Note = ({ note }) => {  
  return [  
    <li>{note.content}</li>  
  ]  
}  
  
export default Note
```

The last line of the module `exports` the declared module, the variable `Note`.

Now the file that is using the component - `App.jsx` - can `import` the module:

```
import Note from './components/Note'  
const App = ({ notes }) => {  
  // ...  
}
```

The component exported by the module is now available for use in the variable `Note`, just as it was earlier.

Note that when importing our own components, their location must be given *in relation to the importing file*:

```
'./components/Note'
```

The period `- . -` in the beginning refers to the current directory, so the module's location is a file called

Note.jsx in the *components* sub-directory of the current directory. The filename extension *.jsx* can be omitted.

Modules have plenty of other uses other than enabling component declarations to be separated into their own files. We will get back to them later in this course.

The current code of the application can be found on [GitHub](#).

Note that the *main* branch of the repository contains the code for a later version of the application. The current code is in the branch [part2-1](#):

The screenshot shows a GitHub repository page for the branch 'part2-1'. At the top, there are navigation links for Code, Issues, Pull requests, Projects, Wiki, Security, Insights, and Settings. Below the header, it says 'No description, website, or topics provided.' and has a 'Manage topics' button. A summary bar shows 3 commits, 2 branches, 0 releases, and 1 contributor. A dropdown menu indicates the branch is 'part2-1'. There are buttons for View #1, Create new file, Upload files, Find File, and Clone or download. On the left, a sidebar shows a 'Switch branches/tags' dropdown with 'master' and 'part2-1' selected. It also lists 'package-lock.json' and 'package.json'. The main area displays commit history:

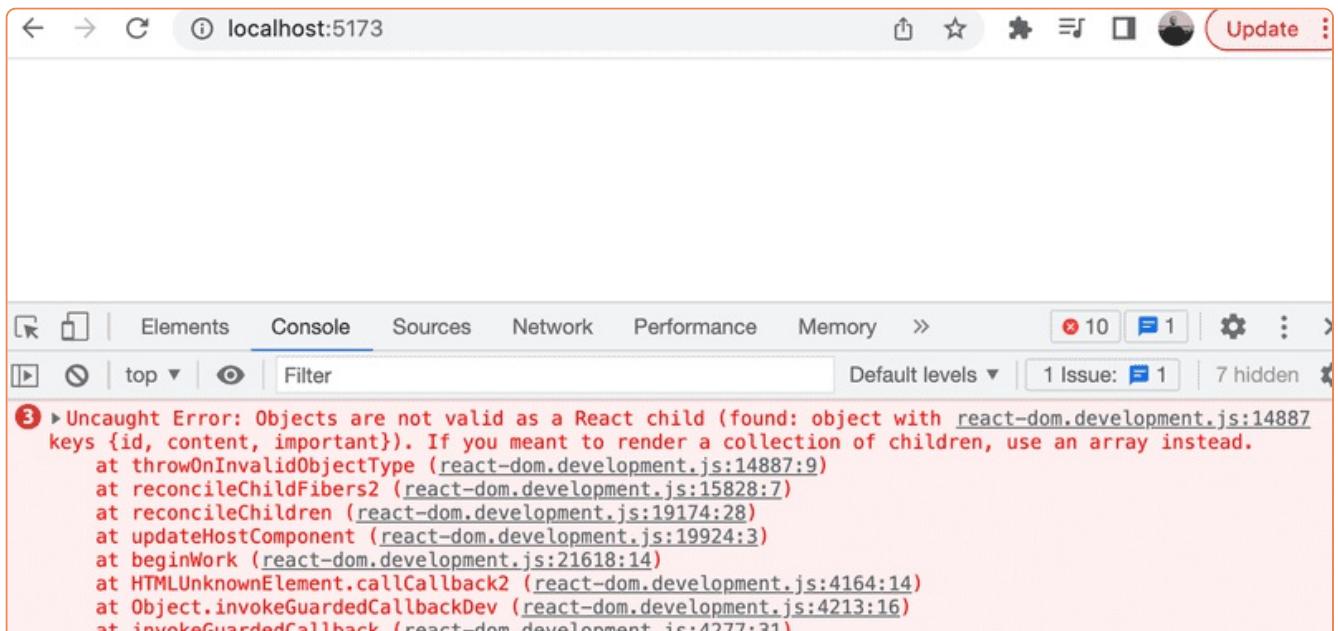
Commit	Time Ago
Initial commit from Create React App	9 minutes ago
part2-1	3 minutes ago
Initial commit from Create React App	9 minutes ago
Initial commit from Create React App	9 minutes ago
Initial commit from Create React App	9 minutes ago

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

When the Application Breaks

Early in your programming career (and even after 30 years of coding like yours truly), what often happens is that the application just completely breaks down. This is even more so the case with dynamically typed languages, such as JavaScript, where the compiler does not check the data type. For instance, function variables or return values.

A "React explosion" can, for example, look like this:



In these situations, your best way out is the `console.log` method.

The piece of code causing the explosion is this:

```
const Course = ({ course }) => (
  <div>
    <Header course={course} />
  </div>
)

const App = () => {
  const course = [
    // ...
  ]

  return [
    <div>
      <Course course={course} />
    </div>
  ]
}
```

We'll hone in on the reason for the breakdown by adding `console.log` commands to the code. Because the first thing to be rendered is the `App` component, it's worth putting the first `console.log` there:

```
const App = () => {
  const course = [
    // ...
  ]

  console.log('App works...')
  return [
    // ...
  ]
}
```



To see the printing in the console, we must scroll up over the long red wall of errors.

```
Open your browser's developer console to further inspect this error.

Elements Network Console Sources Performance Memory Application Security Audits > ✖ 3 ⚠ 1
top Filter Default levels ▾

App toimii... index.

✖ > Uncaught Error: Objects are not valid as a React child (found: object with keys {name, parts}). If you meant to render a collection of children, use an array instead.
    in h1 (at src/index.js:5)
    in Header (at src/index.js:28)
    in div (at src/index.js:27)
    in Course (at src/index.js:56)
    in div (at src/index.js:55)
    in App (at src/index.js:62)
    at invariant (react-dom.development.js:57)
    at throwOnInvalidObjectType (react-dom.development.js:13582)
    at createChild (react-dom.development.js:13809)
    at reconcileChildrenArray (react-dom.development.js:14055)
```

When one thing is found to be working, it's time to log deeper. If the component has been declared as a single statement or a function without a return, it makes printing to the console harder.

```
const Course = ([course]) => [
  <div>
    <Header course={course} />
  </div>
]
```

The component should be changed to its longer form for us to add the printing:

```
const Course = ([course]) => {
  console.log(course)  return [
    <div>
      <Header course={course} />
    </div>
  ]
}
```

Quite often the root of the problem is that the props are expected to be of a different type, or called with a different name than they actually have, and destructuring fails as a result. The problem often begins to solve itself when destructuring is removed and we see what the *props* contain.

```
const Course = [props] => {
  console.log(props)  const { course } = props
  return [
    <div>
      <Header course={course} />
    </div>
  ]
}
```

If the problem has still not been resolved, sadly there isn't much to do apart from continuing to bug-hunt by sprinkling more *console.log* statements around your code.

I added this chapter to the material after the model answer for the next question exploded completely

(due to props being of the wrong type), and I had to debug it using `console.log`.

Web developer's oath

Before the exercises, let me remind what you promised at the end of the previous part.

Programming is 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 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 2.1.-2.5.

The exercises are submitted via GitHub, and by marking the exercises as done in the [submission system](#).

You can submit all of the exercises into the same repository, or use multiple different repositories. If you submit exercises from different parts into the same repository, name your directories well.

The exercises are submitted **One part at a time**. When you have submitted the exercises for a part, you can no longer submit any missed exercises for that part.

Note that this part has more exercises than the ones before, so *do not submit* until you have done all the exercises from this part you want to submit.

2.1: Course information step6

Let's finish the code for rendering course contents from exercises 1.1 - 1.5. You can start with the code from the model answers. The model answers for part 1 can be found by going to the [submission system](#), clicking on *my submissions* at the top, and in the row corresponding to part 1 under the *solutions* column clicking on *show*. To see the solution to the *course info* exercise, click on *index.js* under *kurssitiedot* ("kurssitiedot" means "course info").

Note that if you copy a project from one place to another, you might have to delete the `node_modules` directory and install the dependencies again with the command `npm install` before you can start the application.

Generally, it's not recommended that you copy a project's whole contents and/or add the `node_modules` directory to the version control system.

Let's change the *App* component like so:

```
const App = () => [
  const course = [
    id: 1,
    name: 'Half Stack application development',
    parts: [
```

```

[{
  name: 'Fundamentals of React',
  exercises: 10,
  id: 1
},
{
  name: 'Using props to pass data',
  exercises: 7,
  id: 2
},
{
  name: 'State of a component',
  exercises: 14,
  id: 3
}]

return <Course course={course} />
}

export default App

```

Define a component responsible for formatting a single course called *Course*.

The component structure of the application can be, for example, the following:

```

App
  Course
    Header
    Content
      Part
      Part
      ...

```

Hence, the *Course* component contains the components defined in the previous part, which are responsible for rendering the course name and its parts.

The rendered page can, for example, look as follows:

A screenshot of a web browser window. The address bar shows 'localhost:3000'. The main content area displays the heading 'Half Stack application development' in large bold letters. Below it, there is a list of course names and their exercise counts:

- Fundamentals of React 10
- Using props to pass data 7
- State of a component 14

You don't need the sum of the exercises yet.

The application must work *regardless of the number of parts a course has*, so make sure the application works if you add or remove parts of a course.

Ensure that the console shows no errors!

2.2: Course information step7

Show also the sum of the exercises of the course.

Fundamentals of React 10
Using props to pass data 7
State of a component 14
Redux 11
total of 42 exercises

2.3*: Course information step8

If you haven't done so already, calculate the sum of exercises with the array method [reduce](#).

Pro tip: when your code looks as follows:

```
const total =  
  parts.reduce((s, p) => someMagicHere)
```

and does not work, it's worth it to use `console.log`, which requires the arrow function to be written in its longer form:

```
const total = parts.reduce((s, p) => [  
  console.log('what is happening', s, p),  
  return someMagicHere  
])
```

Not working? : Use your search engine to look up how `reduce` is used in an [Object Array](#).

2.4: Course information step9

Let's extend our application to allow for an *arbitrary number* of courses:

```
const App = () => {  
  const courses = [  
    {  
      name: 'Half Stack application development',  
      id: 1,  
    },  
    {  
      name: 'Half Stack application development',  
      id: 2,  
    },  
  ]  
  return (  
    <ul>  
      {courses.map(course => (  
        <li>{course.name}</li>  
      ))}  
    </ul>  
  )  
}
```

```

parts: [
  [
    {
      name: 'Fundamentals of React',
      exercises: 10,
      id: 1
    },
    [
      {
        name: 'Using props to pass data',
        exercises: 7,
        id: 2
      },
      [
        {
          name: 'State of a component',
          exercises: 14,
          id: 3
        },
        [
          {
            name: 'Redux',
            exercises: 11,
            id: 4
          }
        ]
      ]
    ],
    [
      {
        name: 'Node.js',
        id: 2,
        parts: [
          [
            {
              name: 'Routing',
              exercises: 3,
              id: 1
            },
            [
              {
                name: 'Middlewares',
                exercises: 7,
                id: 2
              }
            ]
          ]
        ]
      }
    ]
  ]
}

return [
  <div>
    // ...
  </div>
]
}

```

The application can, for example, look like this:

Web development curriculum

Half Stack application development

Fundamentals of React 10

Using props to pass data 7

State of a component 14

Redux 11

total of 42 exercises

Node.js

Routing 3

Middlewares 7

total of 10 exercises

2.5: Separate module step10

Declare the *Course* component as a separate module, which is imported by the *App* component. You can include all subcomponents of the course in the same module.

2b: Forms

Let's continue expanding our application by allowing users to add new notes. You can find the code for our current application [here](#).

Saving the notes in the component state

To get our page to update when new notes are added it's best to store the notes in the *App* component's state. Let's import the `useState` function and use it to define a piece of state that gets initialized with the initial notes array passed in the props.

```
import { useState } from 'react'
import Note from './components>Note'

const App = [props] => [
  const [notes, setNotes] = useState(props.notes)
  return [
    <div>
      <h1>Notes</h1>
      <ul>
        {notes.map(note =>
          <Note key={note.id} note={note} />
        )}
      </ul>
    </div>
  ]
]
```

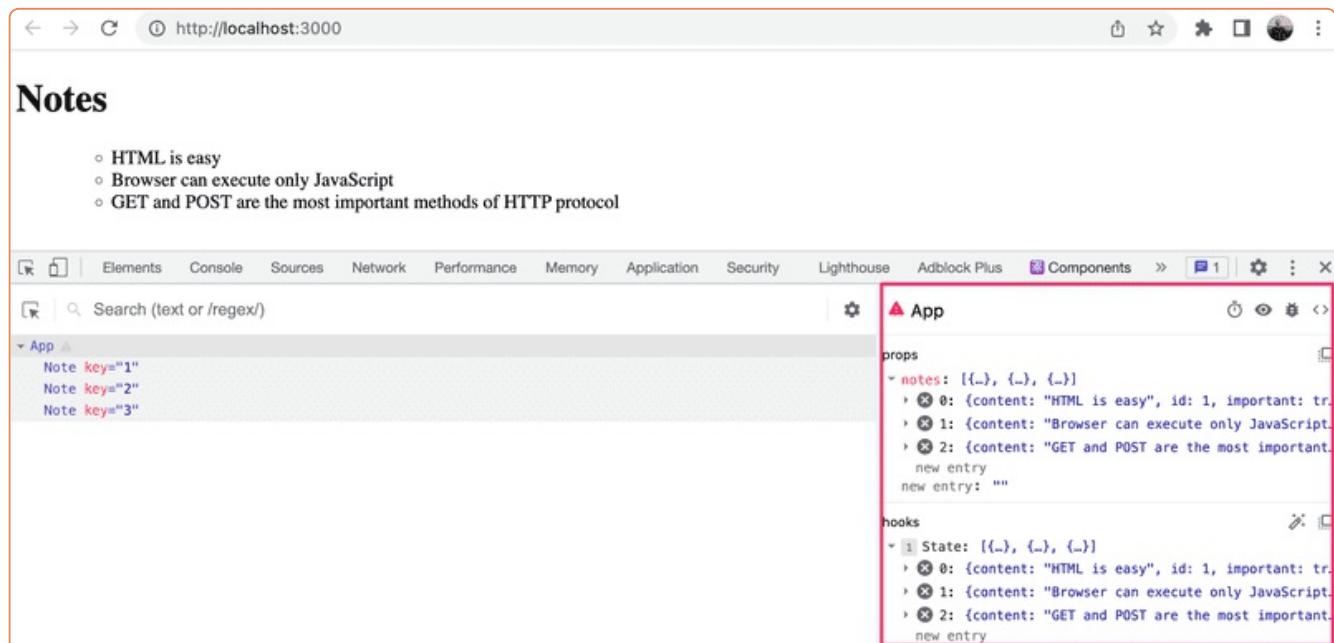
```
export default App
```

The component uses the `useState` function to initialize the piece of state stored in `notes` with the array of notes passed in the props:

```
const App = [props] => [
  const [notes, setNotes] = useState(props.notes)

  // ...
]
```

We can also use React Developer Tools to see that this really happens:



If we wanted to start with an empty list of notes, we would set the initial value as an empty array, and since the props would not be used, we could omit the `props` parameter from the function definition:

```
const App = () => [
  const [notes, setNotes] = useState([])

  // ...
]
```

Let's stick with the initial value passed in the props for the time being.

Next, let's add an HTML `form` to the component that will be used for adding new notes.

```
const App = [props] => [
  const [notes, setNotes] = useState(props.notes)

  const addNote = [event] => {
    event.preventDefault()
    console.log('button clicked', event.target)
  }
  return [
    <form>
      <input type="text" />
      <button type="button" onClick={addNote}>Add Note</button>
    </form>
    ...
  ]
]
```

```

<div>
  <h1>Notes</h1>
  <ul>
    {notes.map(note =>
      <Note key={note.id} note={note} />
    )}
  </ul>
  <form onSubmit={addNote}>          <input />          <button type="submit">sa
ve</button>        </form>        </div>
}

```

We have added the `addNote` function as an event handler to the form element that will be called when the form is submitted, by clicking the submit button.

We use the method discussed in [part 1](#) for defining our event handler:

```

const addNote = (event) => {
  event.preventDefault()
  console.log('button clicked', event.target)
}

```

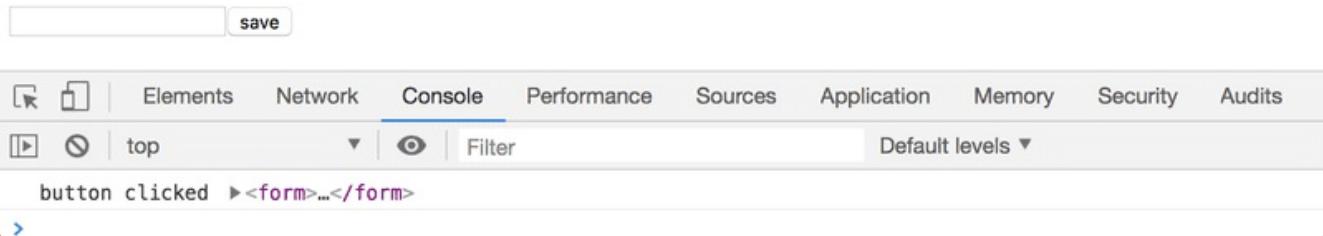
The `event` parameter is the `event` that triggers the call to the event handler function:

The event handler immediately calls the `event.preventDefault()` method, which prevents the default action of submitting a form. The default action would, [among other things](#), cause the page to reload.

The target of the event stored in `event.target` is logged to the console:

Notes

- HTML is easy
- Browser can execute only Javascript
- GET and POST are the most important methods of HTTP protocol



The target in this case is the form that we have defined in our component.

How do we access the data contained in the form's `input` element?

Controlled component

There are many ways to accomplish this; the first method we will take a look at is through the use of so-called [controlled components](#).

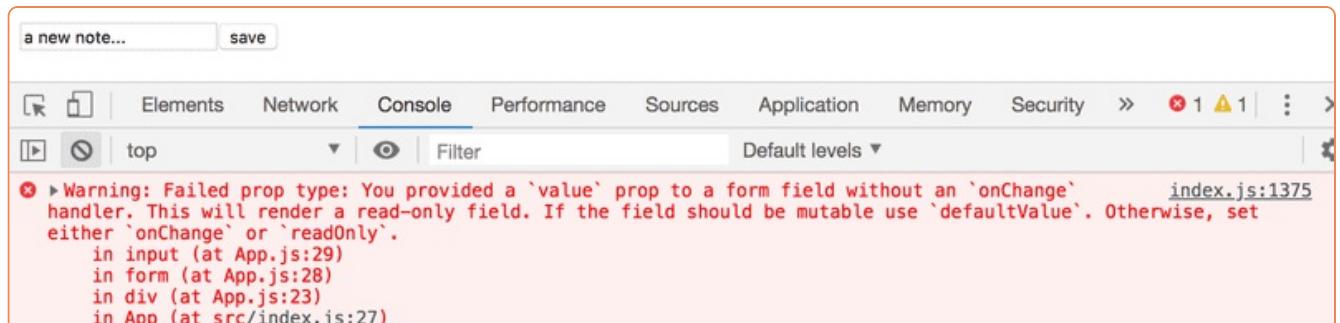
Let's add a new piece of state called `newNote` for storing the user-submitted input **and** let's set it as

the *input* element's *value* attribute:

```
const App = (props) => [
  const [notes, setNotes] = useState(props.notes)
  const [newNote, setNewNote] = useState('a new note...')
  const addNote = (event) => {
    event.preventDefault()
    console.log('button clicked', event.target)
  }

  return [
    <div>
      <h1>Notes</h1>
      <ul>
        {notes.map(note =>
          <Note key={note.id} note={note} />
        )}
      </ul>
      <form onSubmit={addNote}>
        <input value={newNote} />           <button type="submit">save</button>
      </form>
    </div>
  ]
]
```

The placeholder text stored as the initial value of the *newNote* state appears in the *input* element, but the input text can't be edited. The console displays a warning that gives us a clue as to what might be wrong:



Since we assigned a piece of the *App* component's state as the *value* attribute of the *input* element, the *App* component now **controls** the behavior of the *input* element.

To enable editing of the *input* element, we have to register an *event handler* that synchronizes the changes made to the *input* with the component's state:

```
const App = (props) => [
  const [notes, setNotes] = useState(props.notes)
  const [newNote, setNewNote] = useState(
    'a new note...'
  )

  // ...
]
```

```

    const handleNoteChange = (event) => {
      console.log(event.target.value)
      setNewNote(event.target.value)
    }
    return (
      <div>
        <h1>Notes</h1>
        <ul>
          {notes.map(note =>
            <Note key={note.id} note={note} />
          )}
        </ul>
        <form onSubmit={addNote}>
          <input
            value={newNote}
            onChange={handleNoteChange}
          />
          <button type="submit">save</button>
        </form>
      </div>
    )
  }
}

```

We have now registered an event handler to the *onChange* attribute of the form's *input* element:

```

<input
  value={newNote}
  onChange={handleNoteChange}
/>

```

The event handler is called every time *a change occurs in the input element*. The event handler function receives the event object as its *event* parameter:

```

const handleNoteChange = (event) => {
  console.log(event.target.value)
  setNewNote(event.target.value)
}

```

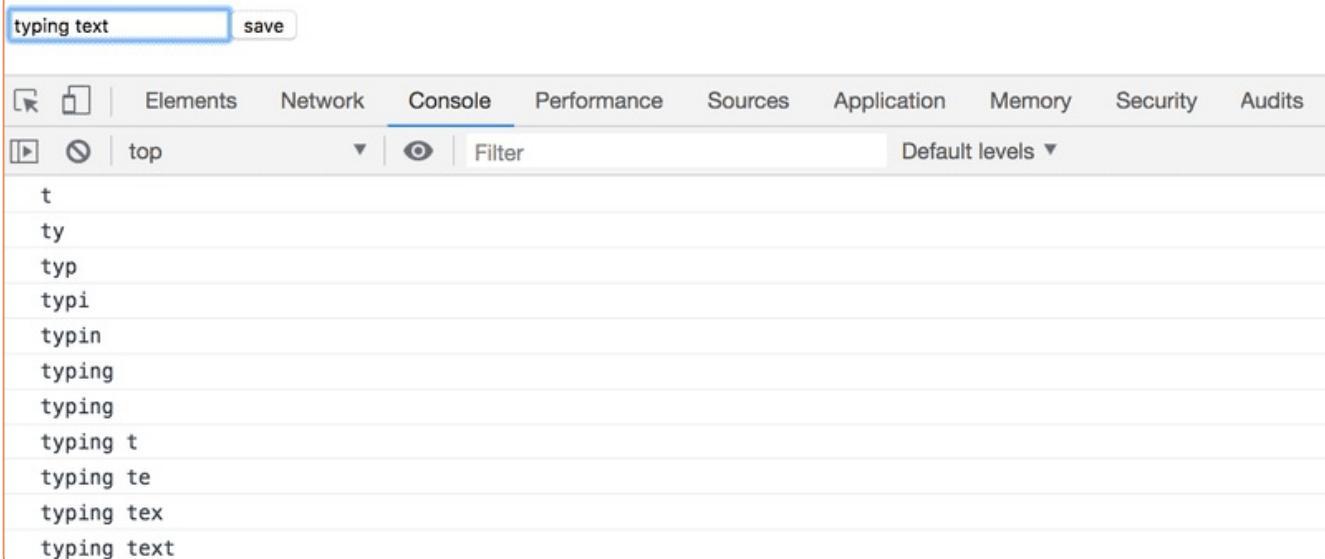
The *target* property of the event object now corresponds to the controlled *input* element, and *event.target.value* refers to the input value of that element.

Note that we did not need to call the *event.preventDefault()* method like we did in the *onSubmit* event handler. This is because no default action occurs on an input change, unlike a form submission.

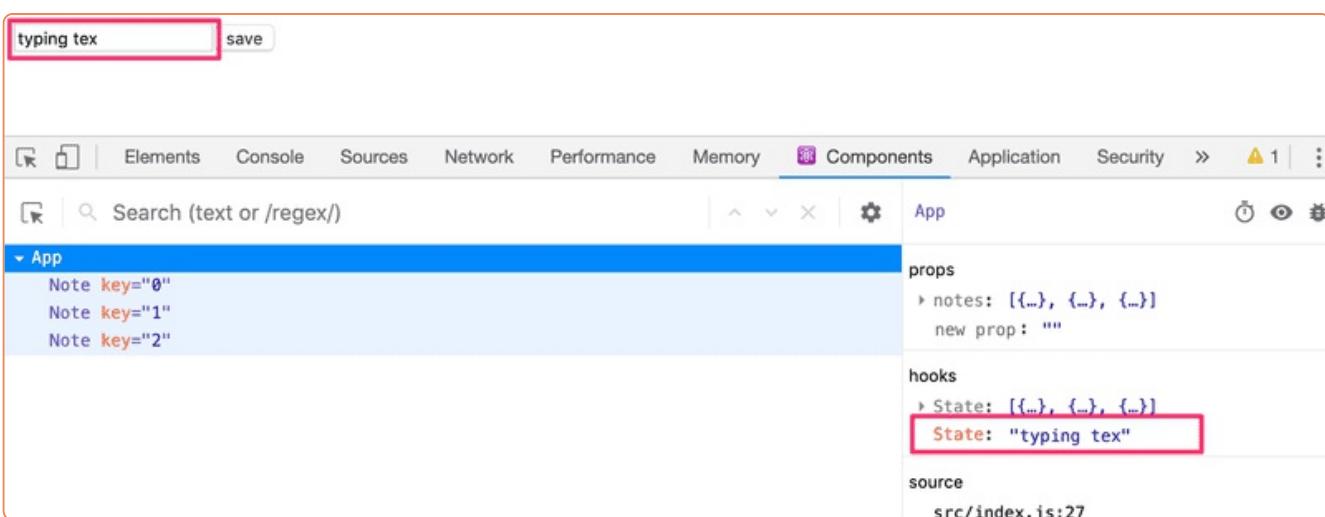
You can follow along in the console to see how the event handler is called:

Notes

- HTML is easy
- Browser can execute only Javascript
- GET and POST are the most important methods of HTTP protocol



You did remember to install [React devtools](#), right? Good. You can directly view how the state changes from the React Devtools tab:



Now the *App* component's *newNote* state reflects the current value of the input, which means that we can complete the *addNote* function for creating new notes:

```
const addNote = (event) => {
  event.preventDefault()
  const noteObject = {
    content: newNote,
    important: Math.random() < 0.5,
    id: notes.length + 1,
  }

  setNotes(notes.concat(noteObject))
  setNewNote('')
}
```

First, we create a new object for the note called `noteObject` that will receive its content from the component's `newNote` state. The unique identifier `id` is generated based on the total number of notes. This method works for our application since notes are never deleted. With the help of the `Math.random()` function, our note has a 50% chance of being marked as important.

The new note is added to the list of notes using the `concat` array method, introduced in [part 1](#):

```
setNotes(notes.concat(noteObject))
```

The method does not mutate the original `notes` array, but rather creates *a new copy of the array with the new item added to the end*. This is important since we must [never mutate state directly](#) in React!

The event handler also resets the value of the controlled input element by calling the `setNewNote` function of the `newNote` state:

```
setNewNote('')
```

You can find the code for our current application in its entirety in the `part2-2` branch of [this GitHub repository](#).

Filtering Displayed Elements

Let's add some new functionality to our application that allows us to only view the important notes.

Let's add a piece of state to the `App` component that keeps track of which notes should be displayed:

```
const App = [props] => [
  const [notes, setNotes] = useState(props.notes)
  const [newNote, setNewNote] = useState('')
  const [showAll, setShowAll] = useState(true)
  // ...
]
```

Let's change the component so that it stores a list of all the notes to be displayed in the `notesToShow` variable. The items on the list depend on the state of the component:

```
import { useState } from 'react'
import Note from './components>Note'

const App = [props] => [
  const [notes, setNotes] = useState(props.notes)
  const [newNote, setNewNote] = useState('')
  const [showAll, setShowAll] = useState(true)

  // ...

  const notesToShow = showAll ? notes : notes.filter(note => note.importan
t === true)
  return [
    // ...
  ]
]
```

```

<div>
  <h1>Notes</h1>
  <ul>
    {notesToShow.map(note => <Note key={note.id} note={note} />)
  }
</ul>
// ...
</div>
}

```

The definition of the `notesToShow` variable is rather compact:

```

const notesToShow = showAll
  ? notes
  : notes.filter(note => note.important === true)

```

The definition uses the [conditional](#) operator also found in many other programming languages.

The operator functions as follows. If we have:

```
const result = condition ? val1 : val2
```

the `result` variable will be set to the value of `val1` if `condition` is true. If `condition` is false, the `result` variable will be set to the value of `val2`.

If the value of `showAll` is false, the `notesToShow` variable will be assigned to a list that only contains notes that have the `important` property set to true. Filtering is done with the help of the array [filter](#) method:

```
notes.filter(note => note.important === true)
```

The comparison operator is redundant, since the value of `note.important` is either `true` or `false`, which means that we can simply write:

```
notes.filter(note => note.important)
```

We showed the comparison operator first to emphasize an important detail: in JavaScript `val1 == val2` does not always work as expected. When performing comparisons, it's therefore safer to exclusively use `val1 === val2`. You can read more about the topic [here](#).

You can test out the filtering functionality by changing the initial value of the `showAll` state.

Next, let's add functionality that enables users to toggle the `showAll` state of the application from the user interface.

The relevant changes are shown below:

```

import { useState } from 'react'
import Note from './components/Note'

```

```

const App = (props) => [
  const [notes, setNotes] = useState(props.notes)
  const [newNote, setNewNote] = useState('')
  const [showAll, setShowAll] = useState(true)

  // ...

  return [
    <div>
      <h1>Notes</h1>
      <div>          <button onClick={() => setShowAll(!showAll)}>{showAll ? 'important' : 'all'}</button>      </div>      <ul>
        {notesToShow.map(note =>
          <Note key={note.id} note={note} />
        )}
      </ul>
      // ...
    </div>
  ]
]
}

```

The displayed notes (all versus important) are controlled with a button. The event handler for the button is so simple that it has been defined directly in the attribute of the button element. The event handler switches the value of *showAll* from true to false and vice versa:

```
() => setShowAll(!showAll)
```

The text of the button depends on the value of the *showAll* state:

```
show [showAll ? 'important' : 'all']
```

You can find the code for our current application in its entirety in the *part2-3* branch of [this GitHub repository](#).

Exercises 2.6.-2.10.

In the first exercise, we will start working on an application that will be further developed in the later exercises. In related sets of exercises, it is sufficient to return the final version of your application. You may also make a separate commit after you have finished each part of the exercise set, but doing so is not required.

2.6: The Phonebook Step 1

Let's create a simple phonebook. ***In this part, we will only be adding names to the phonebook.***

Let us start by implementing the addition of a person to the phonebook.

You can use the code below as a starting point for the *App* component of your application:

```

import { useState } from 'react'

const App = () => {
  const [persons, setPersons] = useState([
    { name: 'Arto Hellas' }
  ])
  const [newName, set newName] = useState('')

  return [
    <div>
      <h2>Phonebook</h2>
      <form>
        <div>
          name: <input />
        </div>
        <div>
          <button type="submit">add</button>
        </div>
      </form>
      <h2>Numbers</h2>
      ...
    </div>
  ]
}

export default App

```

The `newName` state is meant for controlling the form input element.

Sometimes it can be useful to render state and other variables as text for debugging purposes. You can temporarily add the following element to the rendered component:

```
<div>debug: {newName}</div>
```

It's also important to put what we learned in the [debugging React applications](#) chapter of part one into good use. The [React developer tools](#) extension is *incredibly* useful for tracking changes that occur in the application's state.

After finishing this exercise your application should look something like this:

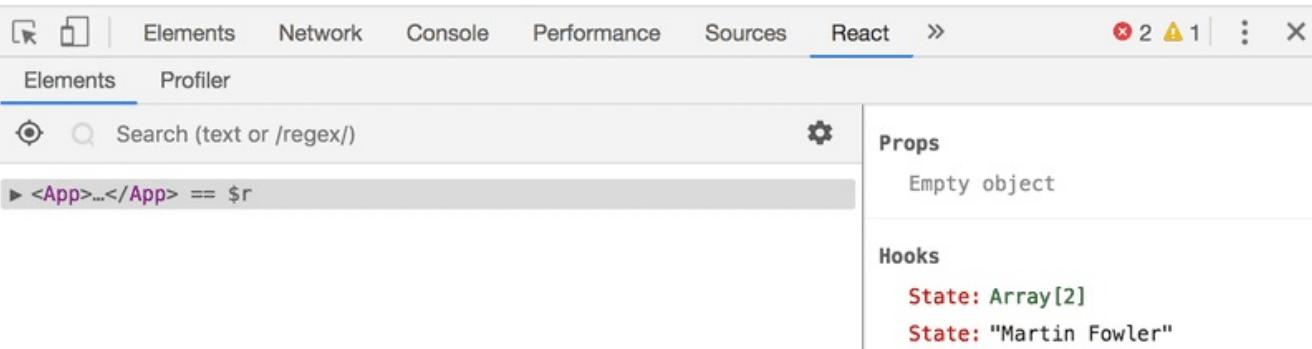
Phonebook

name:

add

Numbers

Arto Hellas
Ada Lovelace



Note the use of the React developer tools extension in the picture above!

NB:

- you can use the person's name as a value of the *key* property
- remember to prevent the default action of submitting HTML forms!

2.7: The Phonebook Step 2

Prevent the user from being able to add names that already exist in the phonebook. JavaScript arrays have numerous suitable [methods](#) for accomplishing this task. Keep in mind [how object equality works](#) in Javascript.

Issue a warning with the `alert` command when such an action is attempted:

The screenshot shows a browser window with the URL 'localhost:3000/'. A modal dialog box is open, displaying the message 'localhost:3000 says' followed by 'Arto Hellas is already added to phonebook'. An 'OK' button is at the bottom right of the dialog. The main content area shows a 'Phonebook' section with a form where 'name: Arto Hellas' is entered, and an 'add' button. Below it is a 'Numbers' section listing 'Arto Hellas'.

Hint: when you are forming strings that contain values from variables, it is recommended to use a [template string](#):

```
'${newName} is already added to phonebook'
```

If the `newName` variable holds the value `Arto Hellas`, the template string expression returns the string

```
`Arto Hellas is already added to phonebook`
```

The same could be done in a more Java-like fashion by using the plus operator:

```
newName + ' is already added to phonebook'
```

Using template strings is the more idiomatic option and the sign of a true JavaScript professional.

2.8: The Phonebook Step 3

Expand your application by allowing users to add phone numbers to the phone book. You will need to add a second `input` element to the form (along with its own event handler):

```
<form>
  <div>name: <input /></div>
  <div>number: <input /></div>
  <div><button type="submit">add</button></div>
</form>
```

At this point, the application could look something like this. The image also displays the application's state with the help of [React developer tools](#):

Phonebook

name: Ada Lovelace
number: 39-44-5323523
add

Numbers

Arto Hellas 040-1234567

React

Props

Empty object

Hooks

State: Array[1]
State: "Ada Lovelace"
State: "Arto Hellas"
State: "39-44-5323523"

2.9*: The Phonebook Step 4

Implement a search field that can be used to filter the list of people by name:

Phonebook

filter shown with

add a new

name:
number:

Numbers

Arto Hellas 040-123456
Ada Lovelace 39-44-5323523

You can implement the search field as an `input` element that is placed outside the HTML form. The filtering logic shown in the image is *case insensitive*, meaning that the search term `arto` also returns results that contain Arto with an uppercase A.

NB: When you are working on new functionality, it's often useful to "hardcode" some dummy data into your application, e.g.

```
const App = () => [
  const [persons, setPersons] = useState([
    { name: 'Arto Hellas', number: '040-123456', id: 1 },
    { name: 'Ada Lovelace', number: '39-44-5323523', id: 2 },
    { name: 'Dan Abramov', number: '12-43-234345', id: 3 },
    { name: 'Mary Poppendieck', number: '39-23-6423122', id: 4 }
  ])
  // ...
]
```

This saves you from having to manually input data into your application for testing out your new functionality.

2.10: The Phonebook Step 5

If you have implemented your application in a single component, refactor it by extracting suitable parts into new components. Maintain the application's state and all event handlers in the `App` root component.

It is sufficient to extract **three** components from the application. Good candidates for separate components are, for example, the search filter, the form for adding new people to the phonebook, a component that renders all people from the phonebook, and a component that renders a single person's details.

The application's root component could look similar to this after the refactoring. The refactored root component below only renders titles and lets the extracted components take care of the rest.

```

const App = () => {
  // ...

  return (
    <div>
      <h2>Phonebook</h2>

      <Filter ... />

      <h3>Add a new</h3>

      <PersonForm
        ...
      />

      <h3>Numbers</h3>

      <Persons ... />
    </div>
  )
}

```

NB: You might run into problems in this exercise if you define your components "in the wrong place". Now would be a good time to rehearse the chapter [do not define a component in another component](#) from the last part.

2c: Getting data from server

For a while now we have only been working on "frontend", i.e. client-side (browser) functionality. We will begin working on "backend", i.e. server-side functionality in the [third part](#) of this course. Nonetheless, we will now take a step in that direction by familiarizing ourselves with how the code executing in the browser communicates with the backend.

Let's use a tool meant to be used during software development called [JSON Server](#) to act as our server.

Create a file named *db.json* in the root directory of the previous *notes* project with the following content:

```

{
  "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  
  },  
  {  
    "id": 1,  
    "content": "HTML is easy",  
    "important": true  
  },  
  {  
    "id": 2,  
    "content": "Browser can execute only JavaScript",  
    "important": false  
  }]  
]
```

You can [install](#) a JSON server globally on your machine using the command `npm install -g json-server`. A global installation requires administrative privileges, which means that it is not possible on faculty computers or freshman laptops.

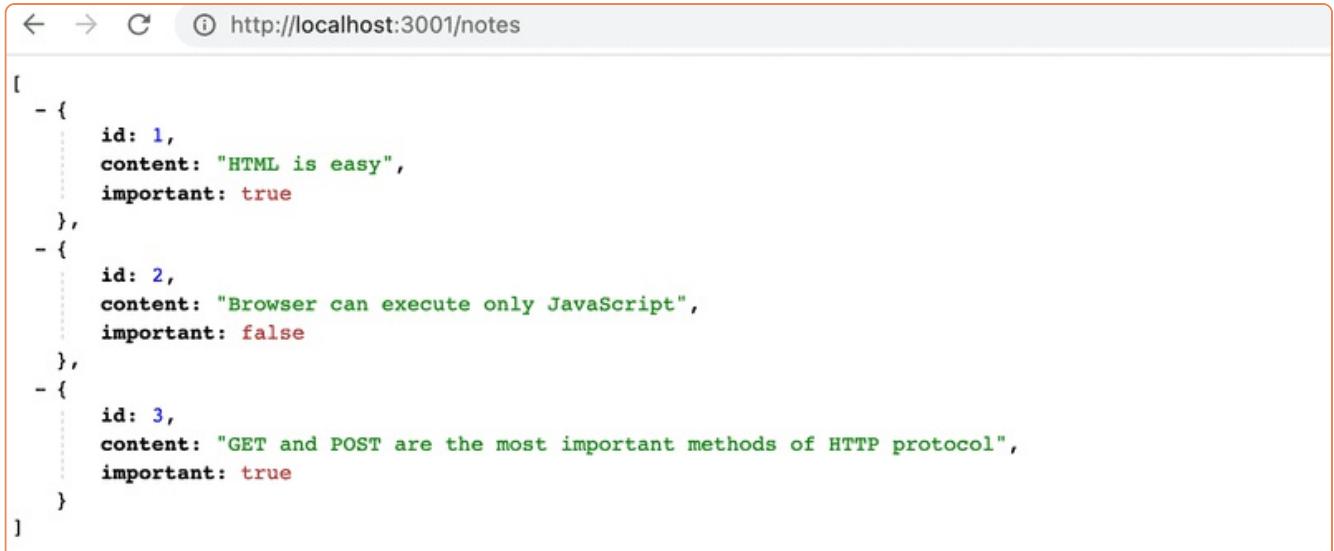
After installing run the following command to run the json-server. The *json-server* starts running on port 3000 by default; we will now define an alternate port 3001, for the json-server. The --watch option automatically looks for any saved changes to db.json

```
json-server --port 3001 --watch db.json
```

However, a global installation is not necessary. From the root directory of your app, we can run the *json-server* using the command `npx`:

```
npx json-server --port 3001 --watch db.json
```

Let's navigate to the address <http://localhost:3001/notes> in the browser. We can see that *json-server* serves the notes we previously wrote to the file in JSON format:



The screenshot shows a browser window with the URL `http://localhost:3001/notes` in the address bar. The page content displays the JSON array of 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  
  }]
```

If your browser doesn't have a way to format the display of JSON-data, then install an appropriate plugin, e.g. [JSONVue](#) to make your life easier.

Going forward, the idea will be to save the notes to the server, which in this case means saving them to the json-server. The React code fetches the notes from the server and renders them to the screen. Whenever a new note is added to the application, the React code also sends it to the server to make the new note persist in "memory".

json-server stores all the data in the `db.json` file, which resides on the server. In the real world, data would be stored in some kind of database. However, json-server is a handy tool that enables the use

of server-side functionality in the development phase without the need to program any of it.

We will get familiar with the principles of implementing server-side functionality in more detail in [part 3](#) of this course.

The browser as a runtime environment

Our first task is fetching the already existing notes to our React application from the address <http://localhost:3001/notes>.

In the part0 [example project](#), we already learned a way to fetch data from a server using JavaScript. The code in the example was fetching the data using [XMLHttpRequest](#), otherwise known as an HTTP request made using an XHR object. This is a technique introduced in 1999, which every browser has supported for a good while now.

The use of XHR is no longer recommended, and browsers already widely support the [fetch](#) method, which is based on so-called [promises](#), instead of the event-driven model used by XHR.

As a reminder from part0 (which one should *remember to not use* without a pressing reason), data was fetched using XHR in the following way:

```
const xhttp = new XMLHttpRequest()

xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    const data = JSON.parse(this.responseText)
    // handle the response that is saved in variable data
  }
}

xhttp.open('GET', '/data.json', true)
xhttp.send()
```

Right at the beginning, we register an *event handler* to the *xhttp* object representing the HTTP request, which will be called by the JavaScript runtime whenever the state of the *xhttp* object changes. If the change in state means that the response to the request has arrived, then the data is handled accordingly.

It is worth noting that the code in the event handler is defined before the request is sent to the server. Despite this, the code within the event handler will be executed at a later point in time. Therefore, the code does not execute synchronously "from top to bottom", but does so *asynchronously*. JavaScript calls the event handler that was registered for the request at some point.

A synchronous way of making requests that's common in Java programming, for instance, would play out as follows (NB, this is not actually working Java code):

```
HTTPRequest request = new HTTPRequest();
String url = "https://studies.cs.helsinki.fi/exampleapp/data.json";
List<Note> notes = request.get(url);

notes.forEach(m => {
```

```
    System.out.println(m.content);
}
```

In Java, the code executes line by line and stops to wait for the HTTP request, which means waiting for the command `request.get(...)` to finish. The data returned by the command, in this case the notes, are then stored in a variable, and we begin manipulating the data in the desired manner.

In contrast, JavaScript engines, or runtime environments follow the [asynchronous model](#). In principle, this requires all [IO operations](#) (with some exceptions) to be executed as non-blocking. This means that code execution continues immediately after calling an IO function, without waiting for it to return.

When an asynchronous operation is completed, or, more specifically, at some point after its completion, the JavaScript engine calls the event handlers registered to the operation.

Currently, JavaScript engines are *single-threaded*, which means that they cannot execute code in parallel. As a result, it is a requirement in practice to use a non-blocking model for executing IO operations. Otherwise, the browser would "freeze" during, for instance, the fetching of data from a server.

Another consequence of this single-threaded nature of JavaScript engines is that if some code execution takes up a lot of time, the browser will get stuck for the duration of the execution. If we added the following code at the top of our application:

```
setTimeout(() => {
  console.log('loop...')
  let i = 0
  while (i < 5000000000) {
    i++
  }
  console.log('end')
}, 5000)
```

everything would work normally for 5 seconds. However, when the function defined as the parameter for `setTimeout` is run, the browser will be stuck for the duration of the execution of the long loop. Even the browser tab cannot be closed during the execution of the loop, at least not in Chrome.

For the browser to remain *responsive*, i.e., to be able to continuously react to user operations with sufficient speed, the code logic needs to be such that no single computation can take too long.

There is a host of additional material on the subject to be found on the internet. One particularly clear presentation of the topic is the keynote by Philip Roberts called [What the heck is the event loop anyway?](#)

In today's browsers, it is possible to run parallelized code with the help of so-called [web workers](#). The event loop of an individual browser window is, however, still only handled by a [single thread](#).

npm

Let's get back to the topic of fetching data from the server.

We could use the previously mentioned promise-based function `fetch` to pull the data from the server. Fetch is a great tool. It is standardized and supported by all modern browsers (excluding IE).

That being said, we will be using the [axios](#) library instead for communication between the browser and server. It functions like fetch but is somewhat more pleasant to use. Another good reason to use axios is our getting familiar with adding external libraries, so-called *npm packages*, to React projects.

Nowadays, practically all JavaScript projects are defined using the node package manager, aka [npm](#). The projects created using Vite also follow the npm format. A clear indicator that a project uses npm is the *package.json* file located at the root of the project:

```
{  
  "name": "notes-frontend",  
  "private": true,  
  "version": "0.0.0",  
  "type": "module",  
  "scripts": {  
    "dev": "vite",  
    "build": "vite build",  
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",  
    "preview": "vite preview"  
  },  
  "dependencies": {  
    "react": "^18.2.0",  
    "react-dom": "^18.2.0"  
  },  
  "devDependencies": {  
    "@types/react": "^18.2.15",  
    "@types/react-dom": "^18.2.7",  
    "@vitejs/plugin-react": "^4.0.3",  
    "eslint": "^8.45.0",  
    "eslint-plugin-react": "^7.32.2",  
    "eslint-plugin-react-hooks": "^4.6.0",  
    "eslint-plugin-react-refresh": "^0.4.3",  
    "vite": "^4.4.5"  
  }  
}
```

At this point, the *dependencies* part is of most interest to us as it defines what *dependencies*, or external libraries, the project has.

We now want to use axios. Theoretically, we could define the library directly in the *package.json* file, but it is better to install it from the command line.

```
npm install axios
```

NB *npm-commands should always be run in the project root directory*, which is where the *package.json* file can be found.

Axios is now included among the other dependencies:

```
{  
  "name": "notes-frontend",  
  "private": true,  
  "version": "0.0.0",  
  "type": "module",  
  "scripts": {  
    "dev": "vite",  
    "build": "vite build",  
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",  
    "preview": "vite preview"  
  },  
  "dependencies": {  
    "react": "^18.2.0",  
    "react-dom": "^18.2.0",  
    "axios": "^1.5.1",  
    "@types/react": "^18.2.15",  
    "@types/react-dom": "^18.2.7",  
    "@vitejs/plugin-react": "^4.0.3",  
    "eslint": "^8.45.0",  
    "eslint-plugin-react": "^7.32.2",  
    "eslint-plugin-react-hooks": "^4.6.0",  
    "eslint-plugin-react-refresh": "^0.4.3",  
    "vite": "^4.4.5"  
  },  
  "devDependencies": {}  
}
```

```

"private": true,
"version": "0.0.0",
"type": "module",
"scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview"
},
"dependencies": [
    "axios": "^1.4.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
],
// ...
}

```

In addition to adding axios to the dependencies, the *npm install* command also *downloaded* the library code. As with other dependencies, the code can be found in the *node_modules* directory located in the root. As one might have noticed, *node_modules* contains a fair amount of interesting stuff.

Let's make another addition. Install *json-server* as a development dependency (only used during development) by executing the command:

```
npm install json-server --save-dev
```

and making a small addition to the *scripts* part of the *package.json* file:

```

{
    // ...
    "scripts": {
        "dev": "vite",
        "build": "vite build",
        "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
        "preview": "vite preview",
        "server": "json-server -p3001 --watch db.json"
    }
}

```

We can now conveniently, without parameter definitions, start the json-server from the project root directory with the command:

```
npm run server
```

We will get more familiar with the *npm* tool in the [third part of the course](#).

NB The previously started json-server must be terminated before starting a new one; otherwise, there will be trouble:

```
Cannot bind to the port 3001. Please specify another port number either through --port argument or through the json-server.json configuration file
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! notes@0.1.0 server: `json-server -p3001 db.json`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the notes@0.1.0 server script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.
```

The red print in the error message informs us about the issue:

Cannot bind to port 3001. Please specify another port number either through --port argument or through the json-server.json configuration file

As we can see, the application is not able to bind itself to the [port](#). The reason being that port 3001 is already occupied by the previously started json-server.

We used the command `npm install` twice, but with slight differences:

```
npm install axios
npm install json-server --save-dev
```

There is a fine difference in the parameters. *axios* is installed as a runtime dependency of the application because the execution of the program requires the existence of the library. On the other hand, *json-server* was installed as a development dependency (`--save-dev`), since the program itself doesn't require it. It is used for assistance during software development. There will be more on different dependencies in the next part of the course.

Axios and promises

Now we are ready to use Axios. Going forward, json-server is assumed to be running on port 3001.

NB: To run json-server and your react app simultaneously, you may need to use two terminal windows. One to keep json-server running and the other to run our React application.

The library can be brought into use the same way other libraries, e.g. React, are, i.e., by using an appropriate *import* statement.

Add the following to the file *main.jsx*:

```
import axios from 'axios'

const promise = axios.get('http://localhost:3001/notes')
console.log(promise)

const promise2 = axios.get('http://localhost:3001/foobar')
console.log(promise2)
```

If you open <http://localhost:5173/> in the browser, this should be printed to the console

```

▼ Promise {<pending>} ⓘ
  ► [[Prototype]]: Promise
  ► [[PromiseState]]: "fulfilled"
  ► [[PromiseResult]]: Object

▼ Promise {<pending>} ⓘ
  ► [[Prototype]]: Promise
  ► [[PromiseState]]: "rejected"
  ► [[PromiseResult]]: AxiosError

▶ GET http://localhost:3001/foobar 404 (Not Found)

▶ Uncaught (in promise)
  ► AxiosError {message: 'Request failed with status code 404', name: 'AxiosError', code: 'ERR_BAD_REQUEST', config: {...}}

```

Axios' method `get` returns a [promise](#).

The documentation on Mozilla's site states the following about promises:

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

In other words, a promise is an object that represents an asynchronous operation. A promise can have three distinct states:

- The promise is *pending*: It means that the final value (one of the following two) is not available yet.
- The promise is *fulfilled*: It means that the operation has been completed and the final value is available, which generally is a successful operation. This state is sometimes also called *resolved*.
- The promise is *rejected*: It means that an error prevented the final value from being determined, which generally represents a failed operation.

The first promise in our example is *fulfilled*, representing a successful `axios.get('http://localhost:3001/notes')` request. The second one, however, is *rejected*, and the console tells us the reason. It looks like we were trying to make an HTTP GET request to a non-existent address.

If, and when, we want to access the result of the operation represented by the promise, we must register an event handler to the promise. This is achieved using the method `then`:

```

const promise = axios.get('http://localhost:3001/notes')

promise.then(response => [
  console.log(response)
])

```

The following is printed to the console:

```
▼ {data: Array(3), status: 200, statusText: 'OK', headers: AxiosHeaders, config: {...}, ...} ⓘ
  ► config: {transitional: {...}, adapter: Array(2), transformRequest: Array(1), transformResponse: Array(1), timeout: 0, ...}
  ▼ data: Array(3)
    ► 0: {id: 1, content: 'HTML is easy', important: true}
    ► 1: {id: 2, content: 'Browser can execute only JavaScript', important: false}
    ► 2: {id: 3, content: 'GET and POST are the most important methods of HTTP protocol', important: true}
      length: 3
    ► [[Prototype]]: Array(0)
  ► headers: AxiosHeaders {cache-control: 'no-cache', content-length: '299', content-type: 'application/json; charset=utf-8', e
  ► request: XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequest}
    status: 200
    statusText: "OK"
  ► [[Prototype]]: Object
```

The JavaScript runtime environment calls the callback function registered by the *then* method providing it with a *response* object as a parameter. The *response* object contains all the essential data related to the response of an HTTP GET request, which would include the returned *data*, *status code*, and *headers*.

Storing the promise object in a variable is generally unnecessary, and it's instead common to chain the *then* method call to the *axios* method call, so that it follows it directly:

```
axios.get('http://localhost:3001/notes').then(response => {
  const notes = response.data
  console.log(notes)
})
```

The callback function now takes the data contained within the response, stores it in a variable, and prints the notes to the console.

A more readable way to format *chained* method calls is to place each call on its own line:

```
axios
  .get('http://localhost:3001/notes')
  .then(response => {
    const notes = response.data
    console.log(notes)
})
```

The data returned by the server is plain text, basically just one long string. The *axios* library is still able to parse the data into a JavaScript array, since the server has specified that the data format is *application/json; charset=utf-8* (see the previous image) using the *content-type* header.

We can finally begin using the data fetched from the server.

Let's try and request the notes from our local server and render them, initially as the *App* component. Please note that this approach has many issues, as we're rendering the entire *App* component only when we successfully retrieve a response:

```
import ReactDOM from 'react-dom/client'
import axios from 'axios'
import App from './App'

axios.get('http://localhost:3001/notes').then(response => [
  const notes = response.data
```

```
ReactDOM.createRoot(document.getElementById('root')).render(<App notes={notes}>
/>)
})
```

This method could be acceptable in some circumstances, but it's somewhat problematic. Let's instead move the fetching of the data into the *App* component.

What's not immediately obvious, however, is where the command `axios.get` should be placed within the component.

Effect-hooks

We have already used [state hooks](#) that were introduced along with React version [16.8.0](#), which provide state to React components defined as functions - the so-called *functional components*. Version 16.8.0 also introduces [effect hooks](#) as a new feature. As per the official docs:

Effects let a component connect to and synchronize with external systems. This includes dealing with network, browser DOM, animations, widgets written using a different UI library, and other non-React code.

As such, effect hooks are precisely the right tool to use when fetching data from a server.

Let's remove the fetching of data from `main.jsx`. Since we're going to be retrieving the notes from the server, there is no longer a need to pass data as props to the *App* component. So `main.jsx` can be simplified to:

```
import ReactDOM from "react-dom/client";
import App from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(<App />);
```

The *App* component changes as follows:

```
import { useState, useEffect } from 'react';
import axios from 'axios';
import Note from './components/Note';

const App = () => {
  const [notes, setNotes] = useState([]);
  const [newNote, setNewNote] = useState('');
  const [showAll, setShowAll] = useState(true);

  useEffect(() => {
    console.log('effect');
    axios.get('http://localhost:3001/notes')
      .then(response => {
        setNotes(response.data);
      })
      .catch(error => {
        console.error(error);
      });
  }, [notes]);
}

export default App;
```

We have also added a few helpful prints, which clarify the progression of the execution.

This is printed to the console:

```
render 0 notes
effect
promise fulfilled
render 3 notes
```

First, the body of the function defining the component is executed and the component is rendered for the first time. At this point *render 0 notes* is printed, meaning data hasn't been fetched from the server yet.

The following function, or effect in React parlance:

```
() => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
}
```

is executed immediately after rendering. The execution of the function results in *effect* being printed to the console, and the command *axios.get* initiates the fetching of data from the server as well as registers the following function as an *event handler* for the operation:

```
response => {
  console.log('promise fulfilled')
  setNotes(response.data)
}
```

When data arrives from the server, the JavaScript runtime calls the function registered as the event handler, which prints *promise fulfilled* to the console and stores the notes received from the server into the state using the function *setNotes(response.data)*.

As always, a call to a state-updating function triggers the re-rendering of the component. As a result, *render 3 notes* is printed to the console, and the notes fetched from the server are rendered to the screen.

Finally, let's take a look at the definition of the effect hook as a whole:

```
useEffect(() => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
})
```

```
}, [])
```

Let's rewrite the code a bit differently.

```
const hook = () => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
}

useEffect(hook, [])
```

Now we can see more clearly that the function `useEffect` takes *two parameters*. The first is a function, the *effect* itself. According to the documentation:

By default, effects run after every completed render, but you can choose to fire it only when certain values have changed.

So by default, the effect is *always* run after the component has been rendered. In our case, however, we only want to execute the effect along with the first render.

The second parameter of `useEffect` is used to [specify how often the effect is run](#). If the second parameter is an empty array `[]`, then the effect is only run along with the first render of the component.

There are many possible use cases for an effect hook other than fetching data from the server. However, this use is sufficient for us, for now.

Think back to the sequence of events we just discussed. Which parts of the code are run? In what order? How often? Understanding the order of events is critical!

Note that we could have also written the code for the effect function this way:

```
useEffect(() => {
  console.log('effect')

  const eventHandler = response => {
    console.log('promise fulfilled')
    setNotes(response.data)
  }

  const promise = axios.get('http://localhost:3001/notes')
  promise.then(eventHandler)
}, [])
```

A reference to an event handler function is assigned to the variable `eventHandler`. The promise returned by the `get` method of Axios is stored in the variable `promise`. The registration of the callback happens by giving the `eventHandler` variable, referring to the event-handler function, as a parameter to the `then` method of the promise. It isn't usually necessary to assign functions and promises to variables, and a more compact way of representing things, as seen below, is sufficient.

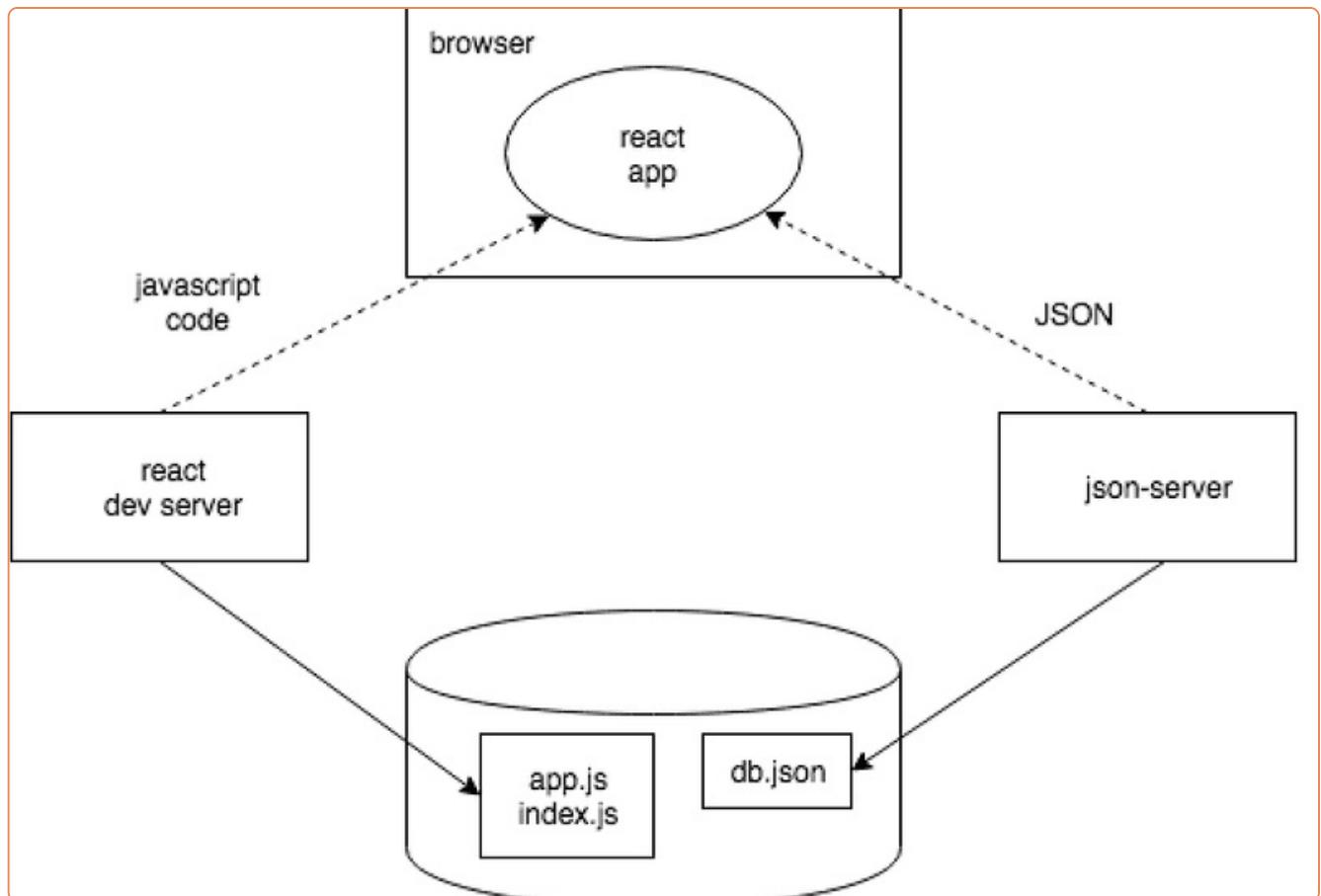
```
useEffect(() => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
}, [])
```

We still have a problem with our application. When adding new notes, they are not stored on the server.

The code for the application, as described so far, can be found in full on [github](#), on branch `part2-4`.

The development runtime environment

The configuration for the whole application has steadily grown more complex. Let's review what happens and where. The following image describes the makeup of the application



The JavaScript code making up our React application is run in the browser. The browser gets the

JavaScript from the *React dev server*, which is the application that runs after running the command `npm run dev`. The dev-server transforms the JavaScript into a format understood by the browser. Among other things, it stitches together JavaScript from different files into one file. We'll discuss the dev-server in more detail in part 7 of the course.

The React application running in the browser fetches the JSON formatted data from *json-server* running on port 3001 on the machine. The server we query the data from - *json-server* - gets its data from the file `db.json`.

At this point in development, all the parts of the application happen to reside on the software developer's machine, otherwise known as localhost. The situation changes when the application is deployed to the internet. We will do this in part 3.

Exercise 2.11.

2.11: The Phonebook Step 6

We continue with developing the phonebook. Store the initial state of the application in the file `db.json`, which should be placed in the root of the project.

```
[{"persons": [{"name": "Arto Hellas", "number": "040-123456", "id": 1}, {"name": "Ada Lovelace", "number": "39-44-5323523", "id": 2}, {"name": "Dan Abramov", "number": "12-43-234345", "id": 3}, {"name": "Mary Poppendieck", "number": "39-23-6423122", "id": 4}]}]
```

Start json-server on port 3001 and make sure that the server returns the list of people by going to the address <http://localhost:3001/persons> in the browser.

If you receive the following error message:

```
events.js:182
```

```
throw er; // Unhandled 'error' event
^

Error: listen EADDRINUSE 0.0.0.0:3001
  at Object._errnoException [util.js:1019:11]
  at _exceptionWithHostPort [util.js:1041:20]
```

it means that port 3001 is already in use by another application, e.g. in use by an already running json-server. Close the other application, or change the port in case that doesn't work.

Modify the application such that the initial state of the data is fetched from the server using the `axios`-library. Complete the fetching with an [Effect hook](#).

2d: Altering data in server

When creating notes in our application, we would naturally want to store them in some backend server. The [json-server](#) package claims to be a so-called REST or RESTful API in its documentation:

Get a full fake REST API with zero coding in less than 30 seconds (seriously)

The json-server does not exactly match the description provided by the textbook [definition](#) of a REST API, but neither do most other APIs claiming to be RESTful.

We will take a closer look at REST in the [next part](#) of the course. But it's important to familiarize ourselves at this point with some of the [conventions](#) used by json-server and REST APIs in general. In particular, we will be taking a look at the conventional use of [routes](#), aka URLs and HTTP request types, in REST.

REST

In REST terminology, we refer to individual data objects, such as the notes in our application, as *resources*. Every resource has a unique address associated with it - its URL. According to a general convention used by json-server, we would be able to locate an individual note at the resource URL `notes/3`, where 3 is the id of the resource. The `notes` URL, on the other hand, would point to a resource collection containing all the notes.

Resources are fetched from the server with HTTP GET requests. For instance, an HTTP GET request to the URL `notes/3` will return the note that has the id number 3. An HTTP GET request to the `notes` URL would return a list of all notes.

Creating a new resource for storing a note is done by making an HTTP POST request to the `notes` URL according to the REST convention that the json-server adheres to. The data for the new note resource is sent in the *body* of the request.

json-server requires all data to be sent in JSON format. What this means in practice is that the data must be a correctly formatted string and that the request must contain the *Content-Type* request header with the value `application/json`.

Sending Data to the Server

Let's make the following changes to the event handler responsible for creating a new note:

```
addNote = event => {
  event.preventDefault()
  const noteObject = {
    content: newNote,
    important: Math.random() < 0.5,
  }

  axios .post('http://localhost:3001/notes', noteObject) .then(response =>
  {
    console.log(response)
  })
}
```

We create a new object for the note but omit the *id* property since it's better to let the server generate ids for our resources.

The object is sent to the server using the axios *post* method. The registered event handler logs the response that is sent back from the server to the console.

When we try to create a new note, the following output pops up in the console:

```
▼{data: {...}, status: 201, statusText: 'Created', headers: AxiosHeaders, config: {...}, ...} ⓘ
  ► config: {transitional: {...}, adapter: Array(2), transformRequest: Array(1), transformResponse: Array(1), timeout: 0, ...}
  ► data: {content: 'POST is used to create new resources', important: false, id: 4}
  ► headers: AxiosHeaders {cache-control: 'no-cache', content-length: '88', content-type: 'application/json; charset=utf-8'}
  ► request: XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequestUpload, status: 201, statusText: "Created"}
```

The newly created note resource is stored in the value of the *data* property of the *response* object.

Quite often it is useful to inspect HTTP requests in the *Network* tab of Chrome developer tools, which was used heavily at the beginning of [part 0](#).

We can use the inspector to check that the headers sent in the POST request are what we expected them to be:

x Headers Payload Preview Response Initiator Timing

▼ General

Request URL: http://localhost:3001/notes
Request Method: POST
Status Code: 201 Created
Remote Address: [::1]:3001
Referrer Policy: strict-origin-when-cross-origin

► Response Headers (16)

View source

▼ Request Headers

Accept: application/json, text/plain, */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Connection: keep-alive
Content-Length: 68
Content-Type: application/json
Host: localhost:3001

Since the data we sent in the POST request was a JavaScript object, axios automatically knew to set the appropriate *application/json* value for the *Content-Type* header.

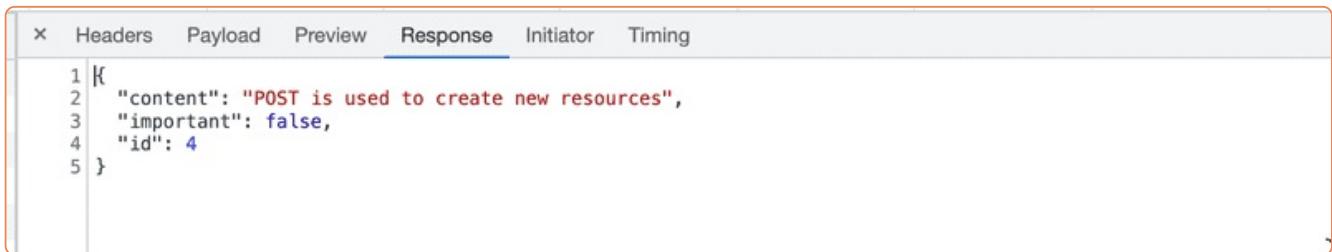
The tab *payload* can be used to check the request data:



A screenshot of the Network tab in a browser's developer tools. The tab bar includes Headers, Payload (which is underlined), Preview, Response, Initiator, and Timing. Under Request Payload, it shows the JSON object sent in the POST request:

```
{content: "POST is used to create new resources", important: false}  
content: "POST is used to create new resources"  
important: false
```

Also the tab *response* is useful, it shows what was the data the server responded with:



A screenshot of the Network tab in a browser's developer tools. The tab bar includes Headers, Payload, Preview, Response (which is underlined), Initiator, and Timing. Under Response, it shows the JSON object returned by the server:

```
{  
  "content": "POST is used to create new resources",  
  "important": false,  
  "id": 4  
}
```

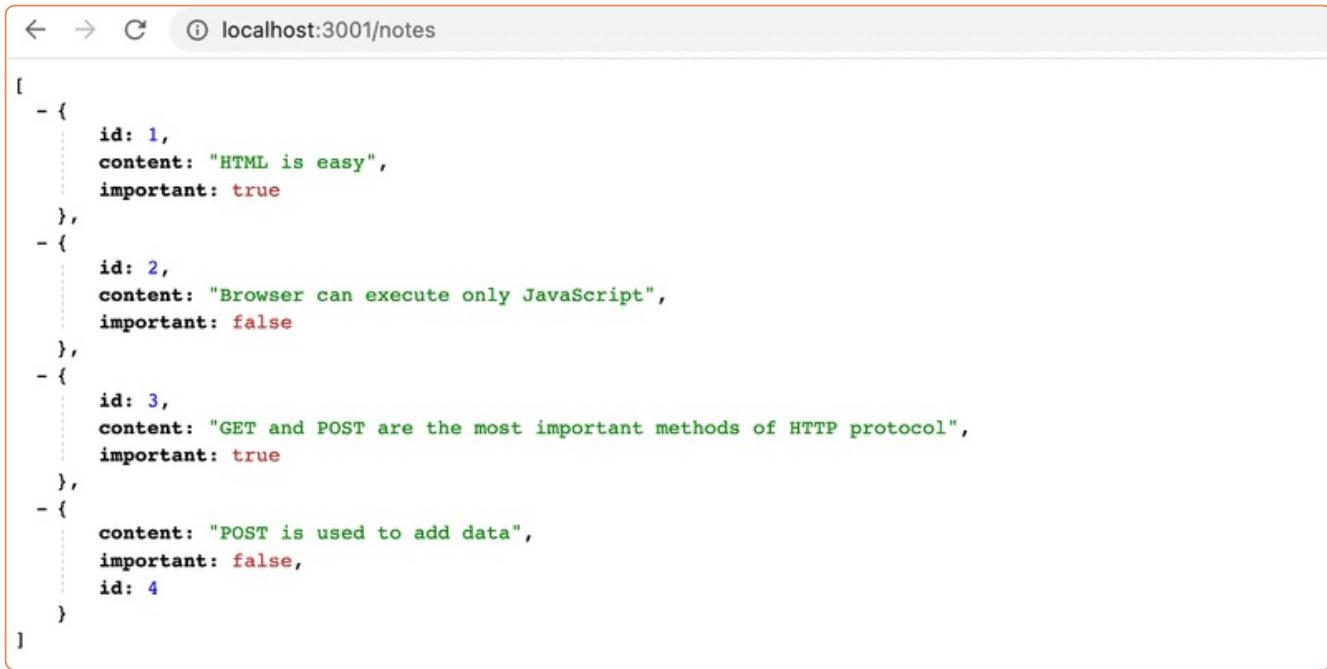
The new note is not rendered to the screen yet. This is because we did not update the state of the App component when we created it. Let's fix this:

```
addNote = event => {  
  event.preventDefault()  
  const noteObject = {  
    content: newNote,  
    important: Math.random() > 0.5,  
  }  
  
  axios  
    .post('http://localhost:3001/notes', noteObject)  
    .then(response => {  
      setNotes(notes.concat(response.data))  
      setNewNote('')  
    })  
}
```

The new note returned by the backend server is added to the list of notes in our application's state in the customary way of using the *setNotes* function and then resetting the note creation form. An **important detail** to remember is that the *concat* method does not change the component's original state, but instead creates a new copy of the list.

Once the data returned by the server starts to have an effect on the behavior of our web applications, we are immediately faced with a whole new set of challenges arising from, for instance, the asynchronicity of communication. This necessitates new debugging strategies, console logging and other means of debugging become increasingly more important. We must also develop a sufficient understanding of the principles of both the JavaScript runtime and React components. Guessing won't be enough.

It's beneficial to inspect the state of the backend server, e.g. through the browser:



A screenshot of a web browser window showing the URL `localhost:3001/notes`. The page displays a JSON array of four note objects. Each note has an `id`, `content`, and `important` boolean property. Note 1 is important, while Notes 2, 3, and 4 are not.

```
[{"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}, {"id": 4, "content": "POST is used to add data", "important": false} ]
```

This makes it possible to verify that all the data we intended to send was actually received by the server.

In the next part of the course, we will learn to implement our own logic in the backend. We will then take a closer look at tools like [Postman](#) that helps us to debug our server applications. However, inspecting the state of the json-server through the browser is sufficient for our current needs.

The code for the current state of our application can be found in the *part2-5* branch on [GitHub](#).

Changing the Importance of Notes

Let's add a button to every note that can be used for toggling its importance.

We make the following changes to the *Note* component:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important' : 'make important'

  return [
    <li>
      {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  ]
}
```

We add a button to the component and assign its event handler as the *toggleImportance* function passed in the component's props.

The *App* component defines an initial version of the *toggleImportanceOf* event handler function and passes it to every *Note* component:

```

const App = () => {
  const [notes, setNotes] = useState([])
  const [newNote, setNewNote] = useState('')
  const [showAll, setShowAll] = useState(true)

  // ...

  const toggleImportanceOf = (id) => {    console.log('importance of ' + id + ' needs to be toggled')  }
  // ...

  return (
    <div>
      <h1>Notes</h1>
      <div>
        <button onClick={() => setShowAll(!showAll)}>
          show {showAll ? 'important' : 'all'}
        </button>
      </div>
      <ul>
        {notesToShow.map(note =>
          <Note
            key={note.id}
            note={note}
            toggleImportance={() => toggleImportanceOf(note.id)} />
        )}
      </ul>
      // ...
    </div>
  )
}

```

Notice how every note receives its own *unique* event handler function since the *id* of every note is unique.

E.g., if *note.id* is 3, the event handler function returned by *toggleImportance(note.id)* will be:

```
() => { console.log('importance of 3 needs to be toggled') }
```

A short reminder here. The string printed by the event handler is defined in a Java-like manner by adding the strings:

```
console.log('importance of ' + id + ' needs to be toggled')
```

The [template string](#) syntax added in ES6 can be used to write similar strings in a much nicer way:

```
console.log(`importance of ${id} needs to be toggled`)
```

We can now use the "dollar-bracket"-syntax to add parts to the string that will evaluate JavaScript expressions, e.g. the value of a variable. Note that we use backticks in template strings instead of

quotation marks used in regular JavaScript strings.

Individual notes stored in the json-server backend can be modified in two different ways by making HTTP requests to the note's unique URL. We can either *replace* the entire note with an HTTP PUT request or only change some of the note's properties with an HTTP PATCH request.

The final form of the event handler function is the following:

```
const toggleImportanceOf = id => {
  const url = `http://localhost:3001/notes/${id}`
  const note = notes.find(n => n.id === id)
  const changedNote = { ...note, important: !note.important }

  axios.put(url, changedNote).then(response => {
    setNotes(notes.map(n => n.id !== id ? n : response.data))
  })
}
```

Almost every line of code in the function body contains important details. The first line defines the unique URL for each note resource based on its id.

The array [find method](#) is used to find the note we want to modify, and we then assign it to the *note* variable.

After this, we create a *new object* that is an exact copy of the old note, apart from the *important* property that has the value flipped (from true to false or from false to true).

The code for creating the new object that uses the [object spread](#) syntax may seem a bit strange at first:

```
const changedNote = { ...note, important: !note.important }
```

In practice, `{ ...note }` creates a new object with copies of all the properties from the *note* object. When we add properties inside the curly braces after the spread object, e.g. `{ ...note, important: true }`, then the value of the *important* property of the new object will be *true*. In our example, the *important* property gets the negation of its previous value in the original object.

There are a few things to point out. Why did we make a copy of the note object we wanted to modify when the following code also appears to work?

```
const note = notes.find(n => n.id === id)
note.important = !note.important

axios.put(url, note).then(response => [
  // ...
])
```

This is not recommended because the variable *note* is a reference to an item in the *notes* array in the component's state, and as we recall we must [never mutate state directly](#) in React.

It's also worth noting that the new object *changedNote* is only a so-called [shallow copy](#), meaning that the values of the new object are the same as the values of the old object. If the values of the old

object were objects themselves, then the copied values in the new object would reference the same objects that were in the old object.

The new note is then sent with a PUT request to the backend where it will replace the old object.

The callback function sets the component's *notes* state to a new array that contains all the items from the previous *notes* array, except for the old note which is replaced by the updated version of it returned by the server:

```
axios.put(url, changedNote).then(response => [
  setNotes(notes.map(note => note.id !== id ? note : response.data))
])
```

This is accomplished with the *map* method:

```
notes.map(note => note.id !== id ? note : response.data)
```

The *map* method creates a new array by mapping every item from the old array into an item in the new array. In our example, the new array is created conditionally so that if *note.id* \neq *id* is true; we simply copy the item from the old array into the new array. If the condition is false, then the note object returned by the server is added to the array instead.

This *map* trick may seem a bit strange at first, but it's worth spending some time wrapping your head around it. We will be using this method many times throughout the course.

Extracting Communication with the Backend into a Separate Module

The *App* component has become somewhat bloated after adding the code for communicating with the backend server. In the spirit of the [single responsibility principle](#), we deem it wise to extract this communication into its own [module](#).

Let's create a *src/services* directory and add a file there called *notes.js*:

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

const getAll = () => [
  return axios.get(baseUrl)
]

const create = newObjet => [
  return axios.post(baseUrl, newObjet)
]

const update = [id, newObjet] => [
  return axios.put(`/${baseUrl}/${id}`, newObjet)
]

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

```
    update: update
}
```

The module returns an object that has three functions (*getAll*, *create*, and *update*) as its properties that deal with notes. The functions directly return the promises returned by the axios methods.

The *App* component uses *import* to get access to the module:

```
import noteService from './services/notes'
const App = () => {
```

The functions of the module can be used directly with the imported variable *noteService* as follows:

```
const App = () => {
  // ...

  useEffect(() => {
    noteService      .getAll()      .then(response => {
      setNotes(response.data)
    })
  }, [])

  const toggleImportanceOf = id => {
    const note = notes.find(n => n.id === id)
    const changedNote = { ...note, important: !note.important }

    noteService      .update(id, changedNote)      .then(response => {
      setNotes(notes.map(note => note.id !== id ? note : response.data))
    })
  }

  const addNote = [event] => {
    event.preventDefault()
    const noteObject = [
      content: newNote,
      important: Math.random() > 0.5
    ]

    noteService      .create(noteObject)      .then(response => {
      setNotes([notes.concat(response.data)])
      setNewNote('')
    })
  }

  // ...
}

export default App
```

We could take our implementation a step further. When the *App* component uses the functions, it receives an object that contains the entire response for the HTTP request:

```
noteService
  .getAll()
  .then(response => {
    setNotes(response.data)
  })
```

The `App` component only uses the `response.data` property of the response object.

The module would be much nicer to use if, instead of the entire HTTP response, we would only get the response data. Using the module would then look like this:

```
noteService
  .getAll()
  .then(initialNotes => [
    setNotes(initialNotes)
  ])
```

We can achieve this by changing the code in the module as follows (the current code contains some copy-paste, but we will tolerate that for now):

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

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

const create = newObject => {
  const request = axios.post(baseUrl, newObject)
  return request.then(response => response.data)
}

const update = (id, newObject) => {
  const request = axios.put(`/${baseUrl}/${id}`, newObject)
  return request.then(response => response.data)
}

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

We no longer return the promise returned by `axios` directly. Instead, we assign the promise to the `request` variable and call its `then` method:

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

The last row in our function is simply a more compact expression of the same code as shown below:

```
const getAll = () => {
```

```

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

```

The modified `getAll` function still returns a promise, as the `then` method of a promise also [returns a promise](#).

After defining the parameter of the `then` method to directly return `response.data`, we have gotten the `getAll` function to work like we wanted it to. When the HTTP request is successful, the promise returns the data sent back in the response from the backend.

We have to update the `App` component to work with the changes made to our module. We have to fix the callback functions given as parameters to the `noteService` object's methods so that they use the directly returned response data:

```

const App = () => [
    // ...

    useEffect(() => [
        noteService
            .getAll()
            .then(initialNotes => {
                setNotes(initialNotes)
            })
    ], [])
]

const toggleImportanceOf = id => [
    const note = notes.find(n => n.id === id)
    const changedNote = { ...note, important: !note.important }

    noteService
        .update(id, changedNote)
        .then(returnedNote => {
            setNotes(notes.map(note => note.id !== id ?
                note : returnedNote))
        })
]

const addNote = (event) => [
    event.preventDefault()
    const noteObject = [
        content: newNote,
        important: Math.random() > 0.5
    ]

    noteService
        .create(noteObject)
        .then(returnedNote => {
            setNotes(notes.concat(returnedNote))
            setNewNote('')
        })
]

```

This is all quite complicated and attempting to explain it may just make it harder to understand. The

internet is full of material discussing the topic, such as [this](#) one.

The "Async and performance" book from the [You do not know JS](#) book series [explains the topic](#) well, but the explanation is many pages long.

Promises are central to modern JavaScript development and it is highly recommended to invest a reasonable amount of time into understanding them.

Cleaner Syntax for Defining Object Literals

The module defining note-related services currently exports an object with the properties `getAll`, `create`, and `update` that are assigned to functions for handling notes.

The module definition was:

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

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

const create = newObject => {
  const request = axios.post(baseUrl, newObject)
  return request.then(response => response.data)
}

const update = [id, newObject] => {
  const request = axios.put(`#${baseUrl}/${id}`, newObject)
  return request.then(response => response.data)
}

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

The module exports the following, rather peculiar looking, object:

```
{
  getAll: getAll,
  create: create,
  update: update
}
```

The labels to the left of the colon in the object definition are the *keys* of the object, whereas the ones to the right of it are *variables* that are defined inside the module.

Since the names of the keys and the assigned variables are the same, we can write the object definition with a more compact syntax:

```
[  
  getAll,  
  create,  
  update  
]
```

As a result, the module definition gets simplified into the following form:

```
import axios from 'axios'  
const baseUrl = 'http://localhost:3001/notes'  
  
const getAll = () => {  
  const request = axios.get(baseUrl)  
  return request.then(response => response.data)  
}  
  
const create = newObject => {  
  const request = axios.post(baseUrl, newObject)  
  return request.then(response => response.data)  
}  
  
const update = (id, newObject) => {  
  const request = axios.put(`/${baseUrl}/${id}`, newObject)  
  return request.then(response => response.data)  
}  
  
export default [ getAll, create, update ]
```

In defining the object using this shorter notation, we make use of a [new feature](#) that was introduced to JavaScript through ES6, enabling a slightly more compact way of defining objects using variables.

To demonstrate this feature, let's consider a situation where we have the following values assigned to variables:

```
const name = 'Leevi'  
const age = 0
```

In older versions of JavaScript we had to define an object like this:

```
const person = {  
  name: name,  
  age: age  
}
```

However, since both the property fields and the variable names in the object are the same, it's enough to simply write the following in ES6 JavaScript:

```
const person = { name, age }
```

The result is identical for both expressions. They both create an object with a `name` property with the value `Leevi` and an `age` property with the value `0`.

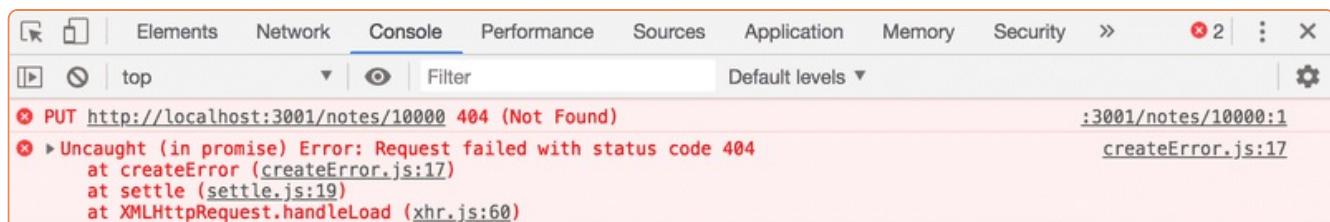
Promises and Errors

If our application allowed users to delete notes, we could end up in a situation where a user tries to change the importance of a note that has already been deleted from the system.

Let's simulate this situation by making the `getAll` function of the note service return a "hardcoded" note that does not actually exist on the backend server:

```
const getAll = () => {
  const request = axios.get(baseUrl)
  const nonExisting = [
    id: 10000,
    content: 'This note is not saved to server',
    important: true,
  ]
  return request.then(response => response.data.concat(nonExisting))
}
```

When we try to change the importance of the hardcoded note, we see the following error message in the console. The error says that the backend server responded to our HTTP PUT request with a status code `404 not found`.



The application should be able to handle these types of error situations gracefully. Users won't be able to tell that an error has occurred unless they happen to have their console open. The only way the error can be seen in the application is that clicking the button does not affect the note's importance.

We had [previously](#) mentioned that a promise can be in one of three different states. When an HTTP request fails, the associated promise is *rejected*. Our current code does not handle this rejection in any way.

The rejection of a promise is [handled](#) by providing the `then` method with a second callback function, which is called in the situation where the promise is rejected.

The more common way of adding a handler for rejected promises is to use the `catch` method.

In practice, the error handler for rejected promises is defined like this:

```
axios
  .get('http://example.com/probably_will_fail')
  .then(response => {
    console.log('success!')
  }
  .catch(error => {
    console.error('error!', error)
  })
}
```

```
})
.catch(error => {
  console.log('fail')
})
```

If the request fails, the event handler registered with the *catch* method gets called.

The *catch* method is often utilized by placing it deeper within the promise chain.

When our application makes an HTTP request, we are in fact creating a [promise chain](#):

```
axios
  .put(`#${baseUrl}/${id}`, newObject)
  .then(response => response.data)
  .then(changedNote => [
    // ...
  ])
```

The *catch* method can be used to define a handler function at the end of a promise chain, which is called once any promise in the chain throws an error and the promise becomes *rejected*.

```
axios
  .put(`#${baseUrl}/${id}`, newObject)
  .then(response => response.data)
  .then(changedNote => [
    // ...
  ])
  .catch(error => {
    console.log('fail')
  })
```

Let's use this feature and register an error handler in the *App* component:

```
const toggleImportanceOf = id => {
  const note = notes.find(n => n.id === id)
  const changedNote = { ...note, important: !note.important }

  noteService
    .update(id, changedNote).then(returnedNote => [
      setNotes(notes.map(note => note.id !== id ? note : returnedNote))
    ])
    .catch(error => {
      alert(`the note '${note.content}' was already deleted from server`)
      setNotes(notes.filter(n => n.id !== id))
    })
}
```

The error message is displayed to the user with the trusty old [alert](#) dialog popup, and the deleted note gets filtered out from the state.

Removing an already deleted note from the application's state is done with the array [filter](#) method, which returns a new array comprising only the items from the list for which the function that was passed as a parameter returns true for:

```
notes.filter[n => n.id !== id]
```

It's probably not a good idea to use alert in more serious React applications. We will soon learn a more advanced way of displaying messages and notifications to users. There are situations, however, where a simple, battle-tested method like *alert* can function as a starting point. A more advanced method could always be added in later, given that there's time and energy for it.

The code for the current state of our application can be found in the *part2-6* branch on [GitHub](#).

Full stack developer's oath

It is again time for the exercises. The complexity of our app is now increasing since besides just taking care of the React components in the frontend, we also have a backend that is persisting the application data.

To cope with the increasing complexity we should extend the web developer's oath to a *Full stack developer's oath*, which reminds us to make sure that the communication between frontend and backend happens as expected.

So here is the updated 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 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 2.12.-2.15.

2.12: The Phonebook step 7

Let's return to our phonebook application.

Currently, the numbers that are added to the phonebook are not saved to a backend server. Fix this situation.

2.13: The Phonebook step 8

Extract the code that handles the communication with the backend into its own module by following the example shown earlier in this part of the course material.

2.14: The Phonebook step 9

Make it possible for users to delete entries from the phonebook. The deletion can be done through a dedicated button for each person in the phonebook list. You can confirm the action from the user by using the `window.confirm` method:

The screenshot shows a web browser window with the URL `localhost:3000`. On the left, there is a "Phonebook" application. It has a form for adding new entries with fields for name and number, and a "add" button. Below this, there is a section titled "Numbers" listing three entries: "Arto Hellas 040-123456", "Ada Lovelace 39-44-5323523", and "Dan Abramov 12-43-234345", each with a "delete" button. A modal dialog box is overlaid on the page, centered over the "Numbers" section. The dialog box contains the text "localhost:3000 says" followed by "Delete Arto Hellas ?" and two buttons: "Cancel" and "OK".

The associated resource for a person in the backend can be deleted by making an HTTP DELETE request to the resource's URL. If we are deleting e.g. a person who has the `id` 2, we would have to make an HTTP DELETE request to the URL `localhost:3001/persons/2`. No data is sent with the request.

You can make an HTTP DELETE request with the `axios` library in the same way that we make all of the other requests.

NB: You can't use the name `delete` for a variable because it's a reserved word in JavaScript. E.g. the following is not possible:

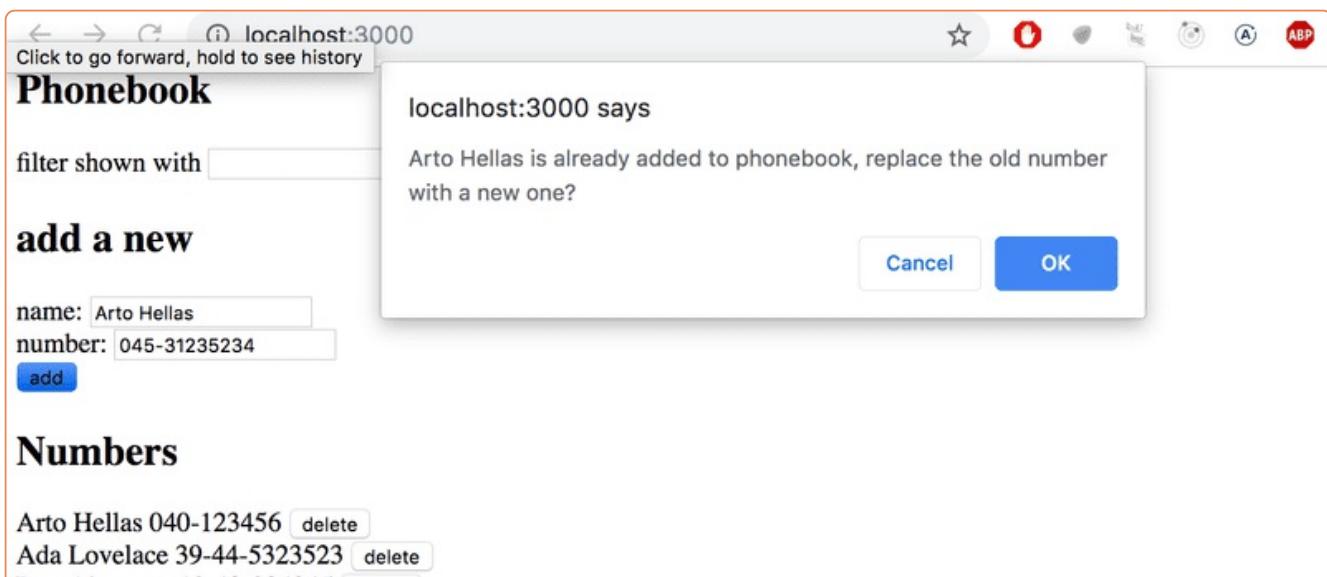
```
// use some other name for variable!
const delete = [id] => [
  // ...
]
```

2.15*: The Phonebook step 10

Why is there a star in the exercise? See [here](#) for the explanation.

Change the functionality so that if a number is added to an already existing user, the new number will replace the old number. It's recommended to use the HTTP PUT method for updating the phone number.

If the person's information is already in the phonebook, the application can ask the user to confirm the action:



2e: Adding styles to React app

The appearance of our current application is quite modest. In [exercise 0.2](#), the assignment was to go through Mozilla's [CSS tutorial](#).

Let's take a look at how we can add styles to a React application. There are several different ways of doing this and we will take a look at the other methods later on. First, we will add CSS to our application the old-school way; in a single file without using a [CSS preprocessor](#) (although this is not entirely true as we will learn later on).

Let's add a new *index.css* file under the *src* directory and then add it to the application by importing it in the *main.jsx* file:

```
import './index.css'
```

Let's add the following CSS rule to the *index.css* file:

```
h1 {  
  color: green;  
}
```

CSS rules comprise of *selectors* and *declarations*. The selector defines which elements the rule should be applied to. The selector above is *h1*, which will match all of the *h1* header tags in our application.

The declaration sets the *color* property to the value *green*.

One CSS rule can contain an arbitrary number of properties. Let's modify the previous rule to make the text cursive, by defining the font style as *italic*:

```
h1 {  
  color: green;  
  font-style: italic;  
}
```

There are many ways of matching elements by using [different types of CSS selectors](#).

If we wanted to target, let's say, each one of the notes with our styles, we could use the selector *li*, as all of the notes are wrapped inside *li* tags:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important'
    : 'make important';

  return [
    <li>
      {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  ]
}
```

Let's add the following rule to our style sheet (since my knowledge of elegant web design is close to zero, the styles don't make much sense):

```
li {
  color: grey;
  padding-top: 3px;
  font-size: 15px;
}
```

Using element types for defining CSS rules is slightly problematic. If our application contained other *li* tags, the same style rule would also be applied to them.

If we want to apply our style specifically to notes, then it is better to use [class selectors](#).

In regular HTML, classes are defined as the value of the *class* attribute:

```
<li class="note">some text...</li>
```

In React we have to use the [className](#) attribute instead of the *class* attribute. With this in mind, let's make the following changes to our *Note* component:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important'
    : 'make important';

  return [
    <li className='note'>      {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  ]
}
```

Class selectors are defined with the *.classname* syntax:

```
.note {  
  color: grey;  
  padding-top: 5px;  
  font-size: 15px;  
}
```

If you now add other *i* elements to the application, they will not be affected by the style rule above.

Improved error message

We previously implemented the error message that was displayed when the user tried to toggle the importance of a deleted note with the *alert* method. Let's implement the error message as its own React component.

The component is quite simple:

```
const Notification = ({ message }) => {  
  if (message === null) {  
    return null  
  }  
  
  return [  
    <div className='error'>  
      {message}  
    </div>  
  ]  
}
```

If the value of the *message* prop is *null*, then nothing is rendered to the screen, and in other cases, the message gets rendered inside of a div element.

Let's add a new piece of state called *errorMessage* to the *App* component. Let's initialize it with some error message so that we can immediately test our component:

```
const App = () => [  
  const [notes, setNotes] = useState([])  
  const [newNote, setNewNote] = useState('')  
  const [showAll, setShowAll] = useState(true)  
  const [errorMessage, setErrorMessage] = useState('some error happened...')  
  // ...  
  
  return [  
    <div>  
      <h1>Notes</h1>  
      <Notification message={errorMessage} />          <div>  
        <button onClick={() => setShowAll(!showAll)}>  
          show {showAll ? 'important' : 'all'}  
        </button>  
      </div>  
      // ...  
    </div>
```

```
}
```

Then let's add a style rule that suits an error message:

```
.error {
  color: red;
  background: lightgrey;
  font-size: 20px;
  border-style: solid;
  border-radius: 5px;
  padding: 10px;
  margin-bottom: 10px;
}
```

Now we are ready to add the logic for displaying the error message. Let's change the `toggleImportanceOf` function in the following way:

```
const toggleImportanceOf = id => {
  const note = notes.find(n => n.id === id)
  const changedNote = [ ...note, important: !note.important ]

  noteService
    .update(id, changedNote).then(returnedNote => [
      setNotes(notes.map(note => note.id !== id ? note : returnedNote))
    ])
    .catch(error => [
      setErrorMessage(`Note '${note.content}' was already removed from server`),
      setTimeout(() => setErrorMessage(null), 5000),
      setNotes(notes.filter(n => n.id !== id))
    ])
}
```

When the error occurs we add a descriptive error message to the `errorMessage` state. At the same time, we start a timer, that will set the `errorMessage` state to `null` after five seconds.

The result looks like this:

A screenshot of a web browser window titled "localhost:3000". The main content area has a red border and displays the text "Note 'This note is not saved to server' was already removed from server". Below this, there is a "show important" button and a list of bullet points. At the bottom right of the content area is a "save" button.

- HTML is easy [make not important](#)
- Browser can execute only Javascript [make important](#)
- GET and POST are the most important methods of HTTP protocol [make not important](#)
- POST is used to add data to a REST api [make important](#)
- Network tab of devtools is most beneficial [make important](#)

[save](#)

The code for the current state of our application can be found in the *part2-7* branch on [GitHub](#).

Inline styles

React also makes it possible to write styles directly in the code as so-called [inline styles](#).

The idea behind defining inline styles is extremely simple. Any React component or element can be provided with a set of CSS properties as a JavaScript object through the [style](#) attribute.

CSS rules are defined slightly differently in JavaScript than in normal CSS files. Let's say that we wanted to give some element the color green and italic font that's 16 pixels in size. In CSS, it would look like this:

```
color: green;
font-style: italic;
font-size: 16px;
```

But as a React inline-style object it would look like this:

```
{ 
  color: 'green',
  fontStyle: 'italic',
  fontSize: 16
}
```

Every CSS property is defined as a separate property of the JavaScript object. Numeric values for pixels can be simply defined as integers. One of the major differences compared to regular CSS, is that hyphenated (kebab case) CSS properties are written in camelCase.

Next, we could add a "bottom block" to our application by creating a *Footer* component and defining the following inline styles for it:

```
const Footer = () => { const footerStyle = { color: 'green', fontStyle: 'italic', fontSize: 16 } return [ <div style={footerStyle}> <br /> <em>Note app, Department of Computer Science, University of Helsinki 2024</em> </div> ] }
const App = () => [
  // ...

  return [
    <div>
      <h1>Notes</h1>

      <Notification message={errorMessage} />

      // ...

      <Footer /> </div>
    ]
]
```

Inline styles come with certain limitations. For instance, so-called [pseudo-classes](#) can't be used straightforwardly.

Inline styles and some of the other ways of adding styles to React components go completely against the grain of old conventions. Traditionally, it has been considered best practice to entirely separate CSS from the content (HTML) and functionality (JavaScript). According to this older school of thought, the goal was to write CSS, HTML, and JavaScript into their separate files.

The philosophy of React is, in fact, the polar opposite of this. Since the separation of CSS, HTML, and JavaScript into separate files did not seem to scale well in larger applications, React bases the division of the application along the lines of its logical functional entities.

The structural units that make up the application's functional entities are React components. A React component defines the HTML for structuring the content, the JavaScript functions for determining functionality, and also the component's styling; all in one place. This is to create individual components that are as independent and reusable as possible.

The code of the final version of our application can be found in the *part2-8* branch on [GitHub](#).

Exercises 2.16.-2.17.

2.16: Phonebook step 11

Use the [improved error message](#) example from part 2 as a guide to show a notification that lasts for a few seconds after a successful operation is executed (a person is added or a number is changed):

Added Juha Tauriainen

filter shown with

add a new

name:
number:

2.17*: Phonebook step 12

Open your application in two browsers. If you delete a person in browser 1 a short while before attempting to change the person's phone number in browser 2, you will get the following error messages:

```
✖ PUT http://localhost:3001/persons/10 404 (Not Found) :3001/persons/10:1
✖ ▶ Uncaught (in promise) Error: Request failed with status code 404
  at createError (createError.js:17)
  at settle (settle.js:19)
  at XMLHttpRequest.handleLoad (xhr.js:78)
```

Fix the issue according to the example shown in [promise and errors](#) in part 2. Modify the example so that the user is shown a message when the operation does not succeed. The messages shown for successful and unsuccessful events should look different:

Information of Edsger Dijkstra has already been removed from server

filter shown with

add a new

name:
number:

Note that even if you handle the exception, the first "404" error message is still printed to the console. But you should not see "Uncaught (in promise) Error".

Couple of important remarks

At the end of this part there are a few more challenging exercises. At this stage, you can skip the exercises if they are too much of a headache, we will come back to the same themes again later. The material is worth reading through in any case.

We have done one thing in our app that is masking away a very typical source of error.

We set the state *notes* to have initial value of an empty array:

```
const App = () => {
  const [notes, setNotes] = useState([])

  // ...
}
```

This is a pretty natural initial value since the notes are a set, that is, there are many notes that the state will store.

If the state would be only saving "one thing", a more proper initial value would be *null* denoting that there is *nothing* in the state at the start. Let us try what happens if we use this initial value:

```
const App = () => {
  const [notes, setNotes] = useState(null)
  // ...
}
```

The app breaks down:

```
✖ ▶ Uncaught TypeError: Cannot read properties of null (reading 'map')
  at App (App.js:53:1)
  at renderWithHooks (react-dom.development.js:16305:1)
  at mountIndeterminateComponent (react-dom.development.js:20074:1)
  at beginWork (react-dom.development.js:21587:1)
  at HTMLUnknownElement.callCallback (react-dom.development.js:4164:1)
  at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:1)
  at invokeGuardedCallback (react-dom.development.js:4277:1)
  at beginWork$1 (react-dom.development.js:27451:1)
  at performUnitOfWork (react-dom.development.js:26557:1)
  at workLoopSync (react-dom.development.js:26466:1)
```

The error message gives the reason and location for the error. The code that caused the problems is the following:

```
// notesToShow gets the value of notes
const notesToShow = showAll
  ? notes
  : notes.filter(note => note.important)

// ...

{notesToShow.map(note =>      <Note key={note.id} note={note} />
)}
```

The error message is

```
Cannot read properties of null (reading 'map')
```

The variable `notesToShow` is first assigned the value of the state `notes` and then the code tries to call method `map` to a nonexisting object, that is, to `null`.

What is the reason for that?

The effect hook uses the function `setNotes` to set `notes` to have the notes that the backend is returning:

```
useEffect(() => {
  noteService
    .getAll()
    .then(initialNotes => {
      setNotes(initialNotes)
    })
}, [])
```

However the problem is that the effect is executed only *after the first render*. And because `notes` has the initial value of `null`:

```
const App = () => {
  const [notes, setNotes] = useState(null)
  // ...
```

on the first render the following code gets executed:

```
notesToShow = notes

// ...

notesToShow.map(note => ...)
```

and this blows up the app since we can not call method `map` of the value `null`.

When we set `notes` to be initially an empty array, there is no error since it is allowed to call `map` to an empty array.

So, the initialization of the state "masked" the problem that is caused by the fact that the data is not yet fetched from the backend.

Another way to circumvent the problem is to use *conditional rendering* and return `null` if the component state is not properly initialized:

```
const App = () => {
  const [notes, setNotes] = useState(null) // ...

  useEffect(() => {
    noteService
```

```

    .getAll()
    .then(initialNotes => {
      setNotes(initialNotes)
    })
  , [])
}

// do not render anything if notes is still null
if (!notes) { return null }
// ...
}

```

So on the first render, nothing is rendered. When the notes arrive from the backend, the effect used function `setNotes` to set the value of the state `notes`. This causes the component to be rendered again, and at the second render, the notes get rendered to the screen.

The method based on conditional rendering is suitable in cases where it is impossible to define the state so that the initial rendering is possible.

The other thing that we still need to have a closer look is the second parameter of the `useEffect`:

```

useEffect(() => {
  noteService
    .getAll()
    .then(initialNotes => {
      setNotes(initialNotes)
    })
  , [])
}

```

The second parameter of `useEffect` is used to [specify how often the effect is run](#). The principle is that the effect is always executed after the first render of the component *and* when the value of the second parameter changes.

If the second parameter is an empty array `[]`, its content never changes and the effect is only run after the first render of the component. This is exactly what we want when we are initializing the app state from the server.

However, there are situations where we want to perform the effect at other times, e.g. when the state of the component changes in a particular way.

Consider the following simple application for querying currency exchange rates from the [Exchange rate API](#):

```

import { useState, useEffect } from 'react'
import axios from 'axios'

const App = () => {
  const [value, setValue] = useState('')
  const [rates, setRates] = useState({})
  const [currency, setCurrency] = useState(null)

  useEffect(() => {
    console.log('effect run, currency is now', currency)
  })
}

```

```

// skip if currency is not defined
if (!currency) {
  console.log('fetching exchange rates...')
  axios
    .get(`https://open.er-api.com/v6/latest/${currency}`)
    .then(response => [
      setRates(response.data.rates)
    ])
}
}, [currency])

const handleChange = (event) => {
  setValue(event.target.value)
}

const onSearch = (event) => [
  event.preventDefault(),
  setCurrency(value)
]

return [
  <div>
    <form onSubmit={onSearch}>
      currency: <input value={value} onChange={handleChange} />
      <button type="submit">exchange rate</button>
    </form>
    <pre>
      {JSON.stringify(rates, null, 2)}
    </pre>
  </div>
]
}

export default App

```

The user interface of the application has a form, in the input field of which the name of the desired currency is written. If the currency exists, the application renders the exchange rates of the currency to other currencies:

The screenshot shows a browser developer tools console at <http://localhost:3000>. A form has 'eur' entered into a field, and a button labeled 'exchange rate' is visible. Below the form, the browser's developer tools are open, specifically the 'Console' tab. The console displays the following log statements:

```

{
  "EUR": 1,
  "AED": 3.973561,
  "AFN": 97.215051,
  "ALL": 116.195247,
  "AMD": 429.346071,
  "ANG": 1.936739,
}
effect run, currency is now null
effect run, currency is now usd
fetching exchange rates...
effect run, currency is now eur
fetching exchange rates...

```

The application sets the name of the currency entered to the form to the state `currency` at the moment the button is pressed.

When the `currency` gets a new value, the application fetches its exchange rates from the API in the effect function:

```

const App = () => [
  // ...
  const [currency, setCurrency] = useState(null)

  useEffect(() => [
    console.log('effect run, currency is now', currency)

    // skip if currency is not defined
    if (currency) {
      console.log('fetching exchange rates...')
      axios
        .get(`https://open.er-api.com/v6/latest/${currency}`)
        .then(response => {
          setRates(response.data.rates)
        })
    }
  ], [currency]) // ...
]

```

The `useEffect` hook now has `[currency]` as the second parameter. The effect function is therefore executed after the first render, and *always* after the table as its second parameter `[currency]` changes. That is, when the state `currency` gets a new value, the content of the table changes and the effect function is executed.

The effect has the following condition

```
if (currency) {
```

```
// exchange rates are fetched
```

which prevents requesting the exchange rates just after the first render when the variable `currency` still has the initial value, i.e. a null value.

So if the user writes e.g. `eur` in the search field, the application uses Axios to perform an HTTP GET request to the address <https://open.er-api.com/v6/latest/eur> and stores the response in the `rates` state.

When the user then enters another value in the search field, e.g. `usd`, the effect function is executed again and the exchange rates of the new currency are requested from the API.

The way presented here for making API requests might seem a bit awkward. This particular application could have been made completely without using the `useEffect`, by making the API requests directly in the form submit handler function:

```
const onSearch = [event] => {
  event.preventDefault()
  axios
    .get(`https://open.er-api.com/v6/latest/${value}`)
    .then(response => [
      setRates(response.data.rates)
    ])
}
```

However, there are situations where that technique would not work. For example, you *might* encounter one such a situation in the exercise 2.20 where the use of `useEffect` could provide a solution. Note that this depends quite much on the approach you selected, e.g. the model solution does not use this trick.

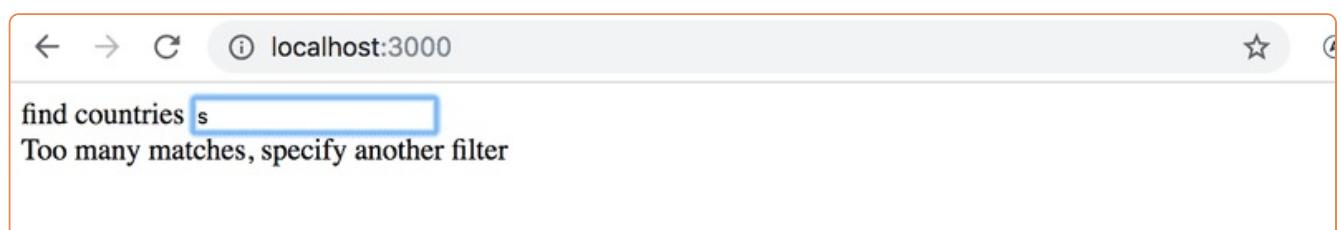
Exercises 2.18.-2.20.

2.18* Data for countries, step 1

At <https://studies.cs.helsinki.fi/restcountries/> you can find a service that offers a lot of information related to different countries in a so-called machine-readable format via the REST API. Make an application that allows you to view information from different countries.

The user interface is very simple. The country to be shown is found by typing a search query into the search field.

If there are too many (over 10) countries that match the query, then the user is prompted to make their query more specific:



If there are ten or fewer countries, but more than one, then all countries matching the query are shown:

A screenshot of a web browser window titled "localhost:3000". The search bar contains the query "find countries sw". Below the search bar, a list of four countries is displayed: Botswana, Swaziland, Sweden, and Switzerland. The word "Switzerland" is preceded by a small star icon.

When there is only one country matching the query, then the basic data of the country (eg. capital and area), its flag and the languages spoken are shown:

A screenshot of a web page showing details for Switzerland. The search bar at the top contains "find countries swi". The main title is "Switzerland". Below the title, it says "capital Bern" and "area 41284". Under the heading "languages:", there is a bulleted list: "• French", "• Swiss German", "• Italian", and "• Romansh". To the left of the text, there is a large red square flag of Switzerland with a white cross in the center.

NB: It is enough that your application works for most countries. Some countries, like *Sudan*, can be hard to support since the name of the country is part of the name of another country, *South Sudan*. You don't need to worry about these edge cases.

2.19*: Data for countries, step 2

There is still a lot to do in this part, so don't get stuck on this exercise!

Improve on the application in the previous exercise, such that when the names of multiple countries are shown on the page there is a button next to the name of the country, which when pressed shows the view for that country:

localhost:3000

find countries sw

Botswana show

Swaziland show

Sweden show

Switzerland show

In this exercise, it is also enough that your application works for most countries. Countries whose name appears in the name of another country, like *Sudan*, can be ignored.

2.20*: Data for countries, step 3

Add to the view showing the data of a single country, the weather report for the capital of that country. There are dozens of providers for weather data. One suggested API is <https://openweathermap.org>. Note that it might take some minutes until a generated API key is valid.

find countries fin

Finland

capital Helsinki
area 338424

languages:

- Finnish
- Swedish



Weather in Helsinki

temperature -3.73 Celcius



wind 1.34 m/s

If you use Open weather map, [here](#) is the description for how to get weather icons.

NB: In some browsers (such as Firefox) the chosen API might send an error response, which indicates that HTTPS encryption is not supported, although the request URL starts with *http://*. This issue can be fixed by completing the exercise using Chrome.

NB: You need an api-key to use almost every weather service. Do not save the api-key to source control! Nor hardcode the api-key to your source code. Instead use an [environment variable](#) to save the key.

Assuming the api-key is *54l41n3n4v41m34rv0*, when the application is started like so:

```
export VITE_SOME_KEY=54l41n3n4v41m34rv0 && npm run dev // For Linux/macOS Bash  
[$env:VITE_SOME_KEY="54l41n3n4v41m34rv0"] -and [npm run dev] // For Windows PowerShell  
set "VITE_SOME_KEY=54l41n3n4v41m34rv0" && npm run dev // For Windows cmd.exe
```

you can access the value of the key from the *import.meta.env* object:

```
const api_key = import.meta.env.VITE_SOME_KEY  
// variable api_key now has the value set in startup
```

Note that you will need to restart the server to apply the changes.

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](#).

3: Programming a server with NodeJS and Express

3a: 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

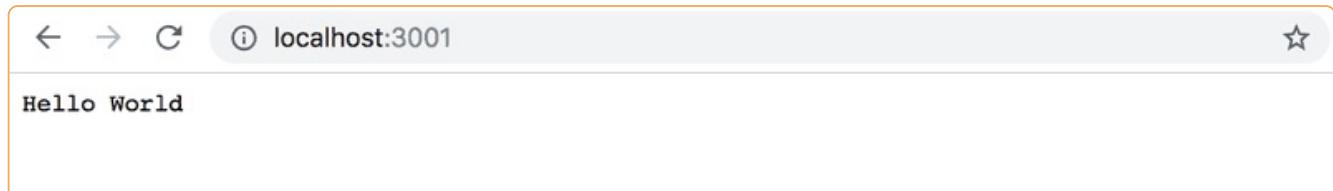
Let's change the application into a web server by editing the `index.js` file as follows:

```
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:

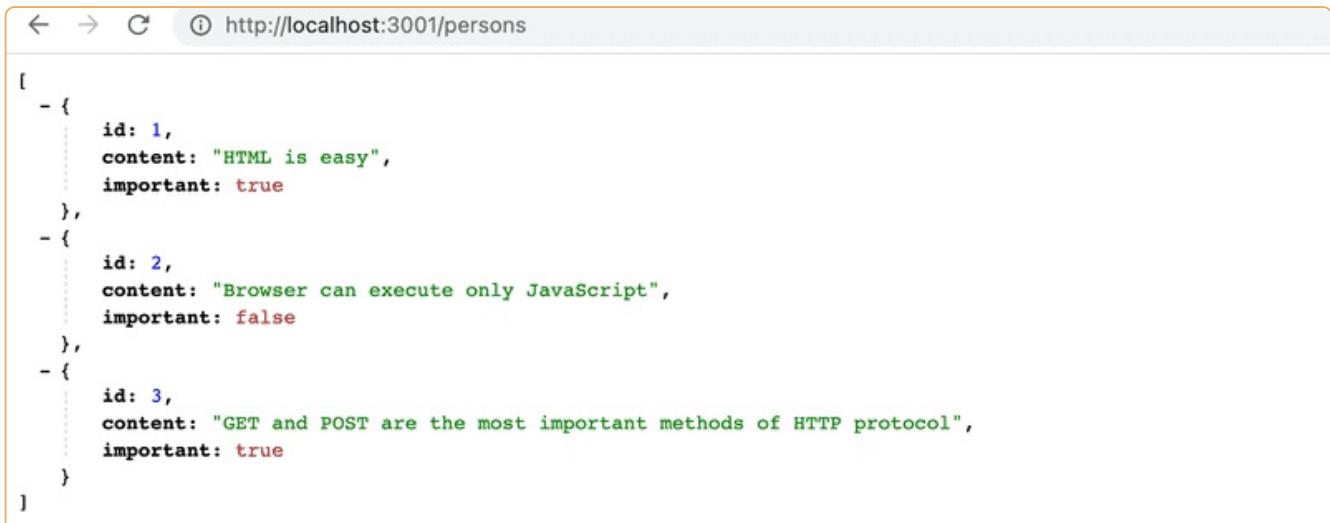
```
const http = require('http')

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:



A screenshot of a web browser window. The address bar shows the URL `http://localhost:3001/persons`. The page content displays a JSON array of three objects, each representing a note. The notes are: {id: 1, content: "HTML is easy", important: true}, {id: 2, content: "Browser can execute only JavaScript", important: false}, and {id: 3, content: "GET and POST are the most important methods of HTTP protocol", important: true}.

```
[ { 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 } ]
```

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
```

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

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

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:

```
→ node_modules ls
accepts          encodeurl        merge-descriptors  range-parser
array-flatten    escape-html      methods           raw-body
body-parser      etag             mime              safe-buffer
bytes            express           mime-db          safer-buffer
content-disposition finalhandler   mime-types       send
content-type     forwarded        ms                serve-static
cookie           fresh            negotiator      setprototypeof
cookie-signature http-errors      on-finished     statuses
debug            iconv-lite       parseurl         type-is
depd             inherits         path-to-regexp  unpipe
destroy          ipaddr.js       proxy-addr      utils-merge
ee-first          media-typer      qs               vary
→ node_modules
```

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"
```

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:

```
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 screenshot shows the Network tab in developer tools with a single request listed. The request is to 'localhost' at port 3001, method 'GET', and status '200 OK'. The response headers include 'Connection: keep-alive', 'Content-Length: 21', and 'Content-Type: text/html; charset=utf-8'. The response body is '<h1>Hello World!</h1>'.

Name	Headers	Preview	Response	Cookies	Timing
localhost	<ul style="list-style-type: none">General<ul style="list-style-type: none">Request URL: http://localhost:3001/Request Method: GETStatus Code: 200 OKRemote Address: [::1]:3001Referrer Policy: no-referrer-when-downgradeResponse Headers<ul style="list-style-type: none">view sourceConnection: keep-aliveContent-Length: 21Content-Type: text/html; charset=utf-8				
1 requests 226 B transferred Fin...					

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`.

The screenshot shows a browser's developer tools Network tab with a request for `/api/notes`. The response is a JSON object containing three 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}]
```

The Headers section shows the following details:

- Request URL: `http://localhost:3001/api/notes`
- Request Method: GET
- Status Code: 200 OK
- Remote Address: `::1:3001`
- Referrer Policy: strict-origin-when-cross-origin

The Response Headers section shows the following headers:

- Connection: keep-alive
- Content-Length: 226
- Content-Type: application/json; charset=utf-8 (highlighted with a red box)
- Date: Wed, 18 Jan 2023 08:03:42 GMT
- ETag: W/"e2-88HXx35CxplEbdoGXpmsfhcPEM"
- Keep-Alive: timeout=5
- X-Powered-By: Express

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))
```

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:

```
X node (node) ⌘1 X ..kai.github.io (zsh) ⌘2 X bundle (fsevent_wat... ⌘3
> person
{ name: 'Arto', age: 35 }
> const json = JSON.stringify(person)
undefined
> json
'{"name":"Arto","age":35}'
> typeof person
'object'
> typeof json
'string'
>
```

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
```

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

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

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
```

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"
  },
  // ...
}
```

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
```

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)
  })
}
```

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
```

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 screenshot shows the Network tab in the Chrome DevTools developer console. A single request is listed under the 'Name' column, labeled '100'. The 'Headers' tab is selected. Under the 'General' section, the Request URL is 'http://localhost:3001/api/notes/100', the Request Method is 'GET', and the Status Code is '200 OK'. The 'Content-Length' header is explicitly highlighted with a red box, showing its value as '0'. Other headers listed include 'Connection: keep-alive' and 'Content-Type: application/json; charset=utf-8'.

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:

The screenshot shows the Postman application interface. At the top, there's a header bar with a 'DEL' button, the URL 'http://localhost:3001/api/notes/2', and a 'No Environment' dropdown. Below the header is a toolbar with 'Save' and other icons. The main area has a 'DELETE' button highlighted with a red box, followed by the URL 'http://localhost:3001/api/notes/2'. To the right are 'Send' and 'Send' dropdown buttons. Below the URL are tabs for 'Params', 'Authorization', 'Headers (7)', 'Body', 'Pre-request Script', 'Tests', 'Settings', and 'Cookies'. Under 'Params', there's a 'Query Params' table:

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

At the bottom, there are tabs for 'Body', 'Cookies', 'Headers (4)', and 'Test Results'. The 'Test Results' tab is selected and shows a status bar with '204 No Content', '10 ms', '134 B', and a 'Save Response' button. Below the status bar are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'Text' (with a dropdown arrow). On the far right of the bottom bar are search and refresh icons.

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.

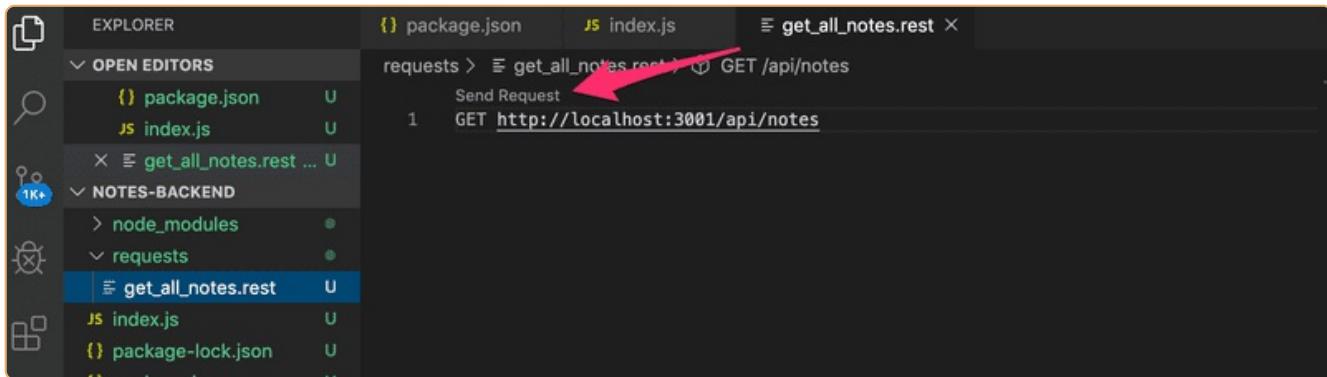
Because the notes in the application are only saved to memory, the list of notes will return to its original state when we restart the application.

The Visual Studio Code REST client

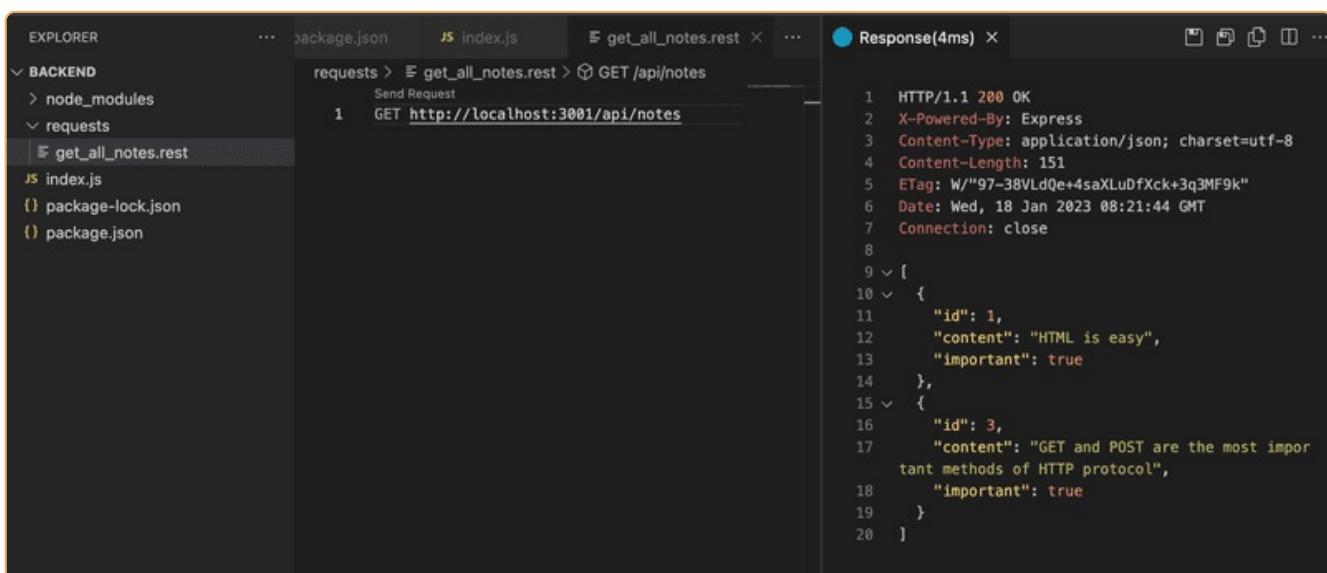
If you use Visual Studio Code, you can use the VS Code [REST client](#) plugin instead of Postman.

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())

```

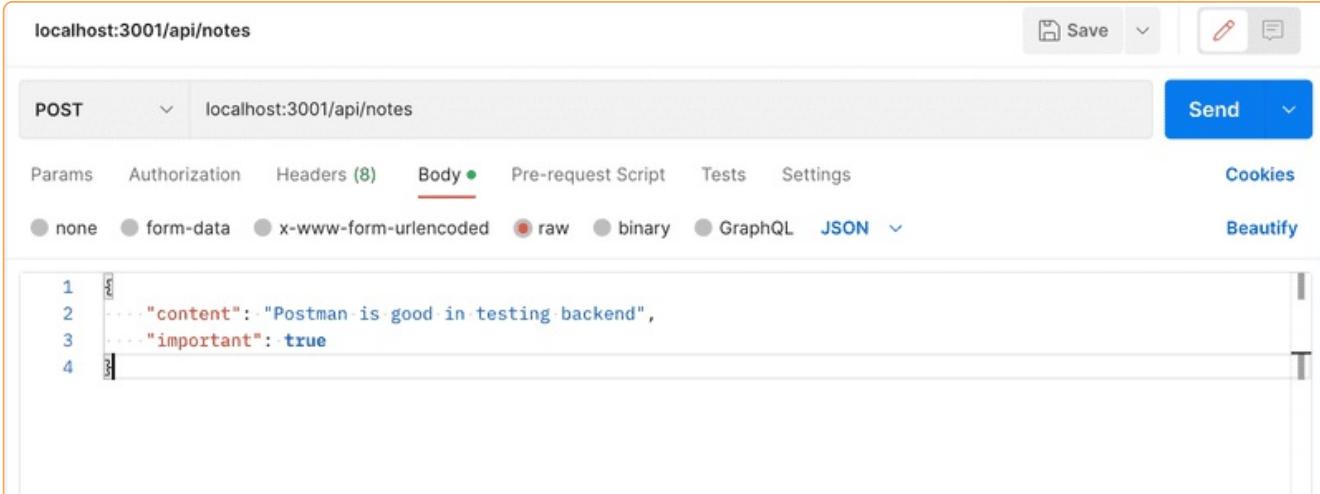
```
//...  
  
app.post('/api/notes', (request, response) => { const note = request.body console.log(note) response.json(note) })
```

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 screenshot shows the Postman interface with a POST request to `localhost:3001/api/notes`. The `Body` tab is selected, displaying the following JSON payload:

```
1 {  
2   "content": "Postman is good in testing backend",  
3   "important": true  
4 }
```

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 screenshot shows a Postman request configuration. The method is POST, the URL is `localhost:3001/api/notes`, and the body is set to `Text`. The request body contains the following JSON:

```
1 {
2   "content": "do not forget to set the correct content type!",
3   "important": true
4 }
```

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

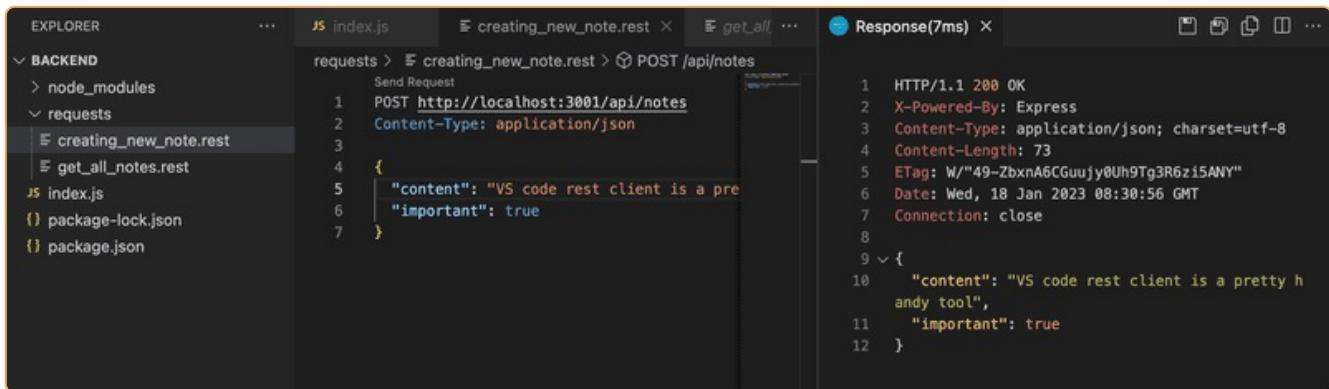
The screenshot shows the Postman Headers section. The `Content-Type` header is listed and highlighted with a red box. Other headers shown include `Postman-Token`, `Content-Length`, `Host`, and `User-Agent`.

The server appears to only receive an empty object:

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

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:



```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 73
ETag: W/"49-ZbxnA6CGuujy0Uh9Tg3R6zi5ANY"
Date: Wed, 18 Jan 2023 08:30:56 GMT
Connection: close
{
    "content": "VS code rest client is a pretty h
    andy tool",
    "important": true
}
```

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
```

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](#).

The screenshot shows a GitHub repository page for the 'part3-1' branch. At the top, there are statistics: 1 commit, 2 branches, 0 releases, and 1 contributor. Below this, there are buttons for 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. A 'Switch branches/tags' dropdown menu is open, showing 'master' and 'part3-1' with 'part3-1' selected. To the right, a list of commits is shown, all made 15 minutes ago. The commits are: 'initial commit', 'initial commit', 'initial commit', 'initial commit', and 'initial commit'. At the bottom of the commit list, there are links for 'package-lock.json' and 'package.json'.

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
  return maxId + 1
]
```

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

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

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:

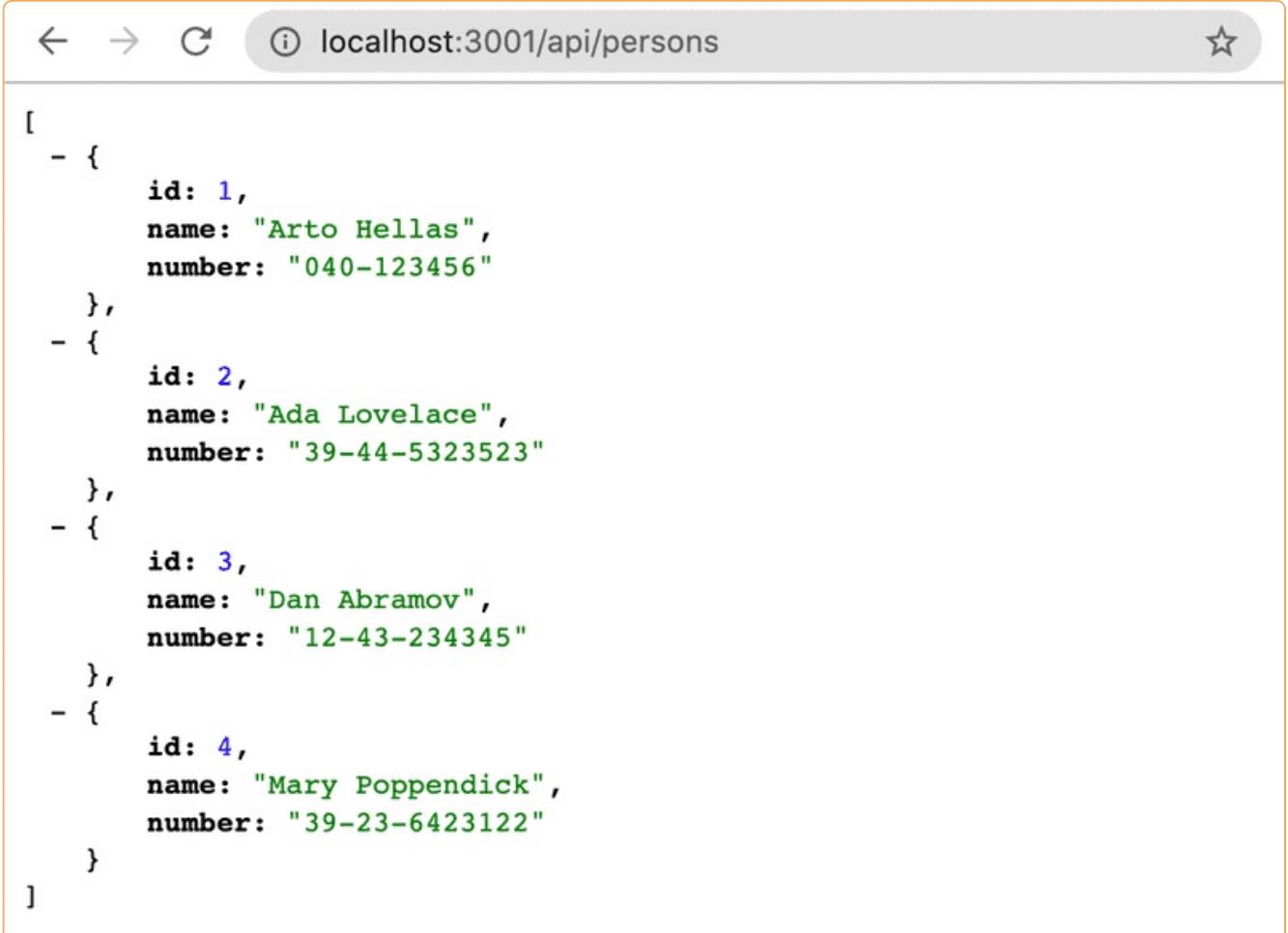
```
[[
  {
    "id": 1,
    "name": "Arto Hellas",
    "number": "040-123456"
  },
  {
    "id": 2,
    "name": "Ada Lovelace",
    "number": "39-44-5323523"
  }
]
```

```

    "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:



The screenshot shows a browser window with the address bar containing "localhost:3001/api/persons". The main content area displays a JSON array of four objects, each representing a person with properties id, name, and number.

```

[
  {
    "id": 1,
    "name": "Arto Hellas",
    "number": "040-123456"
  },
  {
    "id": 2,
    "name": "Ada Lovelace",
    "number": "39-44-5323523"
  },
  {
    "id": 3,
    "name": "Dan Abramov",
    "number": "12-43-234345"
  },
  {
    "id": 4,
    "name": "Mary Poppendick",
    "number": "39-23-6423122"
  }
]

```

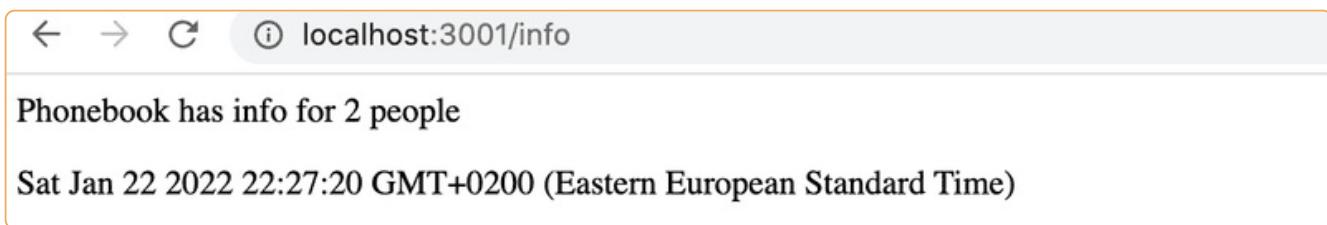
Notice that the forward slash in the route `api/persons` is not a special character, and is just like any other character in the string.

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 `
` 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' }
```

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()
}
```

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)
```

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](#)

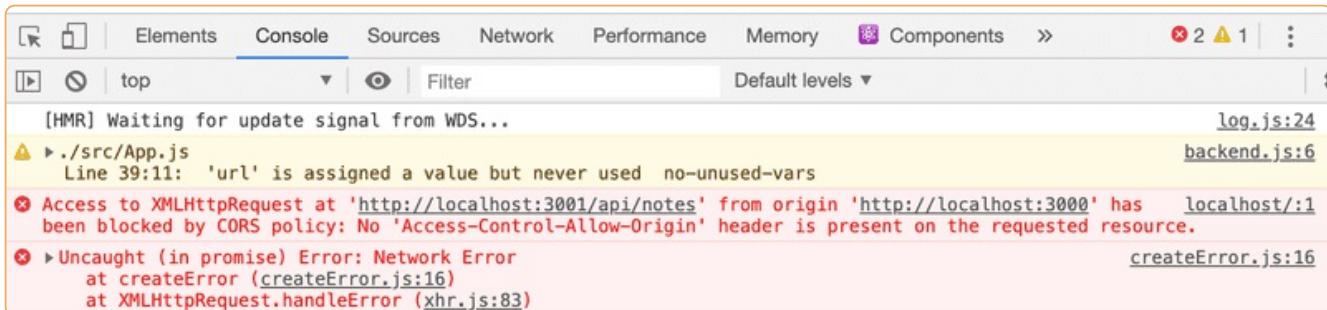
3b: Deploying app to internet

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
```

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

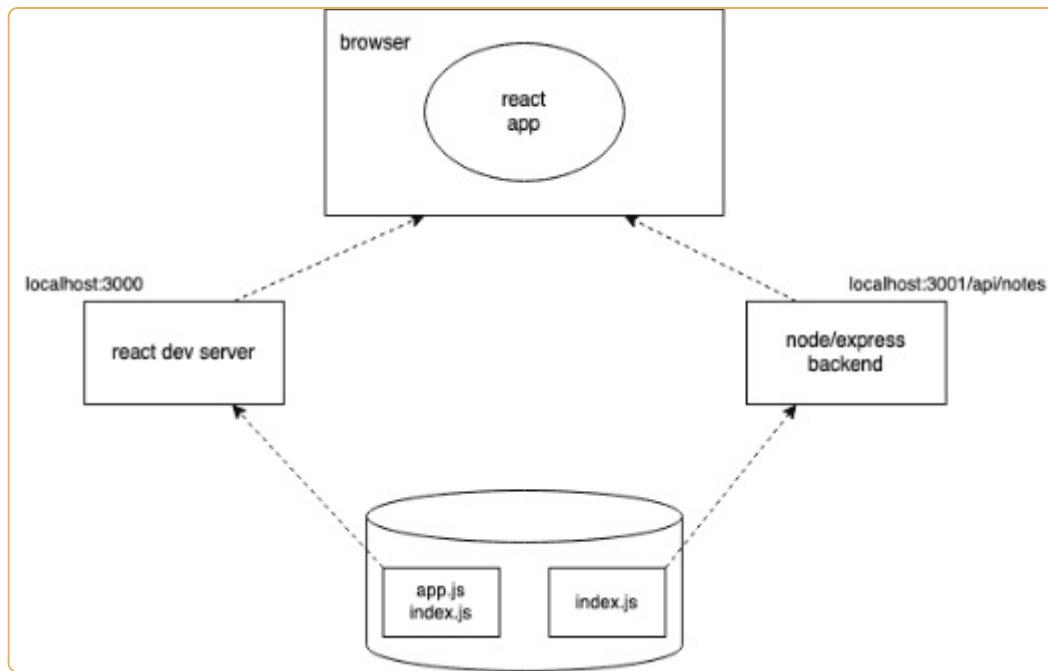
```
const cors = require('cors')

app.use(cors())
```

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](#)
- [Repl.it](#)
- [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}`)
})
```

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
```

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
```

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

[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
```

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
```

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
```

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 fdःaa:0:8a3d::3 [gateway], seq=0 time=65.1ms
35 bytes from fdःaa:0:8a3d::3 [gateway], seq=1 time=28.5ms
35 bytes from fdःaa:0:8a3d::3 [gateway], seq=2 time=29.3ms
...
...
```

then there are no connection problems!

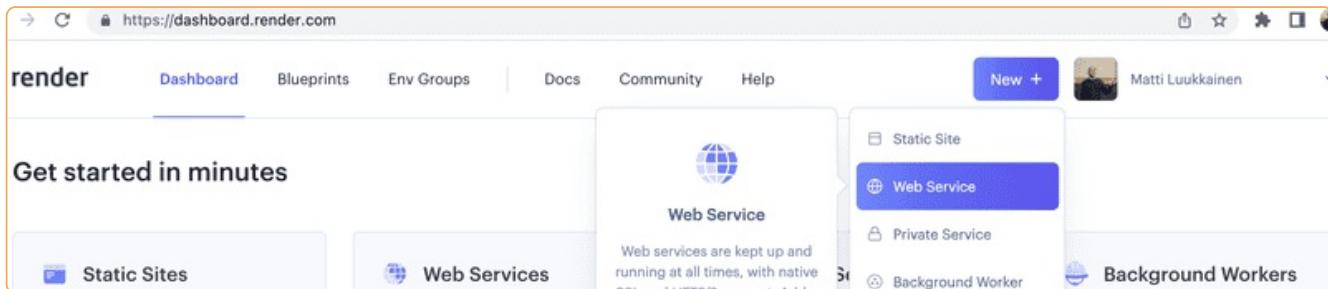
Whenever you make changes to the application, you can take the new version to production with a command

```
fly deploy
```

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 screenshot shows the Render dashboard at <https://dashboard.render.com/select-repo?type=web>. The top navigation bar includes links for render, Dashboard, Blueprints, Env Groups, Docs, Community, and Help. The main content area is titled "Public Git repository" and contains a note: "Use a public repository by entering the URL below. Features like PR Previews and Auto-Deploy are not available if the repository has not been configured for Render." Below this is a text input field containing the URL <https://github.com/mluukkai/render-test>, with a blue "Continue" button to its right.

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:

The screenshot shows the Render "New" web service setup form at <https://dashboard.render.com/web/new>. The top navigation bar includes links for render, Dashboard, Blueprints, Env Groups, Docs, Community, Help, and a "New +" button. The user is Matti Luukkainen. The form fields are as follows:

- Name:** render-test
- Region:** Frankfurt (EU Central)
- Branch:** main
- Root Directory** (Optional): e.g. src
- Environment:** Node
- Build Command:** \$ npm install
- Start Command:** \$ npm start

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

The screenshot shows the Render dashboard for the 'render-test' service. At the top, it displays the service name, instance type (Node), plan (Free Plan), repository (mluukkai/render-test), branch (main), and deployment status (Up to date). Below this, there are sections for Events, Logs, Disks, Environment, Shell, PRs, Jobs, Metrics, Scaling, and Settings. The Logs section shows a log entry from January 18, 2023, at 11:48 AM, indicating an initial build for commit 81a315c. A search bar is available for logs. On the right, there are buttons for 'Connect' and 'Manual Deploy'.

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:

The screenshot shows the Render dashboard for the 'render-test' service. It includes the same top navigation and service details as the previous screenshot. The 'Events' section shows a recent deployment for commit 81a315c. To the right of the deployment log, a dropdown menu is open with three options: 'Deploy latest commit' (which is highlighted with a red border), 'Deploy a specific commit', and 'Clear build cache & deploy'.

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

The screenshot shows the Render Test interface. On the left, a sidebar lists various log categories: Events, Logs (which is selected and highlighted with a red box), Disks, Environment, Shell, PRs, Jobs, Metrics, Scaling, and Settings. At the top, it shows the project name 'render-test' and the URL 'https://render-test-yu7p.onrender.com'. A search bar with the placeholder 'Search logs' and a blue 'Search' button are also present. The main area displays log entries from January 18, 2024, at 12:04:41 PM. The logs show the application starting, listening on port 10000, and handling multiple GET requests to the root path. A pink arrow points to the first log entry where it says 'Server running on port 10000'.

```
Jan 18 12:04:41 PM > node index.js
Jan 18 12:04:41 PM
Jan 18 12:04:43 PM Server running on port 10000
Jan 18 12:04:48 PM Method: GET
Jan 18 12:04:48 PM Path: /
Jan 18 12:04:48 PM Body: {}
Jan 18 12:04:48 PM ---
Jan 18 12:04:57 PM ==> Starting service with 'npm start'
Jan 18 12:05:00 PM
Jan 18 12:05:00 PM > render-test@1.0.0 start /opt/render/project/src
Jan 18 12:05:00 PM > node index.js
Jan 18 12:05:00 PM
Jan 18 12:05:01 PM Server running on port 10000
Jan 18 12:05:28 PM Method: GET
Jan 18 12:05:28 PM Path: /
Jan 18 12:05:28 PM Body: {}
Jan 18 12:05:28 PM ---
```

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=[],ci.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],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

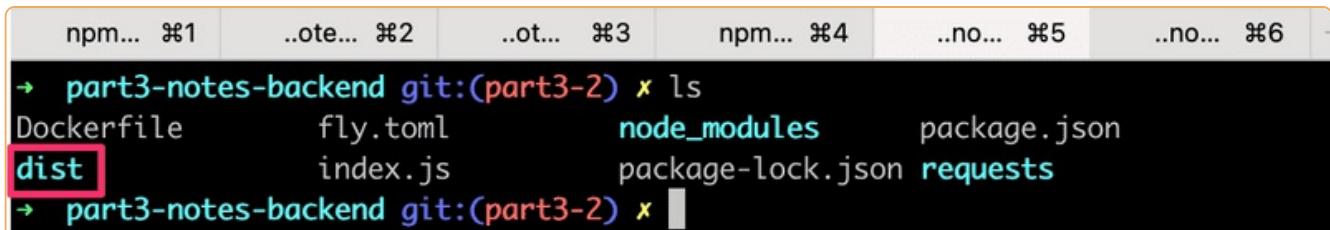
One option for deploying the frontend is to copy the production build (the `dist` directory) to the root of the backend repository and configure the backend to show the frontend's *main page* (the file `dist/index.html`) as its main page.

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:



```
npm... #1 ..ote... #2 ..ot... #3 npm... #4 ..no... #5 ..no... #6
→ part3-notes-backend git:(part3-2) ✘ ls
Dockerfile          fly.toml        node_modules    package.json
dist                index.js       package-lock.json requests
→ part3-notes-backend git:(part3-2) ✘
```

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'))
```

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.

```

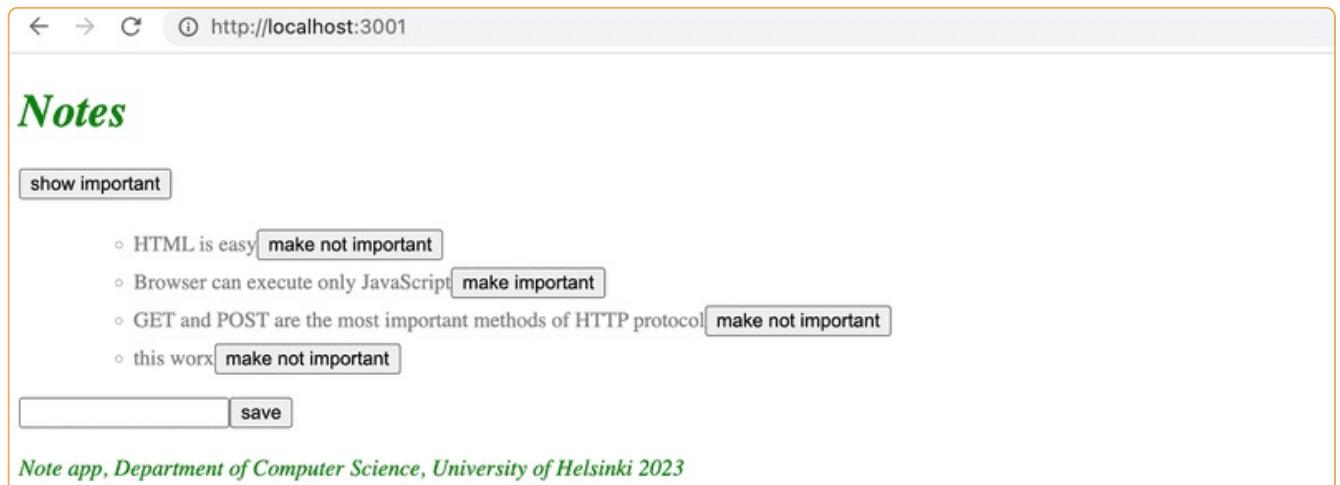
import axios from 'axios'
const baseUrl = '/api/notes'
const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}

// ...

```

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>

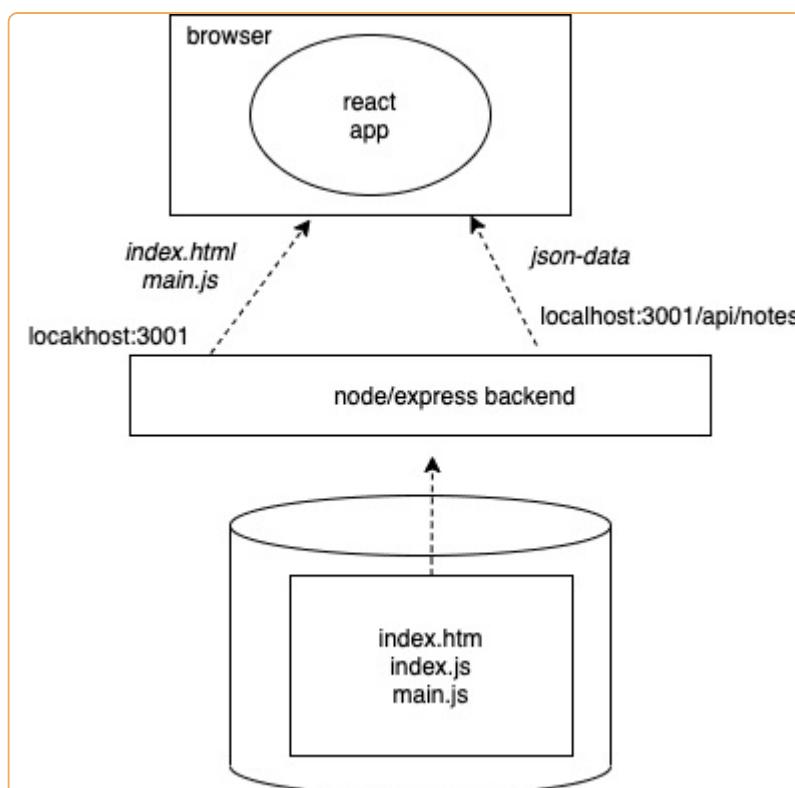
```

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 screenshot shows a browser window with a green header "Notes". Below it is a list of items with "make not important" buttons next to some of them. A "save" button is at the bottom. Below the browser is the developer tools Network tab. It shows a request to "Request URL: http://localhost:3001/api/notes". The response details show: Request Method: GET, Status Code: 200 OK, Remote Address: [::1]:3001, and Referrer Policy: strict-origin-when-cross-origin.

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
```

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.

The screenshot shows a web browser window with the URL <https://notes2023.fly.dev>. The page title is "Notes". There is a button labeled "show important". Below it is a list of three items, each with a "make not important" button:

- HTML is easy
- Browser can execute only JavaScript
- GET and POST are the most important methods of HTTP protocol

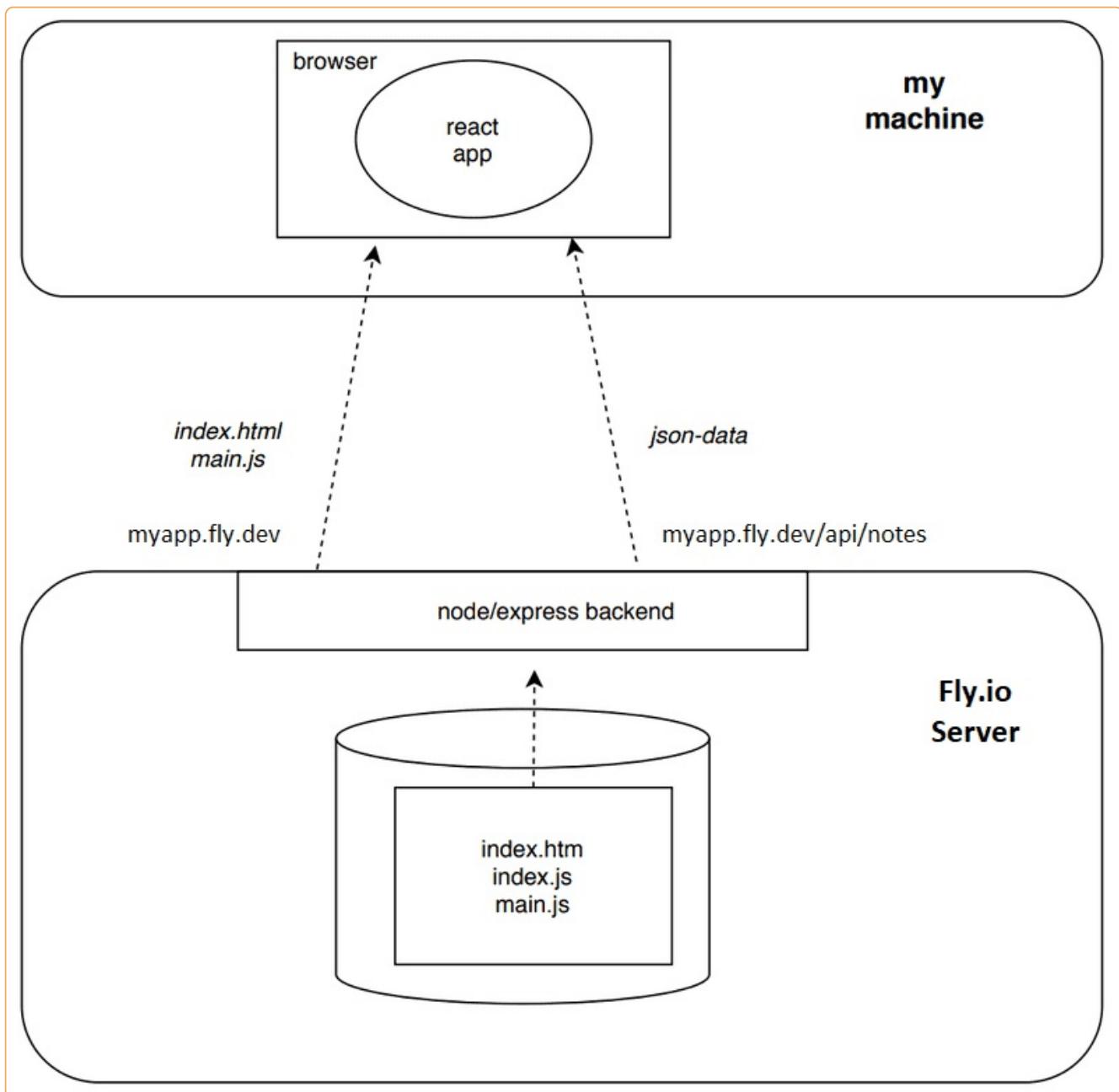
Below the list is a "save" button. At the bottom of the page, there is a footer that reads "Note app, Department of Computer Science, University of Helsinki 2023".

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 && n
pm 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 pathpackage.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 && gi
t 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"
```

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.

The screenshot shows a browser window with the URL `http://localhost:3000`. The page title is **Notes app**. Below the title are two buttons: **show important** and **save**. The main content area displays the text **Note app, Department of Computer Science, University of Helsinki 2023**. At the bottom of the page, there is a footer with links to various departments like Elements, Console, Sources, Network, Performance, Memory, Application, etc. The **Network** tab is currently selected. In the Network tab, there is a table with one row. The row has a red border and contains the following information: Request URL: `http://localhost:3000/api/notes`, Request Method: `GET`, Status Code: `404 Not Found`, Remote Address: `127.0.0.1:3000`, and Referrer Policy: `strict-origin-when-cross-origin`.

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](#).

3c: 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):

The screenshot shows the Visual Studio Code interface with the 'Run' menu open. The 'Start Debugging' option is highlighted. The code editor displays a portion of `index.js` containing a function that logs request details to the console.

```

26 const requestLogger = (request, response, next) => {
27   console.log('Method:', request.method)
28   console.log('Path:', request.path)
29   console.log('Body:', request.body)
30   console.log('---')

```

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 screenshot shows Visual Studio Code in debug mode. The left sidebar displays the 'VARIABLES' pane, which is expanded to show 'Local' variables. The 'CALL STACK' pane shows the current stack trace, indicating the program is 'PAUSED ON BREAKPOINT'. The code editor shows a portion of `index.js` with a red box highlighting the line where the breakpoint was hit: `notes = notes.concat(note)`. The status bar at the bottom indicates the server is running on port 3001.

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

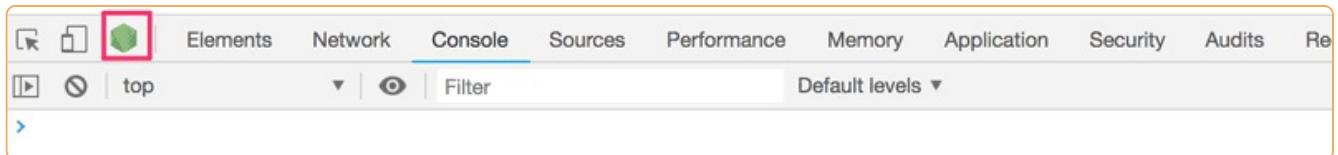
Debugging is also possible with the Chrome developer console by starting your application with the command:

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

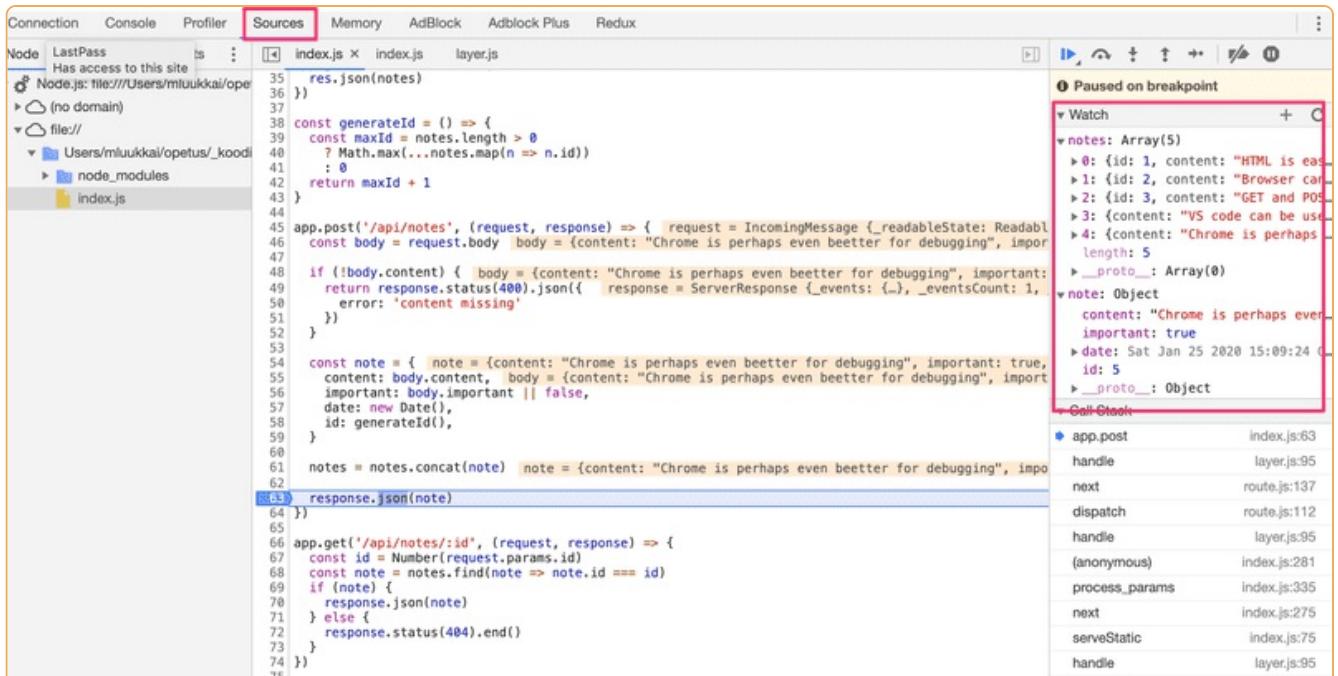
You can also pass the `--inspect` flag to `nodemon`:

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

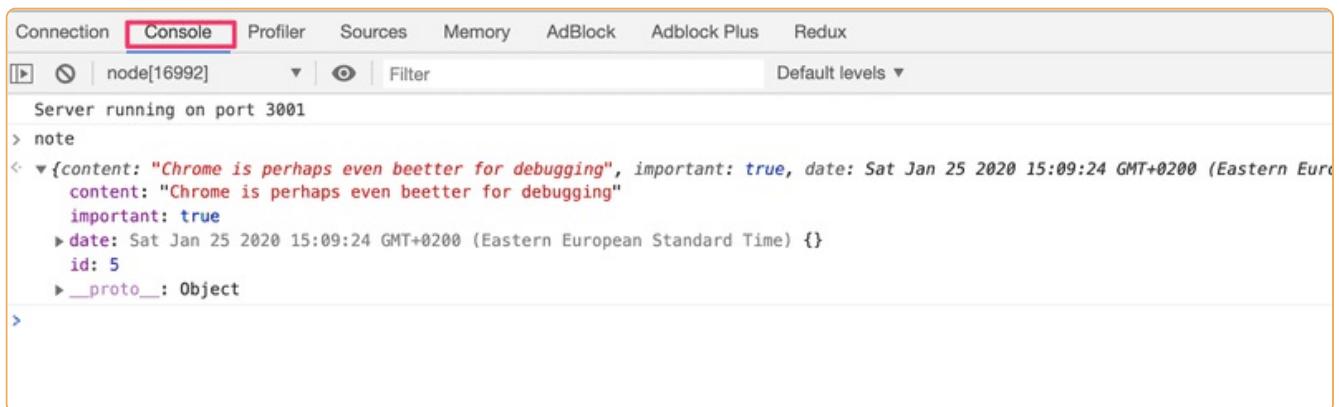
You can access the debugger by clicking the green icon - the node logo - that appears in the Chrome developer console:



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.



A screenshot of a browser's developer tools Console tab. The tab is titled 'Console' and has other tabs like 'Profiler', 'Sources', 'Memory', 'AdBlock', 'Adblock Plus', and 'Redux'. The console output shows:

```
Server running on port 3001
> note
< {content: "Chrome is perhaps even better for debugging", important: true, date: Sat Jan 25 2020 15:09:24 GMT+0200 (Eastern European Standard Time), id: 5}
  content: "Chrome is perhaps even better for debugging"
  important: true
  date: Sat Jan 25 2020 15:09:24 GMT+0200 (Eastern European Standard Time) {}
  id: 5
  __proto__: Object
>
```

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:

Deploy a cloud database

Experience the best of MongoDB on AWS, Azure, and Google Cloud. Choose a deployment option to get started.

PREVIEW

Serverless

For serverless applications that aren't critical with variable traffic. Minimal configuration required.

- ✓ Pay only for the operations you run
- ✓ Resources scale seamlessly to meet your workload
- ✓ Always-on security and backups

[Create](#)

Starting at
\$0.30/1M reads

ADVANCED

Dedicated

For production applications with sophisticated workload requirements. Advanced configuration controls.

- ✓ Network isolation and fine-grained access controls
- ✓ On-demand performance advice
- ✓ Multi-region and multi-cloud options available

[Create](#)

Starting at
\$0.08/hr*

*estimated cost \$56.94/month

FREE

Shared

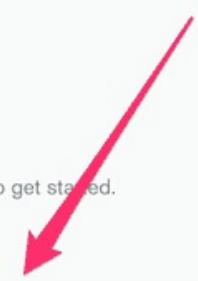
For learning and exploring MongoDB in a cloud environment. Basic configuration options.

- ✓ No credit card required to start
- ✓ Explore with sample datasets
- ✓ Upgrade to dedicated clusters for full functionality

[Create](#)

Starting at
FREE

Pick the cloud provider and location and create the cluster:



PREVIEW Serverless

Dedicated

FREE Shared

For learning and exploring MongoDB in a sandbox environment. Basic configuration controls.

No credit card required to start. Upgrade to dedicated clusters for full functionality.
Explore with sample datasets. Limit of one free cluster per project.

Cloud Provider & Region

AWS, Stockholm (eu-north-1) ▾



Google Cloud

Azure

★ Recommended region ⓘ ⚡ Paid tier region ⓘ

NORTH AMERICA

🇺🇸 Oregon (us-west-2) ★

🇺🇸 N. Virginia (us-east-1) ★

🇺🇸 Ohio (us-east-2) ★ ⓘ

🇺🇸 N. California (us-west-1) ⓘ

EUROPE

🇩🇪 Frankfurt (eu-central-1) ★

🇮🇪 Ireland (eu-west-1) ★

🇬🇧 London (eu-west-2) ★ ⓘ

AUSTRALIA

🇦🇺 Sydney (ap-southeast-2) ★

🇮🇳 Mumbai (ap-south-1)
🇸🇬 Singapore (ap-southeast-1) ★

ASIA

FREE

Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

[Back](#)

[Create 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.

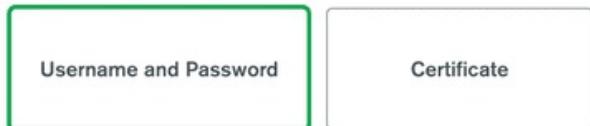
Let's use the *security* tab for creating user credentials for the database. Please note that these are not the same credentials you use for logging into MongoDB Atlas. These will be used for your application to connect to the database.

Security Quickstart

To access data stored in Atlas, you'll need to create users and set up network security controls. [Learn more about security setup](#)

1 How would you like to authenticate your connection?

Your first user will have permission to read and write any data in your project.



Create a database user using a username and password. Users will be given the *read and write to any database privilege* by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

Username	Password
fullstack
Create User	



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:

The screenshot shows the MongoDB Cloud interface for the 'University of Helsinki' project. On the left, there's a sidebar with sections for Deployment, Data Services (Triggers, Data API), and Security (Quickstart, Database Access, Network Access). A red arrow points to the 'Network Access' link. The main area shows a deployment status and a 'Network Access' section with an 'IP Access List' tab selected. A modal window titled 'Add IP Access List Entry' is open. It contains instructions: 'Atlas only allows client connections to a cluster from entries in the project's IP Access List. Each entry should either be a single IP address or a CIDR-notated range of addresses.' Below this are two buttons: 'ADD CURRENT IP ADDRESS' (disabled) and 'ALLOW ACCESS FROM ANYWHERE' (highlighted with a red box). There are fields for 'Access List Entry' (with placeholder 'Enter IP Address or CIDR Notation') and 'Comment' (with placeholder 'Optional comment describing this entry'). A toggle switch indicates the entry is temporary, set to '6 hours'. At the bottom are 'Cancel' and 'Confirm' buttons. A red arrow also points to the 'Allow Access From Anywhere' button in the modal.

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*:

The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with sections for Deployment, Data Lake, Data Services (Triggers, Data API PREVIEW), and Security (Quickstart, Database Access, Network Access, Advanced). The main area is titled 'Database Deployments' and shows 'Cluster0'. It includes a search bar, a 'Connect' button highlighted by a red arrow, and tabs for 'View Monitoring' and 'Browse Collections'. Below these are metrics for throughput (R 0, W 0), connections (0), and network traffic (In 0.0 B/s, Out 0.0 B/s). A call-to-action 'Enhance Your Experience' encourages upgrading to a dedicated cluster. At the bottom, a table provides details about the cluster: VERSION 4.4.11, REGION AWS / Stockholm (eu-north-1), CLUSTER TIER M0 Sandbox (General), TYPE Replica Set - 3 nodes, BACKUPS Inactive, and LINKED REALM None Linked.

and choose: *Connect to your application:*

Connect to Cluster0

✓ Setup connection security **✓ Choose a connection method** **Connect**

- 1 Select your driver and version**
DRIVER: Node.js | VERSION: 4.0 or later
- 2 Add your connection string into your application code**
 Include full driver code example

```
mongodb+srv://fullstack:<password>@cluster0.o1opl.mongodb.net/myFirstDatabase?  
retryWrites=true&w=majority
```

Replace <password> with the password for the **fullstack** user. Replace **myFirstDatabase** with the name of the database that connections will use by default. Ensure any option params are **URL encoded**.

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
```

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
```

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://${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,
})

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]
```

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.

The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with sections for Deployment, Databases (selected), Data Lake, Data Services (Triggers, Data API - PREVIEW), and Security (Quickstart, Database Access, Network Access, Advanced). The main area is titled "Database Deployments" and shows "Cluster0". It includes a search bar, a "Connect" button, "View Monitoring", "Browse Collections" (which has a red arrow pointing to it), and a "...". Below this, there's an "Enhance Your Experience" section with a green "Upgrade" button, and some metrics: R 0, W 0, Connections 0, and a note about throughput and metrics. A red arrow points to the "Browse Collections" button.

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

The screenshot shows the Cluster0 MongoDB interface. At the top, it displays 'VERSION 5.0.14' and 'REGION AWS Stockholm (eu-west-1)'. The navigation bar includes 'Overview', 'Real Time', 'Metrics', 'Collections' (which is underlined in green), 'Search', 'Profiler', 'Performance Advisor', and 'Online'. Below this, a message says 'DATABASES: 1 COLLECTIONS: 1'. On the left sidebar, there's a '+ Create Database' button and a 'Search Namespaces' input field. Under the 'test' database, the 'notes' collection is selected. The main panel shows the 'test.notes' collection with the following details: 'STORAGE SIZE: 36KB', 'LOGICAL DATA SIZE: 166B', 'TOTAL DOCUMENTS: 2', and 'INDEXES TOTAL SIZE: 36KB'. It has tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. A 'FILTER' button with the query '{ field: 'value' }' is present. An 'INSERT DOCUMENT' button is at the top right. Below the filters, the 'QUERY RESULTS: 1-1 OF 1' section displays a single document:

```

_id: ObjectId('63c7e133dd76c0facd00b6cd')
content: "HTML is Easy"
date: 2023-01-18T12:08:19.410+00:00
important: true
__v: 0

```

At the bottom right of the results, there are edit, preview, and copy icons.

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 URL:

```

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

```

Let's run our code again:

The screenshot shows the MongoDB Atlas interface. At the top, there are tabs for Overview, Metrics, Collections (which is selected), Search, Cmd Line Tools, Real Time, Profiler, and P. Below the tabs, it says DATABASES: 1 COLLECTIONS: 1. On the left sidebar, there's a '+ Create Database' button, a NAMESPACES search bar, and a tree view showing a database named 'noteApp' with a collection named 'notes'. In the main area, the collection 'noteApp.notes' is selected. It shows a summary: COLLECTION SIZE: 83B TOTAL DOCUMENTS: 1 INDEXES TOTAL SIZE: 4KB. Below this are buttons for Find, Indexes, Schema Anti-Patterns (0), Aggregation, and Search Ind. A 'FILTER' button with the query '{ field: 'value' }' is present. The 'QUERY RESULTS 1-1 OF 1' section displays a single document:

```
_id: ObjectId("61eab60520442619c1114369")
content: "HTML is Easy"
date: 2022-01-21T13:32:53.201+00:00
important: true
__v: 0
```

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
```

As a result, the application will print:

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

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
```

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

```
node mongo.js yourpassword
```

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

```
phonebook:  
Anna 040-1234556  
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)

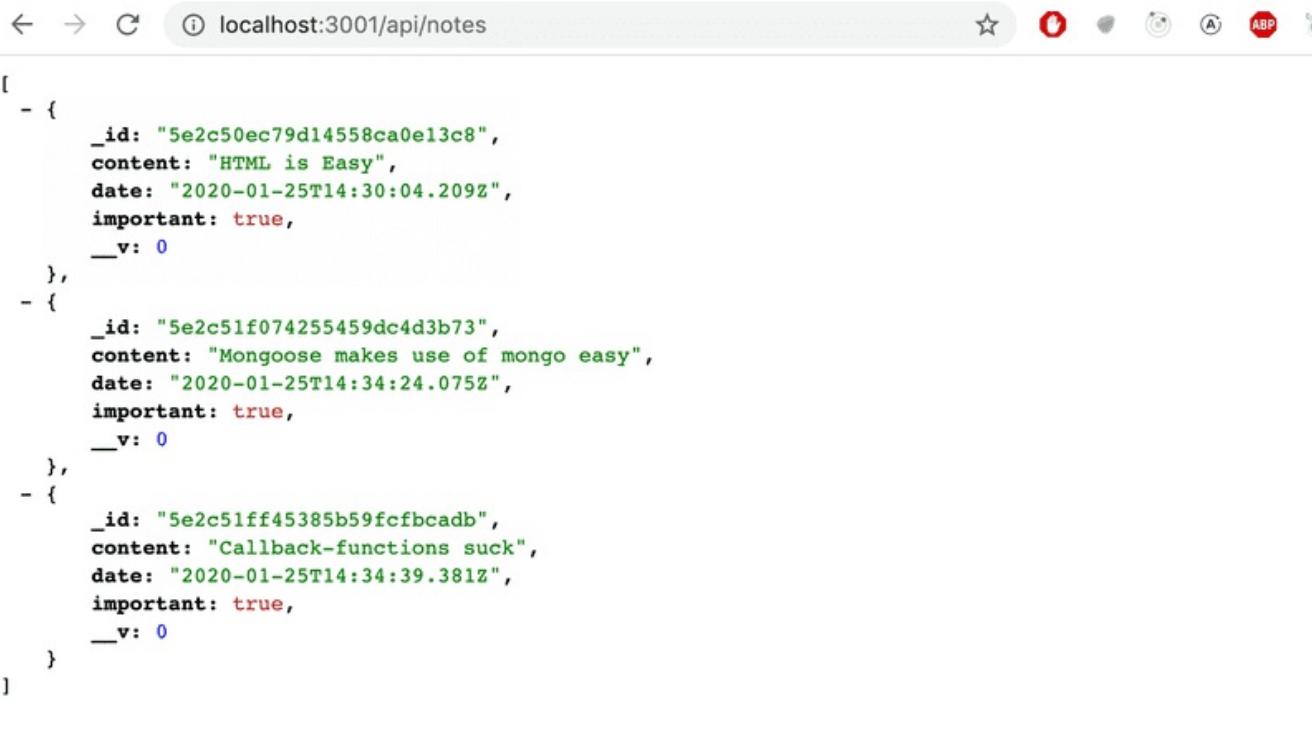
const noteSchema = new mongoose.Schema({
  content: String,
  important: Boolean,
})

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:



```
[{"_id": "5e2c50ec79d14558ca0e13c8", "content": "HTML is Easy", "date": "2020-01-25T14:30:04.209Z", "important": true, "__v": 0}, {"_id": "5e2c51f074255459dc4d3b73", "content": "Mongoose makes use of mongo easy", "date": "2020-01-25T14:34:24.075Z", "important": true, "__v": 0}, {"_id": "5e2c51ff45385b59fcfbcadb", "content": "Callback-functions suck", "date": "2020-01-25T14:34:39.381Z", "important": true, "__v": 0}]
```

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')
```

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')
  ])
  .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:

```
[nodemon] 1.19.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `node index.js`
connecting to mongodb+srv://fullstack:wrongpassword@cluster0-ostce.mongodb.net/note-app?retryWrites=true
Server running on port 3001
error connecting to MongoDB: bad auth Authentication failed.
```

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
```

A more sophisticated way is to use the [dotenv](#) library. You can install the library with the command:

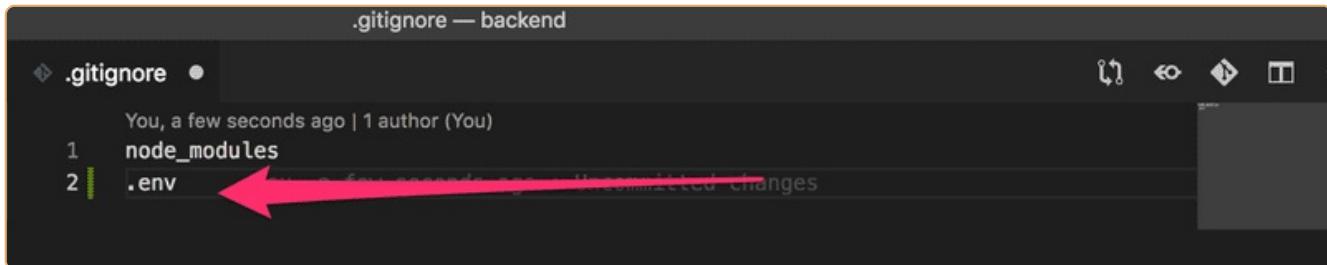
```
npm install dotenv
```

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
```

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!



```
.gitignore — backend
◆ .gitignore ●
You, a few seconds ago | 1 author (You)
1 node_modules
2 .env
```

A screenshot of a GitHub repository showing the .gitignore file. The file contains two entries: 'node_modules' and '.env'. A large red arrow points from the left margin to the '.env' entry.

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:

The screenshot shows the Render web interface. At the top, there's a navigation bar with 'Dashboard', 'Blueprints', 'Env Groups', 'Docs', 'Community', 'Help', 'New +', and a user profile for 'Matti Luukkainen'. Below the navigation, the service name 'render-test' is shown along with its type ('Node'), plan ('Free Plan'), repository ('mluukkai/render-test'), branch ('main'), and URL ('https://render-test-yu7p.onrender.com'). On the left, a sidebar lists 'Events', 'Logs', 'Disks', 'Environment' (which is selected), 'Shell', 'PRs', and 'Jobs'. The main content area is titled 'Environment Variables' and contains a note about using them in code. It shows a table with one row: 'Key' is 'MONGODB_URI' and 'Value' is 'value'. There are buttons for 'Generate', 'Delete', 'Add Environment Variable', and 'Save Changes'.

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:

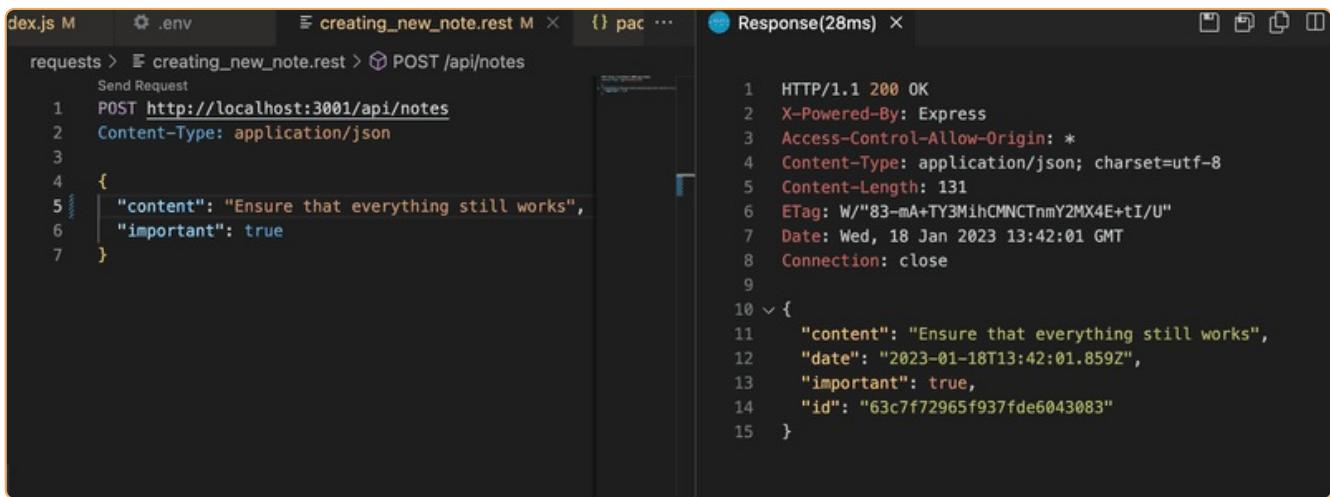
```
response.json(savedNote)
```

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)
  })
})
```

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:



The screenshot shows the VS Code REST Client interface. On the left, there is a code editor window with a file named `dex.js` containing a single line of code: `POST http://localhost:3001/api/notes`. To the right of the code editor is a "requests" panel showing a single request labeled "creating_new_note.rest". Below the requests panel is a "Response(28ms)" panel displaying the server's response. The response is a JSON object with the following fields:

```
1  HTTP/1.1 200 OK
2  X-Powered-By: Express
3  Access-Control-Allow-Origin: *
4  Content-Type: application/json; charset=utf-8
5  Content-Length: 131
6  ETag: W/"83-mA+TY3MihCMNCTnmY2MX4E+tI/U"
7  Date: Wed, 18 Jan 2023 13:42:01 GMT
8  Connection: close
9
10 {
11   "content": "Ensure that everything still works",
12   "date": "2023-01-18T13:42:01.859Z",
13   "important": true,
14   "id": "63c7f72965f937fde6043083"
15 }
```

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/n
ode_modules/mongoose/lib/error/cast.js:27:11)
    at ObjectId.cast (/Users/mluukkai/opetus/_fullstack/osa3-muisiinpan
```

```
ot/node_modules/mongoose/lib/schema/objectid.js:158:13)
```

```
...
```

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:

The screenshot shows a terminal window on the left and a browser window on the right. The terminal window has a red box around its top portion where logs are displayed. It shows several 'nodemon' log entries, including 'restart' and 'start' messages, along with 'Server run' and 'Method: GET' logs. The browser window shows a 404 error page with the message 'This localhost page can't be found' and 'No webpage was found for the web address: http://localhost:3001/api/notes/5a6375b6ddaa4d8c9a132'. The code editor at the top right shows a file named 'Note.js' with some code.

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:



A screenshot of a web browser window. The address bar shows the URL `localhost:3001/api/persons/5c42336b0303b2e44071dc4f`. The main content area displays a JSON object representing a person entry:

```
{
  "name": "Arto Hellas",
  "number": "040-2345823",
  "id": "5c42336b0303b2e44071dc4f"
}
```

3d: 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:

The screenshot shows a Postman request to `localhost:3001/api/notes` with a POST method. The request body contains the following JSON:

```

1+ {
2   "content": "lol",
3   "important": true
4 }

```

The response status is `400 Bad Request`, with a time of `60 ms` and size of `337 B`. The response body is:

```

1+ {
2   "error": "Note validation failed: content: Path `content` (`lol`) is shorter than the minimum allowed length (5)."
3 }

```

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 screenshot shows a browser window with a notes application. The page title is "Notes app". There are two buttons: "show important" and "save". Below the buttons, the text "Note app, Department of Computer Science, University of Helsinki 2023" is displayed. The browser's developer tools Network tab is open, showing a table of network requests. One request for "notes" is listed with the following details:

Name	Status	Type	Initiator	Size
notes	(pending)	xhr	xhr.js:247	

Below the table, the browser's developer tools log tab displays the following error message:

```
2023-01-19T09:51:42Z app[ca96de3d] fra [info]> backend@1.0.0 start
2023-01-19T09:51:42Z app[ca96de3d] fra [info]> node index.js
2023-01-19T09:51:42Z app[ca96de3d] fra [info]connecting to undefined ←
2023-01-19T09:51:43Z app[ca96de3d] fra [info]Server running on port 8080
2023-01-19T09:51:43Z app[ca96de3d] fra [info]error connecting to MongoDB: The 'uri' parameter to 'openUri()' must be a string, got "undefined". Make sure the first parameter to 'mongoose.connect()' or 'mongoose.createConnection()' is a string.
2023-01-19T09:52:23Z runner[0a299bb8] fra [info]Shutting down virtual machine
2023-01-19T09:52:23Z app[0a299bb8] fra [info]Sending signal SIGINT to main child process w/ PID 520
```

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*:

```
2023-01-19T09:51:42Z app[ca96de3d] fra [info]> backend@1.0.0 start
2023-01-19T09:51:42Z app[ca96de3d] fra [info]> node index.js
2023-01-19T09:51:42Z app[ca96de3d] fra [info]connecting to undefined ←
2023-01-19T09:51:43Z app[ca96de3d] fra [info]Server running on port 8080
2023-01-19T09:51:43Z app[ca96de3d] fra [info]error connecting to MongoDB: The 'uri' parameter to 'openUri()' must be a string, got "undefined". Make sure the first parameter to 'mongoose.connect()' or 'mongoose.createConnection()' is a string.
2023-01-19T09:52:23Z runner[0a299bb8] fra [info]Shutting down virtual machine
2023-01-19T09:52:23Z app[0a299bb8] fra [info]Sending signal SIGINT to main child process w/ PID 520
```

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

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

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

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

The screenshot shows the Render Dashboard for a service named 'render-test'. The 'Environment' tab is selected. It displays a table with one row for 'MONGODB_URI'. A red arrow points to the 'Value' column, which contains the placeholder 'value'. There are buttons for 'Generate' and 'Delete' in the same row. Below the table is a button to 'Create Environment Group'. At the bottom right are buttons for 'Add Environment Variable' and 'Save Changes'.

The Render Dashboard shows the server logs:

The screenshot shows the Render Dashboard with the 'Logs' tab selected. The logs pane displays a timestamped log output from a Node.js application. A red arrow points to the line 'Server running on port 10000'. The log output includes:

```
Jan 18 12:04:41 PM > node index.js
Jan 18 12:04:41 PM
Jan 18 12:04:43 PM Server running on port 10000
Jan 18 12:04:48 PM Method: GET
Jan 18 12:04:48 PM Path: /
Jan 18 12:04:48 PM Body: {}
Jan 18 12:04:48 PM ---
Jan 18 12:04:57 PM ==> Starting service with 'npm start'
Jan 18 12:05:00 PM
Jan 18 12:05:00 PM > render-test@1.0.0 start /opt/render/project/src
Jan 18 12:05:00 PM > node index.js
Jan 18 12:05:00 PM
Jan 18 12:05:01 PM Server running on port 10000
Jan 18 12:05:28 PM Method: GET
Jan 18 12:05:28 PM Path: /
Jan 18 12:05:28 PM Body: {}
Jan 18 12:05:28 PM ---
```

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:

Phonebook

Person validation failed: name: Path `name` (ju) is shorter than the minimum allowed length (3).

filter shown with

add a new

name:
number:

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
```

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

```
npx eslint --init
```

We will answer all of the questions:

```

backend git:(part3-7) ✘ npx eslint --init
You can also run this command directly using 'npm init @eslint/config'.
Need to install the following packages:
  @eslint/create-config@0.4.2
Ok to proceed? (y) y
✓ How would you like to use ESLint? · style
✓ What type of modules does your project use? · commonjs
✓ Which framework does your project use? · none
✓ Does your project use TypeScript? · No / Yes
✓ Where does your code run? · browser
✓ How would you like to define a style for your project? · prompt
✓ What format do you want your config file to be in? · JavaScript
✓ What style of indentation do you use? · 4
✓ What quotes do you use for strings? · single
✓ What line endings do you use? · unix
✓ Do you require semicolons? · No / Yes
Successfully created .eslintrc.js file in /Users/mluukkai/opetus/hy-fs/koodi/notes-app/backend

```

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"
      ],
      "files": [
        ".eslintrc.{js,cjs}"
      ],
      "parserOptions": [
        "sourceType": "script"
      ]
    }
  ],
  "parserOptions": [
    "ecmaVersion": "latest"
  ],
  "rules": {}
]

```

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
```

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)

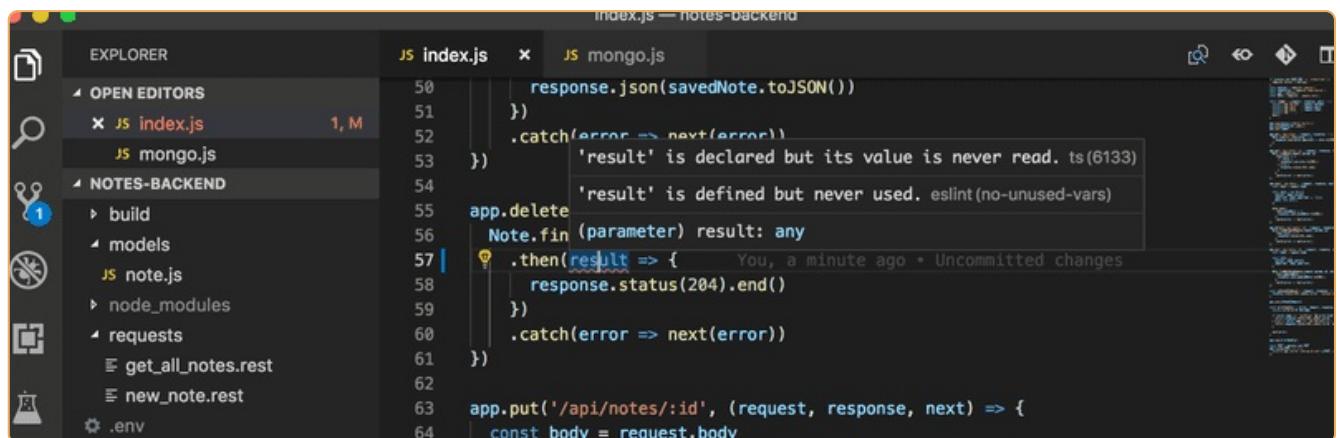
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! notes-backend@1.0.0 lint: `eslint .`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the notes-backend@1.0.0 lint script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.

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) x
```

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',
```

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

```
{
  // ...
  'rules': [
    // ...
    'eqeqeq': 'error',
    'no-trailing-spaces': 'error',
    'object-curly-spacing': [
      'error', 'always'
    ],
    'arrow-spacing': [
      'error', { 'before': true, 'after': true }
    ],
    'no-console': 0
  ],
}
```

NB when you make changes to the `.eslintrc.js` file, it is recommended to run the linter from the

command line. This will verify that the configuration file is correctly formatted:

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

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

/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/.eslintrc.js:30
    "error", {"always"}
          ^
SyntaxError: Unexpected token }

  at new Script (vm.js:74:7)
  at createScript (vm.js:246:10)
  at Object.runInThisContext (vm.js:298:10)
  at Module._compile (internal/modules/cjs/loader.js:646:28)
  at Object.Module._extensions..js (internal/modules/cjs/loader.js:689:10)
  at Module.load (internal/modules/cjs/loader.js:589:32)
  at tryModuleLoad (internal/modules/cjs/loader.js:528:12)
  at Function.Module._load (internal/modules/cjs/loader.js:520:32)
```

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](#).

4: Testing Express servers, user administration

4a: Structure of backend application, introduction to testing

Let's continue our work on the backend of the notes application we started in [part 3](#).

Project structure

Note: 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).

Before we move into the topic of testing, we will modify the structure of our project to adhere to Node.js best practices.

Once we make the changes to the directory structure of our project, we will end up with the following structure:

```
├── index.js
├── app.js
├── dist
│   └── ...
├── controllers
│   └── notes.js
├── models
│   └── note.js
├── package-lock.json
├── package.json
├── utils
│   ├── config.js
│   ├── logger.js
│   └── middleware.js
```

So far we have been using `console.log` and `console.error` to print different information from the code. However, this is not a very good way to do things. Let's separate all printing to the console to its own module `utils/logger.js`:

```
const info = (...params) => {
  console.log(...params)
}

const error = (...params) => {
  console.error(...params)
}

module.exports = [
  info,
  error
]
```

The logger has two functions, `info` for printing normal log messages, and `error` for all error messages.

Extracting logging into its own module is a good idea in several ways. If we wanted to start writing logs to a file or send them to an external logging service like [graylog](#) or [papertrail](#) we would only have to make changes in one place.

The handling of environment variables is extracted into a separate `utils/config.js` file:

```
require('dotenv').config()

const PORT = process.env.PORT
const MONGODB_URI = process.env.MONGODB_URI

module.exports = [
  MONGODB_URI,
  PORT
]
```

The other parts of the application can access the environment variables by importing the configuration module:

```
const config = require('./utils/config')

logger.info(`Server running on port ${config.PORT}`)
```

The contents of the *index.js* file used for starting the application gets simplified as follows:

```
const app = require('./app') // the actual Express application
const config = require('./utils/config')
const logger = require('./utils/logger')

app.listen(config.PORT, () => [
  logger.info(`Server running on port ${config.PORT}`)
])
```

The *index.js* file only imports the actual application from the *app.js* file and then starts the application. The function *info* of the logger-module is used for the console printout telling that the application is running.

Now the Express app and the code taking care of the web server are separated from each other following the [best](#) practices. One of the advantages of this method is that the application can now be tested at the level of HTTP API calls without actually making calls via HTTP over the network, this makes the execution of tests faster.

The route handlers have also been moved into a dedicated module. The event handlers of routes are commonly referred to as *controllers*, and for this reason we have created a new *controllers* directory. All of the routes related to notes are now in the *notes.js* module under the *controllers* directory.

The contents of the *notes.js* module are the following:

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

notesRouter.get('/', (request, response) => {
  Note.find({}).then(notes => [
    response.json(notes)
  ])
})

notesRouter.get('/: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))
  })

notesRouter.post('/+', [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))
])

notesRouter.delete('/:id', [request, response, next] => [
  Note.findByIdAndDelete(request.params.id)
    .then(() => {
      response.status(204).end()
    })
    .catch(error => next(error))
])

notesRouter.put('/: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))
])

module.exports = notesRouter

```

This is almost an exact copy-paste of our previous *index.js* file.

However, there are a few significant changes. At the very beginning of the file we create a new **router** object:

```

const notesRouter = require('express').Router()

//...

```

```
module.exports = notesRouter
```

The module exports the router to be available for all consumers of the module.

All routes are now defined for the router object, similar to what was done before with the object representing the entire application.

It's worth noting that the paths in the route handlers have shortened. In the previous version, we had:

```
app.delete('/api/notes/:id', [request, response, next] => {
```

And in the current version, we have:

```
notesRouter.delete('/:id', [request, response, next] => {
```

So what are these router objects exactly? The Express manual provides the following explanation:

A router object is an isolated instance of middleware and routes. You can think of it as a “mini-application,” capable only of performing middleware and routing functions. Every Express application has a built-in app router.

The router is in fact a *middleware*, that can be used for defining "related routes" in a single place, which is typically placed in its own module.

The *app.js* file that creates the actual application takes the router into use as shown below:

```
const notesRouter = require('./controllers/notes')
app.use('/api/notes', notesRouter)
```

The router we defined earlier is used *if* the URL of the request starts with */api/notes*. For this reason, the notesRouter object must only define the relative parts of the routes, i.e. the empty path / or just the parameter */:id*.

After making these changes, our *app.js* file looks like this:

```
const config = require('./utils/config')
const express = require('express')
const app = express()
const cors = require('cors')
const notesRouter = require('./controllers/notes')
const middleware = require('./utils/middleware')
const logger = require('./utils/logger')
const mongoose = require('mongoose')

mongoose.set('strictQuery', false)

logger.info('connecting to', config.MONGODB_URI)
```

```

mongoose.connect(config.MONGODB_URI)
  .then(() => {
    logger.info('connected to MongoDB')
  })
  .catch((error) => {
    logger.error('error connecting to MongoDB:', error.message)
  })

app.use(cors())
app.use(express.static('dist'))
app.use(express.json())
app.use(middleware.requestLogger)

app.use('/api/notes', notesRouter)

app.use(middleware.unknownEndpoint)
app.use(middleware.errorHandler)

module.exports = app

```

The file takes different middleware into use, and one of these is the *notesRouter* that is attached to the */api/notes* route.

Our custom middleware has been moved to a new *utils/middleware.js* module:

```

const logger = require('./logger')

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

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

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

```

```
module.exports = [
  requestLogger,
  unknownEndpoint,
  errorHandler
]
```

The responsibility of establishing the connection to the database has been given to the *app.js* module. The *note.js* file under the *models* directory only defines the Mongoose schema for notes.

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

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

To recap, the directory structure looks like this after the changes have been made:

```
├── index.js
├── app.js
├── dist
│   └── ...
├── controllers
│   └── notes.js
├── models
│   └── note.js
├── package-lock.json
└── package.json
```

For smaller applications, the structure does not matter that much. Once the application starts to grow in size, you are going to have to establish some kind of structure and separate the different responsibilities of the application into separate modules. This will make developing the application much easier.

There is no strict directory structure or file naming convention that is required for Express applications. In contrast, Ruby on Rails does require a specific structure. Our current structure simply follows some of the best practices that you can come across on the internet.

You can find the code for our current application in its entirety in the *part4-1* branch of [this GitHub repository](#).

If you clone the project for yourself, run the *npm install* command before starting the application with *npm run dev*.

Note on exports

We have used two different kinds of exports in this part. Firstly, e.g. the file *utils/logger.js* does the export as follows:

```
const info = (...params) => {
  console.log(...params)
}

const error = (...params) => {
  console.error(...params)
}

module.exports = { info, error }
```

The file exports *an object* that has two fields, both of which are functions. The functions can be used in two different ways. The first option is to require the whole object and refer to functions through the object using the dot notation:

```
const logger = require('./utils/logger')

logger.info('message')

logger.error('error message')
```

The other option is to destructure the functions to their own variables in the *require* statement:

```
const { info, error } = require('./utils/logger')

info('message')
error('error message')
```

The second way of exporting may be preferable if only a small portion of the exported functions are used in a file. E.g. in file *controller/notes.js* exporting happens as follows:

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

// ...
```

```
module.exports = notesRouter
```

In this case, there is just one "thing" exported, so the only way to use it is the following:

```
const notesRouter = require('./controllers/notes')

// ...

app.use('/api/notes', notesRouter)
```

Now the exported "thing" (in this case a router object) is assigned to a variable and used as such.

Finding the usages of your exports with VS Code

VS Code has a handy feature that allows you to see where your modules have been exported. This can be very helpful for refactoring. For example, if you decide to split a function into two separate functions, your code could break if you don't modify all the usages. This is difficult if you don't know where they are. However, you need to define your exports in a particular way for this to work.

If you right-click on a variable in the location it is exported from and select "Find All References", it will show you everywhere the variable is imported. However, if you assign an object directly to module.exports, it will not work. A workaround is to assign the object you want to export to a named variable and then export the named variable. It also will not work if you destructure where you are importing; you have to import the named variable and then destructure, or just use dot notation to use the functions contained in the named variable.

The nature of VS Code bleeding into how you write your code is probably not ideal, so you need to decide for yourself if the trade-off is worthwhile.

Exercises 4.1.-4.2.

Note: 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).

In the exercises for this part, we will be building a *blog list application*, that allows users to save information about interesting blogs they have stumbled across on the internet. For each listed blog we will save the author, title, URL, and amount of upvotes from users of the application.

4.1 Blog List, step 1

Let's imagine a situation, where you receive an email that contains the following application body and instructions:

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

const blogSchema = new mongoose.Schema({
  title: String,
```

```

    author: String,
    url: String,
    likes: Number
  })

const Blog = mongoose.model('Blog', blogSchema)

const mongoUrl = 'mongodb://localhost/bloglist'
mongoose.connect(mongoUrl)

app.use(cors())
app.use(express.json())

app.get('/api/blogs', (request, response) => {
  Blog
    .find({})
    .then(blogs => {
      response.json(blogs)
    })
})

app.post('/api/blogs', (request, response) => {
  const blog = new Blog(request.body)

  blog
    .save()
    .then(result => {
      response.status(201).json(result)
    })
})

const PORT = 3003
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
})

```

Turn the application into a functioning *npm* project. To keep your development productive, configure the application to be executed with *nodemon*. You can create a new database for your application with MongoDB Atlas, or use the same database from the previous part's exercises.

Verify that it is possible to add blogs to the list with Postman or the VS Code REST client and that the application returns the added blogs at the correct endpoint.

4.2 Blog List, step 2

Refactor the application into separate modules as shown earlier in this part of the course material.

NB refactor your application in baby steps and verify that it works after every change you make. If you try to take a "shortcut" by refactoring many things at once, then [Murphy's law](#) will kick in and it is almost certain that something will break in your application. The "shortcut" will end up taking more time than moving forward slowly and systematically.

One best practice is to commit your code every time it is in a stable state. This makes it easy to rollback to a situation where the application still works.

If you're having issues with `content.body` being `undefined` for seemingly no reason, make sure you didn't forget to add `app.use(express.json())` near the top of the file.

Testing Node applications

We have completely neglected one essential area of software development, and that is automated testing.

Let's start our testing journey by looking at unit tests. The logic of our application is so simple, that there is not much that makes sense to test with unit tests. Let's create a new file `utils/for_testing.js` and write a couple of simple functions that we can use for test writing practice:

```
const reverse = (string) => {
  return string
    .split('')
    .reverse()
    .join('')

}

const average = (array) => {
  const reducer = (sum, item) => {
    return sum + item
  }

  return array.reduce(reducer, 0) / array.length
}

module.exports = [
  reverse,
  average,
]
```

The `average` function uses the array `reduce` method. If the method is not familiar to you yet, then now is a good time to watch the first three videos from the [Functional JavaScript](#) series on YouTube.

There are a large number of test libraries, or *test runners*, available for JavaScript. The old king of test libraries is [Mocha](#), which was replaced a few years ago by [Jest](#). A newcomer to the libraries is [Vitest](#), which bills itself as a new generation of test libraries.

Nowadays, Node also has a built-in test library [node:test](#), which is well suited to the needs of the course.

Let's define the `npm script test` for the test execution:

```
//...
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js",
  "build:ui": "rm -rf build && cd ../frontend/ && npm run build && cp -r build
  ../backend",
  "deploy": "fly deploy",
  "deploy:full": "npm run build:ui && npm run deploy",
  "logs:prod": "fly logs",
  "lint": "eslint .",
  "test": "node --test" },
//...
}
```

Let's create a separate directory for our tests called *tests* and create a new file called *reverse.test.js* with the following contents:

```
const { test } = require('node:test')
const assert = require('node:assert')

const reverse = require('../utils/for_testing').reverse

test('reverse of a', () => {
  const result = reverse('a')

  assert.strictEqual(result, 'a')
})

test('reverse of react', () => {
  const result = reverse('react')

  assert.strictEqual(result, 'tcaer')
})

test('reverse of saippuakauppias', () => {
  const result = reverse('saippuakauppias')

  assert.strictEqual(result, 'saippuakauppias')
})
```

The test defines the keyword *test* and the library *assert*, which is used by the tests to check the results of the functions under test.

In the next row, the test file imports the function to be tested and assigns it to a variable called *reverse*:

```
const reverse = require('../utils/for_testing').reverse
```

Individual test cases are defined with the *test* function. The first argument of the function is the test description as a string. The second argument is a *function*, that defines the functionality for the test case. The functionality for the second test case looks like this:

```

() => {
  const result = reverse('react')

  assert.strictEqual(result, 'tcaer')
}

```

First, we execute the code to be tested, meaning that we generate a reverse for the string *react*. Next, we verify the results with the the method `strictEqual` of the `assert` library.

As expected, all of the tests pass:

```

→ notes-backend git:(part4-2) npm test

> notebackend@1.0.0 test
> node --test

✓ reverse of a (0.443708ms)
✓ reverse of react (0.1045ms)
✓ reverse of saippuakauppias (0.045ms)
i tests 3
i suites 0
i pass 3
i fail 0
i cancelled 0
i skipped 0
i todo 0
i duration_ms 55.315667

```

The library `node:test` expects by default that the names of test files contain `.test`. In this course, we will follow the convention of naming our tests files with the extension `.test.js`.

Let's break the test:

```

test['reverse of react']() => {
  const result = reverse('react')

  assert.strictEqual(result, 'tkaer')
}

```

Running this test results in the following error message:

```

✖ failing tests:

test at tests/reverse.test.js:12:1
✖ reverse of react (0.781584ms)
AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
+ actual - expected
  + 'tcaer'
  - 'tkaer'

```

Let's put the tests for the *average* function, into a new file called `tests/average.test.js`.

```

const { test, describe } = require('node:test')

```

```
// ...

const average = require('../utils/for_testing').average

describe('average', () => {
  test('of one value is the value itself', () => {
    assert.strictEqual(average([1]), 1)
  })

  test('of many is calculated right', () => {
    assert.strictEqual(average([1, 2, 3, 4, 5, 6]), 3.5)
  })

  test('of empty array is zero', () => {
    assert.strictEqual(average([]), 0)
  })
})
})
```

The test reveals that the function does not work correctly with an empty array (this is because in JavaScript dividing by zero results in *NaN*):

```
generatedMessage: true,
code: 'ERR_ASSERTION',
actual: 'tcaer',
expected: 'tkaer',
operator: 'strictEqual'
}

test at tests/reverse.test.js:34:3
✖ of empty array is zero (0.1025ms)
AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
  NaN !== 0
```

Fixing the function is quite easy:

```
const average = array => [
  const reducer = (sum, item) => {
    return sum + item
  }

  return array.length === 0
    ? 0
    : array.reduce(reducer, 0) / array.length
]
```

If the length of the array is 0 then we return 0, and in all other cases, we use the *reduce* method to calculate the average.

There are a few things to notice about the tests that we just wrote. We defined a *describe* block around the tests that were given the name *average*:

```
describe('average', () => {
  // tests
})
```

Describe blocks can be used for grouping tests into logical collections. The test output also uses the name of the describe block:

```
notes-backend git:(part4-2) ✘ npm test

> notebackend@1.0.0 test
> node --test

✓ reverse of a (0.400708ms)
✓ reverse of react (0.050208ms)
✓ reverse of saippuakauppias (0.039625ms)
▶ average
  ✓ of one value is the value itself (0.076042ms)
  ✓ of many is calculated right (0.036083ms)
  ✓ of empty array is zero (0.03275ms)
```

As we will see later on *describe* blocks are necessary when we want to run some shared setup or teardown operations for a group of tests.

Another thing to notice is that we wrote the tests in quite a compact way, without assigning the output of the function being tested to a variable:

```
test['of empty array is zero', () => {
  assert.strictEqual(average([]), 0)
})
```

Exercises 4.3.-4.7.

Let's create a collection of helper functions that are best suited for working with the describe sections of the blog list. Create the functions into a file called *utils/list_helper.js*. Write your tests into an appropriately named test file under the *tests* directory.

4.3: Helper Functions and Unit Tests, step 1

First, define a *dummy* function that receives an array of blog posts as a parameter and always returns the value 1. The contents of the *list_helper.js* file at this point should be the following:

```
const dummy = (blogs) => [
  // ...
]

module.exports = [
  dummy
]
```

Verify that your test configuration works with the following test:

```

const { test, describe } = require('node:test')
const assert = require('node:assert')
const listHelper = require('../utils/list_helper')

test('dummy returns one', () => {
  const blogs = []

  const result = listHelper.dummy(blogs)
  assert.strictEqual(result, 1)
})

```

4.4: Helper Functions and Unit Tests, step 2

Define a new *totalLikes* function that receives a list of blog posts as a parameter. The function returns the total sum of *likes* in all of the blog posts.

Write appropriate tests for the function. It's recommended to put the tests inside of a *describe* block so that the test report output gets grouped nicely:

```

PASS tests/list_helper.test.js
✓ dummy returns 1 (7ms)
total likes
  ✓ of empty list is zero (1ms)
  ✓ when list has only one blog equals the likes of that (1ms)
  ✓ of a bigger list is calculated right (1ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total

```

Defining test inputs for the function can be done like this:

```

describe('total likes', () => {
  const listWithOneBlog = [
    {
      _id: '5a422aa71b54a676234d17f8',
      title: 'Go To Statement Considered Harmful',
      author: 'Edsger W. Dijkstra',
      url: 'https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf',
      likes: 5,
      __v: 0
    }
  ]

  test('when list has only one blog, equals the likes of that', () => {
    const result = listHelper.totalLikes(listWithOneBlog)
    assert.strictEqual(result, 5)
  })
})

```

If defining your own test input list of blogs is too much work, you can use the ready-made list [here](#).

You are bound to run into problems while writing tests. Remember the things that we learned about [debugging](#) in part 3. You can print things to the console with *console.log* even during test execution.

4.5*: Helper Functions and Unit Tests, step 3

Define a new *favoriteBlog* function that receives a list of blogs as a parameter. The function finds out which blog has the most likes. If there are many top favorites, it is enough to return one of them.

The value returned by the function could be in the following format:

```
{  
  title: "Canonical string reduction",  
  author: "Edsger W. Dijkstra",  
  likes: 12  
}
```

NB when you are comparing objects, the [deepStrictEqual](#) method is probably what you want to use, since the [strictEqual](#) tries to verify that the two values are the same value, and not just that they contain the same properties. For differences between various assert module functions, you can refer to [this Stack Overflow answer](#).

Write the tests for this exercise inside of a new *describe* block. Do the same for the remaining exercises as well.

4.6*: Helper Functions and Unit Tests, step 4

This and the next exercise are a little bit more challenging. Finishing these two exercises is not required to advance in the course material, so it may be a good idea to return to these once you're done going through the material for this part in its entirety.

Finishing this exercise can be done without the use of additional libraries. However, this exercise is a great opportunity to learn how to use the [Lodash](#) library.

Define a function called *mostBlogs* that receives an array of blogs as a parameter. The function returns the *author* who has the largest amount of blogs. The return value also contains the number of blogs the top author has:

```
{  
  author: "Robert C. Martin",  
  blogs: 3  
}
```

If there are many top bloggers, then it is enough to return any one of them.

4.7*: Helper Functions and Unit Tests, step 5

Define a function called *mostLikes* that receives an array of blogs as its parameter. The function returns the author, whose blog posts have the largest amount of likes. The return value also contains the total number of likes that the author has received:

```
{  
  author: "Edsger W. Dijkstra",  
  likes: 17  
}
```

If there are many top bloggers, then it is enough to show any one of them.

4b: Testing the backend

We will now start writing tests for the backend. Since the backend does not contain any complicated logic, it doesn't make sense to write [unit tests](#) for it. The only potential thing we could unit test is the `toJSON` method that is used for formatting notes.

In some situations, it can be beneficial to implement some of the backend tests by mocking the database instead of using a real database. One library that could be used for this is [mongodb-memory-server](#).

Since our application's backend is still relatively simple, we will decide to test the entire application through its REST API, so that the database is also included. This kind of testing where multiple components of the system are being tested as a group is called [integration testing](#).

Test environment

In one of the previous chapters of the course material, we mentioned that when your backend server is running in Fly.io or Render, it is in *production* mode.

The convention in Node is to define the execution mode of the application with the `NODE_ENV` environment variable. In our current application, we only load the environment variables defined in the `.env` file if the application is *not* in production mode.

It is common practice to define separate modes for development and testing.

Next, let's change the scripts in our notes application `package.json` file, so that when tests are run, `NODE_ENV` gets the value `test`:

```
{  
  // ...  
  "scripts": {  
    "start": "NODE_ENV=production node index.js",      "dev": "NODE_ENV=development nodemon index.js",  
    "test": "NODE_ENV=test node --test",      "build:ui": "rm -rf build && cd ../frontend/ && npm run build && cp -r build ../backend",  
    "deploy": "fly deploy",  
    "deploy:full": "npm run build:ui && npm run deploy",  
    "logs:prod": "fly logs",  
    "lint": "eslint .",  
  },  
  // ...  
}
```

We specified the mode of the application to be *development* in the `npm run dev` script that uses `nodemon`. We also specified that the default `npm start` command will define the mode as *production*.

There is a slight issue in the way that we have specified the mode of the application in our scripts: it will not work on Windows. We can correct this by installing the [cross-env](#) package as a development dependency with the command:

```
npm install --save-dev cross-env
```

We can then achieve cross-platform compatibility by using the cross-env library in our npm scripts defined in `package.json`:

```
{
  // ...
  "scripts": {
    "start": "cross-env NODE_ENV=production node index.js",
    "dev": "cross-env NODE_ENV=development nodemon index.js",
    "test": "cross-env NODE_ENV=test node --test",
    // ...
  },
  // ...
}
```

NB: If you are deploying this application to Fly.io/Render, keep in mind that if cross-env is saved as a development dependency, it would cause an application error on your web server. To fix this, change cross-env to a production dependency by running this in the command line:

```
npm install cross-env
```

Now we can modify the way that our application runs in different modes. As an example of this, we could define the application to use a separate test database when it is running tests.

We can create our separate test database in MongoDB Atlas. This is not an optimal solution in situations where many people are developing the same application. Test execution in particular typically requires a single database instance that is not used by tests that are running concurrently.

It would be better to run our tests using a database that is installed and running on the developer's local machine. The optimal solution would be to have every test execution use a separate database. This is "relatively simple" to achieve by [running Mongo in-memory](#) or by using [Docker](#) containers. We will not complicate things and will instead continue to use the MongoDB Atlas database.

Let's make some changes to the module that defines the application's configuration in `utils/config.js`:

```
require('dotenv').config()

const PORT = process.env.PORT

const MONGODB_URI = process.env.NODE_ENV === 'test' ? process.env.TEST_MONGODB
  _URI : process.env.MONGODB_URI
module.exports = [
  MONGODB_URI,
  PORT
]
```

The `.env` file has *separate variables* for the database addresses of the development and test databases:

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

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

The `config` module that we have implemented slightly resembles the [node-config](#) package. Writing our implementation is justified since our application is simple, and also because it teaches us valuable lessons.

These are the only changes we need to make to our application's code.

You can find the code for our current application in its entirety in the *part4-2* branch of [this GitHub repository](#).

supertest

Let's use the [supertest](#) package to help us write our tests for testing the API.

We will install the package as a development dependency:

```
npm install --save-dev supertest
```

Let's write our first test in the `tests/note_api.test.js` file:

```
const { test, after } = require('node:test')
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')

const api = supertest(app)

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

after(async () => {
  await mongoose.connection.close()
})
```

The test imports the Express application from the `app.js` module and wraps it with the `supertest` function into a so-called `superagent` object. This object is assigned to the `api` variable and tests can use it for making HTTP requests to the backend.

Our test makes an HTTP GET request to the `api/notes` url and verifies that the request is responded to with the status code 200. The test also verifies that the `Content-Type` header is set to `application/json`, indicating that the data is in the desired format.

Checking the value of the header uses a bit strange looking syntax:

```
- expect(['Content-Type', /application\/json/])
```

The desired value is now defined as [regular expression](#) or in short regex. The regex starts and ends with a slash /, because the desired string *application/json* also contains the same slash, it is preceded by a \ so that it is not interpreted as a regex termination character.

In principle, the test could also have been defined as a string

```
- expect(['Content-Type', 'application/json'])
```

The problem here, however, is that when using a string, the value of the header must be exactly the same. For the regex we defined, it is acceptable that the header *contains* the string in question. The actual value of the header is *application/json; charset=utf-8*, i.e. it also contains information about character encoding. However, our test is not interested in this and therefore it is better to define the test as a regex instead of an exact string.

The test contains some details that we will explore [a bit later on](#). The arrow function that defines the test is preceded by the *async* keyword and the method call for the *api* object is preceded by the *await* keyword. We will write a few tests and then take a closer look at this *async/await* magic. Do not concern yourself with them for now, just be assured that the example tests work correctly. The *async/await* syntax is related to the fact that making a request to the API is an *asynchronous* operation. The *async/await* syntax can be used for writing asynchronous code with the appearance of synchronous code.

Once all the tests (there is currently only one) have finished running we have to close the database connection used by Mongoose. This can be easily achieved with the [after](#) method:

```
after(async () => [
  await mongoose.connection.close()
])
```

One tiny but important detail: at the [beginning](#) of this part we extracted the Express application into the *app.js* file, and the role of the *index.js* file was changed to launch the application at the specified port via *app.listen*:

```
const app = require('./app') // the actual Express app
const config = require('./utils/config')
const logger = require('./utils/logger')

app.listen(config.PORT, () => [
  logger.info(`Server running on port ${config.PORT}`)
])
```

The tests only use the Express application defined in the *app.js* file, which does not listen to any ports:

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

```
const app = require('../app')
const api = supertest(app)
// ...
```

The documentation for `supertest` says the following:

if the server is not already listening for connections then it is bound to an ephemeral port for you so there is no need to keep track of ports.

In other words, `supertest` takes care that the application being tested is started at the port that it uses internally.

Let's add two notes to the test database using the `mongo.js` program (here we must remember to switch to the correct database url).

Let's write a few more tests:

```
test('there are two notes', async () => {
  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, 2)
})

test('the first note is about HTTP methods', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(e => e.content)
  assert.strictEqual(contents.includes('HTML is easy'), true)
})
```

Both tests store the response of the request to the `response` variable, and unlike the previous test that used the methods provided by `supertest` for verifying the status code and headers, this time we are inspecting the response data stored in `response.body` property. Our tests verify the format and content of the response data with the method `strictEqual` of the `assert`-library.

We could simplify the second test a bit, and use the `assert` itself to verify that the note is among the returned ones:

```
test('the first note is about HTTP methods', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(e => e.content)
  // is the argument truthy
  assert(contents.includes('HTML is easy'))
})
```

The benefit of using the `async/await` syntax is starting to become evident. Normally we would have to use callback functions to access the data returned by promises, but with the new syntax things are a

lot more comfortable:

```
const response = await api.get('/api/notes')

// execution gets here only after the HTTP request is complete
// the result of HTTP request is saved in variable response
assert.strictEqual(response.body.length, 2)
```

The middleware that outputs information about the HTTP requests is obstructing the test execution output. Let us modify the logger so that it does not print to the console in test mode:

```
const info = (...params) => {
  if (process.env.NODE_ENV !== 'test') { console.log(...params) }}
```



```
const error = (...params) => {
  if (process.env.NODE_ENV !== 'test') { console.error(...params) }}
```



```
module.exports = [
  info, error
]
```

Initializing the database before tests

Testing appears to be easy and our tests are currently passing. However, our tests are bad as they are dependent on the state of the database, that now happens to have two notes. To make them more robust, we have to reset the database and generate the needed test data in a controlled manner before we run the tests.

Our tests are already using the `after` function of to close the connection to the database after the tests are finished executing. The library `node:test` offers many other functions that can be used for executing operations once before any test is run or every time before a test is run.

Let's initialize the database *before every test* with the `beforeEach` function:

```
const { test, after, beforeEach } = require('node:test')
const Note = require('../models/note')
const initialNotes = [ { content: 'HTML is easy', important: false }, { content: 'Browser can execute only JavaScript', important: true } ]
// ...

beforeEach(async () => {
  await Note.deleteMany({})
  let noteObject = new Note(initialNotes[0])
  await noteObject.save()
  noteObject = new Note(initialNotes[1])
  await noteObject.save()
}) // ...
```

The database is cleared out at the beginning, and after that, we save the two notes stored in the `initialNotes` array to the database. By doing this, we ensure that the database is in the same state before every test is run.

Let's also make the following changes to the last two tests:

```

test('there are two notes', async () => {
  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, initialNotes.length)
})

test('the first note is about HTTP methods', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(e => e.content)
  assert(contents.includes('HTML is easy'))
})

```

Running tests one by one

The `npm test` command executes all of the tests for the application. When we are writing tests, it is usually wise to only execute one or two tests.

There are a few different ways of accomplishing this, one of which is the `only` method. With this method we can define in the code what tests should be executed:

```

test.only('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

test.only('there are two notes', async () => {
  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, 2)
})

```

When tests are run with option `--test-only`, that is, with the command:

```
npm test -- --test-only
```

only the `only` marked tests are executed.

The danger of `only` is that one forgets to remove those from the code.

Another option is to specify the tests that need to be run as arguments of the `npm test` command.

The following command only runs the tests found in the `tests/note_api.test.js` file:

```
npm test -- tests/note_api.test.js
```

The `--tests-by-name-pattern` option can be used for running tests with a specific name:

```
npm test -- --test-name-pattern="the first note is about HTTP methods"
```

The provided argument can refer to the name of the test or the describe block. It can also contain just a part of the name. The following command will run all of the tests that contain *notes* in their name:

```
npm run test -- --test-name-pattern="notes"
```

async/await

Before we write more tests let's take a look at the *async* and *await* keywords.

The *async/await* syntax that was introduced in ES7 makes it possible to use *asynchronous functions that return a promise* in a way that makes the code look synchronous.

As an example, the fetching of notes from the database with promises looks like this:

```
Note.find([]).then[notes => {
  console.log('operation returned the following notes', notes)
}]
```

The *Note.find()* method returns a promise and we can access the result of the operation by registering a callback function with the *then* method.

All of the code we want to execute once the operation finishes is written in the callback function. If we wanted to make several asynchronous function calls in sequence, the situation would soon become painful. The asynchronous calls would have to be made in the callback. This would likely lead to complicated code and could potentially give birth to a so-called [callback hell](#).

By [chaining promises](#) we could keep the situation somewhat under control, and avoid callback hell by creating a fairly clean chain of *then* method calls. We have seen a few of these during the course. To illustrate this, you can view an artificial example of a function that fetches all notes and then deletes the first one:

```
Note.find([])
  .then[notes => {
    return notes[0].deleteOne()
  }]
  .then[response => {
    console.log('the first note is removed')
    // more code here
  }]
})
```

The then-chain is alright, but we can do better. The [generator functions](#) introduced in ES6 provided a [clever way](#) of writing asynchronous code in a way that "looks synchronous". The syntax is a bit clunky and not widely used.

The *async* and *await* keywords introduced in ES7 bring the same functionality as the generators, but in an understandable and syntactically cleaner way to the hands of all citizens of the JavaScript world.

We could fetch all of the notes in the database by utilizing the *await* operator like this:

```
const notes = await Note.find({})

console.log('operation returned the following notes', notes)
```

The code looks exactly like synchronous code. The execution of code pauses at `const notes = await Note.find({})` and waits until the related promise is *fulfilled*, and then continues its execution to the next line. When the execution continues, the result of the operation that returned a promise is assigned to the `notes` variable.

The slightly complicated example presented above could be implemented by using `await` like this:

```
const notes = await Note.find({})
const response = await notes[0].deleteOne()

console.log('the first note is removed')
```

Thanks to the new syntax, the code is a lot simpler than the previous then-chain.

There are a few important details to pay attention to when using `async/await` syntax. To use the `await` operator with asynchronous operations, they have to return a promise. This is not a problem as such, as regular asynchronous functions using callbacks are easy to wrap around promises.

The `await` keyword can't be used just anywhere in JavaScript code. Using `await` is possible only inside of an `async` function.

This means that in order for the previous examples to work, they have to be using `async` functions. Notice the first line in the arrow function definition:

```
const main = async () => {
  const notes = await Note.find({})
  console.log('operation returned the following notes', notes)

  const response = await notes[0].deleteOne()
  console.log('the first note is removed')
}

main()
```

The code declares that the function assigned to `main` is asynchronous. After this, the code calls the function with `main()`.

async/await in the backend

Let's start to change the backend to `async` and `await`. As all of the asynchronous operations are currently done inside of a function, it is enough to change the route handler functions into `async` functions.

The route for fetching all notes gets changed to the following:

```
notesRouter.get('/', async [request, response] => {
  const notes = await Note.find({})
```

```
    response.json(notes)
  })
}
```

We can verify that our refactoring was successful by testing the endpoint through the browser and by running the tests that we wrote earlier.

You can find the code for our current application in its entirety in the *part4-3* branch of [this GitHub repository](#).

More tests and refactoring the backend

When code gets refactored, there is always the risk of [regression](#), meaning that existing functionality may break. Let's refactor the remaining operations by first writing a test for each route of the API.

Let's start with the operation for adding a new note. Let's write a test that adds a new note and verifies that the number of notes returned by the API increases and that the newly added note is in the list.

```
test('a valid note can be added ', async () => {
  const newNote = [
    content: 'async/await simplifies making async calls',
    important: true,
  ]

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(201)
    .expect('Content-Type', '/application/json/')

  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)

  assert.strictEqual(response.body.length, initialNotes.length + 1)

  assert(contents.includes('async/await simplifies making async calls'))
})
}
```

The test fails because we accidentally returned the status code *200 OK* when a new note is created. Let us change that to the status code *201 CREATED*:

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

  const note = new Note([
    content: body.content,
    important: body.important || false,
  ])

  note.save()
```

```

        .then(savedNote => [
          response.status(201).json(savedNote)
        ])
      .catch(error => next(error))
    })
  )

```

Let's also write a test that verifies that a note without content will not be saved into the database.

```

test('note without content is not added', async () => {
  const newNote = [
    important: true
  ]

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(400)

  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, initialNotes.length)
})

```

Both tests check the state stored in the database after the saving operation, by fetching all the notes of the application.

```
const response = await api.get('/api/notes')
```

The same verification steps will repeat in other tests later on, and it is a good idea to extract these steps into helper functions. Let's add the function into a new file called `tests/test_helper.js` which is in the same directory as the test file.

```

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

const initialNotes = [
  {
    content: 'HTML is easy',
    important: false
  },
  {
    content: 'Browser can execute only JavaScript',
    important: true
  }
]

const nonExistingId = async () => {
  const note = new Note({ content: 'willremovethissoon' })
  await note.save()
  await note.deleteOne()

  return note._id.toString()
}

```

```

}

const notesInDb = async () => {
  const notes = await Note.find({})
  return notes.map(note => note.toJSON())
}

module.exports = [
  initialNotes, nonExistingId, notesInDb
]

```

The module defines the `notesInDb` function that can be used for checking the notes stored in the database. The `initialNotes` array containing the initial database state is also in the module. We also define the `nonExistingId` function ahead of time, which can be used for creating a database object ID that does not belong to any note object in the database.

Our tests can now use the helper module and be changed like this:

```

const { test, after, beforeEach } = require('node:test')
const assert = require('node:assert')
const supertest = require('supertest')
const mongoose = require('mongoose')
const helper = require('./test_helper') const app = require('../app')
const api = supertest(app)

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

beforeEach(async () => {
  await Note.deleteMany({})

  let noteObject = new Note(helper.initialNotes[0]) await noteObject.save()

  noteObject = new Note(helper.initialNotes[1]) await noteObject.save()
})

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

test('all notes are returned', async () => {
  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, helper.initialNotes.length)
})

test('a specific note is within the returned notes', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)
})

```

```

    assert(contents.includes('Browser can execute only JavaScript'))
  })

test('a valid note can be added', async () => {
  const newNote = [
    content: 'async/await simplifies making async calls',
    important: true,
  ]

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(201)
    .expect('Content-Type', /application\/json/)

  const notesAtEnd = await helper.notesInDb()  assert.strictEqual(notesAtEnd.length, helper.initialNotes.length + 1)
  const contents = notesAtEnd.map(n => n.content)  assert(contents.includes('async/await simplifies making async calls'))
})

test('note without content is not added', async () => {
  const newNote = [
    important: true
  ]

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(400)

  const notesAtEnd = await helper.notesInDb()
  assert.strictEqual(notesAtEnd.length, helper.initialNotes.length))
}

after(async () => [
  await mongoose.connection.close()
])

```

The code using promises works and the tests pass. We are ready to refactor our code to use the `async/await` syntax.

We make the following changes to the code that takes care of adding a new note (notice that the route handler definition is preceded by the `async` keyword):

```

notesRouter.post('/', async [request, response, next] => {
  const body = request.body

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

```

```

    const savedNote = await note.save()
    response.status(201).json(savedNote)
  })
}

```

There's a slight problem with our code: we don't handle error situations. How should we deal with them?

Error handling and `async/await`

If there's an exception while handling the POST request we end up in a familiar situation:

```

Method: POST
Path:  /api/notes
Body:  { important: true }

...
(node:89372) UnhandledPromiseRejectionWarning: ValidationError: Note validation failed: content: Path `content` is required.
    at new ValidationError (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/error/validation.js:30:11)
    at model.Document.invalidate (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/document.js:2071:32)
    at p.doValidate.skipSchemaValidators (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/document.js:1934:17)
    at /Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/schematype.js:929:9
    at process._tickCallback (internal/process/next_tick.js:172:11)
(node:89372) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing

```

In other words, we end up with an unhandled promise rejection, and the request never receives a response.

With `async/await` the recommended way of dealing with exceptions is the old and familiar `try/catch` mechanism:

```

notesRouter.post('/', async [request, response, next] => {
  const body = request.body

  const note = new Note({
    content: body.content,
    important: body.important || false,
  })
  try {
    const savedNote = await note.save()
    response.status(201).json(savedNote)
  } catch(exception) {
    next(exception)
  }
}

```

The `catch` block simply calls the `next` function, which passes the request handling to the error handling middleware.

After making the change, all of our tests will pass once again.

Next, let's write tests for fetching and removing an individual note:

```

test('a specific note can be viewed', async () => {
  const notesAtStart = await helper.notesInDb()

  const noteToView = notesAtStart[0]

```

```

    const resultNote = await api      .get(`api/notes/${noteToView.id}`)      .expect(200)
      .expect('Content-Type', 'application/json')
    assert.deepStrictEqual(resultNote.body, noteToView)
  })

  test('a note can be deleted', async () => {
    const notesAtStart = await helper.notesInDb()
    const noteToDelete = notesAtStart[0]

    await api      .delete(`api/notes/${noteToDelete.id}`)      .expect(204)
    const notesAtEnd = await helper.notesInDb()

    const contents = notesAtEnd.map(r => r.content)
    assert(!contents.includes(noteToDelete.content))

    assert.strictEqual(notesAtEnd.length, helper.initialNotes.length - 1)
  })
}

```

Both tests share a similar structure. In the initialization phase, they fetch a note from the database. After this, the tests call the actual operation being tested, which is highlighted in the code block. Lastly, the tests verify that the outcome of the operation is as expected.

There is one point worth noting in the first test. Instead of the previously used method `strictEqual`, the method `deepStrictEqual` is used:

```
assert.deepStrictEqual(resultNote.body, noteToView)
```

The reason for this is that `strictEqual` uses the method `Object.is` to compare similarity, i.e. it compares whether the objects are the same. In our case, it is enough to check that the contents of the objects, i.e. the values of their fields, are the same. For this purpose `deepStrictEqual` is suitable.

The tests pass and we can safely refactor the tested routes to use `async/await`:

```

notesRouter.get('/:id', async (request, response, next) => {
  try {
    const note = await Note.findById(request.params.id)
    if (note) {
      response.json(note)
    } else {
      response.status(404).end()
    }
  } catch(exception) {
    next(exception)
  }
})

notesRouter.delete('/:id', async (request, response, next) => {
  try {
    await Note.findByIdAndDelete(request.params.id)
    response.status(204).end()
  } catch(exception) {

```

```
    next(exception)
  }
})
```

You can find the code for our current application in its entirety in the *part4-4* branch of [this GitHub repository](#).

Eliminating the try-catch

Async/await unclutters the code a bit, but the 'price' is the *try/catch* structure required for catching exceptions. All of the route handlers follow the same structure

```
try {
  // do the async operations here
} catch(exception) {
  next(exception)
}
```

One starts to wonder if it would be possible to refactor the code to eliminate the *catch* from the methods?

The [express-async-errors](#) library has a solution for this.

Let's install the library

```
npm install express-async-errors
```

Using the library is *very* easy. You introduce the library in *app.js*, *before* you import your routes:

```
const config = require('./utils/config')
const express = require('express')
require('express-async-errors') const app = express()
const cors = require('cors')
const notesRouter = require('./controllers/notes')
const middleware = require('./utils/middleware')
const logger = require('./utils/logger')
const mongoose = require('mongoose')

// ...

module.exports = app
```

The 'magic' of the library allows us to eliminate the try-catch blocks completely. For example the route for deleting a note

```
notesRouter.delete('/:id', async (request, response, next) => {
  try {
    await Note.findByIdAndDelete(request.params.id)
    response.status(204).end()
  } catch (exception) {
```

```
    next(exception)
  }
})
```

becomes

```
notesRouter.delete('/:id', async [request, response] => {
  await Note.findByIdAndDelete(request.params.id)
  response.status[204].end()
})
```

Because of the library, we do not need the `next(exception)` call anymore. The library handles everything under the hood. If an exception occurs in an `async` route, the execution is automatically passed to the error-handling middleware.

The other routes become:

```
notesRouter.post('/', async [request, response] => {
  const body = request.body

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

  const savedNote = await note.save()
  response.status[201].json(savedNote)
})

notesRouter.get('/:id', async [request, response] => {
  const note = await Note.findById(request.params.id)
  if (note) {
    response.json(note)
  } else {
    response.status[404].end()
  }
})
```

Optimizing the `beforeEach` function

Let's return to writing our tests and take a closer look at the `beforeEach` function that sets up the tests:

```
beforeEach(async () => {
  await Note.deleteMany([])

  let noteObject = new Note(helper.initialNotes[0])
  await noteObject.save()

  noteObject = new Note(helper.initialNotes[1])
```

```
    await noteObject.save()
})
```

The function saves the first two notes from the `helper.initialNotes` array into the database with two separate operations. The solution is alright, but there's a better way of saving multiple objects to the database:

```
beforeEach(async () => {
  await Note.deleteMany({})
  console.log('cleared')

  helper.initialNotes.forEach(async (note) => {
    let noteObject = new Note(note)
    await noteObject.save()
    console.log('saved')
  })
  console.log('done')
})

test('notes are returned as json', async () => {
  console.log('entered test')
  // ...
})
```

We save the notes stored in the array into the database inside of a `forEach` loop. The tests don't quite seem to work however, so we have added some console logs to help us find the problem.

The console displays the following output:

```
cleared
done
entered test
saved
saved
```

Despite our use of the `async/await` syntax, our solution does not work as we expected it to. The test execution begins before the database is initialized!

The problem is that every iteration of the `forEach` loop generates an asynchronous operation, and `beforeEach` won't wait for them to finish executing. In other words, the `await` commands defined inside of the `forEach` loop are not in the `beforeEach` function, but in separate functions that `beforeEach` will not wait for.

Since the execution of tests begins immediately after `beforeEach` has finished executing, the execution of tests begins before the database state is initialized.

One way of fixing this is to wait for all of the asynchronous operations to finish executing with the [Promise.all](#) method:

```
beforeEach(async () => {
  await Note.deleteMany({})
```

```

const noteObjects = helper.initialNotes
  .map(note => new Note(note))
const promiseArray = noteObjects.map(note => note.save())
await Promise.all(promiseArray)
}

```

The solution is quite advanced despite its compact appearance. The `noteObjects` variable is assigned to an array of Mongoose objects that are created with the `Note` constructor for each of the notes in the `helper.initialNotes` array. The next line of code creates a new array that *consists of promises*, that are created by calling the `save` method of each item in the `noteObjects` array. In other words, it is an array of promises for saving each of the items to the database.

The `Promise.all` method can be used for transforming an array of promises into a single promise, that will be *fulfilled* once every promise in the array passed to it as an argument is resolved. The last line of code `await Promise.all(promiseArray)` waits until every promise for saving a note is finished, meaning that the database has been initialized.

The returned values of each promise in the array can still be accessed when using the `Promise.all` method. If we wait for the promises to be resolved with the `await` syntax `const results = await Promise.all(promiseArray)`, the operation will return an array that contains the resolved values for each promise in the `promiseArray`, and they appear in the same order as the promises in the array.

`Promise.all` executes the promises it receives in parallel. If the promises need to be executed in a particular order, this will be problematic. In situations like this, the operations can be executed inside of a `for...of` block, that guarantees a specific execution order.

```

beforeEach(async () => {
  await Note.deleteMany({})

  for (let note of helper.initialNotes) {
    let noteObject = new Note(note)
    await noteObject.save()
  }
})

```

The asynchronous nature of JavaScript can lead to surprising behavior, and for this reason, it is important to pay careful attention when using the `async/await` syntax. Even though the syntax makes it easier to deal with promises, it is still necessary to understand how promises work!

The code for our application can be found on [GitHub](#), branch `part4-5`.

A true full stack developer's oath

Making tests brings yet another layer of challenge to programming. We have to update our full stack developer oath to remind you that systematicity is also key when developing tests.

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 as I expect
- I will keep an eye on the database: does the backend save data there in the right format
- I will progress in small steps
- *I will write lots of console.log statements to make sure I understand how the code and the tests behave 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 is still working
- *If a test does not pass, I make sure that the tested functionality for sure works in the application*
- 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 4.8.-4.12.

Warning: If you find yourself using `async/await` and `then` methods in the same code, it is almost guaranteed that you are doing something wrong. Use one or the other and don't mix the two.

4.8: Blog List Tests, step 1

Use the SuperTest library for writing a test that makes an HTTP GET request to the `/api/blogs` URL. Verify that the blog list application returns the correct amount of blog posts in the JSON format.

Once the test is finished, refactor the route handler to use the `async/await` syntax instead of promises.

Notice that you will have to make similar changes to the code that were made [in the material](#), like defining the test environment so that you can write tests that use separate databases.

NB: when you are writing your tests ***it is better to not execute them all***, only execute the ones you are working on. Read more about this [here](#).

4.9: Blog List Tests, step 2

Write a test that verifies that the unique identifier property of the blog posts is named `id`, by default the database names the property `_id`.

Make the required changes to the code so that it passes the test. The `toJSON` method discussed in part 3 is an appropriate place for defining the `id` parameter.

4.10: Blog List Tests, step 3

Write a test that verifies that making an HTTP POST request to the `/api/blogs` URL successfully creates a new blog post. At the very least, verify that the total number of blogs in the system is increased by one. You can also verify that the content of the blog post is saved correctly to the database.

Once the test is finished, refactor the operation to use `async/await` instead of promises.

4.11*: Blog List Tests, step 4

Write a test that verifies that if the *likes* property is missing from the request, it will default to the value 0. Do not test the other properties of the created blogs yet.

Make the required changes to the code so that it passes the test.

4.12*: Blog List tests, step 5

Write tests related to creating new blogs via the */api/blogs* endpoint, that verify that if the *title* or *url* properties are missing from the request data, the backend responds to the request with the status code *400 Bad Request*.

Make the required changes to the code so that it passes the test.

Refactoring tests

Our test coverage is currently lacking. Some requests like *GET /api/notes/:id* and *DELETE /api/notes/:id* aren't tested when the request is sent with an invalid id. The grouping and organization of tests could also use some improvement, as all tests exist on the same "top level" in the test file. The readability of the test would improve if we group related tests with *describe* blocks.

Below is an example of the test file after making some minor improvements:

```
const { test, after, beforeEach, describe } = require('node:test')
const assert = require('node:assert')
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')
const api = supertest(app)

const helper = require('../test_helper')

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

describe('when there is initially some notes saved', () => {
  beforeEach(async () => {
    await Note.deleteMany({})
    await Note.insertMany(helper.initialNotes)
  })

  test('notes are returned as json', async () => {
    await api
      .get('/api/notes')
      .expect(200)
      .expect('Content-Type', /application\/json/)
  })

  test('all notes are returned', async () => {
    const response = await api.get('/api/notes')

    assert.strictEqual(response.body.length, helper.initialNotes.length)
  })
})
```

```

    })

test('a specific note is within the returned notes', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)
  assert(contents.includes('Browser can execute only JavaScript'))
})

describe('viewing a specific note', () => {

  test('succeeds with a valid id', async () => {
    const notesAtStart = await helper.notesInDb()

    const noteToView = notesAtStart[0]

    const resultNote = await api
      .get(`/api/notes/${noteToView.id}`)
      .expect(200)
      .expect('Content-Type', 'application/json')

    assert.deepStrictEqual(resultNote.body, noteToView)
  })

  test('fails with statuscode 404 if note does not exist', async () => {
    const validNonexistingId = await helper.nonExistingId()

    await api
      .get(`/api/notes/${validNonexistingId}`)
      .expect(404)
  })

  test('fails with statuscode 400 id is invalid', async () => {
    const invalidId = '5a3d5da59070081a82a3445'

    await api
      .get(`/api/notes/${invalidId}`)
      .expect(400)
  })
})

describe('addition of a new note', () => {
  test('succeeds with valid data', async () => {
    const newNote = {
      content: 'async/await simplifies making async calls',
      important: true,
    }

    await api
      .post('/api/notes')
      .send(newNote)
      .expect(201)
      .expect('Content-Type', 'application/json')
  })
})

```

```

const notesAtEnd = await helper.notesInDb()
assert.strictEqual(notesAtEnd.length, helper.initialNotes.length + 1)

const contents = notesAtEnd.map(n => n.content)
assert(contents.includes('async/await simplifies making async calls'))
})

test('fails with status code 400 if data invalid', async () => {
  const newNote = {
    important: true
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(400)

  const notesAtEnd = await helper.notesInDb()

  assert.strictEqual(notesAtEnd.length, helper.initialNotes.length)
})
})

describe('deletion of a note', () => {
  test('succeeds with status code 204 if id is valid', async () => {
    const notesAtStart = await helper.notesInDb()
    const noteToDelete = notesAtStart[0]

    await api
      .delete(`/api/notes/${noteToDelete.id}`)
      .expect(204)

    const notesAtEnd = await helper.notesInDb()

    assert.strictEqual(notesAtEnd.length, helper.initialNotes.length - 1)

    const contents = notesAtEnd.map(r => r.content)
    assert(!contents.includes(noteToDelete.content))
  })
})
})

after(async () => {
  await mongoose.connection.close()
})

```

The test output in the console is grouped according to the *describe* blocks:

- ▶ when there is initially some notes saved
 - ✓ notes are returned as json (784.923458ms)
 - ✓ all notes are returned (61.210958ms)
 - ✓ a specific note is within the returned notes (54.483334ms)
- ▶ viewing a specific note
 - ✓ succeeds with a valid id (75.398542ms)
 - ✓ fails with statuscode 404 if note does not exist (128.202209ms)
 - ✓ fails with statuscode 400 id is invalid (56.257166ms)
- ▶ viewing a specific note (260.737375ms)
- ▶ addition of a new note
 - ✓ succeeds with valid data (103.98325ms)
 - ✓ fails with status code 400 if data invalid (62.044916ms)
- ▶ addition of a new note (166.798125ms)
- ▶ deletion of a note
 - ✓ succeeds with status code 204 if id is valid (152.118417ms)
- ▶ deletion of a note (152.820417ms)
- ▶ when there is initially some notes saved (1482.8545ms)

There is still room for improvement, but it is time to move forward.

This way of testing the API, by making HTTP requests and inspecting the database with Mongoose, is by no means the only nor the best way of conducting API-level integration tests for server applications. There is no universal best way of writing tests, as it all depends on the application being tested and available resources.

You can find the code for our current application in its entirety in the *part4-6* branch of [this GitHub repository](#).

Exercises 4.13.-4.14.

4.13 Blog List Expansions, step 1

Implement functionality for deleting a single blog post resource.

Use the `async/await` syntax. Follow [RESTful](#) conventions when defining the HTTP API.

Implement tests for the functionality.

4.14 Blog List Expansions, step 2

Implement functionality for updating the information of an individual blog post.

Use `async/await`.

The application mostly needs to update the number of *likes* for a blog post. You can implement this functionality the same way that we implemented updating notes in [part 3](#).

Implement tests for the functionality.

4c: User administration

We want to add user authentication and authorization to our application. Users should be stored in the database and every note should be linked to the user who created it. Deleting and editing a note should only be allowed for the user who created it.

Let's start by adding information about users to the database. There is a one-to-many relationship between the user (*User*) and notes (*Note*):



If we were working with a relational database the implementation would be straightforward. Both resources would have their separate database tables, and the id of the user who created a note would be stored in the notes table as a foreign key.

When working with document databases the situation is a bit different, as there are many different ways of modeling the situation.

The existing solution saves every note in the *notes collection* in the database. If we do not want to change this existing collection, then the natural choice is to save users in their own collection, *users* for example.

Like with all document databases, we can use object IDs in Mongo to reference documents in other collections. This is similar to using foreign keys in relational databases.

Traditionally document databases like Mongo do not support *join queries* that are available in relational databases, used for aggregating data from multiple tables. However, starting from version 3.2. Mongo has supported [lookup aggregation queries](#). We will not be taking a look at this functionality in this course.

If we need functionality similar to join queries, we will implement it in our application code by making multiple queries. In certain situations, Mongoose can take care of joining and aggregating data, which gives the appearance of a join query. However, even in these situations, Mongoose makes multiple queries to the database in the background.

References across collections

If we were using a relational database the note would contain a *reference key* to the user who created it. In document databases, we can do the same thing.

Let's assume that the *users* collection contains two users:

```
[  
  {  
    username: 'mluukkai',  
    _id: 123456,  
  },  
  {  
    username: 'hellas',  
    _id: 141414,
```

```
  },
]
```

The `notes` collection contains three notes that all have a `user` field that references a user in the `users` collection:

```
[
  {
    content: 'HTML is easy',
    important: false,
    _id: 221212,
    user: 123456,
  },
  {
    content: 'The most important operations of HTTP protocol are GET and POST',
    important: true,
    _id: 221255,
    user: 123456,
  },
  {
    content: 'A proper dinosaur codes with Java',
    important: false,
    _id: 221244,
    user: 141414,
  },
]
```

Document databases do not demand the foreign key to be stored in the note resources, it could *also* be stored in the users collection, or even both:

```
[
  {
    username: 'mluukkai',
    _id: 123456,
    notes: [221212, 221255],
  },
  {
    username: 'hellas',
    _id: 141414,
    notes: [221244],
  },
]
```

Since users can have many notes, the related ids are stored in an array in the `notes` field.

Document databases also offer a radically different way of organizing the data: In some situations, it might be beneficial to nest the entire notes array as a part of the documents in the users collection:

```
[
  {
    username: 'mluukkai',
    notes: [
      {
        content: 'HTML is easy',
        important: false,
        _id: 221212,
        user: 123456,
      },
      {
        content: 'The most important operations of HTTP protocol are GET and POST',
        important: true,
        _id: 221255,
        user: 123456,
      },
      {
        content: 'A proper dinosaur codes with Java',
        important: false,
        _id: 221244,
        user: 141414,
      },
    ],
  },
]
```

```

_id: 123456,
notes: [
  [
    {
      content: 'HTML is easy',
      important: false
    },
    [
      {
        content: 'The most important operations of HTTP protocol are GET and POST',
        important: true
      }
    ]
  ],
{
  username: 'hellas',
  _id: 141414,
  notes: [
    [
      {
        content:
          'A proper dinosaur codes with Java',
        important: false
      }
    ]
  ]
}

```

In this schema, notes would be tightly nested under users and the database would not generate ids for them.

The structure and schema of the database are not as self-evident as it was with relational databases. The chosen schema must support the use cases of the application the best. This is not a simple design decision to make, as all use cases of the applications are not known when the design decision is made.

Paradoxically, schema-less databases like Mongo require developers to make far more radical design decisions about data organization at the beginning of the project than relational databases with schemas. On average, relational databases offer a more or less suitable way of organizing data for many applications.

Mongoose schema for users

In this case, we decide to store the ids of the notes created by the user in the user document. Let's define the model for representing a user in the *models/user.js* file:

```

const mongoose = require('mongoose')

const userSchema = new mongoose.Schema({
  username: String,
  name: String,
  passwordHash: String,
  notes: [
    [

```

```

        type: mongoose.Schema.Types.ObjectId,
        ref: 'Note'
    },
],
[])
}

userSchema.set('toJSON', [
  transform: (document, returnedObject) => [
    returnedObject.id = returnedObject._id.toString()
    delete returnedObject._id
    delete returnedObject.__v
    // the passwordHash should not be revealed
    delete returnedObject.passwordHash
  ]
])
}

const User = mongoose.model('User', userSchema)

module.exports = User

```

The ids of the notes are stored within the user document as an array of Mongo ids. The definition is as follows:

```
{
  type: mongoose.Schema.Types.ObjectId,
  ref: 'Note'
}
```

The type of the field is *ObjectId* that references *note*-style documents. Mongo does not inherently know that this is a field that references notes, the syntax is purely related to and defined by Mongoose.

Let's expand the schema of the note defined in the *models/note.js* file so that the note contains information about the user who created it:

```

const noteSchema = new mongoose.Schema({
  content: {
    type: String,
    required: true,
    minlength: 5
  },
  important: Boolean,
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' })
}

```

In stark contrast to the conventions of relational databases, *references are now stored in both documents*: the note references the user who created it, and the user has an array of references to all of the notes created by them.

Creating users

Let's implement a route for creating new users. Users have a unique *username*, a *name* and

something called a *passwordHash*. The password hash is the output of a [one-way hash function](#) applied to the user's password. It is never wise to store unencrypted plain text passwords in the database!

Let's install the [bcrypt](#) package for generating the password hashes:

```
npm install bcrypt
```

Creating new users happens in compliance with the RESTful conventions discussed in [part 3](#), by making an HTTP POST request to the *users* path.

Let's define a separate *router* for dealing with users in a new *controllers/users.js* file. Let's take the router into use in our application in the *app.js* file, so that it handles requests made to the */api/users* url:

```
const usersRouter = require('./controllers/users')

// ...

app.use('/api/users', usersRouter)
```

The contents of the file, *controllers/users.js*, that defines the router is as follows:

```
const bcrypt = require('bcrypt')
const usersRouter = require('express').Router()
const User = require('../models/user')

usersRouter.post('/', async (request, response) => {
  const { username, name, password } = request.body

  const saltRounds = 10
  const passwordHash = await bcrypt.hash(password, saltRounds)

  const user = new User({
    username,
    name,
    passwordHash,
  })

  const savedUser = await user.save()

  response.status(201).json(savedUser)
})

module.exports = usersRouter
```

The password sent in the request is *not* stored in the database. We store the *hash* of the password that is generated with the *bcrypt.hash* function.

The fundamentals of [storing passwords](#) are outside the scope of this course material. We will not discuss what the magic number 10 assigned to the *saltRounds* variable means, but you can read more

about it in the linked material.

Our current code does not contain any error handling or input validation for verifying that the username and password are in the desired format.

The new feature can and should initially be tested manually with a tool like Postman. However testing things manually will quickly become too cumbersome, especially once we implement functionality that enforces usernames to be unique.

It takes much less effort to write automated tests, and it will make the development of our application much easier.

Our initial tests could look like this:

```
const bcrypt = require('bcrypt')
const User = require('../models/user')

//...

describe('when there is initially one user in db', () => {
  beforeEach(async () => {
    await User.deleteMany({})

    const passwordHash = await bcrypt.hash('sekret', 10)
    const user = new User({ username: 'root', passwordHash })

    await user.save()
  })
}

test('creation succeeds with a fresh username', async () => {
  const usersAtStart = await helper.usersInDb()

  const newUser = {
    username: 'mluukkai',
    name: 'Matti Luukkainen',
    password: 'salainen'
  }

  await api
    .post('/api/users')
    .send(newUser)
    .expect(201)
    .expect('Content-Type', /application\/json/)

  const usersAtEnd = await helper.usersInDb()
  assert.strictEqual(usersAtEnd.length, usersAtStart.length + 1)

  const usernames = usersAtEnd.map(u => u.username)
  assert(usernames.includes(newUser.username))
})
```

The tests use the `usersInDb()` helper function that we implemented in the `tests/test_helper.js` file. The function is used to help us verify the state of the database after a user is created:

```
const User = require('../models/user')

// ...

const usersInDb = async () => {
  const users = await User.find({})
  return users.map(u => u.toJSON())
}

module.exports = [
  initialNotes,
  nonExistingId,
  notesInDb,
  usersInDb,
]
```

The `beforeEach` block adds a user with the username `root` to the database. We can write a new test that verifies that a new user with the same username can not be created:

```
describe('when there is initially one user in db', () => {
  // ...

  test('creation fails with proper statuscode and message if username already taken', async () => {
    const usersAtStart = await helper.usersInDb()

    const newUser = {
      username: 'root',
      name: 'Superuser',
      password: 'salainen',
    }

    const result = await api
      .post('/api/users')
      .send(newUser)
      .expect(400)
      .expect('Content-Type', /application\/json/)

    const usersAtEnd = await helper.usersInDb()
    assert(result.body.error.includes('expected `username` to be unique'))

    assert.strictEqual(usersAtEnd.length, usersAtStart.length)
  })
})
```

The test case obviously will not pass at this point. We are essentially practicing [test-driven development \(TDD\)](#), where tests for new functionality are written before the functionality is implemented.

Mongoose validations do not provide a direct way to check the uniqueness of a field value. However, it is possible to achieve uniqueness by defining [uniqueness index](#) for a field. The definition is done as follows:

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

const userSchema = mongoose.Schema({
  username: { type: String, required: true, unique: true // this ensure
    s the uniqueness of username },
  name: String,
  passwordHash: String,
  notes: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Note'
    }
  ],
})

// ...
```

However, we want to be careful when using the uniqueness index. If there are already documents in the database that violate the uniqueness condition, [no index will be created](#). So when adding a uniqueness index, make sure that the database is in a healthy state! The test above added the user with username *root* to the database twice, and these must be removed for the index to be formed and the code to work.

Mongoose validations do not detect the index violation, and instead of *ValidationError* they return an error of type *MongoServerError*. We therefore need to extend the error handler for that case:

```
const errorHandler = (error, request, response, next) => {
  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 })
  } else if (error.name === 'MongoServerError' && error.message.includes('E11000
  duplicate key error')) {
    return response.status(400).json({ error: 'expected
    `username` to be unique' })
  }
  next(error)
}
```

After these changes, the tests will pass.

We could also implement other validations into the user creation. We could check that the username is long enough, that the username only consists of permitted characters, or that the password is strong enough. Implementing these functionalities is left as an optional exercise.

Before we move onward, let's add an initial implementation of a route handler that returns all of the users in the database:

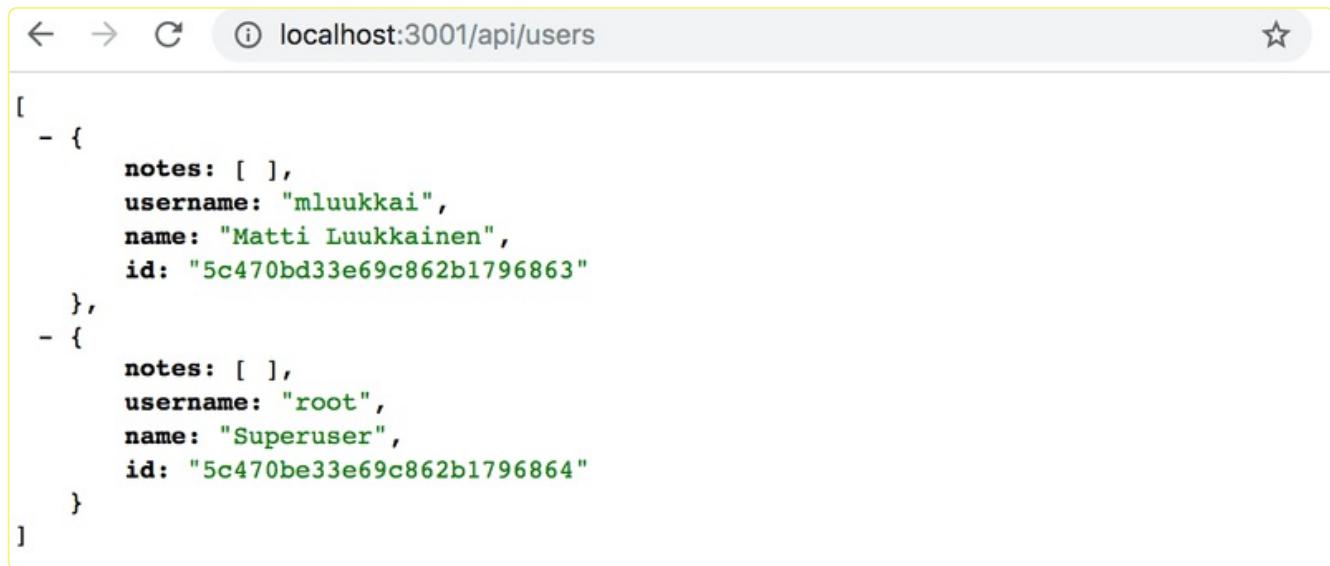
```
usersRouter.get('/', async (request, response) => {
  const users = await User.find({})
```

```
    response.json(users)
  })
}
```

For making new users in a production or development environment, you may send a POST request to [/api/users/](#) via Postman or REST Client in the following format:

```
{
  "username": "root",
  "name": "Superuser",
  "password": "salainen"
}
```

The list looks like this:



You can find the code for our current application in its entirety in the *part4-7* branch of [this GitHub repository](#).

Creating a new note

The code for creating a new note has to be updated so that the note is assigned to the user who created it.

Let's expand our current implementation in *controllers/notes.js* so that the information about the user who created a note is sent in the *userId* field of the request body:

```
const User = require('../models/user')
// ...

notesRouter.post('/', async (request, response) => {
  const body = request.body

  const user = await User.findById(body.userId)
  const note = new Note({
    content: body.content,
    important: body.important === undefined ? false : body.important,
    user
  })

  user.notes.push(note)
  await user.save()

  response.json(note)
})
```

```

    user: user.id })
}

const savedNote = await note.save()
user.notes = user.notes.concat(savedNote._id) await user.save()
response.status[201].json(savedNote)
})
}

```

It's worth noting that the `user` object also changes. The `id` of the note is stored in the `notes` field of the `user` object:

```

const user = await User.findById(body.userId)

// ...

user.notes = user.notes.concat(savedNote._id)
await user.save()

```

Let's try to create a new note

The screenshot shows a Postman interface with a POST request to `localhost:3001/api/notes`. The `Body` tab is active, displaying a JSON object:

```

1 {  
2   "content": "User id of the note creator is at the start sent along the request",  
3   "important": true,  
4   "userId": "5cf61844e8b621fe5c36fc6"  
5 }

```

The operation appears to work. Let's add one more note and then visit the route for fetching all users:

The screenshot shows a browser window with the URL `localhost:3001/api/users`. The response is a JSON array containing two user objects:

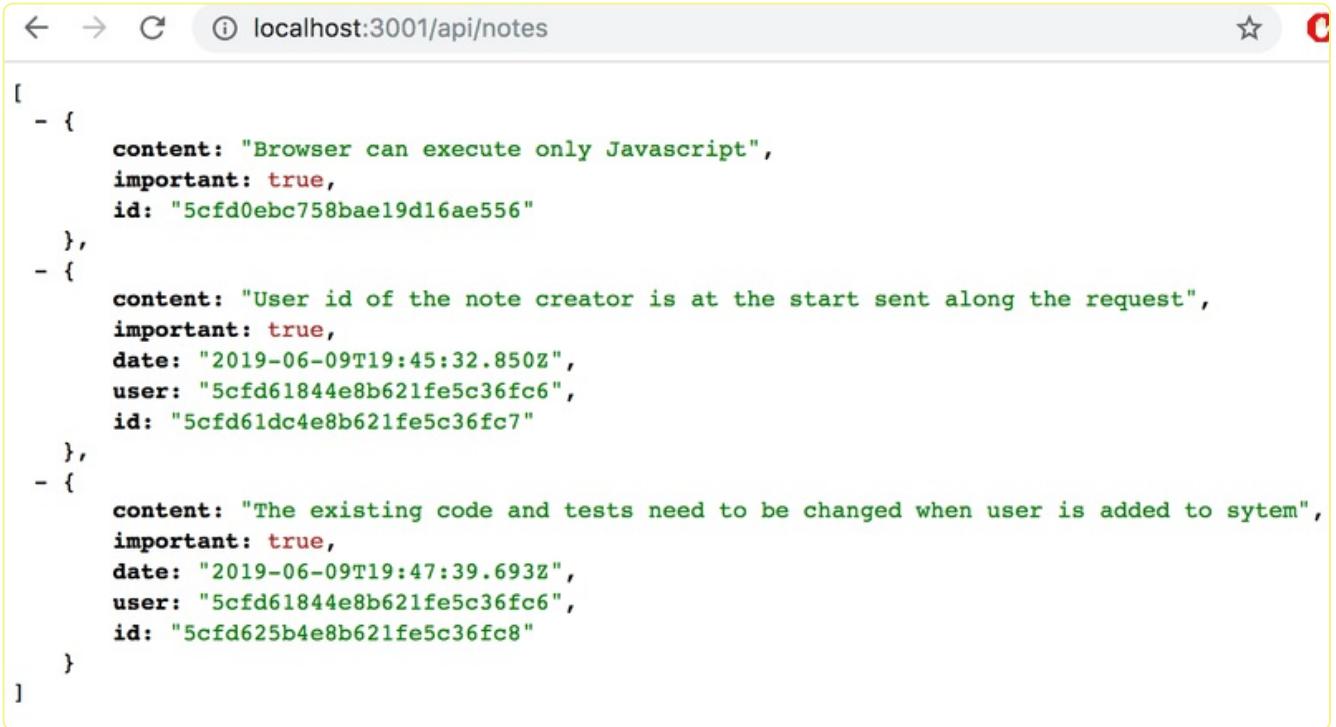
```

[
  {
    "notes": [ ],
    "username": "root",
    "id": "5cf61844e8b621fe5c36fc6"
  },
  {
    "notes": [
      "5cf61dc4e8b621fe5c36fc7",
      "5cf625b4e8b621fe5c36fc8"
    ],
    "username": "mluukkai",
    "name": "Matti Luukkainen",
    "id": "5cf61844e8b621fe5c36fc6"
  }
]

```

We can see that the user has two notes.

Likewise, the ids of the users who created the notes can be seen when we visit the route for fetching all notes:



A screenshot of a web browser window showing the URL "localhost:3001/api/notes". The page displays a JSON array of three note objects. Each note object contains fields: content, important, id, date, user, and another id. The content field for the first note is "Browser can execute only Javascript", and its id is "5cf0ebc758bae19d16ae556". The content field for the second note is "User id of the note creator is at the start sent along the request", and its id is "5cf61dc4e8b621fe5c36fc7". The content field for the third note is "The existing code and tests need to be changed when user is added to system", and its id is "5cf625b4e8b621fe5c36fc8".

```
[  
  - {  
      content: "Browser can execute only Javascript",  
      important: true,  
      id: "5cf0ebc758bae19d16ae556"  
    },  
  - {  
      content: "User id of the note creator is at the start sent along the request",  
      important: true,  
      date: "2019-06-09T19:45:32.850Z",  
      user: "5cf61844e8b621fe5c36fc6",  
      id: "5cf61dc4e8b621fe5c36fc7"  
    },  
  - {  
      content: "The existing code and tests need to be changed when user is added to system",  
      important: true,  
      date: "2019-06-09T19:47:39.693Z",  
      user: "5cf61844e8b621fe5c36fc6",  
      id: "5cf625b4e8b621fe5c36fc8"  
    }  
]
```

Populate

We would like our API to work in such a way, that when an HTTP GET request is made to the `/api/users` route, the user objects would also contain the contents of the user's notes and not just their id. In a relational database, this functionality would be implemented with a *join query*.

As previously mentioned, document databases do not properly support join queries between collections, but the Mongoose library can do some of these joins for us. Mongoose accomplishes the join by doing multiple queries, which is different from join queries in relational databases which are *transactional*, meaning that the state of the database does not change during the time that the query is made. With join queries in Mongoose, nothing can guarantee that the state between the collections being joined is consistent, meaning that if we make a query that joins the user and notes collections, the state of the collections may change during the query.

The Mongoose join is done with the `populate` method. Let's update the route that returns all users first in `controllers/users.js` file:

```
usersRouter.get('/', async [request, response] => {  
  const users = await User.find({}).populate('notes')  
  response.json(users)  
})
```

The `populate` method is chained after the `find` method making the initial query. The argument given to the `populate` method defines that the `ids` referencing `note` objects in the `notes` field of the `user` document will be replaced by the referenced `note` documents.

The result is almost exactly what we wanted:

```
< → C ⓘ http://localhost:3001/api/users

[{"id": "63cbb1878b0220944af3365", "username": "mluukkai", "name": "Matti Luukainen", "notes": [{"content": "User id of the creator is sent along the request", "important": true, "user": "63cbd3f8418ab328603adb28", "id": "63cbd3f8418ab328603adb28"}, {"content": "The existing code and tests need to be changed when user is added to the system", "important": true, "user": "63cbb1878b0220944af3365", "id": "63cbd424418ab328603adb2c"}]}
```

We can use the `populate` method for choosing the fields we want to include from the documents. In addition to the field `id` we are now only interested in `content` and `important`.

The selection of fields is done with the Mongo [syntax](#):

```
usersRouter.get('/', async [request, response] => [
  const users = await User
    .find({})
    .populate('notes', { content: 1, important: 1 })

  response.json(users)
])
```

The result is now exactly like we want it to be:

```
< → C ⓘ http://localhost:3001/api/users

[{"id": "63cbb1878b0220944af3365", "username": "mluukkai", "name": "Matti Luukainen", "notes": [{"content": "User id of the creator is sent along the request", "important": true, "id": "63cbd3f8418ab328603adb28"}, {"content": "The existing code and tests need to be changed when user is added to the system", "important": true, "id": "63cbd424418ab328603adb2c"}]}]
```

Let's also add a suitable population of user information to notes in the `controllers/notes.js` file:

```
notesRouter.get('/', async [request, response] => [
```

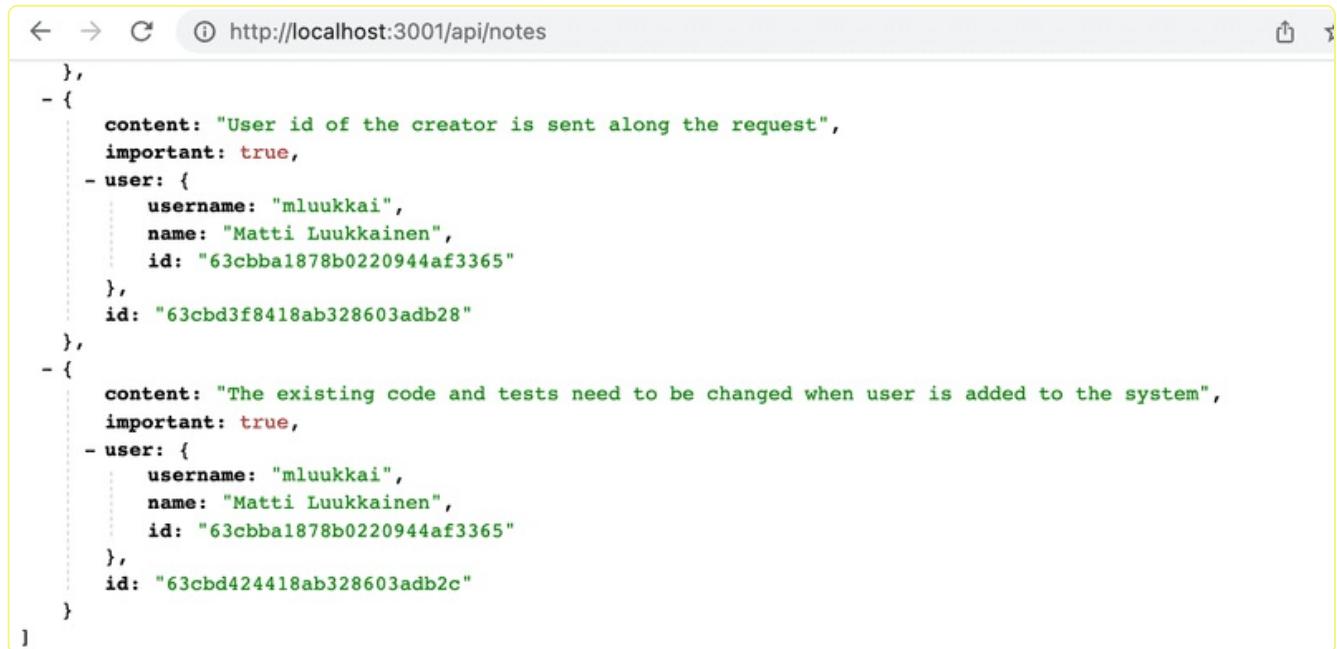
```

    const notes = await Note
      .find({})
      .populate('user', { username: 1, name: 1 })

    response.json(notes)
  })
}

```

Now the user's information is added to the `user` field of note objects.



```

{
  "content": "User id of the creator is sent along the request",
  "important": true,
  "user": {
    "username": "mluukkai",
    "name": "Matti Luukkainen",
    "id": "63cbb1878b0220944af3365"
  },
  "id": "63cbd3f8418ab328603adb28"
},
{
  "content": "The existing code and tests need to be changed when user is added to the system",
  "important": true,
  "user": {
    "username": "mluukkai",
    "name": "Matti Luukkainen",
    "id": "63cbb1878b0220944af3365"
  },
  "id": "63cbd424418ab328603adb2c"
}
]

```

It's important to understand that the database does not know that the ids stored in the `user` field of the notes collection reference documents in the user collection.

The functionality of the `populate` method of Mongoose is based on the fact that we have defined "types" to the references in the Mongoose schema with the `ref` option:

```

const noteSchema = new mongoose.Schema([
  content: {
    type: String,
    required: true,
    minlength: 5
  },
  important: Boolean,
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }
])

```

You can find the code for our current application in its entirety in the `part4-8` branch of [this GitHub repository](#).

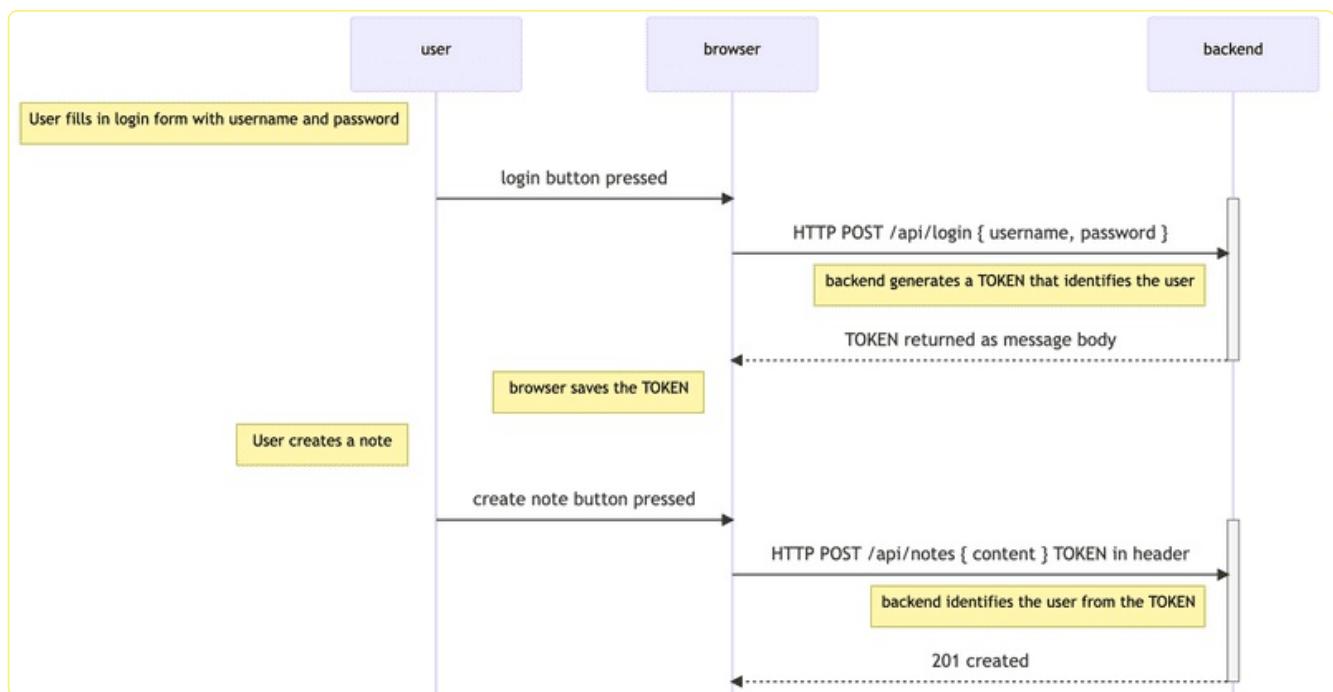
NOTE: At this stage, firstly, some tests will fail. We will leave fixing the tests to a non-compulsory exercise. Secondly, in the deployed notes app, the creating a note feature will stop working as user is not yet linked to the frontend.

4d: Token authentication

Users must be able to log into our application, and when a user is logged in, their user information must automatically be attached to any new notes they create.

We will now implement support for [token-based authentication](#) to the backend.

The principles of token-based authentication are depicted in the following sequence diagram:



- User starts by logging in using a login form implemented with React
 - We will add the login form to the frontend in [part 5](#)
- This causes the React code to send the username and the password to the server address `/api/login` as an HTTP POST request.
- If the username and the password are correct, the server generates a *token* that somehow identifies the logged-in user.
 - The token is signed digitally, making it impossible to falsify (with cryptographic means)
- The backend responds with a status code indicating the operation was successful and returns the token with the response.
- The browser saves the token, for example to the state of a React application.
- When the user creates a new note (or does some other operation requiring identification), the React code sends the token to the server with the request.
- The server uses the token to identify the user

Let's first implement the functionality for logging in. Install the [jsonwebtoken](#) library, which allows us to generate [JSON web tokens](#).

```
npm install jsonwebtoken
```

The code for login functionality goes to the file `controllers/login.js`.

```

const jwt = require('jsonwebtoken')
const bcrypt = require('bcrypt')
const loginRouter = require('express').Router()
const User = require('../models/user')

loginRouter.post('/', async (request, response) => {
  const { username, password } = request.body

  const user = await User.findOne({ username })
  const passwordCorrect = user === null
    ? false
    : await bcrypt.compare(password, user.passwordHash)

  if (!(user && passwordCorrect)) {
    return response.status(401).json({
      error: 'invalid username or password'
    })
  }

  const userForToken = {
    username: user.username,
    id: user._id,
  }

  const token = jwt.sign(userForToken, process.env.SECRET)

  response
    .status(200)
    .send({ token, username: user.username, name: user.name })
})

module.exports = loginRouter

```

The code starts by searching for the user from the database by the *username* attached to the request.

```
const user = await User.findOne({ username })
```

Next, it checks the *password*, also attached to the request.

```

const passwordCorrect = user === null
? false
: await bcrypt.compare(password, user.passwordHash)

```

Because the passwords themselves are not saved to the database, but *hashes* calculated from the passwords, the *bcrypt.compare* method is used to check if the password is correct:

```
await bcrypt.compare(password, user.passwordHash)
```

If the user is not found, or the password is incorrect, the request is responded with the status code

401 unauthorized. The reason for the failure is explained in the response body.

```
if (!user && passwordCorrect) {
  return response.status(401).json({
    error: 'invalid username or password'
  })
}
```

If the password is correct, a token is created with the method `jwt.sign`. The token contains the username and the user id in a digitally signed form.

```
const userForToken = {
  username: user.username,
  id: user._id,
}

const token = jwt.sign(userForToken, process.env.SECRET)
```

The token has been digitally signed using a string from the environment variable `SECRET` as the *secret*. The digital signature ensures that only parties who know the secret can generate a valid token. The value for the environment variable must be set in the `.env` file.

A successful request is responded to with the status code `200 OK`. The generated token and the username of the user are sent back in the response body.

```
response
  .status(200)
  .send({ token, username: user.username, name: user.name })
```

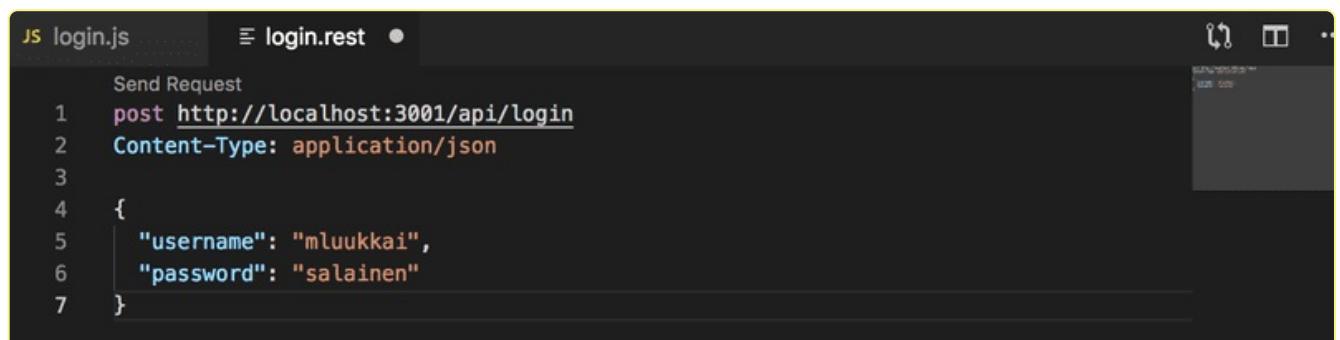
Now the code for login just has to be added to the application by adding the new router to `app.js`.

```
const loginRouter = require('./controllers/login')

// ...

app.use('/api/login', loginRouter)
```

Let's try logging in using VS Code REST-client:



```
JS login.js      login.rest •
Send Request
1 post http://localhost:3001/api/login
2 Content-Type: application/json
3
4 {
5   "username": "mluukkai",
6   "password": "salainen"
7 }
```

It does not work. The following is printed to the console:

```
(node:32911) UnhandledPromiseRejectionWarning: Error: secretOrPrivateKey must have a value
    at Object.module.exports [as sign] (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/jsonwebtoken/sign.js:101:20)
    at loginRouter.post (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/controllers/login.js:26:21)
(node:32911) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). (rejection id: 2)
```

The command `jwt.sign(userForToken, process.env.SECRET)` fails. We forgot to set a value to the environment variable `SECRET`. It can be any string. When we set the value in file `.env` (and restart the server), the login works.

A successful login returns the user details and the token:

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

{
  "username": "mluukkai",
  "password": "mluukkai"
}

HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Content-Type: application/json; charset=utf-8
Content-Length: 237
ETag: W/"ed-HLBLeA6xoDgHs2JITsEyHs9Z56A"
Date: Thu, 30 Jan 2020 11:49:00 GMT
Connection: close

{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJ1c2VybmtzS16Im1sdXVra2FpIiwiWQiOjI1ZTMxZjMyMjc2NGY0NjUxMzYxZWY4ZjQ1LCJpYXQiOjE1DAz0DQ5NDQ9.8L8Eg1JY9yv4U7-5ar7LpezUSUsWZoNsxnwh6qXCmY",
  "username": "mluukkai",
  "name": "Matti Luukkainen"
}
```

A wrong username or password returns an error message and the proper status code:

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

{
  "username": "mluukkai",
  "password": "wrong"
}

HTTP/1.1 401 Unauthorized
X-Powered-By: Express
Access-Control-Allow-Origin: *
Content-Type: application/json; charset=utf-8
Content-Length: 40
ETag: W/"28-o0F+kGSS37KN6k7gEZFvLtWpuSE"
Date: Thu, 30 Jan 2020 12:50:52 GMT
Connection: close

{
  "error": "invalid username or password"
}
```

Limiting creating new notes to logged-in users

Let's change creating new notes so that it is only possible if the post request has a valid token attached. The note is then saved to the notes list of the user identified by the token.

There are several ways of sending the token from the browser to the server. We will use the `Authorization` header. The header also tells which `authentication scheme` is used. This can be necessary if the server offers multiple ways to authenticate. Identifying the scheme tells the server

how the attached credentials should be interpreted.

The *Bearer* scheme is suitable for our needs.

In practice, this means that if the token is, for example, the string `eyJhbGciOiJIUzI1NiIsInR5c2VybmFtZSI6Im1sdXVra2Fpliwiaw`, the Authorization header will have the value:

```
Bearer eyJhbGciOiJIUzI1NiIsInR5c2VybmFtZSI6Im1sdXVra2Fpliwiaw
```

Creating new notes will change like so (*controllers/notes.js*):

```
const jwt = require('jsonwebtoken')
// ...
const getTokenFrom = request => [
  const authorization = request.get('authorization')
  if (authorization && authorization.startsWith('Bearer ')) {
    return authorization.replace('Bearer ', ' ')
  }
]
notesRouter.post('/', async [request, response] => [
  const body = request.body
  const decodedToken = jwt.verify(getTokenFrom(request), process.env.SECRET)
  if (!decodedToken.id) {
    return response.status(401).json({ error: 'token invalid' })
  }
  const user = await User.findById(decodedToken.id)
  const note = new Note({
    content: body.content,
    important: body.important === undefined ? false : body.important,
    user: user._id
  })

  const savedNote = await note.save()
  user.notes = user.notes.concat(savedNote._id)
  await user.save()

  response.json(savedNote)
])
```

The helper function *getTokenFrom* isolates the token from the *authorization* header. The validity of the token is checked with *jwt.verify*. The method also decodes the token, or returns the Object which the token was based on.

```
const decodedToken = jwt.verify(token, process.env.SECRET)
```

If the token is missing or it is invalid, the exception *JsonWebTokenError* is raised. We need to extend the error handling middleware to take care of this particular case:

```
const errorHandler = [error, request, response, next] => [
  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 })
  }
]
```

```

    } else if (error.name === 'MongoServerError' && error.message.includes('E11000
duplicate key error')) {
    return response.status(400).json({ error: 'expected `username` to be unique'
})
} else if (error.name === 'JsonWebTokenError') { return response.status(401
).json({ error: 'token invalid' })
}

next(error)
}

```

The object decoded from the token contains the *username* and *id* fields, which tell the server who made the request.

If the object decoded from the token does not contain the user's identity (*decodedToken.id* is undefined), error status code **401 unauthorized** is returned and the reason for the failure is explained in the response body.

```

if (!decodedToken.id) {
    return response.status(401).json({
        error: 'token invalid'
})
}

```

When the identity of the maker of the request is resolved, the execution continues as before.

A new note can now be created using Postman if the *authorization* header is given the correct value, the string *Bearer eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ*, where the second value is the token returned by the *login* operation.

Using Postman this looks as follows:

The screenshot shows the Postman interface with a POST request to `http://localhost:3001/api/notes`. The `Headers (9)` tab is selected. A red arrow points to the `Authorization` header row, which has a checked checkbox and the value `Bearer eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9eyJ1...`.

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Authorization	Bearer eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9eyJ1...				

and with Visual Studio Code REST client

```

requests > ⚡ create_note.rest > ⚡ POST /api/notes
Send Request | You, 38 seconds ago | 1 author (You)
1 POST http://localhost:3001/api/notes
2 Content-Type: application/json
3 Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlc2VybmFtZSI6Im
4
5 {
6   "content": "Single page apps use token based auth",
7   "important": false
8 }

1 HTTP/1.1 201 Created
2 X-Powered-By: Express
3 Access-Control-Allow-Origin: *
4 Content-Type: application/json; charset=utf-8
5 Content-Length: 135
6 ETag: W/"87-0SGk+ftbONFeLhgI1sGRRQsJXVM"
7 Date: Tue, 13 Feb 2024 16:18:32 GMT
8 Connection: close
9
10 {
11   "content": "Single page apps use token based auth",
12   "important": false,
13   "user": "65cb3eaa7ae575843105587",
14   "id": "65cb96587e615c19038c4bb7"
15 }

```

Current application code can be found on [GitHub](#), branch `part4-9`.

If the application has multiple interfaces requiring identification, JWT's validation should be separated into its own middleware. An existing library like [express-jwt](#) could also be used.

Problems of Token-based authentication

Token authentication is pretty easy to implement, but it contains one problem. Once the API user, eg. a React app gets a token, the API has a blind trust to the token holder. What if the access rights of the token holder should be revoked?

There are two solutions to the problem. The easier one is to limit the validity period of a token:

```

loginRouter.post('/' , async (request, response) => [
  const { username, password } = request.body

  const user = await User.findOne({ username })
  const passwordCorrect = user === null
    ? false
    : await bcrypt.compare(password, user.passwordHash)

  if (!(user && passwordCorrect)) {
    return response.status(401).json({
      error: 'invalid username or password'
    })
  }

  const userForToken = [
    username: user.username,
    id: user._id,
  ]

  // token expires in 60*60 seconds, that is, in one hour
  const token = jwt.sign(userForToken, process.env.SECRET, { expiresIn:
  : 60*60 })
  response
    .status(200)
    .send({ token, username: user.username, name: user.name })
]

```

Once the token expires, the client app needs to get a new token. Usually, this happens by forcing the

user to re-login to the app.

The error handling middleware should be extended to give a proper error in the case of an expired token:

```
const errorHandler = (error, request, response, next) => {
  logger.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 })
  } else if (error.name === 'MongoServerError' && error.message.includes('E11000
duplicate key error')) {
    return response.status(400).json({
      error: 'expected `username` to be unique'
    })
  } else if (error.name === 'JsonWebTokenError') {
    return response.status(401).json({
      error: 'invalid token'
    })
  } else if (error.name === 'TokenExpiredError') {      return response.status(401)
.json({      error: 'token expired'      })
  }
  next(error)
}
```

The shorter the expiration time, the more safe the solution is. So if the token gets into the wrong hands or user access to the system needs to be revoked, the token is only usable for a limited amount of time. On the other hand, a short expiration time forces a potential pain to a user, one must login to the system more frequently.

The other solution is to save info about each token to the backend database and to check for each API request if the access rights corresponding to the tokens are still valid. With this scheme, access rights can be revoked at any time. This kind of solution is often called a *server-side session*.

The negative aspect of server-side sessions is the increased complexity in the backend and also the effect on performance since the token validity needs to be checked for each API request to the database. Database access is considerably slower compared to checking the validity of the token itself. That is why it is quite common to save the session corresponding to a token to a *key-value database* such as [Redis](#), that is limited in functionality compared to eg. MongoDB or a relational database, but extremely fast in some usage scenarios.

When server-side sessions are used, the token is quite often just a random string, that does not include any information about the user as it is quite often the case when jwt-tokens are used. For each API request, the server fetches the relevant information about the identity of the user from the database. It is also quite usual that instead of using Authorization-header, *cookies* are used as the mechanism for transferring the token between the client and the server.

End notes

There have been many changes to the code which have caused a typical problem for a fast-paced

software project: most of the tests have broken. Because this part of the course is already jammed with new information, we will leave fixing the tests to a non-compulsory exercise.

Usernames, passwords and applications using token authentication must always be used over [HTTPS](#). We could use a Node [HTTPS](#) server in our application instead of the [HTTP](#) server (it requires more configuration). On the other hand, the production version of our application is in Fly.io, so our application stays secure: Fly.io routes all traffic between a browser and the Fly.io server over HTTPS.

We will implement login to the frontend in the [next part](#).

NOTE: At this stage, in the deployed notes app, it is expected that the creating a note feature will stop working as the backend login feature is not yet linked to the frontend.

Exercises 4.15.-4.23.

In the next exercises, the basics of user management will be implemented for the Bloglist application. The safest way is to follow the course material from part 4 chapter [User administration](#) to the chapter [Token authentication](#). You can of course also use your creativity.

One more warning: If you notice you are mixing `async/await` and `then` calls, it is 99% certain you are doing something wrong. Use either or, never both.

4.15: Blog List Expansion, step 3

Implement a way to create new users by doing an HTTP POST request to address `api/users`. Users have a *username*, *password* and *name*.

Do not save passwords to the database as clear text, but use the `bcrypt` library like we did in part 4 chapter [Creating users](#).

NB Some Windows users have had problems with `bcrypt`. If you run into problems, remove the library with command

```
npm uninstall bcrypt
```

and install [bcryptjs](#) instead.

Implement a way to see the details of all users by doing a suitable HTTP request.

The list of users can, for example, look as follows:



```
[  
  - {  
      username: "hellas",  
      name: "Arto Hellas",  
      id: "5c4857b1003ad1a6e6626931"  
    },  
  - {  
      username: "mluukkai",  
      name: "Matti Luukkainen",  
      id: "5c4857c4003ad1a6e6626932"  
    }  
]
```

4.16*: Blog List Expansion, step 4

Add a feature which adds the following restrictions to creating new users: Both username and password must be given and both must be at least 3 characters long. The username must be unique.

The operation must respond with a suitable status code and some kind of an error message if an invalid user is created.

NB Do not test password restrictions with Mongoose validations. It is not a good idea because the password received by the backend and the password hash saved to the database are not the same thing. The password length should be validated in the controller as we did in [part 3](#) before using Mongoose validation.

Also, **implement tests** that ensure invalid users are not created and that an invalid add user operation returns a suitable status code and error message.

NB if you decide to define tests on multiple files, you should note that by default each test file is executed in its own process (see *Test execution model* in the [documentation](#)). The consequence of this is that different test files are executed at the same time. Since the tests share the same database, simultaneous execution may cause problems, which can be avoided by executing the tests with the option `--test-concurrency=1`, i.e. defining them to be executed sequentially.

4.17: Blog List Expansion, step 5

Expand blogs so that each blog contains information on the creator of the blog.

Modify adding new blogs so that when a new blog is created, *any* user from the database is designated as its creator (for example the one found first). Implement this according to part 4 chapter [populate](#). Which user is designated as the creator does not matter just yet. The functionality is finished in exercise 4.19.

Modify listing all blogs so that the creator's user information is displayed with the blog:

```
← → ⌂ ⓘ localhost:3003/api/blogs

[ - {
    url: "https://overreacted.io/things-i-dont-know-as-of-2018/",
    title: "Things I Don't Know as of 2018",
    author: "Dan Abramov",
    - user: {
        username: "mluukkai",
        name: "Matti Luukkainen",
        id: "5cfde192de5f227463ff6a4b"
    },
    likes: 0,
    id: "5cfde22ade5f227463ff6a4d"
},
- {
    url: "https://martinfowler.com/articles/distributed-objects-microservices.html",
    title: "Microservices and the First Law of Distributed Objects",
    author: "Martin Fowler",
    - user: {
        username: "hellas",
        name: "Arto Hellas",
        id: "5cfde19dde5f227463ff6a4c"
    },
    likes: 0,
    id: "5cfde2bfde5f227463ff6a4e"
}
]
```

and listing all users also displays the blogs created by each user:

```
← → ⌂ ⓘ localhost:3003/api/users

[ - {
    - blogs: [
        - {
            url: "https://overreacted.io/things-i-dont-know-as-of-2018/",
            title: "Things I Don't Know as of 2018",
            author: "Dan Abramov",
            id: "5cfde22ade5f227463ff6a4d"
        }
    ],
    username: "mluukkai",
    name: "Matti Luukkainen",
    id: "5cfde192de5f227463ff6a4b"
},
- {
    - blogs: [
        - {
            url: "https://martinfowler.com/articles/distributed-objects-microservices.html",
            title: "Microservices and the First Law of Distributed Objects",
            author: "Martin Fowler",
            id: "5cfde2bfde5f227463ff6a4e"
        }
    ],
    username: "hellas",
    name: "Arto Hellas",
    id: "5cfde19dde5f227463ff6a4c"
}
]
```

4.18: Blog List Expansion, step 6

Implement token-based authentication according to part 4 chapter [Token authentication](#).

4.19: Blog List Expansion, step 7

Modify adding new blogs so that it is only possible if a valid token is sent with the HTTP POST request. The user identified by the token is designated as the creator of the blog.

4.20*: Blog List Expansion, step 8

This example from part 4 shows taking the token from the header with the `getTokenFrom` helper function in `controllers/blogs.js`.

If you used the same solution, refactor taking the token to a [middleware](#). The middleware should take the token from the `Authorization` header and assign it to the `token` field of the `request` object.

In other words, if you register this middleware in the `app.js` file before all routes

```
app.use(middleware.tokenExtractor)
```

Routes can access the token with `request.token`:

```
blogsRouter.post('/', async [request, response] => [
  // ...
  const decodedToken = jwt.verify(request.token, process.env.SECRET)
  // ...
])
```

Remember that a normal [middleware function](#) is a function with three parameters, that at the end calls the last parameter `next` to move the control to the next middleware:

```
const tokenExtractor = (request, response, next) => [
  // code that extracts the token

  next()
]
```

4.21*: Blog List Expansion, step 9

Change the delete blog operation so that a blog can be deleted only by the user who added it. Therefore, deleting a blog is possible only if the token sent with the request is the same as that of the blog's creator.

If deleting a blog is attempted without a token or by an invalid user, the operation should return a suitable status code.

Note that if you fetch a blog from the database,

```
const blog = await Blog.findById(...)
```

the field `blog.user` does not contain a string, but an object. So if you want to compare the ID of the object fetched from the database and a string ID, a normal comparison operation does not work. The

ID fetched from the database must be parsed into a string first.

```
if (blog.user.toString() === userid.toString()) { ... }
```

4.22*: Blog List Expansion, step 10

Both the new blog creation and blog deletion need to find out the identity of the user who is doing the operation. The middleware *tokenExtractor* that we did in exercise 4.20 helps but still both the handlers of *post* and *delete* operations need to find out who the user holding a specific token is.

Now create a new middleware *userExtractor*, that finds out the user and sets it to the request object. When you register the middleware in *app.js*

```
app.use(middleware.userExtractor)
```

the user will be set in the field *request.user*:

```
blogsRouter.post('/', async (request, response) => {
  // get user from request object
  const user = request.user
  // ...
}

blogsRouter.delete('/:id', async (request, response) => {
  // get user from request object
  const user = request.user
  // ...
})
```

Note that it is possible to register a middleware only for a specific set of routes. So instead of using *userExtractor* with all the routes,

```
const middleware = require('../utils/middleware')
// ...

// use the middleware in all routes
app.use(middleware.userExtractor)
app.use('/api/blogs', blogsRouter)
app.use('/api/users', usersRouter)
app.use('/api/login', loginRouter)
```

we could register it to be only executed with path */api/blogs* routes:

```
const middleware = require('../utils/middleware')
// ...

// use the middleware only in /api/blogs routes
app.use('/api/blogs', middleware.userExtractor, blogsRouter).use('/api/users', usersRouter)
```

```
app.use('/api/login', loginRouter)
```

As can be seen, this happens by chaining multiple middlewares as the arguments of the function `use`. It would also be possible to register a middleware only for a specific operation:

```
const middleware = require('../utils/middleware');
// ...

router.post('/', middleware.userExtractor, async (request, response) => [
    // ...
])
```

4.23*: Blog List Expansion, step 11

After adding token-based authentication the tests for adding a new blog broke down. Fix them. Also, write a new test to ensure adding a blog fails with the proper status code *401 Unauthorized* if a token is not provided.

This is most likely useful when doing the fix.

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

4e: Legacy: Testing with Jest

This is the old (pre 13th February 2024) content on testing that is using Jest as the testing library. You may continue using this material if you have already started writing tests with Jest. In other case, you should ignore this page.

Testing Node applications

We have completely neglected one essential area of software development, and that is automated testing.

Let's start our testing journey by looking at unit tests. The logic of our application is so simple, that there is not much that makes sense to test with unit tests. Let's create a new file `utils/for_testing.js` and write a couple of simple functions that we can use for test writing practice:

```
const reverse = (string) => {
    return string
        .split('')
        .reverse()
        .join('')

}

const average = (array) => {
    const reducer = (sum, item) => {
        return sum + item
    }

    return array.reduce(reducer, 0)
        / array.length
}
```

```
    return array.reduce(reducer, 0) / array.length
}

module.exports = [
  reverse,
  average,
]
```

The *average* function uses the array `reduce` method. If the method is not familiar to you yet, then now is a good time to watch the first three videos from the [Functional Javascript](#) series on YouTube.

There are many different testing libraries or *test runners* available for JavaScript. In this course we will be using a testing library developed and used internally by Facebook called [jest](#), which resembles the previous king of JavaScript testing libraries [Mocha](#).

Jest is a natural choice for this course, as it works well for testing backends, and it shines when it comes to testing React applications.

Windows users: Jest may not work if the path of the project directory contains a directory that has spaces in its name.

Since tests are only executed during the development of our application, we will install *jest* as a development dependency with the command:

```
npm install --save-dev jest
```

Let's define the *npm script test* to execute tests with Jest and to report about the test execution with the *verbose* style:

```
{
  ...
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "build:ui": "rm -rf build && cd ../frontend/ && npm run build && cp -r build ..../backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs",
    "lint": "eslint .",
    "test": "jest --verbose"
  },
  ...
}
```

Jest requires one to specify that the execution environment is Node. This can be done by adding the following to the end of *package.json*:

```
[  
  // ...  
  "jest": {  
    "testEnvironment": "node"  
  }  
]
```

Let's create a separate directory for our tests called *tests* and create a new file called *reverse.test.js* with the following contents:

```
const reverse = require('../utils/for_testing').reverse  
  
test('reverse of a', () => {  
  const result = reverse('a')  
  
  expect(result).toBe('a')  
})  
  
test('reverse of react', () => {  
  const result = reverse('react')  
  
  expect(result).toBe('tcaer')  
})  
  
test('reverse of releveler', () => {  
  const result = reverse('releveler')  
  
  expect(result).toBe('releveler')  
})
```

The ESLint configuration we added to the project in the previous part complains about the *test* and *expect* commands in our test file since the configuration does not allow *globals*. Let's get rid of the complaints by adding "*jest*: *true*" to the *env* property in the *.eslintrc.js* file.

```
module.exports = [  
  'env': {  
    'commonjs': true,  
    'es2021': true,  
    'node': true,  
    'jest': true,  },  
  // ...  
]
```

In the first row, the test file imports the function to be tested and assigns it to a variable called *reverse*:

```
const reverse = require('../utils/for_testing').reverse
```

Individual test cases are defined with the *test* function. The first parameter of the function is the test description as a string. The second parameter is a *function*, that defines the functionality for the test case. The functionality for the second test case looks like this:

```
() => {
  const result = reverse('react')

  expect(result).toBe('tcaer')
}
```

First, we execute the code to be tested, meaning that we generate a reverse for the string *react*. Next, we verify the results with the *expect* function. Expect wraps the resulting value into an object that offers a collection of *matcher* functions, that can be used for verifying the correctness of the result. Since in this test case we are comparing two strings, we can use the *toBe* matcher.

As expected, all of the tests pass:

```
→ noteapp git:(part4-2) ✘ npm test

> noteapp@1.0.0 test
> jest --verbose

PASS tests/reverse.test.js
  ✓ reverse of a (2 ms)
  ✓ reverse of react (1 ms)
  ✓ reverse of saippuakauppias

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:  0 total
Time:        0.804 s
Ran all test suites.
```

Jest expects by default that the names of test files contain *.test*. In this course, we will follow the convention of naming our tests files with the extension *.test.js*.

Jest has excellent error messages, let's break the test to demonstrate this:

```
test('reverse of react', () => {
  const result = reverse('react')

  expect(result).toBe('tkaer')
})
```

Running this test results in the following error message:

```

→ noteapp git:(part4-2) ✘ npm test

> noteapp@1.0.0 test
> jest --verbose

FAIL | tests/reverse.test.js
  ✓ reverse of a (1 ms)
  ✗ reverse of react (4 ms)
  ✓ reverse of saippuakauppias

● reverse of react

  expect(received).toBe(expected) // Object.is equality

    Expected: "t̄kaer"
    Received: "t̄caer"

    10 |   const result = reverse('react')
    11 |
    > 12 |   expect(result).toBe('t̄kaer')
           ^
    13 | }
    14 |
    15 | test('reverse of saippuakauppias', () => {

```

Let's add a few tests for the `average` function, into a new file `tests/average.test.js`.

```

const average = require('../utils/for_testing').average

describe('average', () => {
  test('of one value is the value itself', () => {
    expect(average([1])).toBe(1)
  })

  test('of many is calculated right', () => {
    expect(average([1, 2, 3, 4, 5, 6])).toBe(3.5)
  })

  test('of empty array is zero', () => {
    expect(average([])).toBe(0)
  })
})

```

The test reveals that the function does not work correctly with an empty array (this is because in JavaScript dividing by zero results in `Nan`):

```

FAIL tests/average.test.js
average
  ✓ of one value is the value itself (2ms)
  ✓ of many is calculated right (1ms)
  ✘ of empty array is zero (10ms)

  • average > of empty array is zero

    expect(received).toBe(expected) // Object.is equality

    Expected: 0
    Received: NaN

    11 |
    12 |   test('of empty array is zero', () => {
    > 13 |     expect(average([])).toBe(0)
          ^
    14 |   })
    15 | }

    at Object.toBe (tests/average.test.js:13:25)

Test Suites: 1 failed, 1 passed, 2 total
Tests:       1 failed, 5 passed, 6 total
Snapshots:  0 total
Time:        0.736s, estimated 1s
Ran all test suites.

```

Fixing the function is quite easy:

```

const average = array => [
  const reducer = (sum, item) => {
    return sum + item
  }

  return array.length === 0
  ? 0
  : array.reduce(reducer, 0) / array.length
]

```

If the length of the array is 0 then we return 0, and in all other cases, we use the *reduce* method to calculate the average.

There are a few things to notice about the tests that we just wrote. We defined a *describe* block around the tests that were given the name *average*:

```

describe('average', () => {
  // tests
})

```

Describe blocks can be used for grouping tests into logical collections. The test output of Jest also uses the name of the describe block:

```
Ran all test suites.
→ noteapp git:(part4-2) ✘ npm test

> noteapp@1.0.0 test
> jest --verbose

PASS tests/average.test.js
  average
    ✓ of one value is the value itself (2 ms)
    ✓ of many is calculated right (1 ms)
    ✓ of empty array is zero (1 ms)

PASS tests/reverse.test.js
  ✓ reverse of a
  ✓ reverse of react
  ✓ reverse of saippuakauppias
```

As we will see later on *describe* blocks are necessary when we want to run some shared setup or teardown operations for a group of tests.

Another thing to notice is that we wrote the tests in quite a compact way, without assigning the output of the function being tested to a variable:

```
test('of empty array is zero', () => [
  expect(average([])).toBe(0)
])
```

Exercises 4.3.-4.7.

Let's create a collection of helper functions that are meant to assist in dealing with the blog list. Create the functions into a file called *utils/list_helper.js*. Write your tests into an appropriately named test file under the *tests* directory.

4.3: Helper Functions and Unit Tests, step 1

First, define a *dummy* function that receives an array of blog posts as a parameter and always returns the value 1. The contents of the *list_helper.js* file at this point should be the following:

```
const dummy = (blogs) => [
  // ...
]

module.exports = [
  dummy
]
```

Verify that your test configuration works with the following test:

```
const listHelper = require('../utils/list_helper')

test('dummy returns one', () => [
  const blogs = []
```

```

    const result = listHelper.dummy(blogs)
    expect(result).toBe(1)
  })
}

```

4.4: Helper Functions and Unit Tests, step 2

Define a new *totalLikes* function that receives a list of blog posts as a parameter. The function returns the total sum of *likes* in all of the blog posts.

Write appropriate tests for the function. It's recommended to put the tests inside of a *describe* block so that the test report output gets grouped nicely:

```

PASS  tests/list_helper.test.js
  ✓ dummy returns 1 (7ms)
  total likes
    ✓ of empty list is zero (1ms)
    ✓ when list has only one blog equals the likes of that (1ms)
    ✓ of a bigger list is calculated right (1ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total

```

Defining test inputs for the function can be done like this:

```

describe('total likes', () => {
  const listWithOneBlog = [
    {
      _id: '5a422aa71b54a676234d17f8',
      title: 'Go To Statement Considered Harmful',
      author: 'Edsger W. Dijkstra',
      url: 'https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf',
      likes: 5,
      __v: 0
    }
  ]

  test('when list has only one blog, equals the likes of that', () => {
    const result = listHelper.totalLikes(listWithOneBlog)
    expect(result).toBe(5)
  })
})

```

If defining your own test input list of blogs is too much work, you can use the ready-made list [here](#).

You are bound to run into problems while writing tests. Remember the things that we learned about [debugging](#) in part 3. You can print things to the console with *console.log* even during test execution. It is even possible to use the debugger while running tests, you can find instructions for that [here](#).

NB: if some test is failing, then it is recommended to only run that test while you are fixing the issue. You can run a single test with the [only](#) method.

Another way of running a single test (or describe block) is to specify the name of the test to be run

with the `-t` flag:

```
npm test -- -t 'when list has only one blog, equals the likes of that'
```

4.5*: Helper Functions and Unit Tests, step 3

Define a new `favoriteBlog` function that receives a list of blogs as a parameter. The function finds out which blog has the most likes. If there are many top favorites, it is enough to return one of them.

The value returned by the function could be in the following format:

```
{  
  title: "Canonical string reduction",  
  author: "Edsger W. Dijkstra",  
  likes: 12  
}
```

NB when you are comparing objects, the `toEqual` method is probably what you want to use, since the `toBe` tries to verify that the two values are the same value, and not just that they contain the same properties.

Write the tests for this exercise inside of a new `describe` block. Do the same for the remaining exercises as well.

4.6*: Helper Functions and Unit Tests, step 4

This and the next exercise are a little bit more challenging. Finishing these two exercises is not required to advance in the course material, so it may be a good idea to return to these once you're done going through the material for this part in its entirety.

Finishing this exercise can be done without the use of additional libraries. However, this exercise is a great opportunity to learn how to use the [Lodash](#) library.

Define a function called `mostBlogs` that receives an array of blogs as a parameter. The function returns the `author` who has the largest amount of blogs. The return value also contains the number of blogs the top author has:

```
{  
  author: "Robert C. Martin",  
  blogs: 3  
}
```

If there are many top bloggers, then it is enough to return any one of them.

4.7*: Helper Functions and Unit Tests, step 5

Define a function called `mostLikes` that receives an array of blogs as its parameter. The function returns the author, whose blog posts have the largest amount of likes. The return value also contains the total number of likes that the author has received:

```
[{"author": "Edsger W. Dijkstra",  
 "likes": 17  
}]
```

If there are many top bloggers, then it is enough to show any one of them.

We will now start writing tests for the backend. Since the backend does not contain any complicated logic, it doesn't make sense to write [unit tests](#) for it. The only potential thing we could unit test is the `toJSON` method that is used for formatting notes.

In some situations, it can be beneficial to implement some of the backend tests by mocking the database instead of using a real database. One library that could be used for this is [mongodb-memory-server](#).

Since our application's backend is still relatively simple, we will decide to test the entire application through its REST API, so that the database is also included. This kind of testing where multiple components of the system are being tested as a group is called [integration testing](#).

Test environment

In one of the previous chapters of the course material, we mentioned that when your backend server is running in Fly.io or Render, it is in *production* mode.

The convention in Node is to define the execution mode of the application with the `NODE_ENV` environment variable. In our current application, we only load the environment variables defined in the `.env` file if the application is *not* in production mode.

It is common practice to define separate modes for development and testing.

Next, let's change the scripts in our notes application `package.json` file, so that when tests are run, `NODE_ENV` gets the value `test`:

```
[{  
  // ...  
  "scripts": [  
    "start": "NODE_ENV=production node index.js",      "dev": "NODE_ENV=development nodemon index.js",  
    "build:ui": "rm -rf build && cd .. frontend/ && npm run build && cp -r build .. backend",  
    "deploy": "fly deploy",  
    "deploy:full": "npm run build:ui && npm run deploy",  
    "logs:prod": "fly logs",  
    "lint": "eslint .",  
    "test": "NODE_ENV=test jest --verbose --runInBand"  ],  
  // ...  
}]
```

We also added the `runInBand` option to the npm script that executes the tests. This option will prevent Jest from running tests in parallel; we will discuss its significance once our tests start using the database.

We specified the mode of the application to be *development* in the `npm run dev` script that uses

nodemon. We also specified that the default `npm start` command will define the mode as *production*.

There is a slight issue in the way that we have specified the mode of the application in our scripts: it will not work on Windows. We can correct this by installing the [cross-env](#) package as a development dependency with the command:

```
npm install --save-dev cross-env
```

We can then achieve cross-platform compatibility by using the `cross-env` library in our npm scripts defined in `package.json`:

```
[{  
  // ...  
  "scripts": {  
    "start": "cross-env NODE_ENV=production node index.js",  
    "dev": "cross-env NODE_ENV=development nodemon index.js",  
    // ...  
    "test": "cross-env NODE_ENV=test jest --verbose --runInBand",  
  },  
  // ...  
}
```

NB: If you are deploying this application to Fly.io/Render, keep in mind that if `cross-env` is saved as a development dependency, it would cause an application error on your web server. To fix this, change `cross-env` to a production dependency by running this in the command line:

```
npm install cross-env
```

Now we can modify the way that our application runs in different modes. As an example of this, we could define the application to use a separate test database when it is running tests.

We can create our separate test database in MongoDB Atlas. This is not an optimal solution in situations where many people are developing the same application. Test execution in particular typically requires a single database instance that is not used by tests that are running concurrently.

It would be better to run our tests using a database that is installed and running on the developer's local machine. The optimal solution would be to have every test execution use a separate database. This is "relatively simple" to achieve by [running Mongo in-memory](#) or by using [Docker](#) containers. We will not complicate things and will instead continue to use the MongoDB Atlas database.

Let's make some changes to the module that defines the application's configuration in `utils/config.js`:

```
require('dotenv').config()  
  
const PORT = process.env.PORT  
  
const MONGODB_URI = process.env.NODE_ENV === 'test' ? process.env.TEST_MONGODB  
_URI : process.env.MONGODB_URI  
module.exports = [  
  MONGODB_URI,  
  PORT
```

The `.env` file has *separate variables* for the database addresses of the development and test databases:

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

TEST_MONGODB_URI=mongodb+srv://fullstack:thepasswordishere@cluster0.oloql.mongodb.net/test
NoteApp?retryWrites=true&w=majority
```

The `config` module that we have implemented slightly resembles the [node-config](#) package. Writing our implementation is justified since our application is simple, and also because it teaches us valuable lessons.

These are the only changes we need to make to our application's code.

You can find the code for our current application in its entirety in the *part4-2* branch of [this GitHub repository](#).

supertest

Let's use the `supertest` package to help us write our tests for testing the API.

We will install the package as a development dependency:

```
npm install --save-dev supertest
```

Let's write our first test in the `tests/note_api.test.js` file:

```
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')

const api = supertest(app)

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

afterAll(async () => {
  await mongoose.connection.close()
})
```

The test imports the Express application from the `app.js` module and wraps it with the `supertest` function into a so-called `superagent` object. This object is assigned to the `api` variable and tests can

use it for making HTTP requests to the backend.

Our test makes an HTTP GET request to the *api/notes* url and verifies that the request is responded to with the status code 200. The test also verifies that the *Content-Type* header is set to *application/json*, indicating that the data is in the desired format.

Checking the value of the header uses a bit strange looking syntax:

```
- expect(['Content-Type', /application\/json/])
```

The desired value is now defined as [regular expression](#) or in short regex. The regex starts and ends with a slash /, because the desired string *application/json* also contains the same slash, it is preceded by a \ so that it is not interpreted as a regex termination character.

In principle, the test could also have been defined as a string

```
- expect(['Content-Type', 'application/json'])
```

The problem here, however, is that when using a string, the value of the header must be exactly the same. For the regex we defined, it is acceptable that the header *contains* the string in question. The actual value of the header is *application/json; charset=utf-8*, i.e. it also contains information about character encoding. However, our test is not interested in this and therefore it is better to define the test as a regex instead of an exact string.

The test contains some details that we will explore [a bit later on](#). The arrow function that defines the test is preceded by the *async* keyword and the method call for the *api* object is preceded by the *await* keyword. We will write a few tests and then take a closer look at this *async/await* magic. Do not concern yourself with them for now, just be assured that the example tests work correctly. The *async/await* syntax is related to the fact that making a request to the API is an *asynchronous* operation. The [*async/await* syntax](#) can be used for writing asynchronous code with the appearance of synchronous code.

Once all the tests (there is currently only one) have finished running we have to close the database connection used by Mongoose. This can be easily achieved with the [*afterAll*](#) method:

```
afterAll(async () => {
  await mongoose.connection.close()
})
```

When running your tests you may run across the following console warning:

```
Jest did not exit one second after the test run has completed.
```

```
This usually means that there are asynchronous operations that weren't stopped in your tests. Consider running Jest with '--detectOpenHandles' to troubleshoot this issue.
```

The problem is quite likely caused by the Mongoose version 6.x, the problem does not appear when version 5.x or 7.x is used. [Mongoose documentation](#) does not recommend testing Mongoose applications with Jest.

[One way](#) to get rid of this is to add to the directory *tests* a file *teardown.js* with the following content

```
module.exports = () => {
  process.exit(0)
}
```

and by extending the Jest definitions in the *package.json* as follows

```
{
  // ...
  "jest": {
    "testEnvironment": "node",
    "globalTeardown": "./tests/teardown.js"
  }
}
```

Another error you may come across is your test takes longer than the default Jest test timeout of 5000 ms. This can be solved by adding a third parameter to the test function:

```
test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
    , 100000)
```

This third parameter sets the timeout to 100000 ms. A long timeout ensures that our test won't fail due to the time it takes to run. (A long timeout may not be what you want for tests based on performance or speed, but this is fine for our example tests).

If you still encounter issues with mongoose timeouts, set `bufferTimeoutMS` variable to a value significantly higher than 10000 (10 seconds). You could set it like this at the top, right after the `require` statements. `mongoose.set("bufferTimeoutMS", 30000)`

One tiny but important detail: at the [beginning](#) of this part we extracted the Express application into the *app.js* file, and the role of the *index.js* file was changed to launch the application at the specified port via `app.listen`:

```
const app = require('./app') // the actual Express app
const config = require('./utils/config')
const logger = require('./utils/logger')

app.listen(config.PORT, () => [
  logger.info(`Server running on port ${config.PORT}`)
])
```

The tests only use the Express application defined in the *app.js* file, which does not listen to any ports:

```
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')
```

```
const api = supertest(app)
// ...
```

The documentation for *supertest* says the following:

if the server is not already listening for connections then it is bound to an ephemeral port for you so there is no need to keep track of ports.

In other words, *supertest* takes care that the application being tested is started at the port that it uses internally.

Let's add two notes to the test database using the *mongo.js* program (here we must remember to switch to the correct database url).

Let's write a few more tests:

```
test('there are two notes', async () => {
  const response = await api.get('/api/notes')

  expect(response.body).toHaveLength(2)
})

test('the first note is about HTTP methods', async () => {
  const response = await api.get('/api/notes')

  expect(response.body[0].content).toBe('HTML is easy')
})
```

Both tests store the response of the request to the *response* variable, and unlike the previous test that used the methods provided by *supertest* for verifying the status code and headers, this time we are inspecting the response data stored in *response.body* property. Our tests verify the format and content of the response data with the *expect* method of Jest.

The benefit of using the *async/await* syntax is starting to become evident. Normally we would have to use callback functions to access the data returned by promises, but with the new syntax things are a lot more comfortable:

```
const response = await api.get('/api/notes')

// execution gets here only after the HTTP request is complete
// the result of HTTP request is saved in variable response
expect(response.body).toHaveLength(2)
```

The middleware that outputs information about the HTTP requests is obstructing the test execution output. Let us modify the logger so that it does not print to the console in test mode:

```
const info = (...params) => {
  if (process.env.NODE_ENV !== 'test') {
    console.log(...params)
  }
}
```

```

const error = (...params) => {
  if (process.env.NODE_ENV !== 'test') { console.error(...params) }
}

module.exports = [
  info,
  error
]

```

Initializing the database before tests

Testing appears to be easy and our tests are currently passing. However, our tests are bad as they are dependent on the state of the database, that now happens to have two notes. To make them more robust, we have to reset the database and generate the needed test data in a controlled manner before we run the tests.

Our tests are already using the `afterAll` function of Jest to close the connection to the database after the tests are finished executing. Jest offers many other `functions` that can be used for executing operations once before any test is run or every time before a test is run.

Let's initialize the database *before every test* with the `beforeEach` function:

```

const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')
const api = supertest(app)
const Note = require('../models/note')
const initialNotes = [ { content: 'HTML is easy', important: false },
  { content: 'Browser can execute only JavaScript', important: true } ]
beforeEach(async () => {
  await Note.deleteMany({})
  let noteObject = new Note(initialNotes[0])
  await noteObject.save()
  noteObject = new Note(initialNotes[1])
  await noteObject.save()
})

```

The database is cleared out at the beginning, and after that, we save the two notes stored in the `initialNotes` array to the database. By doing this, we ensure that the database is in the same state before every test is run.

Let's also make the following changes to the last two tests:

```

test('all notes are returned', async () => {
  const response = await api.get('/api/notes')
  expect(response.body).toHaveLength(initialNotes.length)
})

test('a specific note is within the returned notes', async () => {
  const response = await api.get('/api/notes')
  const contents = response.body.map(r => r.content)
  expect(contents).toContain('Browser can execute only JavaScript')
})

```

Pay special attention to the `expect` in the latter test. The `response.body.map(r => r.content)` command is used to create an array containing the content of every note returned by the API. The `toContain` method is used for checking that the note given to it as a parameter is in the list of notes returned by the API.

Running tests one by one

The `npm test` command executes all of the tests for the application. When we are writing tests, it is usually wise to only execute one or two tests. Jest offers a few different ways of accomplishing this, one of which is the `only` method. If tests are written across many files, this method is not great.

A better option is to specify the tests that need to be run as parameters of the `npm test` command.

The following command only runs the tests found in the `tests/note_api.test.js` file:

```
npm test -- tests/note_api.test.js
```

The `-t` option can be used for running tests with a specific name:

```
npm test -- -t "a specific note is within the returned notes"
```

The provided parameter can refer to the name of the test or the `describe` block. The parameter can also contain just a part of the name. The following command will run all of the tests that contain `notes` in their name:

```
npm test -- -t 'notes'
```

NB: When running a single test, the mongoose connection might stay open if no tests using the connection are run. The problem might be because supertest primes the connection, but Jest does not run the `afterAll` portion of the code.

async/await

Before we write more tests let's take a look at the `async` and `await` keywords.

The `async/await` syntax that was introduced in ES7 makes it possible to use *asynchronous functions that return a promise* in a way that makes the code look synchronous.

As an example, the fetching of notes from the database with promises looks like this:

```
Note.find([]).then(notes => {
  console.log('operation returned the following notes', notes)
})
```

The `Note.find()` method returns a promise and we can access the result of the operation by registering a callback function with the `then` method.

All of the code we want to execute once the operation finishes is written in the callback function. If we wanted to make several asynchronous function calls in sequence, the situation would soon become painful. The asynchronous calls would have to be made in the callback. This would likely lead

to complicated code and could potentially give birth to a so-called [callback hell](#).

By [chaining promises](#) we could keep the situation somewhat under control, and avoid callback hell by creating a fairly clean chain of *then* method calls. We have seen a few of these during the course. To illustrate this, you can view an artificial example of a function that fetches all notes and then deletes the first one:

```
Note.find({})
  .then(notes => [
    return notes[0].deleteOne()
  ])
  .then(response => [
    console.log('the first note is removed')
    // more code here
  ])
}
```

The then-chain is alright, but we can do better. The [generator functions](#) introduced in ES6 provided a [clever way](#) of writing asynchronous code in a way that "looks synchronous". The syntax is a bit clunky and not widely used.

The *async* and *await* keywords introduced in ES7 bring the same functionality as the generators, but in an understandable and syntactically cleaner way to the hands of all citizens of the JavaScript world.

We could fetch all of the notes in the database by utilizing the [await](#) operator like this:

```
const notes = await Note.find({})

console.log('operation returned the following notes', notes)
```

The code looks exactly like synchronous code. The execution of code pauses at *const notes = await Note.find({})* and waits until the related promise is *fulfilled*, and then continues its execution to the next line. When the execution continues, the result of the operation that returned a promise is assigned to the *notes* variable.

The slightly complicated example presented above could be implemented by using *await* like this:

```
const notes = await Note.find({})
const response = await notes[0].deleteOne()

console.log('the first note is removed')
```

Thanks to the new syntax, the code is a lot simpler than the previous then-chain.

There are a few important details to pay attention to when using *async/await* syntax. To use the *await* operator with asynchronous operations, they have to return a promise. This is not a problem as such, as regular asynchronous functions using callbacks are easy to wrap around promises.

The *await* keyword can't be used just anywhere in JavaScript code. Using *await* is possible only inside of an [*async*](#) function.

This means that in order for the previous examples to work, they have to be using *async* functions.

Notice the first line in the arrow function definition:

```
const main = async () => {
  const notes = await Note.find({})
  console.log('operation returned the following notes', notes)

  const response = await notes[0].deleteOne()
  console.log('the first note is removed')

}

main()
```

The code declares that the function assigned to `main` is asynchronous. After this, the code calls the function with `main()`.

async/await in the backend

Let's start to change the backend to `async` and `await`. As all of the asynchronous operations are currently done inside of a function, it is enough to change the route handler functions into `async` functions.

The route for fetching all notes gets changed to the following:

```
notesRouter.get('/', async [request, response] => {
  const notes = await Note.find({})
  response.json(notes)
})
```

We can verify that our refactoring was successful by testing the endpoint through the browser and by running the tests that we wrote earlier.

You can find the code for our current application in its entirety in the `part4-3` branch of [this GitHub repository](#).

More tests and refactoring the backend

When code gets refactored, there is always the risk of [regression](#), meaning that existing functionality may break. Let's refactor the remaining operations by first writing a test for each route of the API.

Let's start with the operation for adding a new note. Let's write a test that adds a new note and verifies that the number of notes returned by the API increases and that the newly added note is in the list.

```
test('a valid note can be added', async () => {
  const newNote = [
    content: 'async/await simplifies making async calls',
    important: true,
  ]

  await api
    .post('/api/notes')
    .send(newNote)
```

```

    .expect(201)
    .expect('Content-Type', /application\/json/)

  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)

  expect(response.body).toHaveLength(initialNotes.length + 1)
  expect(contents).toContain(
    'async/await simplifies making async calls'
  )
}
})

```

Test fails since we are by accident returning the status code *200 OK* when a new note is created. Let us change that to *201 CREATED*:

```

notesRouter.post('/', (request, response, next) => {
  const body = request.body

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

  note.save()
    .then(savedNote => {
      response.status(201).json(savedNote)
    })
    .catch(error => next(error))
}
)

```

Let's also write a test that verifies that a note without content will not be saved into the database.

```

test('note without content is not added', async () => {
  const newNote = {
    important: true
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(400)

  const response = await api.get('/api/notes')

  expect(response.body).toHaveLength(initialNotes.length)
}
)

```

Both tests check the state stored in the database after the saving operation, by fetching all the notes of the application.

```
const response = await api.get('/api/notes')
```

The same verification steps will repeat in other tests later on, and it is a good idea to extract these steps into helper functions. Let's add the function into a new file called `tests/test_helper.js` which is in the same directory as the test file.

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

const initialNotes = [
  {
    content: 'HTML is easy',
    important: false
  },
  {
    content: 'Browser can execute only JavaScript',
    important: true
  }
]

const nonExistingId = async () => {
  const note = new Note({ content: 'willremovethissoon' })
  await note.save()
  await note.deleteOne()

  return note._id.toString()
}

const notesInDb = async () => {
  const notes = await Note.find({})
  return notes.map(note => note.toJSON())
}

module.exports = [
  initialNotes, nonExistingId, notesInDb
]
```

The module defines the `notesInDb` function that can be used for checking the notes stored in the database. The `initialNotes` array containing the initial database state is also in the module. We also define the `nonExistingId` function ahead of time, which can be used for creating a database object ID that does not belong to any note object in the database.

Our tests can now use the helper module and be changed like this:

```
const supertest = require('supertest')
const mongoose = require('mongoose')
const helper = require('./test_helper') const app = require('../app')
const api = supertest(app)

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

beforeEach(async () => {
```

```

    await Note.deleteMany([])

    let noteObject = new Note(helper.initialNotes[0]) await noteObject.save()

    noteObject = new Note(helper.initialNotes[1]) await noteObject.save()
})

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

test('all notes are returned', async () => {
  const response = await api.get('/api/notes')

  expect(response.body).toHaveLength(helper.initialNotes.length)
})

test('a specific note is within the returned notes', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)

  expect(contents).toContain(
    'Browser can execute only JavaScript'
)
})

test('a valid note can be added', async () => {
  const newNote = [
    content: 'async/await simplifies making async calls',
    important: true
  ]

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(201)
    .expect('Content-Type', /application\/json/)

  const notesAtEnd = await helper.notesInDb() expect(notesAtEnd).toHaveLength(helper.initialNotes.length + 1)
  const contents = notesAtEnd.map(n => n.content) expect(contents).toContain(
    'async/await simplifies making async calls'
)
})

test('note without content is not added', async () => {
  const newNote = [
    important: true
  ]
})

```

```

    await api
      .post('/api/notes')
      .send(newNote)
      .expect(400)

    const notesAtEnd = await helper.notesInDb()
    expect(notesAtEnd).toHaveLength(helper.initialNotes.length))

afterAll(async () => {
  await mongoose.connection.close()
})

```

The code using promises works and the tests pass. We are ready to refactor our code to use the `async/await` syntax.

We make the following changes to the code that takes care of adding a new note (notice that the route handler definition is preceded by the `async` keyword):

```

notesRouter.post('/', async (request, response, next) => {
  const body = request.body

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

  const savedNote = await note.save()
  response.status(201).json(savedNote)
})

```

There's a slight problem with our code: we don't handle error situations. How should we deal with them?

Error handling and `async/await`

If there's an exception while handling the POST request we end up in a familiar situation:

```

Method: POST
Path:  /api/notes
Body:  { important: true }

(node:89372) UnhandledPromiseRejectionWarning: ValidationError: Note validation failed: content: Path `content` is required.
    at new ValidationError (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/error/validation.js:30:11)
    at model.Document.invalidate (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/document.js:2071:32)
    at p.doValidate.skipSchemaValidators (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/document.js:1934:17)
    at /Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/schematype.js:929:9
    at process._tickCallback (internal/process/next_tick.js:172:11)
(node:89372) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing

```

In other words, we end up with an unhandled promise rejection, and the request never receives a response.

With `async/await` the recommended way of dealing with exceptions is the old and familiar `try/catch` mechanism:

```
notesRouter.post('/r', async [request, responder, next] => {
  const body = request.body

  const note = new Note({
    content: body.content,
    important: body.important || false,
  })
  try {
    const savedNote = await note.save()
    responder.status(201).json(savedNote)
  } catch(exception) {
    next(exception)
  }
})
```

The catch block simply calls the `next` function, which passes the request handling to the error handling middleware.

After making the change, all of our tests will pass once again.

Next, let's write tests for fetching and removing an individual note:

```
test('a specific note can be viewed', async () => {
  const notesAtStart = await helper.notesInDb()

  const noteToView = notesAtStart[0]

  const resultNote = await api.get(`api/notes/${noteToView.id}`).expect(200).expect('Content-Type', 'application/json')
  expect(resultNote.body).toEqual(noteToView)
})

test('a note can be deleted', async () => {
  const notesAtStart = await helper.notesInDb()
  const noteToDelete = notesAtStart[0]

  await api.delete(`api/notes/${noteToDelete.id}`).expect(204)
  const notesAtEnd = await helper.notesInDb()

  expect(notesAtEnd).toHaveLength(helper.initialNotes.length - 1)
}

const contents = notesAtEnd.map(r => r.content)

expect(contents).not.toContain(noteToDelete.content)
})
```

Both tests share a similar structure. In the initialization phase, they fetch a note from the database. After this, the tests call the actual operation being tested, which is highlighted in the code block. Lastly, the tests verify that the outcome of the operation is as expected.

The tests pass and we can safely refactor the tested routes to use `async/await`:

```

notesRouter.get('/:id', async [request, response, next] => {
  try {
    const note = await Note.findById(request.params.id)
    if (note) {
      response.json(note)
    } else {
      response.status(404).end()
    }
  } catch(exception) {
    next(exception)
  }
})

notesRouter.delete('/:id', async [request, response, next] => {
  try {
    await Note.findByIdAndDelete(request.params.id)
    response.status(204).end()
  } catch(exception) {
    next(exception)
  }
})

```

You can find the code for our current application in its entirety in the *part4-4* branch of [this GitHub repository](#).

Eliminating the try-catch

Async/await unclutters the code a bit, but the 'price' is the *try/catch* structure required for catching exceptions. All of the route handlers follow the same structure

```

try {
  // do the async operations here
} catch(exception) {
  next(exception)
}

```

One starts to wonder if it would be possible to refactor the code to eliminate the *catch* from the methods?

The [express-async-errors](#) library has a solution for this.

Let's install the library

```
npm install express-async-errors
```

Using the library is *very* easy. You introduce the library in *app.js*, *before* you import your routes:

```

const config = require('./utils/config')
const express = require('express')
require('express-async-errors') const app = express()

```

```

const cors = require('cors')
const notesRouter = require('./controllers/notes')
const middleware = require('./utils/middleware')
const logger = require('./utils/logger')
const mongoose = require('mongoose')

// ...

module.exports = app

```

The 'magic' of the library allows us to eliminate the try-catch blocks completely. For example the route for deleting a note

```

notesRouter.delete('/:id', async (request, response, next) => {
  try {
    await Note.findByIdAndDelete(request.params.id)
    response.status(204).end()
  } catch (exception) {
    next(exception)
  }
})

```

becomes

```

notesRouter.delete('/:id', async (request, response) => {
  await Note.findByIdAndDelete(request.params.id)
  response.status(204).end()
})

```

Because of the library, we do not need the `next(exception)` call anymore. The library handles everything under the hood. If an exception occurs in an `async` route, the execution is automatically passed to the error handling middleware.

The other routes become:

```

notesRouter.post('/', async (request, response) => {
  const body = request.body

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

  const savedNote = await note.save()
  response.status(201).json(savedNote)
})

notesRouter.get('/:id', async (request, response) => {
  const note = await Note.findById(request.params.id)
  if (note) {
    response.json(note)
  }
})

```

```
    } else {
      response.status(404).end()
    }
  })
}
```

Optimizing the `beforeEach` function

Let's return to writing our tests and take a closer look at the `beforeEach` function that sets up the tests:

```
beforeEach(async () => [
  await Note.deleteMany([])

  let noteObject = new Note(helper.initialNotes[0])
  await noteObject.save()

  noteObject = new Note(helper.initialNotes[1])
  await noteObject.save()
])
```

The function saves the first two notes from the `helper.initialNotes` array into the database with two separate operations. The solution is alright, but there's a better way of saving multiple objects to the database:

```
beforeEach(async () => [
  await Note.deleteMany([])
  console.log('cleared')

  helper.initialNotes.forEach(async (note) => {
    let noteObject = new Note(note)
    await noteObject.save()
    console.log('saved')
  })
  console.log('done')
])

test('notes are returned as json', async () => [
  console.log('entered test')
  // ...
])
```

We save the notes stored in the array into the database inside of a `forEach` loop. The tests don't quite seem to work however, so we have added some console logs to help us find the problem.

The console displays the following output:

```
cleared
done
entered test
saved
```

Despite our use of the `async/await` syntax, our solution does not work as we expected it to. The test execution begins before the database is initialized!

The problem is that every iteration of the `forEach` loop generates an asynchronous operation, and `beforeEach` won't wait for them to finish executing. In other words, the `await` commands defined inside of the `forEach` loop are not in the `beforeEach` function, but in separate functions that `beforeEach` will not wait for.

Since the execution of tests begins immediately after `beforeEach` has finished executing, the execution of tests begins before the database state is initialized.

One way of fixing this is to wait for all of the asynchronous operations to finish executing with the `Promise.all` method:

```
beforeEach(async () => {
  await Note.deleteMany({})

  const noteObjects = helper.initialNotes
    .map(note => new Note(note))
  const promiseArray = noteObjects.map(note => note.save())
  await Promise.all(promiseArray)
})
```

The solution is quite advanced despite its compact appearance. The `noteObjects` variable is assigned to an array of Mongoose objects that are created with the `Note` constructor for each of the notes in the `helper.initialNotes` array. The next line of code creates a new array that *consists of promises*, that are created by calling the `save` method of each item in the `noteObjects` array. In other words, it is an array of promises for saving each of the items to the database.

The `Promise.all` method can be used for transforming an array of promises into a single promise, that will be *fulfilled* once every promise in the array passed to it as a parameter is resolved. The last line of code `await Promise.all(promiseArray)` waits until every promise for saving a note is finished, meaning that the database has been initialized.

The returned values of each promise in the array can still be accessed when using the `Promise.all` method. If we wait for the promises to be resolved with the `await` syntax `const results = await Promise.all(promiseArray)`, the operation will return an array that contains the resolved values for each promise in the `promiseArray`, and they appear in the same order as the promises in the array.

`Promise.all` executes the promises it receives in parallel. If the promises need to be executed in a particular order, this will be problematic. In situations like this, the operations can be executed inside of a `for...of` block, that guarantees a specific execution order.

```
beforeEach(async () => {
  await Note.deleteMany({})
```

```
for (let note of helper.initialNotes) {
  let noteObject = new Note(note)
  await noteObject.save()
}

})
```

The asynchronous nature of JavaScript can lead to surprising behavior, and for this reason, it is important to pay careful attention when using the `async/await` syntax. Even though the syntax makes it easier to deal with promises, it is still necessary to understand how promises work!

The code for our application can be found on [GitHub](#), branch `part4-5`.

A true full stack developer's oath

Making tests brings yet another layer of challenge to programming. We have to update our full stack developer oath to remind you that systematicity is also key when developing tests.

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 as I expect
- I will keep an eye on the database: does the backend save data there in the right format
- I will progress in small steps
- *I will write lots of `console.log` statements to make sure I understand how the code and the tests behave 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
- *If a test does not pass, I make sure that the tested functionality for sure works in the application*
- 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 4.8.-4.12.

NB: the material uses the `toContain` matcher in several places to verify that an array contains a specific element. It's worth noting that the method uses the `==` operator for comparing and matching elements, which means that it is often not well-suited for matching objects. In most cases, the appropriate method for verifying objects in arrays is the `toContainEqual` matcher. However, the model solutions don't check for objects in arrays with matchers, so using the method is not required for solving the exercises.

Warning: If you find yourself using `async/await` and `then` methods in the same code, it is almost guaranteed that you are doing something wrong. Use one or the other and don't mix the two.

4.8: Blog List Tests, step 1

Use the `supertest` package for writing a test that makes an HTTP GET request to the `/api/blogs` URL.

Verify that the blog list application returns the correct amount of blog posts in the JSON format.

Once the test is finished, refactor the route handler to use the `async/await` syntax instead of promises.

Notice that you will have to make similar changes to the code that were made [in the material](#), like defining the test environment so that you can write tests that use separate databases.

NB: When running the tests, you may run into the following warning:

```
console.warn node_modules/mongoose/lib/helpers/printJestWarning.js:4
  Mongoose: looks like you're trying to test a Mongoose app with Jest's default jsdom test environment. Please make sure you read Mongoose's docs on configuring Jest to test Node.js apps: http://mongoosejs.com/docs/jest.html
```

[One way](#) to get rid of this is to add to the `tests` directory a file `teardown.js` with the following content

```
module.exports = () => [
  process.exit(0)
]
```

and by extending the Jest definitions in the `package.json` as follows

```
{
  // ...
  "jest": {
    "testEnvironment": "node",
    "globalTeardown": "./tests/teardown.js"
  }
}
```

NB: when you are writing your tests ***it is better to not execute them all***, only execute the ones you are working on. Read more about this [here](#).

4.9: Blog List Tests, step 2

Write a test that verifies that the unique identifier property of the blog posts is named `id`, by default the database names the property `_id`. Verifying the existence of a property is easily done with Jest's [toBeDefined](#) matcher.

Make the required changes to the code so that it passes the test. The [toJSON](#) method discussed in part 3 is an appropriate place for defining the `id` parameter.

4.10: Blog List Tests, step 3

Write a test that verifies that making an HTTP POST request to the `/api/blogs` URL successfully creates a new blog post. At the very least, verify that the total number of blogs in the system is increased by one. You can also verify that the content of the blog post is saved correctly to the database.

Once the test is finished, refactor the operation to use `async/await` instead of promises.

4.11*: Blog List Tests, step 4

Write a test that verifies that if the *likes* property is missing from the request, it will default to the value 0. Do not test the other properties of the created blogs yet.

Make the required changes to the code so that it passes the test.

4.12*: Blog List tests, step 5

Write tests related to creating new blogs via the */api/blogs* endpoint, that verify that if the *title* or *url* properties are missing from the request data, the backend responds to the request with the status code *400 Bad Request*.

Make the required changes to the code so that it passes the test.

Refactoring tests

Our test coverage is currently lacking. Some requests like *GET /api/notes/:id* and *DELETE /api/notes/:id* aren't tested when the request is sent with an invalid id. The grouping and organization of tests could also use some improvement, as all tests exist on the same "top level" in the test file. The readability of the test would improve if we group related tests with *describe* blocks.

Below is an example of the test file after making some minor improvements:

```
const supertest = require('supertest')
const mongoose = require('mongoose')
const helper = require('./test_helper')
const app = require('../app')
const api = supertest(app)

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

beforeEach(async () => {
  await Note.deleteMany({})
  await Note.insertMany(helper.initialNotes)
})

describe('when there is initially some notes saved', () => {
  test('notes are returned as json', async () => {
    await api
      .get('/api/notes')
      .expect(200)
      .expect('Content-Type', /application\/json/)
  })

  test('all notes are returned', async () => {
    const response = await api.get('/api/notes')

    expect(response.body).toHaveLength(helper.initialNotes.length)
  })

  test('a specific note is within the returned notes', async () => {
    const response = await api.get('/api/notes')
```

```

    const contents = response.body.map(r => r.content)

    expect(contents).toContain(
      'Browser can execute only JavaScript'
    )
  })
}

describe('viewing a specific note', () => {
  test('succeeds with a valid id', async () => {
    const notesAtStart = await helper.notesInDb()

    const noteToView = notesAtStart[0]

    const resultNote = await api
      .get(`/api/notes/${noteToView.id}`)
      .expect(200)
      .expect('Content-Type', /application\/json/)

    expect(resultNote.body).toEqual(noteToView)
  })

  test('fails with statuscode 404 if note does not exist', async () => {
    const validNonexistingId = await helper.nonExistingId()

    await api
      .get(`/api/notes/${validNonexistingId}`)
      .expect(404)
  })

  test('fails with statuscode 400 if id is invalid', async () => {
    const invalidId = '5a3d5da59070081a82a3445'

    await api
      .get(`/api/notes/${invalidId}`)
      .expect(400)
  })
}

describe('addition of a new note', () => {
  test('succeeds with valid data', async () => {
    const newNote = {
      content: 'async/await simplifies making async calls',
      important: true,
    }

    await api
      .post('/api/notes')
      .send(newNote)
      .expect(201)
      .expect('Content-Type', /application\/json/)

    const notesAtEnd = await helper.notesInDb()
  })
})

```

```

expect(notesAtEnd).toHaveLength(helper.initialNotes.length + 1)

const contents = notesAtEnd.map(n => n.content)
expect(contents).toContain(
  'async/await simplifies making async calls'
)
})

})

test('fails with status code 400 if data invalid', async () => {
  const newNote = {
    important: true
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(400)

  const notesAtEnd = await helper.notesInDb()

  expect(notesAtEnd).toHaveLength(helper.initialNotes.length)
})
})

describe('deletion of a note', () => {
  test('succeeds with status code 204 if id is valid', async () => {
    const notesAtStart = await helper.notesInDb()
    const noteToDelete = notesAtStart[0]

    await api
      .delete(`/api/notes/${noteToDelete.id}`)
      .expect(204)

    const notesAtEnd = await helper.notesInDb()

    expect(notesAtEnd).toHaveLength(
      helper.initialNotes.length - 1
    )

    const contents = notesAtEnd.map(r => r.content)

    expect(contents).not.toContain(noteToDelete.content)
  })
})

afterAll(async () => {
  await mongoose.connection.close()
})
})

```

The test output in the console is grouped according to the *describe* blocks:

```
PASS tests/note_api.test.js (5.473s)
when there is initially some notes saved
  ✓ notes are returned as json (1737ms)
  ✓ all notes are returned (180ms)
  ✓ a specific note is within the returned notes (208ms)
viewing a specific note
  ✓ succeeds with a valid id (226ms)
  ✓ fails with statuscode 404 if note does not exist (297ms)
  ✓ fails with statuscode 400 if id is invalid (114ms)
addition of a new note
  ✓ succeeds with valid data (243ms)
  ✓ fails with status code 400 if data invalid (168ms)
deletion of a note
  ✓ succeeds with status code 200 if id is valid (272ms)
```

There is still room for improvement, but it is time to move forward.

This way of testing the API, by making HTTP requests and inspecting the database with Mongoose, is by no means the only nor the best way of conducting API-level integration tests for server applications. There is no universal best way of writing tests, as it all depends on the application being tested and available resources.

You can find the code for our current application in its entirety in the *part4-6* branch of [this GitHub repository](#).

Exercises 4.13.-4.14.

4.13 Blog List Expansions, step 1

Implement functionality for deleting a single blog post resource.

Use the `async/await` syntax. Follow [RESTful](#) conventions when defining the HTTP API.

Implement tests for the functionality.

4.14 Blog List Expansions, step 2

Implement functionality for updating the information of an individual blog post.

Use `async/await`.

The application mostly needs to update the number of *likes* for a blog post. You can implement this functionality the same way that we implemented updating notes in [part 3](#).

Implement tests for the functionality.

5: Testing React apps

5a: Login in frontend

In the last two parts, we have mainly concentrated on the backend. The frontend that we developed in [part 2](#) does not yet support the user management we implemented to the backend in part 4.

At the moment the frontend shows existing notes and lets users change the state of a note from

important to not important and vice versa. New notes cannot be added anymore because of the changes made to the backend in part 4: the backend now expects that a token verifying a user's identity is sent with the new note.

We'll now implement a part of the required user management functionality in the frontend. Let's begin with the user login. Throughout this part, we will assume that new users will not be added from the frontend.

Handling login

A login form has now been added to the top of the page:

The screenshot shows a web browser window with the URL `http://localhost:3000` in the address bar. The page title is "Notes app". Below the title, there is a heading "Login". Under the heading, there are two input fields: one for "username" and one for "password", both with placeholder text. Below these fields is a "login" button. To the right of the password field is a link labeled "show important". Below the login section is a list of five items, each preceded by a radio button. Each item has a "make not important" button to its right. At the bottom of the list is a "save" button. The entire page has a light gray background with some shadows and rounded corners.

The code of the *App* component now looks as follows:

```
const App = () => {
  const [notes, setNotes] = useState([])
  const [newNote, setNewNote] = useState('')
  const [showAll, setShowAll] = useState(true)
  const [errorMessage, setErrorMessage] = useState(null)
  const [username, setUsername] = useState('')  const [password, setPassword] =
useState('')
  useEffect(() => {
    noteService
      .getAll().then(initialNotes => [
        setNotes(initialNotes)
      ])
    , []
  }

  // ...

  const handleLogin = (event) => {    event.preventDefault()    console.log('log
ing in with', username, password)  }
```

```

    return [
      <div>
        <h1>Notes</h1>

        <Notification message={errorMessage} />

        <form onSubmit={handleLogin}>           <div>           username           <i
      input           type="text"           value={username}           name="Username"
      e"           onChange={({ target }) => setUsername(target.value)}           />
      </div>           <div>           password           <input
      password"           value={password}           name="Password"           type="p
      onCha
      nge={({ target }) => setPassword(target.value)}           />           </div>
      <button type="submit">login</button>           </form>
      // ...
    </div>
  }

export default App

```

The current application code can be found on [GitHub](#), in the branch *part5-1*. If you clone the repo, don't forget to run *npm install* before attempting to run the frontend.

The frontend will not display any notes if it's not connected to the backend. You can start the backend with *npm run dev* in its folder from Part 4. This will run the backend on port 3001. While that is active, in a separate terminal window you can start the frontend with *npm start*, and now you can see the notes that are saved in your MongoDB database from Part 4.

Keep this in mind from now on.

The login form is handled the same way we handled forms in [part 2](#). The app state has fields for *username* and *password* to store the data from the form. The form fields have event handlers, which synchronize changes in the field to the state of the *App* component. The event handlers are simple: An object is given to them as a parameter, and they destructure the field *target* from the object and save its value to the state.

```
  ({ target }) => setUsername(target.value)
```

The method *handleLogin*, which is responsible for handling the data in the form, is yet to be implemented.

Logging in is done by sending an HTTP POST request to the server address *api/login*. Let's separate the code responsible for this request into its own module, to file *services/login.js*.

We'll use *async/await* syntax instead of promises for the HTTP request:

```

import axios from 'axios'
const baseUrl = '/api/login'

const login = async credentials => {
  const response = await axios.post(baseUrl, credentials)

```

```

        return response.data
    }

export default { login }

```

The method for handling the login can be implemented as follows:

```

import loginService from './services/login'
const App = () => {
    // ...
    const [username, setUsername] = useState('')
    const [password, setPassword] = useState('')
    const [user, setUser] = useState(null)
    const handleLogin = async (event) => {
        event.preventDefault()
        try {
            const user = await loginService.login({
                username, password,
            })
            setUser(user)
            setUsername('')
            setPassword('')
        } catch (exception) {
            setErrorMessage('Wrong credentials')
            setTimeout(() => {
                setErrorMessage(null)
            }, 5000)
        }
    }
    // ...
}

```

If the login is successful, the form fields are emptied *and* the server response (including a *token* and the user details) is saved to the *user* field of the application's state.

If the login fails or running the function *loginService.login* results in an error, the user is notified.

The user is not notified about a successful login in any way. Let's modify the application to show the login form only *if the user is not logged-in*, so when *user == null*. The form for adding new notes is shown only if the *user is logged-in*, so when *user* contains the user's details.

Let's add two helper functions to the *App* component for generating the forms:

```

const App = () => {
    // ...

    const loginForm = () => (
        <form onSubmit={handleLogin}>
            <div>
                username
                <input
                    type="text"
                    value={username}
                    name="Username"
                    onChange={({ target }) => setUsername(target.value)}
                />
            </div>
            <div>
                password
                <input
                    type="password"
                    value={password}

```

```

        name="Password"
        onChange={({ target }) => setPassword(target.value)}
      />
    </div>
    <button type="submit">login</button>
  </form>
}

const noteForm = () => [
  <form onSubmit={addNote}>
    <input
      value={newNote}
      onChange={handleNoteChange}
    />
    <button type="submit">save</button>
  </form>
]

return [
  // ...
]
}

```

and conditionally render them:

```

const App = () => [
  // ...

  const loginForm = () => [
    // ...
  ]

  const noteForm = () => [
    // ...
  ]

  return [
    <div>
      <h1>Notes</h1>

      <Notification message={errorMessage} />

      {user === null && loginForm()}       {user !== null && noteForm()}
      <div>
        <button onClick={() => setShowAll(!showAll)}>
          show {showAll ? 'important' : 'all'}
        </button>
      </div>
      <ul>
        {notesToShow.map([note, i] =>
          <Note
            key={i}>

```

```

        note={note}
        toggleImportance={() => toggleImportanceOf(note.id)}
      />
    ) []
  </ul>

  <Footer />
</div>
)
]

```

A slightly odd looking, but commonly used [React trick](#) is used to render the forms conditionally:

```

{
  user === null && loginForm()
}

```

If the first statement evaluates to false or is [falsy](#), the second statement (generating the form) is not executed at all.

We can make this even more straightforward by using the [conditional operator](#):

```

return [
  <div>
    <h1>Notes</h1>

    <Notification message={errorMessage} />

    {user === null ?
      loginForm() :
      noteForm()
    }

    <h2>Notes</h2>

    // ...
  </div>
]

```

If `user === null` is [truthy](#), `loginForm()` is executed. If not, `noteForm()` is.

Let's do one more modification. If the user is logged in, their name is shown on the screen:

```

return [
  <div>
    <h1>Notes</h1>

    <Notification message={errorMessage} />

    {user === null ?

```

```

loginForm() :
<div>
  <p>{user.name} logged-in</p>
  {noteForm()}
</div>
}

<h2>Notes</h2>

// ...

</div>
}

```

The solution isn't perfect, but we'll leave it like this for now.

Our main component *App* is at the moment way too large. The changes we did now are a clear sign that the forms should be refactored into their own components. However, we will leave that for an optional exercise.

The current application code can be found on [GitHub](#), in the branch *part5-2*.

Creating new notes

The token returned with a successful login is saved to the application's state - the *user*'s field *token*:

```

const handleLogin = async (event) => {
  event.preventDefault()
  try {
    const user = await loginService.login({
      username, password,
    })

    setUser(user)      setUsername('')
    setPassword('')
  } catch (exception) {
    // ...
  }
}

```

Let's fix creating new notes so it works with the backend. This means adding the token of the logged-in user to the Authorization header of the HTTP request.

The *noteService* module changes like so:

```

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

let token = null
const setToken = newToken => { token = `Bearer ${newToken}` }
const getAll = () => {

```

```

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

const create = async newObject => {
  const config = { headers: { Authorization: `token` } }
  const response = await axios.post(baseUrl, newObject, config)
  return response.data
}

const update = (id, newObject) => {
  const request = axios.put(`/${baseUrl}/${id}`, newObject)
  return request.then(response => response.data)
}

export default { getAll, create, update, setToken }

```

The noteService module contains a private variable called *token*. Its value can be changed with the *setToken* function, which is exported by the module. *create*, now with `async/await` syntax, sets the token to the *Authorization* header. The header is given to axios as the third parameter of the *post* method.

The event handler responsible for login must be changed to call the method `noteService.setToken(user.token)` with a successful login:

```

const handleLogin = async (event) => {
  event.preventDefault()
  try {
    const user = await loginService.login({
      username, password,
    })

    noteService.setToken(user.token)      setUser(user)
    setUsername('')
    setPassword('')
  } catch (exception) {
    // ...
  }
}

```

And now adding new notes works again!

Saving the token to the browser's local storage

Our application has a small flaw: if the browser is refreshed (eg. pressing F5), the user's login information disappears.

This problem is easily solved by saving the login details to [local storage](#). Local Storage is a [key-value](#) database in the browser.

It is very easy to use. A *value* corresponding to a certain *key* is saved to the database with the method

`setItem`. For example:

```
window.localStorage.setItem('name', 'juha tauriainen')
```

saves the string given as the second parameter as the value of the key *name*.

The value of a key can be found with the method `getItem`:

```
window.localStorage.getItem('name')
```

while `removeItem` removes a key.

Values in the local storage are persisted even when the page is re-rendered. The storage is `origin`-specific so each web application has its own storage.

Let's extend our application so that it saves the details of a logged-in user to the local storage.

Values saved to the storage are `DOMstrings`, so we cannot save a JavaScript object as it is. The object has to be parsed to JSON first, with the method `JSON.stringify`. Correspondingly, when a JSON object is read from the local storage, it has to be parsed back to JavaScript with `JSON.parse`.

Changes to the login method are as follows:

```
const handleLogin = async (event) => {
  event.preventDefault()
  try {
    const user = await loginService.login([
      username, password,
    ])

    window.localStorage.setItem('loggedNoteappUser', JSON.stringify(user))
    noteService.setToken(user.token)
    setUser(user)
    setUsername('')
    setPassword('')
  } catch (exception) {
    // ...
  }
}
```

The details of a logged-in user are now saved to the local storage, and they can be viewed on the console (by typing `window.localStorage` in it):

Notes

Matti Luukkainen logged in

The screenshot shows a browser's developer tools open to the 'Console' tab. At the top, there are buttons for 'save' and 'show important'. Below them is a list of bullet points:

- Browser can execute only Javascript [make not important](#)
- User id of the note creator is at the start sent along the request [make not important](#)
- The existing code and tests need to be changed when user is added to system [make not important](#)
- Testing the token authentication [make not important](#)
- Single Page Apps use token authentication [make not important](#)

The main area shows the browser's local storage:

```
> window.localStorage
< Storage {loggedNoteappUser: "{\"token\":\"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...\", \"username\":\"mluukkai\", \"name\":\"Matti Luukkainen\"}", loggedBlogappUser: "{\"token\":\"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...\", \"username\":\"mluukkai\", \"name\":\"Matti Luukkainen\"}", length: 2}
>
```

You can also inspect the local storage using the developer tools. On Chrome, go to the *Application* tab and select *Local Storage* (more details [here](#)). On Firefox go to the *Storage* tab and select *Local Storage* (details [here](#)).

We still have to modify our application so that when we enter the page, the application checks if user details of a logged-in user can already be found on the local storage. If they are there, the details are saved to the state of the application and to *noteService*.

The right way to do this is with an [effect hook](#): a mechanism we first encountered in [part 2](#), and used to fetch notes from the server.

We can have multiple effect hooks, so let's create a second one to handle the first loading of the page:

```
const App = () => {
  const [notes, setNotes] = useState([])
  const [newNote, setNewNote] = useState('')
  const [showAll, setShowAll] = useState(true)
  const [errorMessage, setErrorMessage] = useState(null)
  const [username, setUsername] = useState('')
  const [password, setPassword] = useState('')
  const [user, setUser] = useState(null)

  useEffect(() => {
    noteService
      .getAll().then(initialNotes => [
        setNotes(initialNotes)
      ])
  }, [])

  useEffect(() => {
    const loggedUserJSON = window.localStorage.getItem('loggedNoteappUser')
    if (loggedUserJSON) {
      const user = JSON.parse(loggedUserJSON)
      setUser(user)
      noteService.setToken(user.token)
    }
  }, [])
}
```

The empty array as the parameter of the effect ensures that the effect is executed only when the component is rendered [for the first time](#).

Now a user stays logged in to the application forever. We should probably add a *logout* functionality, which removes the login details from the local storage. We will however leave it as an exercise.

It's possible to log out a user using the console, and that is enough for now. You can log out with the command:

```
window.localStorage.removeItem('loggedNoteappUser')
```

or with the command which empties *localStorage* completely:

```
window.localStorage.clear()
```

The current application code can be found on [GitHub](#), in the branch *part5-3*.

Exercises 5.1.-5.4.

We will now create a frontend for the blog list backend we created in the last part. You can use [this application](#) from GitHub as the base of your solution. You need to connect your backend with a proxy as shown in [part 3](#).

It is enough to submit your finished solution. You can commit after each exercise, but that is not necessary.

The first few exercises revise everything we have learned about React so far. They can be challenging, especially if your backend is incomplete. It might be best to use the backend that we marked as the answer for part 4.

While doing the exercises, remember all of the debugging methods we have talked about, especially keeping an eye on the console.

Warning: If you notice you are mixing in the *async/await* and *then* commands, it's 99.9% certain you are doing something wrong. Use either or, never both.

5.1: Blog List Frontend, step 1

Clone the application from [GitHub](#) with the command:

```
git clone https://github.com/fullstack-hy2020/bloglist-frontend
```

Remove the git configuration of the cloned application

```
cd bloglist-frontend // go to cloned repository
rm -rf .git
```

The application is started the usual way, but you have to install its dependencies first:

```
npm install
```

```
npm run dev
```

Implement login functionality to the frontend. The token returned with a successful login is saved to the application's state *user*.

If a user is not logged in, *only* the login form is visible.

The screenshot shows a browser window with the address bar set to 'localhost:3000'. The main content area has a heading 'log in to application'. Below it is a form with two input fields: 'username' containing 'mluukkai' and 'password' containing '...'. A 'login' button is at the bottom of the form.

If the user is logged-in, the name of the user and a list of blogs is shown.

The screenshot shows a browser window with the address bar set to 'localhost:3000'. The main content area has a heading 'blogs'. Below it is a message 'Matti Luukkainen logged in'. Underneath the message is a list of two items: 'Things I Don't Know as of 2018 Dan Abramov' and 'Microservices and the First Law of Distributed Objects Martin Fowler'.

User details of the logged-in user do not have to be saved to the local storage yet.

NB You can implement the conditional rendering of the login form like this for example:

```
if (user === null) {
  return [
    <div>
      <h2>Log in to application</h2>
      <form>
        //...
      </form>
    </div>
  ]
}

return [
  <div>
    <h2>blogs</h2>
    {blogs.map(blog =>
      <Blog key={blog.id} blog={blog} />
    )}
  </div>
]
```



5.2: Blog List Frontend, step 2

Make the login 'permanent' by using the local storage. Also, implement a way to log out.

A screenshot of a web browser window. The address bar shows 'localhost:3000'. The main content area displays a heading 'blogs' in large bold letters. Below it, a user 'Matti Luukkainen' is logged in, with a 'logout' button next to the name. Two blog entries are listed:

- 'Things I Don't Know as of 2018 Dan Abramov'
- 'Microservices and the First Law of Distributed Objects Martin Fowler'

Ensure the browser does not remember the details of the user after logging out.

5.3: Blog List Frontend, step 3

Expand your application to allow a logged-in user to add new blogs:

A screenshot of a web browser window. The address bar shows 'localhost:3000'. The main content area displays a heading 'blogs' in large bold letters. Below it, a user 'Matti Luukkainen' is logged in, with a 'logout' button next to the name. A section titled 'create new' contains three input fields: 'title:' with a text input field, 'author:' with a text input field, and 'url:' with a text input field. Below these fields is a 'create' button. At the bottom of the page, two blog entries are listed:

- 'Things I Don't Know as of 2018 Dan Abramov'
- 'Microservices and the First Law of Distributed Objects Martin Fowler'

5.4: Blog List Frontend, step 4

Implement notifications that inform the user about successful and unsuccessful operations at the top of the page. For example, when a new blog is added, the following notification can be shown:

A screenshot of a web browser window titled "localhost:3000". The main content area has a green header bar containing the text "a new blog You're NOT gonna need it! by Ron Jeffries added". Below this, the user "Matti Luukkainen" is logged in, with a "logout" button. A "create new" section contains input fields for "title", "author", and "url", along with a "create" button. Below these fields, three blog entries are listed: "Things I Don't Know as of 2018 Dan Abramov", "Microservices and the First Law of Distributed Objects Martin Fowler", and "You're NOT gonna need it! Ron Jeffries".

Failed login can show the following notification:

A screenshot of a web browser window titled "localhost:3000". The main content area has a red header bar containing the text "wrong username or password". Below this, there are input fields for "username" (containing "mluukkai") and "password" (containing two dots). A "login" button is located below the password field.

The notifications must be visible for a few seconds. It is not compulsory to add colors.

A note on using local storage

At the [end](#) of the last part, we mentioned that the challenge of token-based authentication is how to cope with the situation when the API access of the token holder to the API needs to be revoked.

There are two solutions to the problem. The first one is to limit the validity period of a token. This forces the user to re-login to the app once the token has expired. The other approach is to save the validity information of each token to the backend database. This solution is often called a *server-side session*.

No matter how the validity of tokens is checked and ensured, saving a token in the local storage might contain a security risk if the application has a security vulnerability that allows [Cross Site Scripting \(XSS\)](#) attacks. An XSS attack is possible if the application would allow a user to inject arbitrary JavaScript code (e.g. using a form) that the app would then execute. When using React sensibly it should not be possible since [React sanitizes](#) all text that it renders, meaning that it is not

executing the rendered content as JavaScript.

If one wants to play safe, the best option is to not store a token in local storage. This might be an option in situations where leaking a token might have tragic consequences.

It has been suggested that the identity of a signed-in user should be saved as [httpOnly cookies](#), so that JavaScript code could not have any access to the token. The drawback of this solution is that it would make implementing SPA applications a bit more complex. One would need at least to implement a separate page for logging in.

However, it is good to notice that even the use of httpOnly cookies does not guarantee anything. It has even been suggested that httpOnly cookies are [not any safer than](#) the use of local storage.

So no matter the used solution the most important thing is to [minimize the risk](#) of XSS attacks altogether.

5b: props.children and proptypes

Displaying the login form only when appropriate

Let's modify the application so that the login form is not displayed by default:



A screenshot of a web browser window titled "localhost:3000". The main content area displays the word "Notes" in green. Below it, there are two buttons: "log in" and "show important". A large red arrow points from the right side towards the "log in" button. In the bottom half of the screen, there is a list of bullet points:

- Browser can execute only Javascript [make important](#)
- User id of the note creator is at the start sent along the request [make not important](#)
- The existing code and tests need to be changed when user is added to system [make not important](#)
- Testing the token authentication [make not important](#)

The login form appears when the user presses the *login* button:



A screenshot of a web browser window titled "localhost:3000". The main content area displays the word "Notes" in green. Below it, the word "Login" is displayed in large black letters. Under "Login", there are two input fields: "username" containing "mluukkai" and "password" containing "=". Below the input fields are two buttons: "login" and "cancel". A large red arrow points from the right side towards the "cancel" button.

The user can close the login form by clicking the *cancel* button.

Let's start by extracting the login form into its own component:

```
const LoginForm = ({  
  handleSubmit,  
  handleUsernameChange,  
  handlePasswordChange,  
  username,  
  password  
}) => {  
  return (  
    <div>  
      <h2>Login</h2>  
  
      <form onSubmit={handleSubmit}>  
        <div>  
          username  
          <input  
            value={username}  
            onChange={handleUsernameChange}>  
        />  
      </div>  
      <div>  
        password  
        <input  
          type="password"  
          value={password}  
          onChange={handlePasswordChange}>  
      />  
    </div>  
    <button type="submit">login</button>  
  </form>  
</div>  
)  
}  
  
export default LoginForm
```

The state and all the functions related to it are defined outside of the component and are passed to the component as props.

Notice that the props are assigned to variables through *destructuring*, which means that instead of writing:

```
const LoginForm = (props) => {  
  return (  
    <div>  
      <h2>Login</h2>  
      <form onSubmit={props.handleSubmit}>  
        <div>  
          username  
          <input  
            value={props.username}>
```

```

        onChange={props.handleChange}
        name="username"
      />
    </div>
    // ...
    <button type="submit">login</button>
  </form>
</div>
}
}

```

where the properties of the `props` object are accessed through e.g. `props.handleSubmit`, the properties are assigned directly to their own variables.

One fast way of implementing the functionality is to change the `loginForm` function of the `App` component like so:

```

const App = () => [
  const [loginVisible, setLoginVisible] = useState(false)
  // ...

  const loginForm = () => {
    const hideWhenVisible = { display: loginVisible ? 'none' : '' }
    const showWhenVisible = { display: loginVisible ? '' : 'none' }

    return [
      <div>
        <div style={hideWhenVisible}>
          <button onClick={() => setLoginVisible(true)}>log in</button>
        </div>
        <div style={showWhenVisible}>
          <LoginForm
            username={username}
            password={password}
            handleUsernameChange={({ target }) => setUsername(target.value)}
            handlePasswordChange={({ target }) => setPassword(target.value)}
            handleSubmit={handleLogin}
          />
          <button onClick={() => setLoginVisible(false)}>cancel</button>
        </div>
      </div>
    ]
  }
  // ...
]

```

The `App` component state now contains the boolean `loginVisible`, which defines if the login form should be shown to the user or not.

The value of `loginVisible` is toggled with two buttons. Both buttons have their event handlers defined directly in the component:

```

<button onClick={() => setLoginVisible(true)}>log in</button>

<button onClick={() => setLoginVisible(false)}>cancel</button>

```

The visibility of the component is defined by giving the component an [inline](#) style rule, where the value of the [display](#) property is *none* if we do not want the component to be displayed:

```

const hideWhenVisible = { display: loginVisible ? 'none' : '' }
const showWhenVisible = { display: loginVisible ? '' : 'none' }

<div style={hideWhenVisible}>
  // button
</div>

<div style={showWhenVisible}>
  // button
</div>

```

We are once again using the "question mark" ternary operator. If *loginVisible* is *true*, then the CSS rule of the component will be:

```
display: 'none'
```

If *loginVisible* is *false*, then *display* will not receive any value related to the visibility of the component.

The components children, aka. `props.children`

The code related to managing the visibility of the login form could be considered to be its own logical entity, and for this reason, it would be good to extract it from the *App* component into a separate component.

Our goal is to implement a new *Toggable* component that can be used in the following way:

```

<Toggable buttonLabel='login'>
  <LoginForm
    username={username}
    password={password}
    handleUsernameChange={({ target }) => setUsername(target.value)}
    handlePasswordChange={({ target }) => setPassword(target.value)}
    handleSubmit={handleLogin}
  />
</Toggable>

```

The way that the component is used is slightly different from our previous components. The component has both opening and closing tags that surround a *LoginForm* component. In React terminology *LoginForm* is a child component of *Toggable*.

We can add any React elements we want between the opening and closing tags of *Toggable*, like this for example:

```
<Toggable buttonLabel="reveal">
  <p>this line is at start hidden</p>
  <p>also this is hidden</p>
</Toggable>
```

The code for the *Toggable* component is shown below:

```
import { useState } from 'react'

const Toggable = (props) => {
  const [visible, setVisible] = useState(false)

  const hideWhenVisible = { display: visible ? 'none' : '' }
  const showWhenVisible = { display: visible ? '' : 'none' }

  const toggleVisibility = () => {
    setVisible(!visible)
  }

  return [
    <div>
      <div style={hideWhenVisible}>
        <button onClick={toggleVisibility}>{props.buttonLabel}</button>
      </div>
      <div style={showWhenVisible}>
        {props.children}
        <button onClick={toggleVisibility}>cancel</button>
      </div>
    </div>
  ]
}

export default Toggable
```

The new and interesting part of the code is `props.children`, which is used for referencing the child components of the component. The child components are the React elements that we define between the opening and closing tags of a component.

This time the children are rendered in the code that is used for rendering the component itself:

```
<div style={showWhenVisible}>
  {props.children}
  <button onClick={toggleVisibility}>cancel</button>
</div>
```

Unlike the "normal" props we've seen before, `children` is automatically added by React and always exists. If a component is defined with an automatically closing `/>` tag, like this:

```
<Note
  key={note.id}>
```

```
    note={note}
    toggleImportance={() => toggleImportanceOf(note.id)}
  />
```

Then `props.children` is an empty array.

The `Toggable` component is reusable and we can use it to add similar visibility toggling functionality to the form that is used for creating new notes.

Before we do that, let's extract the form for creating notes into a component:

```
const NoteForm = ({ onSubmit, handleChange, value }) => {
  return (
    <div>
      <h2>Create a new note</h2>

      <form onSubmit={onSubmit}>
        <input
          value={value}
          onChange={handleChange}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}
```

Next let's define the form component inside of a `Toggable` component:

```
<Toggable buttonLabel="new note">
  <NoteForm
    onSubmit={addNote}
    value={newNote}
    handleChange={handleNoteChange}
  />
</Toggable>
```

You can find the code for our current application in its entirety in the `part5-4` branch of [this GitHub repository](#).

State of the forms

The state of the application currently is in the `App` component.

React documentation says the [following](#) about where to place the state:

Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as lifting state up, and it's one of the most common things you will do writing React code.

If we think about the state of the forms, so for example the contents of a new note before it has been created, the *App* component does not need it for anything. We could just as well move the state of the forms to the corresponding components.

The component for creating a new note changes like so:

```
import { useState } from 'react'

const NoteForm = ({ createNote }) => {
  const [newNote, setNewNote] = useState('')

  const addNote = (event) => {
    event.preventDefault()
    createNote({
      content: newNote,
      important: true
    })

    setNewNote('')
  }

  return (
    <div>
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={event => setNewNote(event.target.value)}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}

export default NoteForm
```

NOTE At the same time, we changed the behavior of the application so that new notes are important by default, i.e. the field *important* gets the value *true*.

The *newNote* state variable and the event handler responsible for changing it have been moved from the *App* component to the component responsible for the note form.

There is only one prop left, the *createNote* function, which the form calls when a new note is created.

The *App* component becomes simpler now that we have got rid of the *newNote* state and its event handler. The *addNote* function for creating new notes receives a new note as a parameter, and the function is the only prop we send to the form:

```
const App = () => {
  // ...
```

```

const addNote = (noteObject) => {
  noteService
    .create(noteObject)
    .then(returnedNote => {
      setNotes(notes.concat(returnedNote))
    })
}

// ...
const noteForm = () => (
  <Toggable buttonLabel='new note'>
    <NoteForm createNote={addNote} />
  </Toggable>
)

// ...

```

We could do the same for the log in form, but we'll leave that for an optional exercise.

The application code can be found on [GitHub](#), branch *part5-5*.

References to components with ref

Our current implementation is quite good; it has one aspect that could be improved.

After a new note is created, it would make sense to hide the new note form. Currently, the form stays visible. There is a slight problem with hiding it, the visibility is controlled with the *visible* state variable inside of the *Toggable* component.

One solution to this would be to move control of the *Toggable* component's state outside the component. However, we won't do that now, because we want the component to be responsible for its own state. So we have to find another solution, and find a mechanism to change the state of the component externally.

There are several different ways to implement access to a component's functions from outside the component, but let's use the *ref* mechanism of React, which offers a reference to the component.

Let's make the following changes to the *App* component:

```

import { useState, useEffect, useRef } from 'react'
const App = () => [
  // ...
  const noteFormRef = useRef()
  const noteForm = () => (
    <Toggable buttonLabel='new note' ref={noteFormRef}>           <NoteForm createNo
    te={addNote} />
    </Toggable>
  )
  // ...
]

```

The *useRef* hook is used to create a *noteFormRef* reference, that is assigned to the *Toggable*

component containing the creation note form. The `noteFormRef` variable acts as a reference to the component. This hook ensures the same reference (ref) that is kept throughout re-renders of the component.

We also make the following changes to the `Toggable` component:

```
import { useState, forwardRef, useImperativeHandle } from 'react'
const Toggable = forwardRef((props, refs) => {
  const [visible, setVisible] = useState(false)

  const hideWhenVisible = { display: visible ? 'none' : '' }
  const showWhenVisible = { display: visible ? '' : 'none' }

  const toggleVisibility = () => {
    setVisible(!visible)
  }

  useImperativeHandle(refs, () => {
    return {
      toggleVisibility
    }
  })
}

export default Toggable
```

The function that creates the component is wrapped inside of a `forwardRef` function call. This way the component can access the ref that is assigned to it.

The component uses the `useImperativeHandle` hook to make its `toggleVisibility` function available outside of the component.

We can now hide the form by calling `noteFormRef.current.toggleVisibility()` after a new note has been created:

```
const App = () => {
  // ...
  const addNote = (noteObject) => {
    noteFormRef.current.toggleVisibility()
    noteService
      .create(noteObject)
      .then(returnedNote => {
        setNotes(notes.concat(returnedNote))
      })
  }
  // ...
}
```

To recap, the `useImperativeHandle` function is a React hook, that is used for defining functions in a component, which can be invoked from outside of the component.

This trick works for changing the state of a component, but it looks a bit unpleasant. We could have accomplished the same functionality with slightly cleaner code using "old React" class-based components. We will take a look at these class components during part 7 of the course material. So far this is the only situation where using React hooks leads to code that is not cleaner than with class components.

There are also [other use cases](#) for refs than accessing React components.

You can find the code for our current application in its entirety in the *part5-6* branch of [this GitHub repository](#).

One point about components

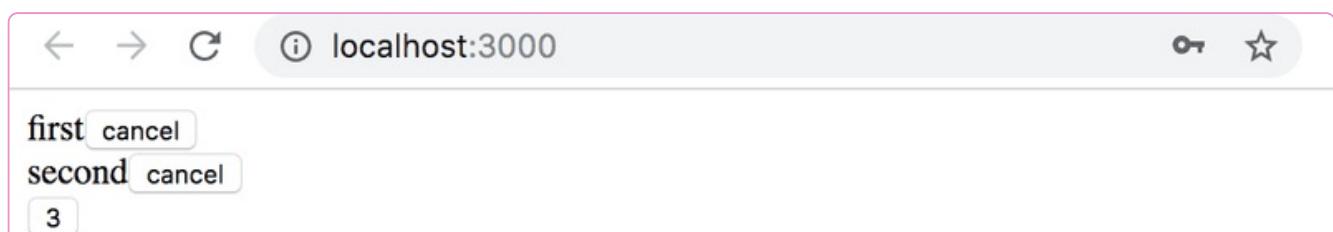
When we define a component in React:

```
const Toggable = () => ...  
// ...  
[]
```

And use it like this:

```
<div>  
  <Toggable buttonLabel="1" ref={toggable1}>  
    first  
  </Toggable>  
  
  <Toggable buttonLabel="2" ref={toggable2}>  
    second  
  </Toggable>  
  
  <Toggable buttonLabel="3" ref={toggable3}>  
    third  
  </Toggable>  
</div>
```

We create *three separate instances of the component* that all have their separate state:



The `ref` attribute is used for assigning a reference to each of the components in the variables `toggable1`, `toggable2` and `toggable3`.

The updated full stack developer's oath

The number of moving parts increases. At the same time, the likelihood of ending up in a situation where we are looking for a bug in the wrong place increases. So we need to be even more systematic.

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
- *when I suspect that there is a bug in the frontend, I'll make sure that the backend works as expected*
- *when I suspect that there is a bug in the backend, I'll make sure that the frontend works as expected*
- I will write lots of `console.log` statements to make sure I understand how the code and the tests behave and to help pinpoint problems
- If my code does not work, I will not write more code. Instead, I'll start deleting it until it works or will just return to a state where everything was still working
- If a test does not pass, I'll make sure that the tested functionality works properly in the application
- 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 5.5.-5.11.

5.5 Blog List Frontend, step 5

Change the form for creating blog posts so that it is only displayed when appropriate. Use functionality similar to what was shown [earlier in this part of the course material](#). If you wish to do so, you can use the *Toggable* component defined in part 5.

By default the form is not visible

A screenshot of a web browser window showing a blog list. The URL in the address bar is `localhost:3000`. The page title is "blogs". A user "Matti Luukkainen" is logged in, with a "logout" button. A red arrow points to a "new note" button. Below the button, two blog titles are listed: "Things I Don't Know as of 2018 Dan Abramov" and "Microservices and the First Law of Distributed Objects Martin Fowler".

It expands when button *create new blog* is clicked

A screenshot of a web browser window titled "localhost:3000". The page displays a "blogs" section with a "create new" button. Below it is a form with three input fields: "title:", "author:", and "url:". At the bottom of the form are two buttons: "create" and "cancel". A large red arrow points to the "create" button.

The form hides again after a new blog is created.

5.6 Blog List Frontend, step 6

Separate the form for creating a new blog into its own component (if you have not already done so), and move all the states required for creating a new blog to this component.

The component must work like the *NoteForm* component from the [material](#) of this part.

5.7 Blog List Frontend, step 7

Let's add a button to each blog, which controls whether all of the details about the blog are shown or not.

Full details of the blog open when the button is clicked.

A screenshot of a web browser window titled "localhost:3000". The page displays a "blogs" section. It shows a list of blog entries. The first entry is expanded, showing its title ("The Practical Test Pyramid Ham Vocke"), author ("Dan Abramov"), URL ("https://overreacted.io/on-let-vs-const/"), and a "hide" button. Below this, it shows the number of likes ("likes 0") and a "like" button. The second entry is collapsed, showing only its title ("Java Ternary Operator examples mykong"). Each entry has a "view" button at the end.

And the details are hidden when the button is clicked again.

At this point, the *like* button does not need to do anything.

The application shown in the picture has a bit of additional CSS to improve its appearance.

It is easy to add styles to the application as shown in part 2 using [inline](#) styles:

```
const Blog = ({ blog }) => {
  const blogStyle = {
    paddingTop: 10,
    paddingLeft: 2,
    border: 'solid',
    borderWidth: 1,
    marginBottom: 5
  }

  return (
    <div style={blogStyle}>      <div>
      {blog.title} {blog.author}
      </div>
      // ...
    </div>
  )
}
```

NB: Even though the functionality implemented in this part is almost identical to the functionality provided by the *Toggable* component, it can't be used directly to achieve the desired behavior. The easiest solution would be to add a state to the blog component that controls if the details are being displayed or not.

5.8: Blog List Frontend, step 8

Implement the functionality for the like button. Likes are increased by making an HTTP *PUT* request to the unique address of the blog post in the backend.

Since the backend operation replaces the entire blog post, you will have to send all of its fields in the request body. If you wanted to add a like to the following blog post:

```
{
  _id: "5a43fde2cbd20b12a2c34e91",
  user: {
    _id: "5a43e6b6c37f3d065eaaa581",
    username: "mluukkai",
    name: "Matti Luukkainen"
  },
  likes: 0,
  author: "Joel Spolsky",
  title: "The Joel Test: 12 Steps to Better Code",
  url: "https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/"
}
```

You would have to make an HTTP PUT request to the address `/api/blogs/5a43fde2cbd20b12a2c34e91` with the following request data:

```
{  
  user: "5a43e6b6c37f3d065eaaa581",  
  likes: 1,  
  author: "Joel Spolsky",  
  title: "The Joel Test: 12 Steps to Better Code",  
  url: "https://www.joelonsoftware.com/2000/08/09/the-j Joel Test: 12 steps to better code/"  
}
```

The backend has to be updated too to handle the user reference.

5.9: Blog List Frontend, step 9

We notice that something is wrong. When a blog is liked in the app, the name of the user that added the blog is not shown in its details:

localhost:5173

FP vs. OO List Processing Robert C. Martin [hide](#)
<https://blog.cleancoder.com/uncle-bob/2018/12/17/FPvsOO-List-processing.html>
likes 0 [like](#)
Matti Luukainen

On let vs const Dan Abramov [hide](#)
<https://overreacted.io/on-let-vs-const/>
likes 1 [like](#)

When the browser is reloaded, the information of the person is displayed. This is not acceptable, find out where the problem is and make the necessary correction.

Of course, it is possible that you have already done everything correctly and the problem does not occur in your code. In that case, you can move on.

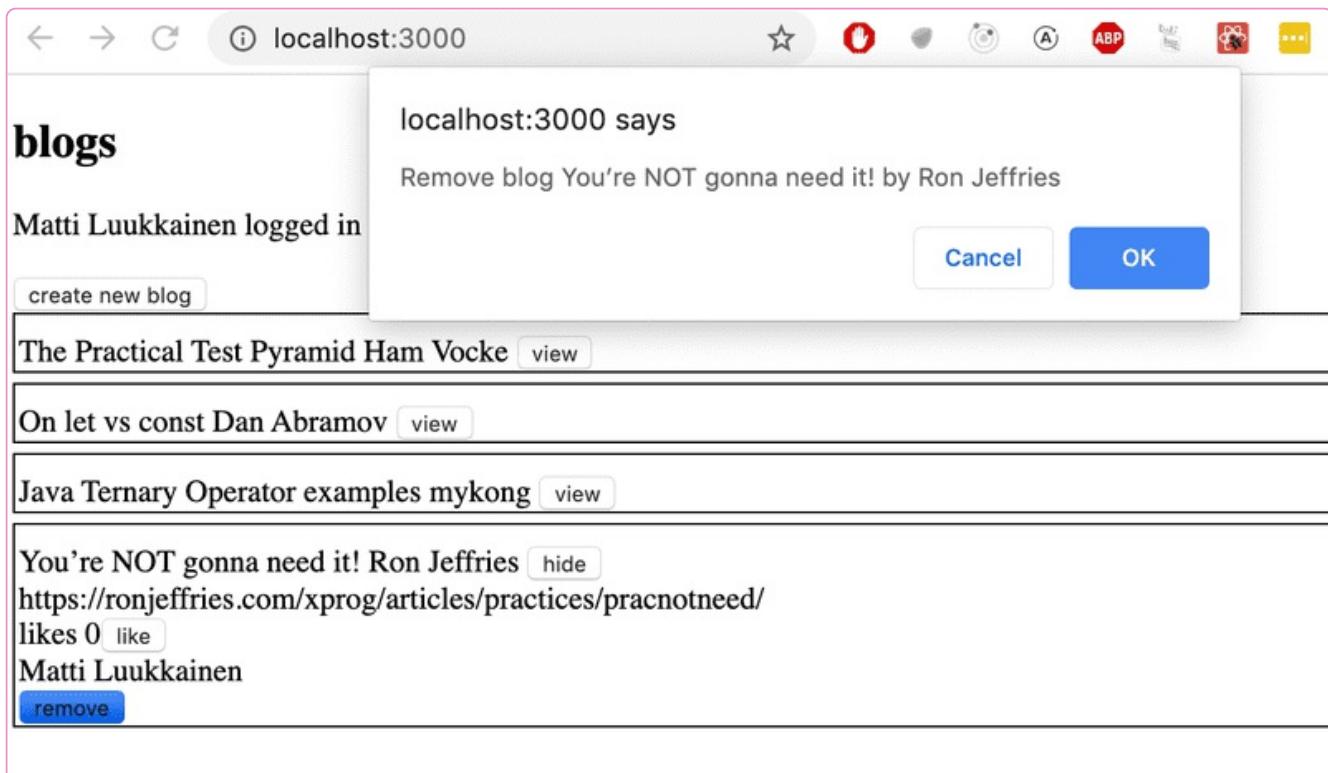
5.10: Blog List Frontend, step 10

Modify the application to sort the blog posts by the number of *likes*. The Sorting can be done with the array `sort` method.

5.11: Blog List Frontend, step 11

Add a new button for deleting blog posts. Also, implement the logic for deleting blog posts in the frontend.

Your application could look something like this:



The confirmation dialog for deleting a blog post is easy to implement with the `window.confirm` function.

Show the button for deleting a blog post only if the blog post was added by the user.

PropTypes

The `Toggable` component assumes that it is given the text for the button via the `buttonLabel` prop. If we forget to define it to the component:

```
<Toggable> buttonLabel forgotten... </Toggable>
```

The application works, but the browser renders a button that has no label text.

We would like to enforce that when the `Toggable` component is used, the button label text prop must be given a value.

The expected and required props of a component can be defined with the `prop-types` package. Let's install the package:

```
npm install prop-types
```

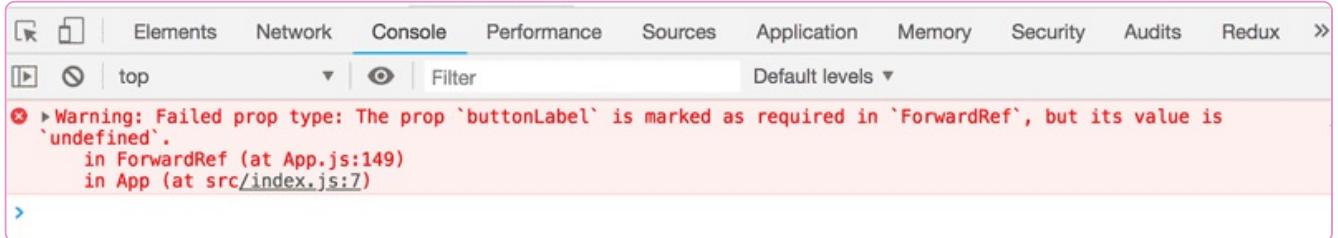
We can define the `buttonLabel` prop as a mandatory or *required* string-type prop as shown below:

```
import PropTypes from 'prop-types'

const Toggable = React.forwardRef((props, ref) => [
  // ...
])
```

```
Toggable.propTypes = {
  buttonLabel: PropTypes.string.isRequired
}
```

The console will display the following error message if the prop is left undefined:



The application still works and nothing forces us to define props despite the PropTypes definitions. Mind you, it is extremely unprofessional to leave *any* red output in the browser console.

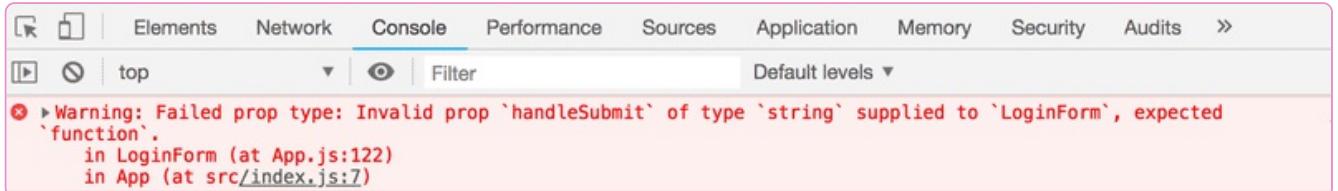
Let's also define PropTypes to the *LoginForm* component:

```
import PropTypes from 'prop-types'

const LoginForm = ({
  handleSubmit,
  handleUsernameChange,
  handlePasswordChange,
  username,
  password
}) => [
  // ...
]

LoginForm.propTypes = {
  handleSubmit: PropTypes.func.isRequired,
  handleUsernameChange: PropTypes.func.isRequired,
  handlePasswordChange: PropTypes.func.isRequired,
  username: PropTypes.string.isRequired,
  password: PropTypes.string.isRequired
}
```

If the type of a passed prop is wrong, e.g. if we try to define the *handleSubmit* prop as a string, then this will result in the following warning:



ESlint

In part 3 we configured the [ESlint](#) code style tool to the backend. Let's take ESlint to use in the frontend as well.

Vite has installed ESLint to the project by default, so all that's left for us to do is define our desired configuration in the `.eslintrc.cjs` file.

Let's create a `.eslintrc.cjs` file with the following contents:

```
module.exports = [
  root: true,
  env: {
    browser: true,
    es2020: true,
  },
  extends: [
    'eslint:recommended',
    'plugin:react/recommended',
    'plugin:react/jsx-runtime',
    'plugin:react-hooks/recommended',
  ],
  ignorePatterns: ['dist', '.eslintrc.cjs'],
  parserOptions: { ecmaVersion: 'latest', sourceType: 'module' },
  settings: { react: { version: '18.2' } },
  plugins: ['react-refresh'],
  rules: {
    "indent": [
      "error",
      2
    ],
    "linebreak-style": [
      "error",
      "unix"
    ],
    "quotes": [
      "error",
      "single"
    ],
    "semi": [
      "error",
      "never"
    ],
    "eqeqeq": "error",
    "no-trailing-spaces": "error",
    "object-curly-spacing": [
      "error",
      "always"
    ],
    "arrow-spacing": [
      "error",
      { "before": true, "after": true }
    ],
    "no-console": 0,
    "react/react-in-jsx-scope": "off",
    "react/prop-types": 0,
    "no-unused-vars": 0
  }
]
```

NOTE: If you are using Visual Studio Code together with ESLint plugin, you might need to add a workspace setting for it to work. If you are seeing `Failed to load plugin react: Cannot find module 'eslint-plugin-react'` additional configuration is needed. Adding the line `"eslint.workingDirectories": [{ "mode": "auto" }]` to `settings.json` in the workspace seems to work. See [here](#) for more information.

Let's create `.eslintignore` file with the following contents to the repository root

```
node_modules
dist
.eslintrc.cjs
vite.config.js
```

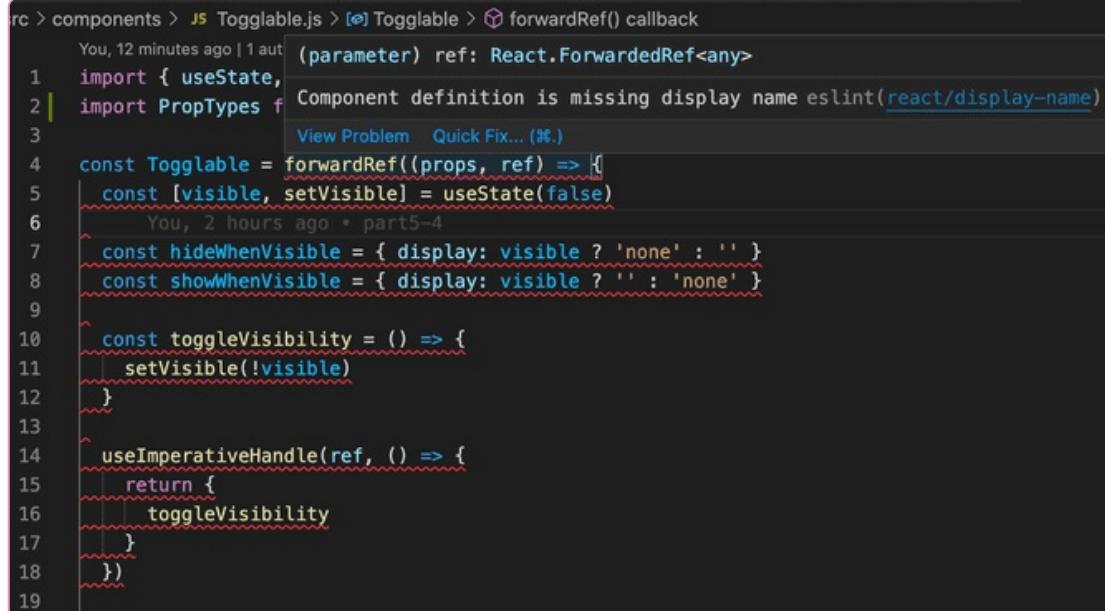
Now the directories `dist` and `node_modules` will be skipped when linting.

As usual, you can perform the linting either from the command line with the command

```
npm run lint
```

or using the editor's Eslint plugin.

Component `Toggable` causes a nasty-looking warning `Component definition is missing display name`:



The screenshot shows a code editor with a dark theme. A tooltip is displayed over the word `ref` in line 1, which reads: `(parameter) ref: React.ForwardedRef<any>`. Below the tooltip, a message says: `Component definition is missing display name eslint(react/display-name)`. The code itself is a component named `Toggable` using `useState` and `useImperativeHandle`.

```
rc > components > JS Toggable.js > [x] Toggable > ⓘ forwardRef() callback
You, 12 minutes ago | 1 aut
1 import { useState,
2 import PropTypes f Component definition is missing display name eslint(react/display-name)
3
4 const Toggable = forwardRef((props, ref) => {
5   const [visible, setVisible] = useState(false)
6   You, 2 hours ago * part5-4
7   const hideWhenVisible = { display: visible ? 'none' : '' }
8   const showWhenVisible = { display: visible ? '' : 'none' }
9
10  const toggleVisibility = () => {
11    setVisible(!visible)
12  }
13
14  useImperativeHandle(ref, () => {
15    return {
16      toggleVisibility
17    }
18  })
19
```

The react-devtools also reveals that the component does not have a name:

Matti Luukkainen logged in

new note
show important

- HTML is Easy [make not important]
- Mongoose makes use of mongo easy [make not important]

Elements Console Sources Network Performance Memory Components Application Security Audits

Search (text or /regex/)

App

Notification

Anonymous ForwardRef

NoteForm

Note key="0"
Note key="1"
Note key="2"
Note key="3"
Note key="4"

Fortunately, this is easy to fix

```
import { useState, useImperativeHandle } from 'react'  
import PropTypes from 'prop-types'  
  
const Toggable = React.forwardRef((props, ref) => {  
  // ...  
})  
  
Toggable.displayName = 'Toggable'  
export default Toggable
```

You can find the code for our current application in its entirety in the *part5-7* branch of [this GitHub repository](#).

Exercise 5.12.

5.12: Blog List Frontend, step 12

Define PropTypes for one of the components of your application, and add ESLint to the project. Define the configuration according to your liking. Fix all of the linter errors.

Vite has installed ESLint to the project by default, so all that's left for you to do is define your desired configuration in the `.eslintrc.cjs` file.

5c: Testing React apps

The test library used in this part was changed on March 3, 2024 from Jest to Vitest. If you already started this part using Jest, you can see [here](#) the old content.

There are many different ways of testing React applications. Let's take a look at them next.

The course previously used the [Jest](#) library developed by Facebook to test React components. We are now using the new generation of testing tools from Vite developers called [Vitest](#). Apart from the configurations, the libraries provide the same programming interface, so there is virtually no difference in the test code.

Let's start by installing Vitest and the [jsdom](#) library simulating a web browser:

```
npm install --save-dev vitest jsdom
```

In addition to Vitest, we also need another testing library that will help us render components for testing purposes. The current best option for this is [react-testing-library](#) which has seen rapid growth in popularity in recent times. It is also worth extending the expressive power of the tests with the library [jest-dom](#).

Let's install the libraries with the command:

```
npm install --save-dev @testing-library/react @testing-library/jest-dom
```

Before we can do the first test, we need some configurations.

We add a script to the *package.json* file to run the tests:

```
{
  "scripts": {
    // ...
    "test": "vitest run"
  }
  // ...
}
```

Let's create a file *testSetup.js* in the project root with the following content

```
import { afterEach } from 'vitest'
import { cleanup } from '@testing-library/react'
import '@testing-library/jest-dom/vitest'

afterEach(() => [
  cleanup()
])
```

Now, after each test, the function *cleanup* is executed to reset jsdom, which is simulating the browser.

Expand the *vite.config.js* file as follows

```
export default defineConfig({
  // ...
  test: {
    environment: 'jsdom',
    globals: true,
```

```
    setupFiles: './testSetup.js'  
  }  
})
```

With `globals: true`, there is no need to import keywords such as `describe`, `test` and `expect` into the tests.

Let's first write tests for the component that is responsible for rendering a note:

```
const Note = ({ note, toggleImportance }) => [  
  const label = note.important  
    ? 'make not important'  
    : 'make important'  
  
  return [  
    <li className='note'> {note.content}  
      <button onClick={toggleImportance}>{label}</button>  
    </li>  
  ]  
]
```

Notice that the `li` element has the value `note` for the `CSS` attribute `className`, that could be used to access the component in our tests.

Rendering the component for tests

We will write our test in the `src/components/Note.test.jsx` file, which is in the same directory as the component itself.

The first test verifies that the component renders the contents of the note:

```
import { render, screen } from '@testing-library/react'  
import Note from './Note'  
  
test('renders content', () => [  
  const note = [  
    content: 'Component testing is done with react-testing-library',  
    important: true  
  ]  
  
  render(<Note note={note} />)  
  
  const element = screen.getByText('Component testing is done with react-testing  
-library')  
  expect(element).toBeDefined()  
])
```

After the initial configuration, the test renders the component with the `render` function provided by the `react-testing-library`:

```
render[<Note note={note} />]
```

Normally React components are rendered to the **DOM**. The render method we used renders the components in a format that is suitable for tests without rendering them to the DOM.

We can use the object `screen` to access the rendered component. We use screen's method `getByText` to search for an element that has the note content and ensure that it exists:

```
const element = screen.getByText('Component testing is done with react-testing  
-library')  
expect(element).toBeDefined()
```

The existence of an element is checked using Vitest's `expect` command. Expect generates an assertion for its argument, the validity of which can be tested using various condition functions. Now we used `toBeDefined` which tests whether the *element* argument of expect exists.

Run the test with command `npm test`:

```
$ npm test  
  
> notes-frontend@0.0.0 test  
> vitest  
  
DEV  v1.3.1  /Users/mluukkai/opetus/2024-fs/part3/notes-frontend  
  
✓  src/components/Note.test.jsx [1]  
  ✓  renders content  
  
Test Files  1 passed [1]  
  Tests  1 passed [1]  
  Start at  17:05:37  
  Duration  812ms [transform 31ms, setup 220ms, collect 11ms, tests 14ms, environment 395ms, prepare 70ms]  
  
PASS  Waiting for file changes...
```

Eslint complains about the keywords `test` and `expect` in the tests. The problem can be solved by installing `eslint-plugin-vitest-globals`:

```
npm install --save-dev eslint-plugin-vitest-globals
```

and enable the plugin by editing the `.eslint.cjs` file as follows:

```
module.exports = [  
  root: true,  
  env: {  
    browser: true,
```

```
es2020: true,
"vitest-globals/env": true },
extends: [
  'eslint:recommended',
  'plugin:react/recommended',
  'plugin:react/jsx-runtime',
  'plugin:react-hooks/recommended',
  'plugin:vitest-globals/recommended'],
// ...
]
```

Test file location

In React there are (at least) [two different conventions](#) for the test file's location. We created our test files according to the current standard by placing them in the same directory as the component being tested.

The other convention is to store the test files "normally" in a separate *test* directory. Whichever convention we choose, it is almost guaranteed to be wrong according to someone's opinion.

I do not like this way of storing tests and application code in the same directory. The reason we choose to follow this convention is that it is configured by default in applications created by Vite or `create-react-app`.

Searching for content in a component

The `react-testing-library` package offers many different ways of investigating the content of the component being tested. In reality, the `expect` in our test is not needed at all:

```
import { render, screen } from '@testing-library/react'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.getByText('Component testing is done with react-testing-library')
  expect(element).toBeDefined()
})
```

Test fails if `getByText` does not find the element it is looking for.

We could also use [CSS-selectors](#) to find rendered elements by using the method `querySelector` of the object `container` that is one of the fields returned by the `render`:

```
import { render, screen } from '@testing-library/react'
```

```

import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const { container } = render(<Note note={note} />)
  const div = container.querySelector('.note') expect(div).toHaveTextContent(
    'Component testing is done with react-testing-library' ))
})

```

NB: A more consistent way of selecting elements is using a [data attribute](#) that is specifically defined for testing purposes. Using *react-testing-library*, we can leverage the [getByTestId](#) method to select elements with a specified *data-testid* attribute.

Debugging tests

We typically run into many different kinds of problems when writing our tests.

Object *screen* has method [debug](#) that can be used to print the HTML of a component to the terminal. If we change the test as follows:

```

import { render, screen } from '@testing-library/react'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  screen.debug()
  // ...
})

```

the HTML gets printed to the console:

```

console.log
<body>
  <div>
    <li
      class="note"
    >
      Component testing is done with react-testing-library
      <button>
        make not important
      </button>
    
```

```
</li>
</div>
</body>
```

It is also possible to use the same method to print a wanted element to console:

```
import { render, screen } from '@testing-library/react'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />

  const element = screen.getByText('Component testing is done with react-testing
-library')

  screen.debug(element)
  expect(element).toBeDefined()
})
```

Now the HTML of the wanted element gets printed:

```
<li
  class="note"
>
  Component testing is done with react-testing-library
  <button>
    make not important
  </button>
</li>
```

Clicking buttons in tests

In addition to displaying content, the *Note* component also makes sure that when the button associated with the note is pressed, the *toggleImportance* event handler function gets called.

Let us install a library [user-event](#) that makes simulating user input a bit easier:

```
npm install --save-dev @testing-library/user-event
```

Testing this functionality can be accomplished like this:

```
import { render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event' import Note from './Note'
```

```
// ...

test('clicking the button calls event handler once', async () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const mockHandler = vi.fn()
  render(
    <Note note={note} toggleImportance={mockHandler} />
  )

  const user = userEvent.setup()  const button = screen.getByText('make not important')
  await user.click(button)
  expect(mockHandler.mock.calls).toHaveLength(1))
})
```

There are a few interesting things related to this test. The event handler is a [mock](#) function defined with Vitest:

```
const mockHandler = vi.fn()
```

A [session](#) is started to interact with the rendered component:

```
const user = userEvent.setup()
```

The test finds the button *based on the text* from the rendered component and clicks the element:

```
const button = screen.getByText('make not important')
await user.click(button)
```

Clicking happens with the method [click](#) of the userEvent-library.

The expectation of the test uses [toHaveLength](#) to verify that the *mock function* has been called exactly once:

```
expect(mockHandler.mock.calls).toHaveLength(1)
```

The calls to the mock function are saved to the array [mock.calls](#) within the mock function object.

[Mock objects and functions](#) are commonly used [stub](#) components in testing that are used for replacing dependencies of the components being tested. Mocks make it possible to return hardcoded responses, and to verify the number of times the mock functions are called and with what parameters.

In our example, the mock function is a perfect choice since it can be easily used for verifying that the method gets called exactly once.

Tests for the *Toggable* component

Let's write a few tests for the *Toggable* component. Let's add the *toggableContent* CSS classname to

the div that returns the child components.

```
const Toggable = forwardRef((props, ref) => {
  // ...

  return [
    <div>
      <div style={hideWhenVisible}>
        <button onClick={toggleVisibility}>
          {props.buttonLabel}
        </button>
      </div>
      <div style={showWhenVisible} className="toggableContent"> {props.children}
        <button onClick={toggleVisibility}>cancel</button>
      </div>
    </div>
  ]
})
```

The tests are shown below:

```
import { render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import Toggable from './Toggable'

describe('<Toggable />', () => {
  let container

  beforeEach(() => {
    container = render(
      <Toggable buttonLabel="show...">
        <div className="testDiv" >
          toggable content
        </div>
      </Toggable>
    ).container
  })

  test('renders its children', async () => {
    await screen.findAllByText('toggable content')
  })

  test('at start the children are not displayed', () => {
    const div = container.querySelector('.toggableContent')
    expect(div).toHaveStyle('display: none')
  })

  test('after clicking the button, children are displayed', async () => {
    const user = userEvent.setup()
    const button = screen.getByText('show...')
    user.click(button)
    await screen.update()
    const div = container.querySelector('.toggableContent')
    expect(div).not.toHaveStyle('display: none')
  })
})
```

```

    await user.click(button)

    const div = container.querySelector('.toggableContent')
    expect(div).not.toHaveStyle('display: none')
  })
})

```

The `beforeEach` function gets called before each test, which then renders the `Toggable` component and saves the field `container` of the returned value.

The first test verifies that the `Toggable` component renders its child component

```

<div className="testDiv">
  togglable content
</div>

```

The remaining tests use the `toHaveStyle` method to verify that the child component of the `Toggable` component is not visible initially, by checking that the style of the `div` element contains `{ display: 'none' }`. Another test verifies that when the button is pressed the component is visible, meaning that the style for hiding it *is no longer* assigned to the component.

Let's also add a test that can be used to verify that the visible content can be hidden by clicking the second button of the component:

```

describe('<Toggable />', () => [
  // ...

  test('toggled content can be closed', async () => {
    const user = userEvent.setup()
    const button = screen.getByText('show...')
    await user.click(button)

    const closeButton = screen.getByText('cancel')
    await user.click(closeButton)

    const div = container.querySelector('.toggableContent')
    expect(div).toHaveStyle('display: none')
  })
])

```

Testing the forms

We already used the `click` function of the `user-event` in our previous tests to click buttons.

```

const user = userEvent.setup()
const button = screen.getByText('show...')
await user.click(button)

```

We can also simulate text input with `userEvent`.

Let's make a test for the *NoteForm* component. The code of the component is as follows.

```
import { useState } from 'react'

const NoteForm = ({ createNote }) => {
  const [newNote, setNewNote] = useState('')

  const handleChange = (event) => {
    setNewNote(event.target.value)
  }

  const addNote = (event) => {
    event.preventDefault()
    createNote({
      content: newNote,
      important: true,
    })
    setNewNote('')
  }

  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleChange}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}

export default NoteForm
```

The form works by calling the function received as props *createNote*, with the details of the new note.

The test is as follows:

```
import { render, screen } from '@testing-library/react'
import NoteForm from './NoteForm'
import userEvent from '@testing-library/user-event'

test('<NoteForm /> updates parent state and calls onSubmit', async () => {
  const createNote = vi.fn()
  const user = userEvent.setup()

  render(<NoteForm createNote={createNote} />)
```

```

const input = screen.getByRole('textbox')
const sendButton = screen.getText('save')

await user.type(input, 'testing a form...')
await user.click(sendButton)

expect(createNote.mock.calls).toHaveLength(1)
expect(createNote.mock.calls[0][0].content).toBe('testing a form...')

})

```

Tests get access to the input field using the function `getByRole`.

The method `type` of the `userEvent` is used to write text to the input field.

The first test expectation ensures that submitting the form calls the `createNote` method. The second expectation checks that the event handler is called with the right parameters - that a note with the correct content is created when the form is filled.

It's worth noting that the good old `console.log` works as usual in the tests. For example, if you want to see what the calls stored by the mock-object look like, you can do the following

```

test('<NoteForm /> updates parent state and calls onSubmit', async() => {
  const user = userEvent.setup()
  const createNote = vi.fn()

  render(<NoteForm createNote={createNote} />

  const input = screen.getByRole('textbox')
  const sendButton = screen.getText('save')

  await user.type(input, 'testing a form...')
  await user.click(sendButton)

  console.log(createNote.mock.calls))
}

```

In the middle of running the tests, the following is printed in the console:

```
[ [ { content: 'testing a form...', important: true } ] ]
```

About finding the elements

Let us assume that the form has two input fields

```

const NoteForm = ({ createNote }) => {
  // ...

  return [
    <div className="formDiv">
      <h2>Create a new note</h2>

```

```

<form onSubmit={addNote}>
  <input
    value={newNote}
    onChange={handleChange}
  />
  <input type="text" value="..." onChange={...} />
  <button type="submit">save</button>
</form>
</div>
)
}

```

Now the approach that our test uses to find the input field

```
const input = screen.getByRole('textbox')
```

would cause an error:

```

FAIL src/components/NoteForm.test.js
● <NoteForm /> updates parent state and calls onSubmit

  TestingLibraryElementError: Found multiple elements with the role "textbox"

  Here are the matching elements:

  Ignored nodes: comments, <script />, <style />
<input
  value=""
/>

  Ignored nodes: comments, <script />, <style />
<input />

  (If this is intentional, then use the `*AllBy*` variant of the query (like
queryAllByText`, `getAllByText`, or `findAllByText`)).
```

The error message suggests using `getAllByRole`. The test could be fixed as follows:

```
const inputs = screen.getAllByRole('textbox')

await user.type(inputs[0], 'testing a form...')
```

Method `getAllByRole` now returns an array and the right input field is the first element of the array. However, this approach is a bit suspicious since it relies on the order of the input fields.

Quite often input fields have a `placeholder` text that hints user what kind of input is expected. Let us add a placeholder to our form:

```
const NoteForm = [createNote] => [
```

```
// ...

return [
  <div className="formDiv">
    <h2>Create a new note</h2>

    <form onSubmit={addNote}>
      <input
        value={newNote}
        onChange={handleChange}
        placeholder='write note content here' />
      <input
        value={...}
        onChange={...}
      />
      <button type="submit">save</button>
    </form>
  </div>
]
```

Now finding the right input field is easy with the method `getByPlaceholderText`:

```
test('<NoteForm /> updates parent state and calls onSubmit', () => {
  const createNote = vi.fn()

  render(<NoteForm createNote={createNote} />)

  const input = screen.getByPlaceholderText('write note content here')  const sendButton = screen.getByText('save')

  userEvent.type(input, 'testing a form...')
  userEvent.click(sendButton)

  expect(createNote.mock.calls).toHaveLength(1)
  expect(createNote.mock.calls[0][0].content).toBe('testing a form...')
})
```

The most flexible way of finding elements in tests is the method `querySelector` of the `container` object, which is returned by `render`, as was mentioned [earlier in this part](#). Any CSS selector can be used with this method for searching elements in tests.

Consider eg. that we would define a unique `id` to the input field:

```
const NoteForm = ({ createNote }) => [
  // ...

  return [
    <div className="formDiv">
      <h2>Create a new note</h2>
```

```

    <form onSubmit={addNote}>
      <input
        value={newNote}
        onChange={handleChange}
        id='note-input'          />
      <input
        value={...}
        onChange={...}
      />
      <button type="submit">save</button>
    </form>
  </div>
}

}

```

The input element could now be found in the test as follows:

```

const [ container ] = render(<NoteForm createNote={createNote} />)

const input = container.querySelector('#note-input')

```

However, we shall stick to the approach of using *getByPlaceholderText* in the test.

Let us look at a couple of details before moving on. Let us assume that a component would render text to an HTML element as follows:

```

const Note = ({ note, toggleImportance }) => [
  const label = note.important
    ? 'make not important' : 'make important'

  return [
    <li className='note'>
      Your awesome note: {note.content}           <button onClick={toggleImportance}>{label}</button>
    </li>
  ]
]

export default Note

```

the *getByText* method that the test uses does *not* find the element

```

test('renders content', () => [
  const note = {
    content: 'Does not work anymore :(',
    important: true
  }

  render(<Note note={note} />)
]

```

```
const element = screen.getByText('Does not work anymore :(')

expect(element).toBeInTheDocument()
})
```

The `getByText` method looks for an element that has the **same text** that it has as a parameter, and nothing more. If we want to look for an element that *contains* the text, we could use an extra option:

```
const element = screen.getByText(
  'Does not work anymore :(', { exact: false }
)
```

or we could use the `findByText` method:

```
const element = await screen.findByText('Does not work anymore :(')
```

It is important to notice that, unlike the other `ByText` methods, `findByText` returns a promise!

There are situations where yet another form of the `queryByText` method is useful. The method returns the element but *it does not cause an exception* if it is not found.

We could eg. use the method to ensure that something *is not rendered* to the component:

```
test('does not render this', () => {
  const note = {
    content: 'This is a reminder',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.queryByText('do not want this thing to be rendered')
  expect(element).toBeNull()
})
```

Test coverage

We can easily find out the `coverage` of our tests by running them with the command.

```
npm test -- --coverage
```

The first time you run the command, Vitest will ask you if you want to install the required library `@vitest/coverage-v8`. Install it, and run the command again:

% Coverage report from v8					
File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Line #s
All files	22.92	55.55	41.66	22.92	
src	0	0	0	0	
App.jsx	0	0	0	0	1-150
main.jsx	0	0	0	0	1-5
src/components	51.28	71.42	62.5	51.28	
Footer.jsx	0	0	0	0	1-16
LoginForm.jsx	0	0	0	0	1-44
Note.jsx	100	50	100	100	3
NoteForm.jsx	100	100	100	100	
Notification.jsx	0	0	0	0	1-13
Toggable.jsx	92.3	100	100	92.3	15-17
src/services	0	0	0	0	
login.js	0	0	0	0	1-9
notes.js	0	0	0	0	1-29

A HTML report will be generated to the `coverage` directory. The report will tell us the lines of untested code in each component:

All files / src/components Toggable.jsx

92.3% Statements 36/39 100% Branches 6/6 100% Functions 1/1 92.3% Lines 36/39

Press `n` or `j` to go to the next uncovered block, `b`, `p` or `k` for the previous block.

```

1 1x import { useState, useImperativeHandle, forwardRef } from 'react'
2 1x import PropTypes from 'prop-types'
3 1x
4 1x const Toggable = forwardRef((props, ref) => {
5 7x   const [visible, setVisible] = useState(false)
6 7x
7 7x   const hideWhenVisible = { display: visible ? 'none' : '' }
8 7x   const showWhenVisible = { display: visible ? '' : 'none' }
9 7x
10 7x   const toggleVisibility = () => {
11 3x     setVisible(!visible)
12 3x   }
13 7x
14 7x   useImperativeHandle(ref, () => {
15    return {
16      toggleVisibility
17    }
18 7x   })
19 7x
20 7x   return (
21 7x     <div>
22 7x       <div style={hideWhenVisible}>
23 7x         <button onClick={toggleVisibility}>{props.buttonLabel}</button>
24 7x       </div>
25 7x       <div style={showWhenVisible} className="toggableContent">
26 7x         {props.children}
27 7x         <button onClick={toggleVisibility}>cancel</button>
28 7x       </div>
29 7x     </div>
30 7x   )
31 7x
32 7x
33 7x
34 7x
35 7x
36 7x
37 7x
38 7x
39 7x
40 7x
41 7x
42 7x
43 7x
44 7x
45 7x
46 7x
47 7x
48 7x
49 7x
50 7x
51 7x
52 7x
53 7x
54 7x
55 7x
56 7x
57 7x
58 7x
59 7x
60 7x
61 7x
62 7x
63 7x
64 7x
65 7x
66 7x
67 7x
68 7x
69 7x
70 7x
71 7x
72 7x
73 7x
74 7x
75 7x
76 7x
77 7x
78 7x
79 7x
80 7x
81 7x
82 7x
83 7x
84 7x
85 7x
86 7x
87 7x
88 7x
89 7x
90 7x
91 7x
92 7x
93 7x
94 7x
95 7x
96 7x
97 7x
98 7x
99 7x
100 7x
101 7x
102 7x
103 7x
104 7x
105 7x
106 7x
107 7x
108 7x
109 7x
110 7x
111 7x
112 7x
113 7x
114 7x
115 7x
116 7x
117 7x
118 7x
119 7x
120 7x
121 7x
122 7x
123 7x
124 7x
125 7x
126 7x
127 7x
128 7x
129 7x
130 7x
131 7x
132 7x
133 7x
134 7x
135 7x
136 7x
137 7x
138 7x
139 7x
140 7x
141 7x
142 7x
143 7x
144 7x
145 7x
146 7x
147 7x
148 7x
149 7x
150 7x
151 7x
152 7x
153 7x
154 7x
155 7x
156 7x
157 7x
158 7x
159 7x
160 7x
161 7x
162 7x
163 7x
164 7x
165 7x
166 7x
167 7x
168 7x
169 7x
170 7x
171 7x
172 7x
173 7x
174 7x
175 7x
176 7x
177 7x
178 7x
179 7x
180 7x
181 7x
182 7x
183 7x
184 7x
185 7x
186 7x
187 7x
188 7x
189 7x
190 7x
191 7x
192 7x
193 7x
194 7x
195 7x
196 7x
197 7x
198 7x
199 7x
200 7x
201 7x
202 7x
203 7x
204 7x
205 7x
206 7x
207 7x
208 7x
209 7x
210 7x
211 7x
212 7x
213 7x
214 7x
215 7x
216 7x
217 7x
218 7x
219 7x
220 7x
221 7x
222 7x
223 7x
224 7x
225 7x
226 7x
227 7x
228 7x
229 7x
230 7x
231 7x
232 7x
233 7x
234 7x
235 7x
236 7x
237 7x
238 7x
239 7x
240 7x
241 7x
242 7x
243 7x
244 7x
245 7x
246 7x
247 7x
248 7x
249 7x
250 7x
251 7x
252 7x
253 7x
254 7x
255 7x
256 7x
257 7x
258 7x
259 7x
260 7x
261 7x
262 7x
263 7x
264 7x
265 7x
266 7x
267 7x
268 7x
269 7x
270 7x
271 7x
272 7x
273 7x
274 7x
275 7x
276 7x
277 7x
278 7x
279 7x
280 7x
281 7x
282 7x
283 7x
284 7x
285 7x
286 7x
287 7x
288 7x
289 7x
290 7x
291 7x
292 7x
293 7x
294 7x
295 7x
296 7x
297 7x
298 7x
299 7x
300 7x
301 7x
302 7x
303 7x
304 7x
305 7x
306 7x
307 7x
308 7x
309 7x
310 7x
311 7x
312 7x
313 7x
314 7x
315 7x
316 7x
317 7x
318 7x
319 7x
320 7x
321 7x
322 7x
323 7x
324 7x
325 7x
326 7x
327 7x
328 7x
329 7x
330 7x
331 7x
332 7x
333 7x
334 7x
335 7x
336 7x
337 7x
338 7x
339 7x
340 7x
341 7x
342 7x
343 7x
344 7x
345 7x
346 7x
347 7x
348 7x
349 7x
350 7x
351 7x
352 7x
353 7x
354 7x
355 7x
356 7x
357 7x
358 7x
359 7x
360 7x
361 7x
362 7x
363 7x
364 7x
365 7x
366 7x
367 7x
368 7x
369 7x
370 7x
371 7x
372 7x
373 7x
374 7x
375 7x
376 7x
377 7x
378 7x
379 7x
380 7x
381 7x
382 7x
383 7x
384 7x
385 7x
386 7x
387 7x
388 7x
389 7x
390 7x
391 7x
392 7x
393 7x
394 7x
395 7x
396 7x
397 7x
398 7x
399 7x
400 7x
401 7x
402 7x
403 7x
404 7x
405 7x
406 7x
407 7x
408 7x
409 7x
410 7x
411 7x
412 7x
413 7x
414 7x
415 7x
416 7x
417 7x
418 7x
419 7x
420 7x
421 7x
422 7x
423 7x
424 7x
425 7x
426 7x
427 7x
428 7x
429 7x
430 7x
431 7x
432 7x
433 7x
434 7x
435 7x
436 7x
437 7x
438 7x
439 7x
440 7x
441 7x
442 7x
443 7x
444 7x
445 7x
446 7x
447 7x
448 7x
449 7x
450 7x
451 7x
452 7x
453 7x
454 7x
455 7x
456 7x
457 7x
458 7x
459 7x
460 7x
461 7x
462 7x
463 7x
464 7x
465 7x
466 7x
467 7x
468 7x
469 7x
470 7x
471 7x
472 7x
473 7x
474 7x
475 7x
476 7x
477 7x
478 7x
479 7x
480 7x
481 7x
482 7x
483 7x
484 7x
485 7x
486 7x
487 7x
488 7x
489 7x
490 7x
491 7x
492 7x
493 7x
494 7x
495 7x
496 7x
497 7x
498 7x
499 7x
500 7x
501 7x
502 7x
503 7x
504 7x
505 7x
506 7x
507 7x
508 7x
509 7x
510 7x
511 7x
512 7x
513 7x
514 7x
515 7x
516 7x
517 7x
518 7x
519 7x
520 7x
521 7x
522 7x
523 7x
524 7x
525 7x
526 7x
527 7x
528 7x
529 7x
530 7x
531 7x
532 7x
533 7x
534 7x
535 7x
536 7x
537 7x
538 7x
539 7x
540 7x
541 7x
542 7x
543 7x
544 7x
545 7x
546 7x
547 7x
548 7x
549 7x
550 7x
551 7x
552 7x
553 7x
554 7x
555 7x
556 7x
557 7x
558 7x
559 7x
560 7x
561 7x
562 7x
563 7x
564 7x
565 7x
566 7x
567 7x
568 7x
569 7x
570 7x
571 7x
572 7x
573 7x
574 7x
575 7x
576 7x
577 7x
578 7x
579 7x
580 7x
581 7x
582 7x
583 7x
584 7x
585 7x
586 7x
587 7x
588 7x
589 7x
590 7x
591 7x
592 7x
593 7x
594 7x
595 7x
596 7x
597 7x
598 7x
599 7x
600 7x
601 7x
602 7x
603 7x
604 7x
605 7x
606 7x
607 7x
608 7x
609 7x
610 7x
611 7x
612 7x
613 7x
614 7x
615 7x
616 7x
617 7x
618 7x
619 7x
620 7x
621 7x
622 7x
623 7x
624 7x
625 7x
626 7x
627 7x
628 7x
629 7x
630 7x
631 7x
632 7x
633 7x
634 7x
635 7x
636 7x
637 7x
638 7x
639 7x
640 7x
641 7x
642 7x
643 7x
644 7x
645 7x
646 7x
647 7x
648 7x
649 7x
650 7x
651 7x
652 7x
653 7x
654 7x
655 7x
656 7x
657 7x
658 7x
659 7x
660 7x
661 7x
662 7x
663 7x
664 7x
665 7x
666 7x
667 7x
668 7x
669 7x
670 7x
671 7x
672 7x
673 7x
674 7x
675 7x
676 7x
677 7x
678 7x
679 7x
680 7x
681 7x
682 7x
683 7x
684 7x
685 7x
686 7x
687 7x
688 7x
689 7x
690 7x
691 7x
692 7x
693 7x
694 7x
695 7x
696 7x
697 7x
698 7x
699 7x
700 7x
701 7x
702 7x
703 7x
704 7x
705 7x
706 7x
707 7x
708 7x
709 7x
710 7x
711 7x
712 7x
713 7x
714 7x
715 7x
716 7x
717 7x
718 7x
719 7x
720 7x
721 7x
722 7x
723 7x
724 7x
725 7x
726 7x
727 7x
728 7x
729 7x
730 7x
731 7x
732 7x
733 7x
734 7x
735 7x
736 7x
737 7x
738 7x
739 7x
740 7x
741 7x
742 7x
743 7x
744 7x
745 7x
746 7x
747 7x
748 7x
749 7x
750 7x
751 7x
752 7x
753 7x
754 7x
755 7x
756 7x
757 7x
758 7x
759 7x
760 7x
761 7x
762 7x
763 7x
764 7x
765 7x
766 7x
767 7x
768 7x
769 7x
770 7x
771 7x
772 7x
773 7x
774 7x
775 7x
776 7x
777 7x
778 7x
779 7x
780 7x
781 7x
782 7x
783 7x
784 7x
785 7x
786 7x
787 7x
788 7x
789 7x
790 7x
791 7x
792 7x
793 7x
794 7x
795 7x
796 7x
797 7x
798 7x
799 7x
800 7x
801 7x
802 7x
803 7x
804 7x
805 7x
806 7x
807 7x
808 7x
809 7x
810 7x
811 7x
812 7x
813 7x
814 7x
815 7x
816 7x
817 7x
818 7x
819 7x
820 7x
821 7x
822 7x
823 7x
824 7x
825 7x
826 7x
827 7x
828 7x
829 7x
830 7x
831 7x
832 7x
833 7x
834 7x
835 7x
836 7x
837 7x
838 7x
839 7x
840 7x
841 7x
842 7x
843 7x
844 7x
845 7x
846 7x
847 7x
848 7x
849 7x
850 7x
851 7x
852 7x
853 7x
854 7x
855 7x
856 7x
857 7x
858 7x
859 7x
860 7x
861 7x
862 7x
863 7x
864 7x
865 7x
866 7x
867 7x
868 7x
869 7x
870 7x
871 7x
872 7x
873 7x
874 7x
875 7x
876 7x
877 7x
878 7x
879 7x
880 7x
881 7x
882 7x
883 7x
884 7x
885 7x
886 7x
887 7x
888 7x
889 7x
890 7x
891 7x
892 7x
893 7x
894 7x
895 7x
896 7x
897 7x
898 7x
899 7x
900 7x
901 7x
902 7x
903 7x
904 7x
905 7x
906 7x
907 7x
908 7x
909 7x
910 7x
911 7x
912 7x
913 7x
914 7x
915 7x
916 7x
917 7x
918 7x
919 7x
920 7x
921 7x
922 7x
923 7x
924 7x
925 7x
926 7x
927 7x
928 7x
929 7x
930 7x
931 7x
932 7x
933 7x
934 7x
935 7x
936 7x
937 7x
938 7x
939 7x
940 7x
941 7x
942 7x
943 7x
944 7x
945 7x
946 7x
947 7x
948 7x
949 7x
950 7x
951 7x
952 7x
953 7x
954 7x
955 7x
956 7x
957 7x
958 7x
959 7x
960 7x
961 7x
962 7x
963 7x
964 7x
965 7x
966 7x
967 7x
968 7x
969 7x
970 7x
971 7x
972 7x
973 7x
974 7x
975 7x
976 7x
977 7x
978 7x
979 7x
980 7x
981 7x
982 7x
983 7x
984 7x
985 7x
986 7x
987 7x
988 7x
989 7x
990 7x
991 7x
992 7x
993 7x
994 7x
995 7x
996 7x
997 7x
998 7x
999 7x
1000 7x
1001 7x
1002 7x
1003 7x
1004 7x
1005 7x
1006 7x
1007 7x
1008 7x
1009 7x
1010 7x
1011 7x
1012 7x
1013 7x
1014 7x
1015 7x
1016 7x
1017 7x
1018 7x
1019 7x
1020 7x
1021 7x
1022 7x
1023 7x
1024 7x
1025 7x
1026 7x
1027 7x
1028 7x
1029 7x
1030 7x
1031 7x
1032 7x
1033 7x
1034 7x
1035 7x
1036 7x
1037 7x
1038 7x
1039 7x
1040 7x
1041 7x
1042 7x
1043 7x
1044 7x
1045 7x
1046 7x
1047 7x
1048 7x
1049 7x
1050 7x
1051 7x
1052 7x
1053 7x
1054 7x
1055 7x
1056 7x
1057 7x
1058 7x
1059 7x
1060 7x
1061 7x
1062 7x
1063 7x
1064 7x
1065 7x
1066 7x
1067 7x
1068 7x
1069 7x
1070 7x
1071 7x
1072 7x
1073 7x
1074 7x
1075 7x
1076 7x
1077 7x
1078 7x
1079 7x
1080 7x
1081 7x
1082 7x
1083 7x
1084 7x
1085 7x
1086 7x
1087 7x
1088 7x
1089 7x
1090 7x
1091 7x
1092 7x
1093 7x
1094 7x
1095 7x
1096 7x
1097 7x
1098 7x
1099 7x
1100 7x
1101 7x
1102 7x
1103 7x
1104 7x
1105 7x
1106 7x
1107 7x
1108 7x
1109 7x
1110 7x
1111 7x
1112 7x
1113 7x
1114 7x
1115 7x
1116 7x
1117 7x
1118 7x
1119 7x
1120 7x
1121 7x
1122 7x
1123 7x
1124 7x
1125 7x
1126 7x
1127 7x
1128 7x
1129 7x
1130 7x
1131 7x
1132 7x
1133 7x
1134 7x
1135 7x
1136 7x
1137 7x
1138 7x
1139 7x
1140 7x
1141 7x
1142 7x
1143 7x
1144 7x
1145 7x
1146 7x
1147 7x
1148 7x
1149 7x
1150 7x
1151 7x
1152 7x
1153 7x
1154 7x
1155 7x
1156 7x
1157 7x
1158 7x
1159 7x
1160 7x
1161 7x
1162 7x
1163 7x
1164 7x
1165 7x
1166 7x
1167 7x
1168 7x
1169 7x
1170 7x
1171 7x
1172 7x
1173 7x
1174 7x
1175 7x
1176 7x
1177 7x
1178 7x
1179 7x
1180 7x
1181 7x
1182 7x
1183 7x
1184 7x
1185 7x
1186 7x
1187 7x
1188 7x
1189 7x
1190 7x
1191 7x
1192 7x
1193 7x
1194 7x
1195 7x
1196 7x
1197 7x
1198 7x
1199 7x
1200 7x
1201 7x
1202 7x
1203 7x
1204 7x
1205 7x
1206 7x
1207 7x
1208 7x
1209 7x
1210 7x
1211 7x
1212 7x
1213 7x
1214 7x
1215 7x
1216 7x
1217 7x
1218 7x
1219 7x
1220 7x
1221 7x
1222 7x
1223 7x
1224 7x
1225 7x
1226 7x
1227 7x
1228 7x
1229 7x
1230 7x
1231 7x
1232 7x
1233 7x
1234 7x
1235 7x
1236 7x
1237 7x
1238 7x
1239 7x
1240 7x
1241 7x
1242 7x
1243 7x
1244 7x
1245 7x
1246 7x
1247 7x
1248 7x
1249 7x
1250 7x
1251 7x
1252 7x
1253 7x
1254 7x
1255 7x
1256 7x
1257 7x
1258 7x
1259 7x
1260 7x
1261 7x
1262 7x
1263 7x
1264 7x
1265 7x
1266 7x
1267 7x
1268 7x
1269 7x
1270 7x
1271 7x
1272 7x
1273 7x
1274 7x
1275 7x
1276 7x
1277 7x
1278 7x
1279 7x
1280 7x
1281 7x
1282 7x
1283 7x
1284 7x
1285 7x
1286 7x
1287 7x
1288 7x
1289 7x
1290 7x
1291 7x
1292 7x
1293 7x
1294 7x
1295 7x
1296 7x
1297 7x
1298 7x
1299 7x
1300 7x
1301 7x
1302 7x
1303 7x
1304 7x
1305 7x
1306 7x
1307 7x
1308 7x
1309 7x
1310 7x
1311 7x
1312 7x
1313 7x
1314 7x
1315 7x
1316 7x
1317 7x
1318 7x
1319 7x
1320 7x
1321 7x
1322 7x
1323 7x
1324 7x
1325 7x
1326 7x
1327 7x
1328 7x
1329 7x
1330 7x
1331 7x
1332 7x
1333 7x
1334 7x
1335 7x
1336 7x
1337 7x
1338 7x
1339 7x
1340 7x
1341 7x
```

but does not render its URL or number of likes by default.

Add CSS classes to the component to help the testing as necessary.

5.14: Blog List Tests, step 2

Make a test, which checks that the blog's URL and number of likes are shown when the button controlling the shown details has been clicked.

5.15: Blog List Tests, step 3

Make a test, which ensures that if the *like* button is clicked twice, the event handler the component received as props is called twice.

5.16: Blog List Tests, step 4

Make a test for the new blog form. The test should check, that the form calls the event handler it received as props with the right details when a new blog is created.

Frontend integration tests

In the previous part of the course material, we wrote integration tests for the backend that tested its logic and connected the database through the API provided by the backend. When writing these tests, we made the conscious decision not to write unit tests, as the code for that backend is fairly simple, and it is likely that bugs in our application occur in more complicated scenarios than unit tests are well suited for.

So far all of our tests for the frontend have been unit tests that have validated the correct functioning of individual components. Unit testing is useful at times, but even a comprehensive suite of unit tests is not enough to validate that the application works as a whole.

We could also make integration tests for the frontend. Integration testing tests the collaboration of multiple components. It is considerably more difficult than unit testing, as we would have to for example mock data from the server. We chose to concentrate on making end-to-end tests to test the whole application. We will work on the end-to-end tests in the last chapter of this part.

Snapshot testing

Vitest offers a completely different alternative to "traditional" testing called [snapshot](#) testing. The interesting feature of snapshot testing is that developers do not need to define any tests themselves, it is simple enough to adopt snapshot testing.

The fundamental principle is to compare the HTML code defined by the component after it has changed to the HTML code that existed before it was changed.

If the snapshot notices some change in the HTML defined by the component, then either it is new functionality or a "bug" caused by accident. Snapshot tests notify the developer if the HTML code of the component changes. The developer has to tell Vitest if the change was desired or undesired. If the change to the HTML code is unexpected, it strongly implies a bug, and the developer can become aware of these potential issues easily thanks to snapshot testing.

5d: End to end testing: Playwright

So far we have tested the backend as a whole on an API level using integration tests and tested some frontend components using unit tests.

Next, we will look into one way to test the [system as a whole](#) using *End to End* (E2E) tests.

We can do E2E testing of a web application using a browser and a testing library. There are multiple libraries available. One example is [Selenium](#), which can be used with almost any browser. Another browser option is so-called [headless browsers](#), which are browsers with no graphical user interface. For example, Chrome can be used in headless mode.

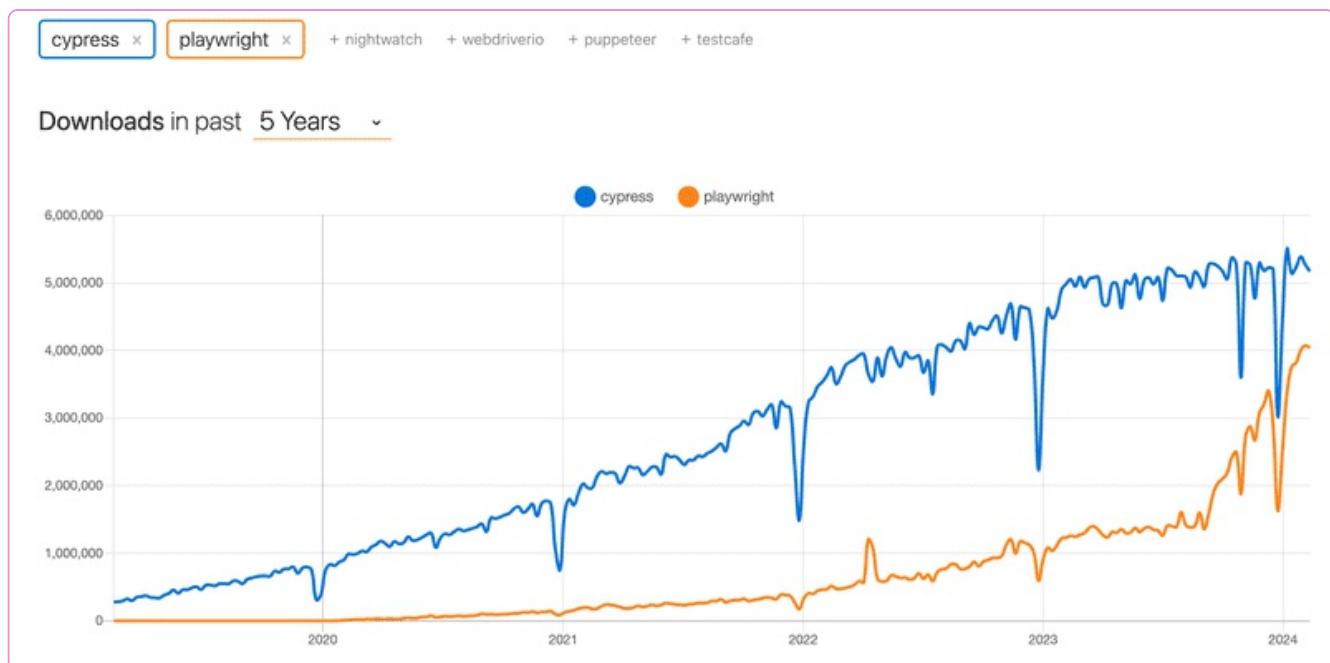
E2E tests are potentially the most useful category of tests because they test the system through the same interface as real users use.

They do have some drawbacks too. Configuring E2E tests is more challenging than unit or integration tests. They also tend to be quite slow, and with a large system, their execution time can be minutes or even hours. This is bad for development because during coding it is beneficial to be able to run tests as often as possible in case of code [regressions](#).

E2E tests can also be [flaky](#). Some tests might pass one time and fail another, even if the code does not change at all.

Perhaps the two easiest libraries for End to End testing at the moment are [Cypress](#) and [Playwright](#).

From the statistics on [npmtrtrends.com](#) we see that Cypress, which dominated the market for the last five years, is still clearly the number one, but Playwright is on a rapid rise:



This course has been using Cypress for years. Now Playwright is a new addition. You can choose whether to complete the E2E testing part of the course with Cypress or Playwright. The operating principles of both libraries are very similar, so your choice is not very important. However, Playwright is now the preferred E2E library for the course.

If your choice is Playwright, please proceed. If you end up using Cypress, go [here](#).

Playwright

So [Playwright](#) is a newcomer to the End to End tests, which started to explode in popularity towards the end of 2023. Playwright is roughly on a par with Cypress in terms of ease of use. The libraries are slightly different in terms of how they work. Cypress is radically different from most libraries suitable for E2E testing, as Cypress tests are run entirely within the browser. Playwright's tests, on the other hand, are executed in the Node process, which is connected to the browser via programming interfaces.

Many blogs have been written about library comparisons, e.g. [this](#) and [this](#).

It is difficult to say which library is better. One advantage of Playwright is its browser support; Playwright supports Chrome, Firefox and Webkit-based browsers like Safari. Currently, Cypress includes support for all these browsers, although Webkit support is experimental and does not support all of Cypress features. At the time of writing (1.3.2024), my personal preference leans slightly towards Playwright.

Now let's explore Playwright.

Initializing tests

Unlike the backend tests or unit tests done on the React front-end, End to End tests do not need to be located in the same npm project where the code is. Let's make a completely separate project for the E2E tests with the `npm init` command. Then install Playwright by running in the new project directory the command:

```
npm init playwright@latest
```

The installation script will ask a few questions, answer them as follows:

```
→ e2e npm init playwright@latest
Getting started with writing end-to-end tests with Playwright:
Initializing project in '..'
✓ Do you want to use TypeScript or JavaScript? · JavaScript
✓ Where to put your end-to-end tests? · tests
✓ Add a GitHub Actions workflow? (y/N) · false
✓ Install Playwright browsers (can be done manually via 'npx playwright install')? (Y/n) · true
Installing Playwright Test (npm install --save-dev @playwright/test)...
```

Let's define an npm script for running tests and test reports in `package.json`:

```
{
  // ...
  "scripts": {
    "test": "playwright test",
    "test:report": "playwright show-report"
  },
  // ...
}
```

During installation, the following is printed to the console:

```
And check out the following files:
- ./tests/example.spec.js - Example end-to-end test
```

```
- ./tests-examples/demo-todo-app.spec.js - Demo Todo App end-to-end tests
- ./playwright.config.js - Playwright Test configuration
```

that is, the location of a few example tests for the project that the installation has created.

Let's run the tests:

```
$ npm test

> notes-e2e@1.0.0 test
> playwright test

Running 6 tests using 5 workers
6 passed [3.9s]

To open last HTML report run:

npx playwright show-report
```

The tests pass. A more detailed test report can be opened either with the command suggested by the output, or with the npm script we just defined:

```
npm run test:report
```

Tests can also be run via the graphical UI with the command:

```
npm run test -- --ui
```

Sample tests look like this:

```
const { test, expect } = require('@playwright/test');

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Expect a title "to contain" a substring.
  await expect(page).toHaveTitle(/Playwright/);
});

test('get started link', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Click the get started link.
  await page.getByRole('link', { name: 'Get started' }).click();
  // Expects page to have a heading with the name of Installation.
  await expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
});
```

The first line of the test functions says that the tests are testing the page at <https://playwright.dev/>.

Testing our own code

Now let's remove the sample tests and start testing our own application.

Playwright tests assume that the system under test is running when the tests are executed. Unlike, for example, backend integration tests, Playwright tests *do not start* the system under test during testing.

Let's make an npm script for the *backend*, which will enable it to be started in testing mode, i.e. so that `NODE_ENV` gets the value `test`.

Let's start the frontend and backend, and create the first test file for the application tests/note app.spec.js:

First, the test opens the application with the method `page.goto`. After this, it uses the `page.getText` to get a `locator` that corresponds to the element where the text `Notes` is found.

The method `toBeVisible` ensures that the element corresponding to the locator is visible at the page.

The second check is done without using the auxiliary variable.

We notice that the year has changed. Let's change the test as follows:

```

const { test, expect } = require('@playwright/test')

test('front page can be opened', async ({ page }) => {
  await page.goto('http://localhost:5173')

  const locator = await page.getByText('Notes')
  await expect(locator).toBeVisible()
  await expect(page.getByText('Note app, Department of Computer Science, University of Helsinki 2024')).toBeVisible()
})

```

As expected, the test fails. Playwright opens the test report in the browser and it becomes clear that Playwright has actually performed the tests with three different browsers: Chrome, Firefox and Webkit, i.e. the browser engine used by Safari:

The screenshot shows a test report interface. At the top, there's a search bar, a status summary (All 3, Passed 0, Failed 3), and a date (21/02/2022). Below this, a tree view shows a single test file: note_app.spec.js. Under this file, three test cases are listed, each marked as failed (indicated by a red 'X'). Each failed test is associated with a browser engine: chromium, firefox, and webkit. The test cases are: 'front page can be opened'.

By clicking on the report of one of the browsers, we can see a more detailed error message:

The screenshot shows a detailed error report for the 'front page can be opened' test in the chromium browser. The title of the report is 'front page can be opened'. It includes the file path (note_app.spec.js:3) and the duration (5.3s). There are two buttons: 'Run' (disabled) and 'chromium' (highlighted). The 'Errors' section is expanded, showing the following error message:

```

Error: Timed out 5000ms waiting for expect(locator).toBeVisible()

Locator: getByText('Note app, Department of Computer Science, University of Helsinki 2024')
Expected: visible
Received: hidden
Call log:
- expect.toBeVisible with timeout 5000ms
- waiting for getByText('Note app, Department of Computer Science, University of Helsinki 2024')

6 |   const locator = await page.getByText('Notes')
7 |   await expect(locator).toBeVisible()
> 8 |   await expect(page.getByText('Note app, Department of Computer Science, University of Helsinki 2024')).toBeVisible()
9 |

```

at /Users/mluukkai/opetus/2024-fs/part3/e2e/tests/note_app.spec.js:8:105

In the big picture, it is of course a very good thing that the testing takes place with all three

commonly used browser engines, but this is slow, and when developing the tests it is probably best to carry them out mainly with only one browser. You can define the browser engine to be used with the command line parameter:

```
npm test -- --project chromium
```

Now let's correct the outdated year in the frontend code that caused the error.

Before we continue, let's add a *describe* block to the tests:

```
const { test, describe, expect } = require('@playwright/test')

describe('Note app', () => {
  test('front page can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    const locator = await page.getByText('Notes')
    await expect(locator).toBeVisible()
    await expect(page.getByText('Note app, Department of Computer Science, University of Helsinki 2024')).toBeVisible()
  })
})
```

Before we move on, let's break the tests one more time. We notice that the execution of the tests is quite fast when they pass, but much slower if they do not pass. The reason for this is that Playwright's policy is to wait for searched elements until [they are rendered and ready for action](#). If the element is not found, a *TimeoutError* is raised and the test fails. Playwright waits for elements by default for 5 or 30 seconds [depending on the functions used in testing](#).

When developing tests, it may be wiser to reduce the waiting time to a few seconds. According to the [documentation](#), this can be done by changing the file *playwright.config.js* as follows:

```
module.exports = defineConfig({
  timeout: 3000,
  fullyParallel: false, workers: 1, // ...
})
```

We also made two other changes to the file, and specified that all tests [be executed one at a time](#). With the default configuration, the execution happens in parallel, and since our tests use a database, parallel execution causes problems.

Writing on the form

Let's write a new test that tries to log into the application. Let's assume that a user is stored in the database, with username *mluukkai* and password *salainen*.

Let's start by opening the login form.

```
describe('Note app', () => {
  // ...
```

```

test('login form can be opened', async ({ page }) => {
  await page.goto('http://localhost:5173')

  await page.getByRole('button', { name: 'log in' }).click()
})
})

```

The test first uses the method `page.getByRole` to retrieve the button based on its text. The method returns the `Locator` corresponding to the Button element. Pressing the button is performed using the Locator method `click()`.

When developing tests, you could use Playwright's [UI mode](#), i.e. the user interface version. Let's start the tests in UI mode as follows:

```
npm test -- --ui
```

We now see that the test finds the button

The screenshot shows the Playwright UI mode interface. On the left, a tree view displays a test file `note_app.spec.js` with a single test case `login form can be opened` that has passed. The timeline at the top shows the execution flow from 0ms to 350ms. The main area shows a browser preview of a "Notes" application. The browser window title is "Notes" and the URL is "http://localhost:5173". The page content includes a "Show important" button, a note about browser capabilities, and a footer with "Note app, Department of Computer Science, University of Helsinki 2024". Below the browser preview, there is a code editor showing the test script. The line `await page.getByRole('button', { name: 'log in' }).click()` is highlighted in blue, indicating it is currently being executed.

```

note_app.spec.js
8   await expect(locator).toContainText('Notes')
9   await expect(locator).toContainText('Note app, Department of Computer Science, Univers
10  })
11
12 test('login form can be opened', async ({ page }) => {
13   await page.goto('http://localhost:5173')
14
15   await page.getByRole('button', { name: 'log in' }).click()
16 }
17 )

```

After clicking, the form will appear

The screenshot shows a browser window with a test runner interface on the left and a browser view on the right. The browser view displays a login page for 'Note app'. The page has a 'username' field, a 'password' field, a 'login' button, a 'cancel' button, and a 'show important' link. Below the form, there is a note from 'Note app, Department of Computer Science, University of Helsinki 2024'.

When the form is opened, the test should look for the text fields and enter the username and password in them. Let's make the first attempt using the method `page.getByRole`:

```
describe('Note app', () => {
  // ...

  test('login form can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByRole('textbox').fill('mluukkai')
  })
})
```

This results to an error:

```
Error: locator.fill: Error: strict mode violation: getByRole('textbox') resolved
to 2 elements:
  1) <input value="" /> aka locator('div').filter({ hasText: /^username$/ }).getB
yRole('textbox')
  2) <input value="" type="password" /> aka locator('input[type="password"]')
```

The problem now is that `getByRole` finds two text fields, and calling the `fill` method fails, because it assumes that there is only one text field found. One way around the problem is to use the methods `first` and `last`:

```
describe('Note app', () => {
  // ...

  test('login form can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByRole('textbox').first().fill('mluukkai')      await page.getBy
Role('textbox').last().fill('salainen')      await page.getByRole('button', { name: 'log in' }).click()
  })
})
```

```
'login' }).click()      await expect(page.getText('Matti Luukkainen logged in')) .toBeVisible() })
})
```

After writing in the text fields, the test presses the *login* button and checks that the application renders the logged-in user's information on the screen.

If there were more than two text fields, using the methods *first* and *last* would not be enough. One possibility would be to use the *all* method, which turns the found locators into an array that can be indexed:

```
describe('Note app', () => {
  // ...
  test('login form can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    await page.getByRole('button', { name: 'log in' }).click()
    const textboxes = await page.getByRole('textbox').all()    await textboxes[0]
.fill('mluukkai')    await textboxes[1].fill('salainen')
    await page.getByRole('button', { name: 'login' }).click()

    await expect(page.getText('Matti Luukkainen logged in')).toBeVisible()
  })
})
```

Both this and the previous version of the test work. However, both are problematic to the extent that if the registration form is changed, the tests may break, as they rely on the fields to be on the page in a certain order.

A better solution is to define unique test id attributes for the fields, to search for them in the tests using the method *getByTestId*.

Let's expand the login form as follows

```
const LoginForm = ({ ... }) => {
  return [
    <div>
      <h2>Login</h2>
      <form onSubmit={handleSubmit}>
        <div>
          username
          <input
            data-testid='username'
            value={username}
            onChange={handleUsernameChange}
          />
        </div>
        <div>
          password
          <input
            data-testid='password'
            type="password"
          />
        </div>
      </form>
    </div>
  ]
}
```

```

        value={password}
        onChange={handlePasswordChange}
      />
    </div>
    <button type="submit">
      login
    </button>
  </form>
</div>
)
}

```

Test changes as follows:

```

describe('Note app', () => {
  // ...

  test('login form can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    await page.getByRole('button', { name: 'log in' }).click()
    await page.getTestId('username').fill('mluukkai')    await page.getTestId('password').fill('salainen')
    await page.getByRole('button', { name: 'login' }).click()

    await expect(page.getByText('Matti Luukkainen logged in')).toBeVisible()
  })
})

```

Note that passing the test at this stage requires that there is a user in the *test* database of the backend with username *mluukkai* and password *salainen*. Create a user if needed!

Since both tests start in the same way, i.e. by opening the page <http://localhost:5173>, it is recommended to isolate the common part in the *beforeEach* block that is executed before each test:

```

const { test, describe, expect, beforeEach } = require('@playwright/test')

describe('Note app', () => {
  beforeEach(async ({ page }) => {    await page.goto('http://localhost:5173')  })

  test('front page can be opened', async ({ page }) => {
    const locator = await page.getByText('Notes')
    await expect(locator).toBeVisible()
    await expect(page.getByText('Note app, Department of Computer Science, University of Helsinki 2024')).toBeVisible()
  })

  test('login form can be opened', async ({ page }) => {
    await page.getByRole('button', { name: 'log in' }).click()
    await page.getTestId('username').fill('mluukkai')
  })
})

```

```
    await page.getByTestId('password').fill('salainen')
    await page.getByRole('button', { name: 'login' }).click()
    await expect(page.getText('Matti Luukkainen logged in')).toBeVisible()
  })
})
```

Testing note creation

Next, let's create a test that adds a new note to the application:

```
const { test, describe, expect, beforeEach } = require('@playwright/test')

describe('Note app', () => {
  // ...

  describe('when logged in', () => {
    beforeEach(async ({ page }) => {
      await page.getByRole('button', { name: 'log in' }).click()
      await page.getByTestId('username').fill('mluukkai')
      await page.getByTestId('password').fill('salainen')
      await page.getByRole('button', { name: 'login' }).click()
    })

    test('a new note can be created', async ({ page }) => {
      await page.getByRole('button', { name: 'new note' }).click()
      await page.getByRole('textbox').fill('a note created by playwright')
      await page.getByRole('button', { name: 'save' }).click()
      await expect(page.getText('a note created by playwright')).toBeVisible()
    })
  })
})
```

The test is defined in its own *describe* block. Creating a note requires that the user is logged in, which is handled in the *beforeEach* block.

The test trusts that when creating a new note, there is only one input field on the page, so it searches for it as follows:

```
page.getByRole('textbox')
```

If there were more fields, the test would break. Because of this, it would be better to add a test-id to the form input and search for it in the test based on this id.

Note: the test will only pass the first time. The reason for this is that its expectation

```
await expect(page.getText('a note created by playwright')).toBeVisible()
```

causes problems when the same note is created in the application more than once. The problem will be solved in the next chapter.

The structure of the tests looks like this:

```
const { test, describe, expect, beforeEach } = require('@playwright/test')

describe('Note app', () => {
  // ...

  test('user can log in', async ({ page }) => {
    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByTestId('username').fill('mluukkai')
    await page.getByTestId('password').fill('salainen')
    await page.getByRole('button', { name: 'login' }).click()
    await expect(page.getText('Matti Luukkainen logged in')).toBeVisible()
  })

  describe('when logged in', () => {
    beforeEach(async ({ page }) => {
      await page.getByRole('button', { name: 'log in' }).click()
      await page.getByTestId('username').fill('mluukkai')
      await page.getByTestId('password').fill('salainen')
      await page.getByRole('button', { name: 'login' }).click()
    })

    test('a new note can be created', async ({ page }) => {
      await page.getByRole('button', { name: 'new note' }).click()
      await page.getByRole('textbox').fill('a note created by playwright')
      await page.getByRole('button', { name: 'save' }).click()
      await expect(page.getText('a note created by playwright')).toBeVisible()
    })
  })
})
```

Since we have prevented the tests from running in parallel, Playwright runs the tests in the order they appear in the test code. That is, first the test *user can log in*, where the user logs into the application, is performed. After this the test *a new note can be created* gets executed, which also does a log in, in the *beforeEach* block. Why is this done, isn't the user already logged in thanks to the previous test? No, because the execution of *each* test starts from the browser's "zero state", all changes made to the browser's state by the previous tests are reset.

Controlling the state of the database

If the tests need to be able to modify the server's database, the situation immediately becomes more complicated. Ideally, the server's database should be the same each time we run the tests, so our tests can be reliably and easily repeatable.

As with unit and integration tests, with E2E tests it is best to empty the database and possibly format it before the tests are run. The challenge with E2E tests is that they do not have access to the database.

The solution is to create API endpoints for the backend tests. We can empty the database using these endpoints. Let's create a new router for the tests inside the *controllers* folder, in the *testing.js* file

```

const router = require('express').Router()
const Note = require('../models/note')
const User = require('../models/user')

router.post('/reset', async (request, response) => {
  await Note.deleteMany({})
  await User.deleteMany({})

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

module.exports = router

```

and add it to the backend only *if the application is run in test-mode*:

```

// ...

app.use('/api/login', loginRouter)
app.use('/api/users', usersRouter)
app.use('/api/notes', notesRouter)

if (process.env.NODE_ENV === 'test') { const testingRouter = require('./controllers/testing')
  app.use('/api/testing', testingRouter)
}

app.use(middleware.unknownEndpoint)
app.use(middleware.errorHandler)

module.exports = app

```

After the changes, an HTTP POST request to the `/api/testing/reset` endpoint empties the database. Make sure your backend is running in test mode by starting it with this command (previously configured in the package.json file):

```
npm run start:test
```

The modified backend code can be found on the [GitHub](#) branch *part5-1*.

Next, we will change the `beforeEach` block so that it empties the server's database before tests are run.

Currently, it is not possible to add new users through the frontend's UI, so we add a new user to the backend from the `beforeEach` block.

```

describe('Note app', () => {
  beforeEach(async ({ page, request }) => {
    await request.post('http://localhost:3001/api/testing/reset')
    await request.post('http://localhost:3001/api/users', {
      data: {
        name: 'Matti Luukkainen',
        username: 'mluukkai',
        password: 'salainen'
      }
    })
  })
})

```

```

    }
})

await page.goto('http://localhost:5173')
})

test('front page can be opened', () => {
    // ...
})

test('user can login', () => {
    // ...
})

describe('when logged in', () => {
    // ...
})
})

```

During initialization, the test makes HTTP requests to the backend with the method `post` of the parameter `request`.

Unlike before, now the testing of the backend always starts from the same state, i.e. there is one user and no notes in the database.

Let's make a test that checks that the importance of the notes can be changed.

There are a few different approaches to taking the test.

In the following, we first look for a note and click on its button that has text *make not important*. After this, we check that the note contains the button with *make important*.

```

describe('Note app', () => {
    // ...

    describe('when logged in', () => {
        // ...

        describe('and a note exists', () => {           beforeEach(async ({ page }) => {
            await page.getByRole('button', { name: 'new note' }).click()           await
            page.getByRole('textbox').fill('another note by playwright')           await page.
            getByRole('button', { name: 'save' }).click()           test('importance ca
            n be changed', async ({ page }) => {           await page.getByRole('button', { name
            : 'make not important' }).click()           await expect(page.getText('make impor
            tant')).toBeVisible()           })
        })
    })
})

```

The first command first searches for the component where there is the text *another note by playwright* and inside it the button *make not important* and clicks on it.

The second command ensures that the text of the same button has changed to *make important*.

The current code for the tests is on [GitHub](#), in branch *part5-1*.

Test for failed login

Now let's do a test that ensures that the login attempt fails if the password is wrong.

The first version of the test looks like this:

```
describe('Note app', () => {
  // ...

  test('login fails with wrong password', async ({ page }) => {
    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByTestId('username').fill('mluukkai')
    await page.getByTestId('password').fill('wrong')
    await page.getByRole('button', { name: 'login' }).click()

    await expect(page.getText('wrong credentials')).toBeVisible()
  })
  // ...
})
```

The test verifies with the method `page.getText` that the application prints an error message.

The application renders the error message to an element containing the CSS class *error*:

```
const Notification = ({ message }) => {
  if (message === null) {
    return null
  }

  return (
    <div className="error">      {message}
    </div>
  )
}
```

We could refine the test to ensure that the error message is printed exactly in the right place, i.e. in the element containing the CSS class *error*:

```
test('login fails with wrong password', async ({ page }) => {
  // ...

  const errorDiv = await page.locator('.error')  await expect(errorDiv).toContainText('wrong credentials')
})
```

So the test uses the `page.locator` method to find the component containing the CSS class *error* and

stores it in a variable. The correctness of the text associated with the component can be verified with the expectation `toContainText`. Note that the `CSS class selector` starts with a dot, so the `error` class selector is `.error`.

It is possible to test the application's CSS styles with matcher `toHaveCSS`. We can, for example, make sure that the color of the error message is red, and that there is a border around it:

```
test('login fails with wrong password', async ({ page }) => {
  // ...

  const errorDiv = await page.locator('.error')
  await expect(errorDiv).toContainText('wrong credentials')
  await expect(errorDiv).toHaveCSS('border-style', 'solid')    await expect(errorDiv).toHaveCSS('color', 'rgb(255, 0, 0)')}
```

Colors must be defined to Playwright as `rgb` codes.

Let's finalize the test so that it also ensures that the application **does not render** the text describing a successful login '*Matti Luukkainen logged in*':

```
test('login fails with wrong password', async ({ page }) =>{
  await page.getByRole('button', { name: 'log in' }).click()
  await page.getByTestId('username').fill('mluukkai')
  await page.getByTestId('password').fill('wrong')
  await page.getByRole('button', { name: 'login' }).click()

  const errorDiv = await page.locator('.error')
  await expect(errorDiv).toContainText('wrong credentials')
  await expect(errorDiv).toHaveCSS('border-style', 'solid')
  await expect(errorDiv).toHaveCSS('color', 'rgb(255, 0, 0)')

  await expect(page.getText('Matti Luukkainen logged in')).not.toBeVisible()}
```

Running tests one by one

By default, Playwright always runs all tests, and as the number of tests increases, it becomes time-consuming. When developing a new test or debugging a broken one, the test can be defined instead than with the command `test`, with the command `test.only`, in which case Playwright will run only that test:

```
describe([], () => {
  // this is the only test executed!
  test.only('login fails with wrong password', async ({ page }) => {    // ...
  })

  // this test is skipped...
  test('user can login with correct credentials', async ({ page }) => {
    // ...
  })
})
```

```
})
```

When the test is ready, *only* can and **should** be deleted.

Another option to run a single test is to use a command line parameter:

```
npm test -- -g "login fails with wrong password"
```

Helper functions for tests

Our application tests currently look like this:

```
const { test, describe, expect, beforeEach } = require('@playwright/test')

describe('Note app', () => {
    // ...

    test('user can login with correct credentials', async ({ page }) => {
        await page.getByRole('button', { name: 'log in' }).click()
        await page.getByTestId('username').fill('mluukkai')
        await page.getByTestId('password').fill('salainen')
        await page.getByRole('button', { name: 'login' }).click()
        await expect(page.getText('Matti Luukkainen logged in')).toBeVisible()
    })

    test('login fails with wrong password', async ({ page }) => {
        // ...
    })

    describe('when logged in', () => {
        beforeEach(async ({ page, request }) => {
            await page.getByRole('button', { name: 'log in' }).click()
            await page.getByTestId('username').fill('mluukkai')
            await page.getByTestId('password').fill('salainen')
            await page.getByRole('button', { name: 'login' }).click()
        })

        test('a new note can be created', async ({ page }) => {
            // ...
        })
    })
})
```

First, the login function is tested. After this, another *describe* block contains a set of tests that assume that the user is logged in, the login is handled inside the initializing *beforeEach* block.

As already stated earlier, each test is executed starting from the initial state (where the database is cleared and one user is created there), so even though the test is defined after another test in the code, it does not start from the same state where the tests in the code executed earlier have left!

It is also worth striving for having non-repetitive code in tests. Let's isolate the code that handles the login as a helper function, which is placed e.g. in the file `tests/helper.js`:

```
const loginWith = async [page, username, password] => {
  await page.getByRole('button', { name: 'log in' }).click()
  await page.getByTestId('username').fill(username)
  await page.getByTestId('password').fill(password)
  await page.getByRole('button', { name: 'login' }).click()
}

export { loginWith }
```

The test becomes simpler and clearer:

```
const { loginWith } = require('./helper')

describe('Note app', () => {
  test('user can log in', async ({ page }) => {
    await loginWith(page, 'mluukkai', 'salainen')      await expect(page.getText('Matti Luukkainen logged in')).toBeVisible()
  })

  describe('when logged in', () => {
    beforeEach(async ({ page }) => {
      await loginWith(page, 'mluukkai', 'salainen')
    })

    test('a new note can be created', () => {
      // ...
    })
  })
})
```

Playwright also offers a [solution](#) where the login is performed once before the tests, and each test starts from a state where the application is already logged in. In order for us to take advantage of this method, the initialization of the application's test data should be done a bit differently than now. In the current solution, the database is reset before each test, and because of this, logging in just once before the tests is impossible. In order for us to use the pre-test login provided by Playwright, the user should be initialized only once before the tests. We stick to our current solution for the sake of simplicity.

The corresponding repeating code actually also applies to creating a new note. For that, there is a test that creates a note using a form. Also in the `beforeEach` initialization block of the test that tests changing the importance of the note, a note is created using the form:

```
describe('Note app', function() {
  // ...

  describe('when logged in', () => {
    test('a new note can be created', async ({ page }) => {
```

```

        await page.getByRole('button', { name: 'new note' }).click()
        await page.getByRole('textbox').fill('a note created by playwright')
        await page.getByRole('button', { name: 'save' }).click()
        await expect(page.getText('a note created by playwright')).toBeVisible()
    })

describe('and a note exists', () => {
    beforeEach(async ({ page }) => {
        await page.getByRole('button', { name: 'new note' }).click()
        await page.getByRole('textbox').fill('another note by playwright')
        await page.getByRole('button', { name: 'save' }).click()
    })

    test('it can be made important', async ({ page }) => {
        // ...
    })
})
})
})
})

```

Creation of a note is also isolated as its helper function. The file `tests/helper.js` expands as follows:

```

const loginWith = async (page, username, password) => {
    await page.getByRole('button', { name: 'log in' }).click()
    await page.getTestId('username').fill(username)
    await page.getTestId('password').fill(password)
    await page.getByRole('button', { name: 'login' }).click()
}

const createNote = async (page, content) => {
    await page.getByRole('button', { name: 'new note' }).click()
    await page.getByRole('textbox').fill(content)
    await page.getByRole('button', { name: 'save' }).click()
}
export { loginWith, createNote }

```

The tests are simplified as follows:

```

describe('Note app', () => {
    // ...

    describe('when logged in', () => {
        beforeEach(async ({ page }) => {
            await loginWith(page, 'mluukkai', 'salainen')
        })

        test('a new note can be created', async ({ page }) => {
            await createNote(page, 'a note created by playwright', true)
            await expect(page.getText('a note created by playwright')).toBeVisible()
        })

        describe('and a note exists', () => {

```

```

beforeEach(async ([page]) => {
    await createNote(page, 'another note by playwright', true)
})

test('importance can be changed', async ([page]) => {
    await page.getByRole('button', { name: 'make not important' }).click()
    await expect(page.getText('make important')).toBeVisible()
})
})
})
})

```

There is one more annoying feature in our tests. The frontend address `http://localhost:5173` and the backend address `http://localhost:3001` are hardcoded for tests. Of these, the address of the backend is actually useless, because a proxy has been defined in the Vite configuration of the frontend, which forwards all requests made by the frontend to the address `http://localhost:5173/api` to the backend:

```

export default defineConfig({
  server: [
    proxy: [
      '/api': {
        target: 'http://localhost:3001',
        changeOrigin: true,
      },
    ],
    // ...
  ],
})

```

So we can replace all the addresses in the tests from `http://localhost:3001/api/...` to `http://localhost:5173/api/...`

We can now define the `baseUrl` for the application in the tests configuration file `playwright.config.js`:

```

module.exports = defineConfig([
  // ...
  use: [
    baseURL: 'http://localhost:5173',
  ],
  // ...
])

```

All the commands in the tests that use the application url, e.g.

```

await page.goto('http://localhost:5173')
await page.post('http://localhost:5173/api/tests/reset')

```

can be transformed into:

```

await page.goto('/')
await page.post('/api/tests/reset')

```

The current code for the tests is on [GitHub](#), branch *part5-2*.

Note importance change revisited

Let's take a look at the test we did earlier, which verifies that it is possible to change the importance of a note.

Let's change the initialization block of the test so that it creates two notes instead of one:

```
describe('when logged in', () => {
  // ...
  describe('and several notes exists', () => {
    beforeEach(async ({ page }) => {
      await createNote(page, 'first note', true)           await createNote(page, 'second note', true)
    })

    test('one of those can be made nonimportant', async ({ page }) => {
      const otherNoteElement = await page.getText('first note')

      await otherNoteElement
        .getByRole('button', { name: 'make not important' }).click()
      await expect(otherNoteElement.getText('make important')).toBeVisible()
    })
  })
})
```

The test first searches for the element corresponding to the first created note using the method `page.getText` and stores it in a variable. After this, a button with the text *make not important* is searched inside the element and the button is pressed. Finally, the test verifies that the button's text has changed to *make important*.

The test could also have been written without the auxiliary variable:

```
test('one of those can be made nonimportant', async ({ page }) => {
  await page.getText('first note')
    .getByRole('button', { name: 'make not important' }).click()

  await expect(page.getText('first note').getText('make important'))
    .toBeVisible()
})
```

Let's change the *Note* component so that the note text is rendered inside a *span* element

```
const Note = ({ note, toggleImportance }) => [
  const label = note.important
    ? 'make not important' : 'make important'

  return [
    <li className='note'>
      <span>{note.content}</span>      <button onClick={toggleImportance}>{label}</button>
  ]
]
```

```
</button>
    </li>
}
}
```

Tests break! The reason for the problem is that the command `await page.getText('second note')` now returns a `span` element containing only text, and the button is outside of it.

One way to fix the problem is as follows:

```
test('one of those can be made nonimportant', async ([page]) => {
  const otherNoteText = await page.getText('first note')  const otherNoteElement = await otherNoteText.locator('..')
  await otherNoteElement.getByRole('button', { name: 'make not important' }).click()
  await expect(otherNoteElement.getText('make important')).toBeVisible()
})
```

The first line now looks for the `span` element containing the text associated with the first created note. In the second line, the function `locator` is used and `..` is given as an argument, which retrieves the element's parent element. The locator function is very flexible, and we take advantage of the fact that accepts [as argument](#) not only CSS selectors but also [XPath](#) selector. It would be possible to express the same with CSS, but in this case XPath provides the simplest way to find the parent of an element.

Of course, the test can also be written using only one auxiliary variable:

```
test('one of those can be made nonimportant', async ([page]) => {
  const secondNoteElement = await page.getText('second note').locator('..')
  await secondNoteElement.getByRole('button', { name: 'make not important' }).click()
  await expect(secondNoteElement.getText('make important')).toBeVisible()
})
```

Let's change the test so that three notes are created, the importance is changed in the second created note:

```
describe('when logged in', () => {
  beforeEach(async ([page]) => {
    await loginWith(page, 'mluukkai', 'salainen')
  })

  test('a new note can be created', async ([page]) => {
    await createNote(page, 'a note created by playwright', true)
    await expect(page.getText('a note created by playwright')).toBeVisible()
  })

  describe('and a note exists', () => {
    beforeEach(async ([page]) => {
      await createNote(page, 'first note', true)
    })
```

```

        await createNote(page, 'second note', true)
        await createNote(page, 'third note', true)    })

test('importance can be changed', async ([ page ]) => [
  const otherNoteText = await page.getText('second note')           const other
dNoteElement = await otherNoteText.locator('..')

  await otherdNoteElement.getByRole('button', { name: 'make not important' }).click()
  await expect(otherdNoteElement.getByText('make important')).toBeVisible()
])
})
})
})
})

```

For some reason the test starts working unreliably, sometimes it passes and sometimes it doesn't. It's time to roll up your sleeves and learn how to debug tests.

Test development and debugging

If, and when the tests don't pass and you suspect that the fault is in the tests instead of in the code, you should run the tests in `debug` mode.

The following command runs the problematic test in debug mode:

```
npm test -- -g'importance can be changed' --debug
```

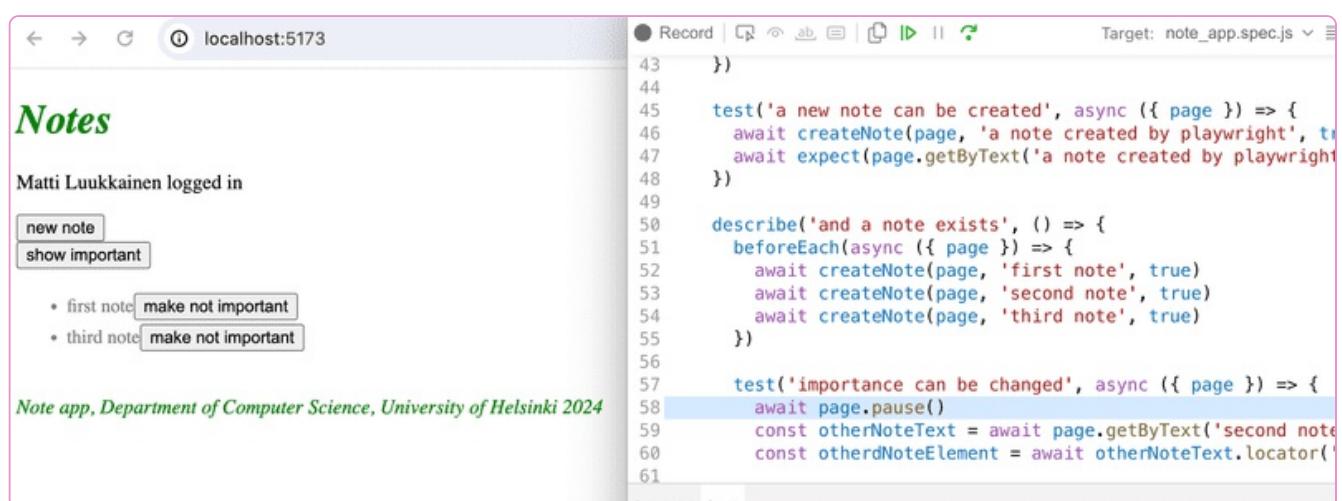
Playwright-inspector shows the progress of the tests step by step. The arrow-dot button at the top takes the tests one step further. The elements found by the locators and the interaction with the browser are visualized in the browser:

The screenshot shows a browser window at `localhost:5173` displaying a notes application. On the left, there's a sidebar with 'Notes' and a message 'Matti Luukkainen logged in'. Below it is a 'Create a new note' form with a 'save' button and a 'getByRole('textbox')' input field. A tooltip on the input field says 'first note make not important'. On the right, a 'Playwright Inspector' window is open, showing the source code of the test file. The line `await page.getByRole('textbox').fill(content)` is highlighted in blue, indicating the current step of the test execution. The inspector also shows 'Locator' and 'Log' tabs at the bottom.

By default, debug steps through the test command by command. If it is a complex test, it can be quite a burden to step through the test to the point of interest. This can be avoided by using the command `await page.pause()`:

Now in the test you can go to `page.pause()` in one step, by pressing the green arrow symbol in the inspector.

When we now run the test and jump to the `page.pause()` command, we find an interesting fact:



It seems that the browser *does not render* all the notes created in the block *beforeEach*. What is the

problem?

The reason for the problem is that when the test creates one note, it starts creating the next one even before the server has responded, and the added note is rendered on the screen. This in turn can cause some notes to be lost (in the picture, this happened to the second note created), since the browser is re-rendered when the server responds, based on the state of the notes at the start of that insert operation.

The problem can be solved by "slowing down" the insert operations by using the `waitFor` command after the insert to wait for the inserted note to render:

```
const createNote = async [page, content] => {
  await page.getByRole('button', { name: 'new note' }).click()
  await page.getByRole('textbox').fill(content)
  await page.getByRole('button', { name: 'save' }).click()
  await pageByText(content).waitFor()
```

Instead of, or alongside debugging mode, running tests in UI mode can be useful. As already mentioned, tests are started in UI mode as follows:

```
npm run test -- --ui
```

Almost the same as UI mode is use of the Playwright's [Trace Viewer](#). The idea is that a "visual trace" of the tests is saved, which can be viewed if necessary after the tests have been completed. A trace is saved by running the tests as follows:

```
npm run test -- --trace on
```

If necessary, Trace can be viewed with the command

```
npx playwright show-report
```

or with the npm script we defined `npm run test:report`

Trace looks practically the same as running tests in UI mode.

UI mode and Trace Viewer also offer the possibility of assisted search for locators. This is done by pressing the double circle on the left side of the lower bar, and then by clicking on the desired user interface element. Playwright displays the element locator:

Actions	Metadata	Action	Before	After
✓ Passed	1.5s			
✗ Before Hooks	1.3s			
> beforeEach hook	604ms			
> beforeEach hook	131ms			
> beforeEach hook	569ms			
locator.click getByText('seco...')	9ms			
expect.toBeVisible getByT...	106ms			
page.pause	0ms			
> After Hooks	38ms			

Notes

Matti Luukkainen logged in

new note
show important

- first note make not important
- second note make important
- third note make not important

locator('li').filter({ hasText: 'third notemake not important' }).getByRole('button')

Note app, Department of Computer Science, University of Helsinki 2024

Locator

```
locator('li').filter({ hasText: 'third notemake not important' }).getByRole('button')
```

Playwright suggests the following as the locator for the third note

```
page.locator('li').filter({ hasText: 'third note' }).getByRole('button')
```

The method `page.locator` is called with the argument `li`, i.e. we search for all `li` elements on the page, of which there are three in total. After this, using the `locator.filter` method, we narrow down to the `li` element that contains the text `third note` and the button element inside it is taken using the `locator.getByRole` method.

The locator generated by Playwright is somewhat different from the locator used by our tests, which was

```
page.getByText('first note').locator('..').getByRole('button', { name: 'make not important' })
```

Which of the locators is better is probably a matter of taste.

Playwright also includes a `test generator` that makes it possible to "record" a test through the user interface. The test generator is started with the command:

```
npx playwright codegen http://localhost:5173/
```

When the `Record` mode is on, the test generator "records" the user's interaction in the Playwright inspector, from where it is possible to copy the locators and actions to the tests:

The screenshot shows a browser window at localhost:5173 displaying a note-taking application. The page title is "Notes". A sidebar on the left contains a "new note" button and a list of notes with checkboxes: "first note" (unchecked), "second note" (checked), "third note" (unchecked), and "playwright is easy to use" (unchecked). Below the sidebar is a note content area with the text "Note app, Department of Computer Science, University of Helsinki 2024". To the right of the browser is the "Playwright Inspector" window, which has a "Record" tab active. It displays the following JavaScript test code:

```

1 import { test, expect } from '@playwright/test';
2
3 test('test', async ({ page }) => {
4     await page.goto('http://localhost:5173/');
5     await page.getByRole('button', { name: 'log in' }).click();
6     await page.getByTestId('username').click();
7     await page.getByTestId('username').fill('mluukkai');
8     await page.getByTestId('password').click();
9     await page.getByTestId('password').fill('salainen');
10    await page.getByTestId('login-button').click();
11    await page.getByRole('button', { name: 'new note' }).click();
12    await page.getByRole('textbox').click();
13    await page.getByRole('textbox').fill('playwright is easy to use');
14    await page.getByRole('button', { name: 'save' }).click();
15 });

```

The "Log" tab in the inspector is currently inactive.

Instead of the command line, Playwright can also be used via the [VS Code](#) plugin. The plugin offers many convenient features, e.g. use of breakpoints when debugging tests.

To avoid problem situations and increase understanding, it is definitely worth browsing Playwright's high-quality [documentation](#). The most important sections are listed below:

- the section about [locators](#) gives good hints for finding elements in test
- section [actions](#) tells how it is possible to simulate the interaction with the browser in tests
- the section about [assertions](#) demonstrates the different expectations Playwright offers for testing

In-depth details can be found in the [API](#) description, particularly useful are the class [Page](#) corresponding to the browser window of the application under test, and the class [Locator](#) corresponding to the elements searched for in the tests.

The final version of the tests is in full on [GitHub](#), in branch *part5-3*.

The final version of the frontend code is in its entirety on [GitHub](#), in branch *part5-9*.

Exercises 5.17.-5.23.

In the last exercises of this part, let's do some E2E tests for the blog application. The material above should be enough to do most of the exercises. However, you should definitely read Playwright's [documentation](#) and [API description](#), at least the sections mentioned at the end of the previous chapter.

5.17: Blog List End To End Testing, step 1

Create a new npm project for tests and configure Playwright there.

Make a test to ensure that the application displays the login form by default.

The body of the test should be as follows:

```
const { test, expect, beforeEach, describe } = require('@playwright/test')

describe('Blog app', () => {
  beforeEach(async ({ page }) => {
    await page.goto('http://localhost:5173')
  })

  test('Login form is shown', async ({ page }) => {
    // ...
  })
})
```

5.18: Blog List End To End Testing, step 2

Do the tests for login. Test both successful and failed login. For tests, create a user in the *beforeEach* block.

The body of the tests expands as follows

```
const { test, expect, beforeEach, describe } = require('@playwright/test')

describe('Blog app', () => {
  beforeEach(async ({ page, request }) => {
    // empty the db here
    // create a user for the backend here
    // ...
  })

  test('Login form is shown', async ({ page }) => {
    // ...
  })

  describe('Login', () => {
    test('succeeds with correct credentials', async ({ page }) => {
      // ...
    })

    test('fails with wrong credentials', async ({ page }) => {
      // ...
    })
  })
})
```

The *beforeEach* block must empty the database using, for example, the `reset` method we used in the [material](#).

5.19: Blog List End To End Testing, step 3

Create a test that verifies that a logged in user can create a blog. The body of the test may look like

the following

```
describe('When logged in', () => {
  beforeEach(async ({ page }) => {
    // ...
  })

  test('a new blog can be created', async ({ page }) => {
    // ...
  })
})
```

The test should ensure that the created blog is visible in the list of blogs.

5.20: Blog List End To End Testing, step 4

Do a test that makes sure the blog can be liked.

5.21: Blog List End To End Testing, step 5

Make a test that ensures that the user who added the blog can delete the blog. If you use the `window.confirm` dialog in the delete operation, you may have to Google how to use the dialog in the Playwright tests.

5.22: Blog List End To End Testing, step 6

Make a test that ensures that only the user who added the blog sees the blog's delete button.

5.23: Blog List End To End Testing, step 7

Do a test that ensures that the blogs are arranged in the order according to the likes, the blog with the most likes first.

This task is significantly more challenging than the previous ones.

This was the last task of the section and it's time to push the code to GitHub and mark the completed tasks in the [exercise submission system](#).

5e: End to end testing: Cypress

[Cypress](#) has been the most popular E2E testing library for the past few years, but Playwright is rapidly gaining ground. This course has been using Cypress for years. Now Playwright is a new addition. You can choose whether to complete the E2E testing part of the course with Cypress or Playwright. The operating principles of both libraries are very similar, so your choice is not very important. However, Playwright is now the preferred E2E library for the course.

If your choice is Cypress, please proceed. If you end up using Playwright, go [here](#).

Cypress

E2E library [Cypress](#) has become popular within the last years. Cypress is exceptionally easy to use,

and when compared to Selenium, for example, it requires a lot less hassle and headache. Its operating principle is radically different than most E2E testing libraries because Cypress tests are run completely within the browser. Other libraries run the tests in a Node process, which is connected to the browser through an API.

Let's make some end-to-end tests for our note application.

Unlike the backend tests or unit tests done on the React front-end, End to End tests do not need to be located in the same npm project where the code is. Let's make a completely separate project for the E2E tests with the *npm init* command. Then install Cypress to *the new project* as a development dependency

```
npm install --save-dev cypress
```

and by adding an npm-script to run it:

```
{
  // ...
  "scripts": {
    "cypress:open": "cypress open"
  }
}
```

We also made a small change to the script that starts the application, without the change Cypress can not access the app.

Unlike the frontend's unit tests, Cypress tests can be in the frontend or the backend repository, or even in their separate repository.

The tests require that the system being tested is running. Unlike our backend integration tests, Cypress tests *do not start* the system when they are run.

Let's add an npm script to *the backend* which starts it in test mode, or so that *NODE_ENV* is *test*.

```
{
  // ...
  "scripts": {
    "start": "NODE_ENV=production node index.js",
    "dev": "NODE_ENV=development nodemon index.js",
    "build:ui": "rm -rf build && cd .. frontend/ && npm run build && cp -r build
      .. backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs",
    "lint": "eslint .",
    "test": "jest --verbose --runInBand",
    "start:test": "NODE_ENV=test node index.js"
  }
}
```

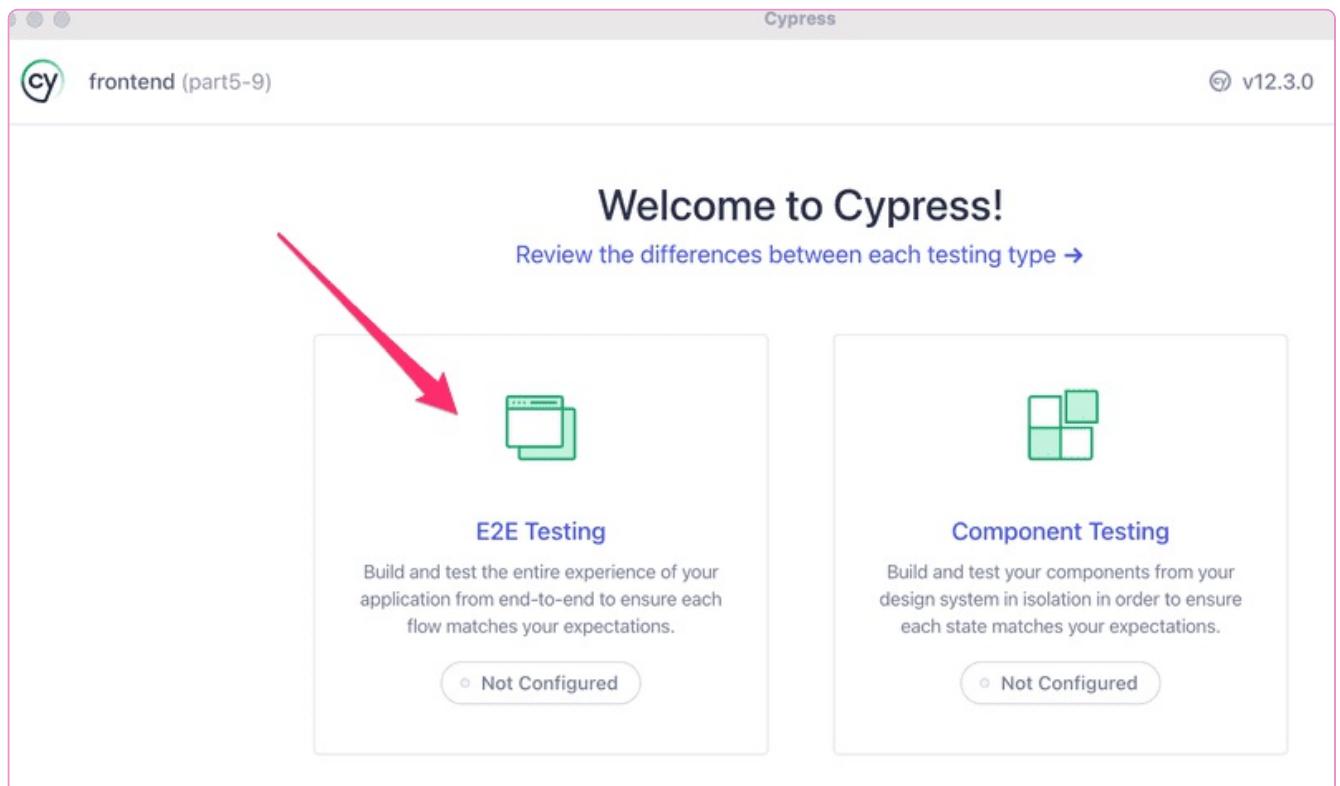
NB To get Cypress working with WSL2 one might need to do some additional configuring first. These

two [links](#) are great places to [start](#).

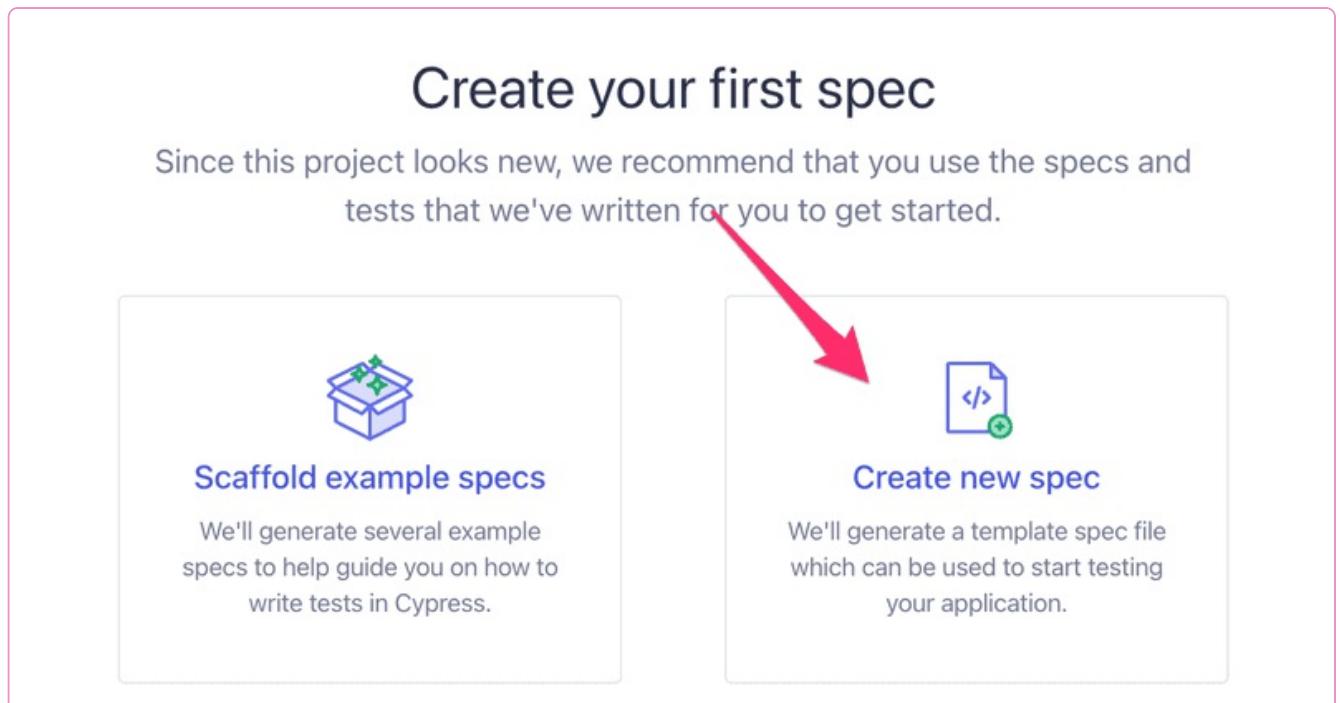
When both the backend and frontend are running, we can start Cypress with the command

```
npm run cypress:open
```

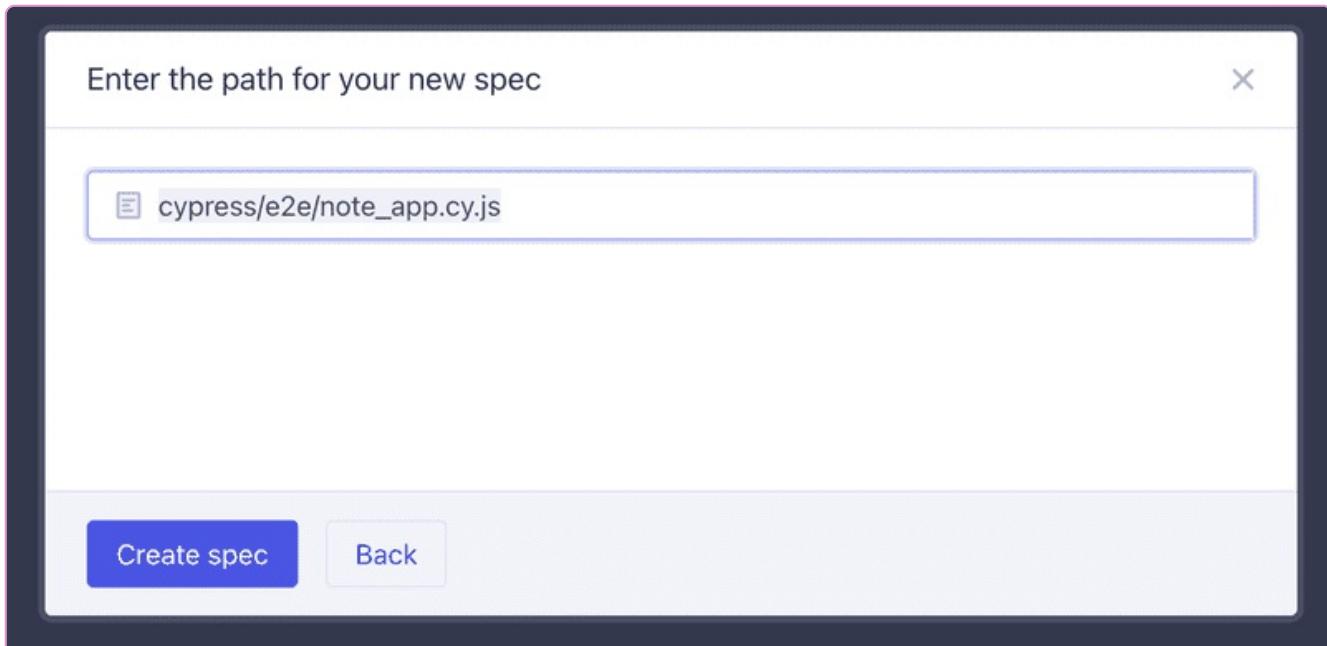
Cypress asks what type of tests we are doing. Let us answer "E2E Testing":



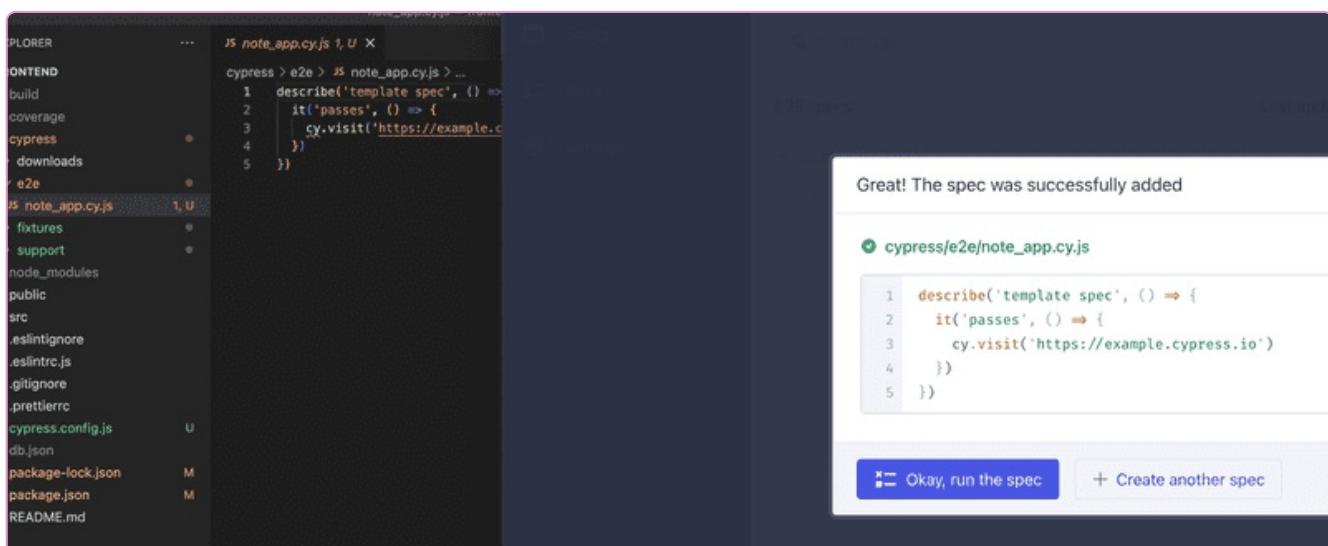
Next a browser is selected (e.g. Chrome) and then we click "Create new spec":



Let us create the test file `cypress/e2e/note_app.cy.js`:



We could edit the tests in Cypress but let us rather use VS Code:



We can now close the edit view of Cypress.

Let us change the test content as follows:

```
describe('Note app', function() {  
  it('front page can be opened', function() {  
    cy.visit('http://localhost:5173')  
    cy.contains('Notes')  
    cy.contains('Note app, Department of Computer Science, University of Helsinki  
i 2023')  
  })  
})
```

The test is run by clicking on the test in Cypress:

Running the test shows how the application behaves as the test is run:

The structure of the test should look familiar. They use *describe* blocks to group different test cases, just like Jest. The test cases have been defined with the *it* method. Cypress borrowed these parts from the [Mocha](#) testing library that it uses under the hood.

`cy.visit` and `cy.contains` are Cypress commands, and their purpose is quite obvious. `cy.visit` opens the web address given to it as a parameter in the browser used by the test. `cy.contains` searches for the string it received as a parameter in the page.

We could have declared the test using an arrow function

```
describe('Note app', () => {
  it('front page can be opened', () => {
    cy.visit('http://localhost:5173')
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki 2023')
  })
})
```

However, Mocha [recommends](#) that arrow functions are not used, because they might cause some issues in certain situations.

If `cy.contains` does not find the text it is searching for, the test does not pass. So if we extend our test like so

```
describe('Note app', function() {
  it('front page can be opened', function() {
    cy.visit('http://localhost:5173')
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki 2023')
  })

  it('front page contains random text', function() {
    cy.visit('http://localhost:5173')
    cy.contains('wtf is this app?')
  })
})
```

the test fails

The screenshot shows the Cypress Test Runner interface. On the left, the test file `note_app.cy.js` is open, displaying a failing test for 'front page contains random text'. The assertion failed because the content 'wtf is this app?' was not found. On the right, a browser window shows the 'Notes app' application running on localhost:3000. The page title is 'Notes app' and it contains a note: 'Browser can execute only JavaScript'. A tooltip indicates that the browser can only execute JavaScript.

Let's remove the failing code from the test.

The variable `cy` our tests use gives us a nasty Eslint error

The screenshot shows a Cypress test file in VS Code. The code defines a test with a describe block and an it block. The it block uses the `cy` variable to visit a URL and check for text. An ESLint error message is displayed in the gutter: "'cy' is not defined. eslint(no-undef)". The cursor is positioned at the start of the first line of the test code.

We can get rid of it by installing [eslint-plugin-cypress](#) as a development dependency

```
npm install eslint-plugin-cypress --save-dev
```

and changing the configuration in `.eslintrc.cjs` like so:

```
module.exports = [
  "env": {
    "browser": true,
    "es2020": true,
    "jest/globals": true,
    "cypress/globals": true
  },
  "extends": [
    "plugin:cypress/recommended"
  ],
  "parser": "babel-eslint",
  "plugins": [
    "cypress"
  ],
  "rules": {
    "no-undef": "error"
  }
]
```

```
"extends": [
  // ...
],
"parserOptions": [
  // ...
],
"plugins": [
  "react", "jest", "cypress"
],
"rules": [
  // ...
]
}
```

Writing to a form

Let's extend our tests so that our new test tries to log in to our application. We assume our backend contains a user with the username *mluukkai* and password *salainen*.

The test begins by opening the login form.

```
describe('Note app', function() {
  // ...

  it('login form can be opened', function() {
    cy.visit('http://localhost:5173')
    cy.contains('log in').click()
  })
})
```

The test first searches for the login button by its text and clicks the button with the command `cy.click`.

Both of our tests begin the same way, by opening the page <http://localhost:5173>, so we should extract the shared code into a `beforeEach` block run before each test:

```
describe('Note app', function() {
  beforeEach(function() { cy.visit('http://localhost:5173') })
  it('front page can be opened', function() {
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki 2023')
  })

  it('login form can be opened', function() {
    cy.contains('log in').click()
  })
})
```

The login field contains two *input* fields, which the test should write into.

The `cy.get` command allows for searching elements by CSS selectors.

We can access the first and the last input field on the page, and write to them with the command `cy.type` like so:

```
it('user can login', function () {
  cy.contains('log in').click()
  cy.get('input:first').type('mluukkai')
  cy.get('input:last').type('salainen')
})
```

The test works. The problem is if we later add more input fields, the test will break because it expects the fields it needs to be the first and the last on the page.

It would be better to give our inputs unique *IDs* and use those to find them. We change our login form like so:

```
const LoginForm = ({ ... }) => {
  return (
    <div>
      <h2>Login</h2>
      <form onSubmit={handleSubmit}>
        <div>
          username
          <input
            id='username' value={username}
            onChange={handleUsernameChange}>
        />
        </div>
        <div>
          password
          <input
            id='password' type="password"
            value={password}
            onChange={handlePasswordChange}>
        />
        </div>
        <button id="login-button" type="submit"> login
        </button>
      </form>
    </div>
  )
}
```

We also added an ID to our submit button so we can access it in our tests.

The test becomes:

```
describe('Note app', function() {
  // ...
  it('user can log in', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')    cy.get('#password').type('salainen')
```

```
    cy.get('#login-button').click()
    cy.contains('Matti Luukkainen logged in')
  })
}
```

The last row ensures that the login was successful.

Note that the CSS's **ID selector** is `#`, so if we want to search for an element with the ID `username` the CSS selector is `#username`.

Please note that passing the test at this stage requires that there is a user in the test database of the backend test environment, whose username is `mluukkai` and the password is `salainen`. Create a user if needed!

Testing new note form

Next, let's add tests to test the "new note" functionality:

```
describe('Note app', function() {
  // ...
  describe('when logged in', function() {
    beforeEach(function() {
      cy.contains('log in').click()
      cy.get('input:first').type('mluukkai')
      cy.get('input:last').type('salainen')
      cy.get('#login-button').click()
    })
    it('a new note can be created', function() {
      cy.contains('new note').click()
      cy.get('input').type('a note created by cypress')
      cy.contains('save').click()
      cy.contains('a note created by cypress')
    })
  })
})})
```

The test has been defined in its own `describe` block. Only logged-in users can create new notes, so we added logging in to the application to a `beforeEach` block.

The test trusts that when creating a new note the page contains only one input, so it searches for it like so:

```
cy.get('input')
```

If the page contained more inputs, the test would break

The screenshot shows a Cypress test run interface. The left pane displays the test code and its execution results. The right pane shows a note from the Note app, Department of Computer Science, University of Helsinki 2022.

TEST BODY

```
1 contains new note
2 -click
3 get input
4 -type a note created by cypress
```

CypressError

cy.type() can only be called on a single element. Your subject contained 2 elements. Learn more

cypress/integration/note_app.spec.js:30:23

```
28 |     it('a new note can be created', f
29 |       cy.contains('new note').click()
> 30 |       cy.get('input').type('a note cr
|           ^
31 |       cy.contains('save').click()
32 |       cy.contains('a note created by
33 |     })
```

[View stack trace](#) [Print to console](#)

Note app, Department of Computer Science, University of Helsinki 2022

* a note created by cypress [make important](#)

Due to this problem, it would again be better to give the input an *ID* and search for the element by its ID.

The structure of the tests looks like so:

```
describe('Note app', function() {
  // ...

  it('user can log in', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('salainen')
    cy.get('#login-button').click()

    cy.contains('Matti Luukkainen logged in')
  })

  describe('when logged in', function() {
    beforeEach(function() {
      cy.contains('log in').click()
      cy.get('input:first').type('mluukkai')
      cy.get('input:last').type('salainen')
      cy.get('#login-button').click()
    })

    it('a new note can be created', function() {
      // ...
    })
  })
})
```

Cypress runs the tests in the order they are in the code. So first it runs *user can log in*, where the user logs in. Then cypress will run *a new note can be created* for which a *beforeEach* block logs in as well. Why do this? Isn't the user logged in after the first test? No, because *each* test starts from zero as far as the browser is concerned. All changes to the browser's state are reversed after each test.

Controlling the state of the database

If the tests need to be able to modify the server's database, the situation immediately becomes more complicated. Ideally, the server's database should be the same each time we run the tests, so our tests can be reliably and easily repeatable.

As with unit and integration tests, with E2E tests it is best to empty the database and possibly format it before the tests are run. The challenge with E2E tests is that they do not have access to the database.

The solution is to create API endpoints for the backend tests. We can empty the database using these endpoints. Let's create a new router for the tests inside the *controllers* folder, in the *testing.js* file

```
const testingRouter = require('express').Router()
const Note = require('../models/note')
const User = require('../models/user')

testingRouter.post('/reset', async (request, response) => [
    await Note.deleteMany({})
    await User.deleteMany({})

    response.status(204).end()
])

module.exports = testingRouter
```

and add it to the backend only *if the application is run in test-mode*:

```
// ...

app.use('/api/login', loginRouter)
app.use('/api/users', usersRouter)
app.use('/api/notes', notesRouter)

if (process.env.NODE_ENV === 'test') {
    const testingRouter = require('./controllers/testing')
    app.use('/api/testing', testingRouter)
}

app.use(middleware.unknownEndpoint)
app.use(middleware.errorHandler)

module.exports = app
```

After the changes, an HTTP POST request to the */api/testing/reset* endpoint empties the database. Make sure your backend is running in test mode by starting it with this command (previously configured in the package.json file):

```
npm run start:test
```

The modified backend code can be found on the [GitHub](#) branch *part5-1*.

Next, we will change the *beforeEach* block so that it empties the server's database before tests are run.

Currently, it is not possible to add new users through the frontend's UI, so we add a new user to the backend from the *beforeEach* block.

```
describe('Note app', function() {
  beforeEach(function() {
    cy.request('POST', 'http://localhost:3001/api/testing/reset')      const user =
    {      name: 'Matti Luukkainen',      username: 'mluukkai',      password: 'sal
ainen'    }    cy.request('POST', 'http://localhost:3001/api/users/', user)
    cy.visit('http://localhost:5173')
  })

  it('front page can be opened', function() {
    // ...
  })

  it('user can login', function() {
    // ...
  })
}

describe('when logged in', function() {
  // ...
})
```

During the formatting, the test does HTTP requests to the backend with `cy.request`.

Unlike earlier, now the testing starts with the backend in the same state every time. The backend will contain one user and no notes.

Let's add one more test for checking that we can change the importance of notes.

A while ago we changed the frontend so that a new note is important by default, so the *important* field is *true*:

```
const NoteForm = ([ createNote ]) => [
  // ...

  const addNote = [event] => {
    event.preventDefault()
    createNote([
      content: newNote,
      important: true
    ])

    setNewNote('')
  }
]
```

```
// ...  
})
```

There are multiple ways to test this. In the following example, we first search for a note and click its *make not important* button. Then we check that the note now contains a *make important* button.

```
describe('Note app', function() {  
    // ...  
  
    describe('when logged in', function() {  
        // ...  
  
        describe('and a note exists', function () {  
            beforeEach(function () {  
                cy.contains('new note').click()  
                cy.get('input').type('another note cypress')  
                cy.contains('save').click()  
            })  
  
            it('it can be made not important', function () {  
                cy.contains('another note cypress')  
                    .contains('make not important')  
                    .click()  
  
                cy.contains('another note cypress')  
                    .contains('make important')  
            })  
        })  
    })  
})
```

The first command searches for a component containing the text *another note cypress*, and then for a *make not important* button within it. It then clicks the button.

The second command checks that the text on the button has changed to *make important*.

Failed login test

Let's make a test to ensure that a login attempt fails if the password is wrong.

Cypress will run all tests each time by default, and as the number of tests increases, it starts to become quite time-consuming. When developing a new test or when debugging a broken test, we can define the test with *it.only* instead of *it*, so that Cypress will only run the required test. When the test is working, we can remove *.only*.

First version of our tests is as follows:

```
describe('Note app', function() {  
    // ...  
  
    it.only('login fails with wrong password', function() {
```

```

    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('wrong')
    cy.get('#login-button').click()

    cy.contains('wrong credentials')
  })

  // ...
})}

```

The test uses `cy.contains` to ensure that the application prints an error message.

The application renders the error message to a component with the CSS class `error`:

```

const Notification = ({ message }) => {
  if (message === null) {
    return null
  }

  return (
    <div className="error">      {message}
    </div>
  )
}

```

We could make the test ensure that the error message is rendered to the correct component, that is, the component with the CSS class `error`:

```

it('login fails with wrong password', function() {
  // ...

  cy.get('.error').contains('wrong credentials'))
})

```

First, we use `cy.get` to search for a component with the CSS class `error`. Then we check that the error message can be found in this component. Note that the [CSS class selectors](#) start with a full stop, so the selector for the class `error` is `.error`.

We could do the same using the `should` syntax:

```

it('login fails with wrong password', function() {
  // ...

  cy.get('.error').should('contain', 'wrong credentials'))
})

```

Using `should` is a bit trickier than using `contains`, but it allows for more diverse tests than `contains` which works based on text content only.

A list of the most common assertions which can be used with `should` can be found [here](#).

We can, for example, make sure that the error message is red and it has a border:

```
it('login fails with wrong password', function() {
  // ...

  cy.get('.error').should('contain', 'wrong credentials')
  cy.get('.error').should('have.css', 'color', 'rgb(255, 0, 0)')
  cy.get('.error').should('have.css', 'border-style', 'solid')
})
```

Cypress requires the colors to be given as `rgb`.

Because all tests are for the same component we accessed using `cy.get`, we can chain them using `and`.

```
it('login fails with wrong password', function() {
  // ...

  cy.get('.error')
    .should('contain', 'wrong credentials')
    .and('have.css', 'color', 'rgb(255, 0, 0)')
    .and('have.css', 'border-style', 'solid')
})
```

Let's finish the test so that it also checks that the application does not render the success message '*Matti Luukkainen logged in*':

```
it('login fails with wrong password', function() {
  cy.contains('log in').click()
  cy.get('#username').type('mluukkai')
  cy.get('#password').type('wrong')
  cy.get('#login-button').click()

  cy.get('.error')
    .should('contain', 'wrong credentials')
    .and('have.css', 'color', 'rgb(255, 0, 0)')
    .and('have.css', 'border-style', 'solid')

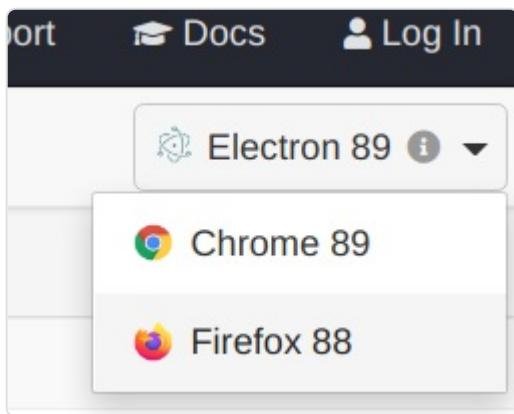
  cy.get('html').should('not.contain', 'Matti Luukkainen logged in'))
})
```

The command `should` is most often used by chaining it after the command `get` (or another similar command that can be chained). The `cy.get('html')` used in the test practically means the visible content of the entire application.

We would also check the same by chaining the command `contains` with the command `should` with a slightly different parameter:

```
cy.contains('Matti Luukkainen logged in').should('not.exist')
```

NOTE: Some CSS properties [behave differently on Firefox](#). If you run the tests with Firefox:



then tests that involve, for example, `border-style`, `border-radius` and `padding`, will pass in Chrome or Electron, but fail in Firefox:

The screenshot shows a Cypress test run for a POST request to '/api/login'. The test code includes several assertions for the error div's properties. The last assertion, at line 12, fails with an `AssertionError`. The error message indicates that the CSS property `border-style` was expected to be `'solid'` but was actually `''`.

```
(xhr) POST 401 /api/login
9 get .error
10 - assert expected <div.error> to contain wrong
username or password
11 - assert expected <div.error> to have CSS property
color with the value rgb(255, 0, 0)
12 - assert expected <div.error> to have CSS property border-
style with the value solid , but the value was ''
```

● Assertion Error

```
Timed out retrying after 4000ms: expected '<div.error>' to have CSS property 'border-style' with the value 'solid', but the value was ''
```

cypress/integration/blog_app.spec.js:30:8

```
28 |     cy.get('.error').should('contain', 'wrong username
29 |         .and('have.css', 'color', 'rgb(255, 0, 0)')
> 30 |         .and('have.css', 'border-style', 'solid')
|             ^
```

Bypassing the UI

Currently, we have the following tests:

```
describe('Note app', function() {
  it('user can login', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('salainen')
    cy.get('#login-button').click()
```

```

    cy.contains('Matti Luukkainen logged in')
  })

  it('login fails with wrong password', function() {
    // ...
  })

  describe('when logged in', function() {
    beforeEach(function() {
      cy.contains('log in').click()
      cy.get('input:first').type('mluukkai')
      cy.get('input:last').type('salainen')
      cy.get('#login-button').click()
    })

    it('a new note can be created', function() {
      // ...
    })
  })
})

```

First, we test logging in. Then, in their own describe block, we have a bunch of tests, which expect the user to be logged in in the *beforeEach* block.

As we said above, each test starts from zero! Tests do not start from the state where the previous tests ended.

The Cypress documentation gives us the following advice: [Fully test the login flow – but only once](#). So instead of logging in a user using the form in the *beforeEach* block, we are going to bypass the UI and do a HTTP request to the backend to log in. The reason for this is that logging in with a HTTP request is much faster than filling out a form.

Our situation is a bit more complicated than in the example in the Cypress documentation because when a user logs in, our application saves their details to the localStorage. However, Cypress can handle that as well. The code is the following:

```

describe('when logged in', function() {
  beforeEach(function() {
    cy.request('POST', 'http://localhost:3001/api/login', {
      username: 'mluukkai',
      password: 'salainen'
    }).then(response => {
      localStorage.setItem('loggedNoteappUser', JSON.stringify(response.body))
      cy.visit('http://localhost:5173')
    })
  })

  it('a new note can be created', function() {
    // ...
  })
})

```

We can access to the response of a `cy.request` with the `then` method. Under the hood `cy.request`, like

all Cypress commands, are [asynchronous](#). The callback function saves the details of a logged-in user to localStorage, and reloads the page. Now there is no difference to a user logging in with the login form.

If and when we write new tests to our application, we have to use the login code in multiple places, we should make it a [custom command](#).

Custom commands are declared in *cypress/support/commands.js*. The code for logging in is as follows:

```
Cypress.Commands.add('login', ({username, password}) => {
  cy.request('POST', 'http://localhost:3001/api/login', {
    username,
    password
  }).then(({body}) => {
    localStorage.setItem('loggedNoteappUser', JSON.stringify(body))
    cy.visit('http://localhost:5173')
  })
})
```

Using our custom command is easy, and our test becomes cleaner:

```
describe('when logged in', function() {
  beforeEach(function() {
    cy.login({username: 'mluukkai', password: 'salainen'})
  })

  it('a new note can be created', function() {
    // ...
  })
  // ...
})
```

The same applies to creating a new note now that we think about it. We have a test, which makes a new note using the form. We also make a new note in the *beforeEach* block of the test that changes the importance of a note:

```
describe('Note app', function() {
  // ...

  describe('when logged in', function() {
    it('a new note can be created', function() {
      cy.contains('new note').click()
      cy.get('input').type('a note created by cypress')
      cy.contains('save').click()

      cy.contains('a note created by cypress')
    })
  })

  describe('and a note exists', function() {
    beforeEach(function() {
      cy.contains('new note').click()
    })
  })
})
```

```

        cy.get('input').type('another note cypress')
        cy.contains('save').click()
    })

    it('it can be made important', function () {
        // ...
    })
})
})
})
})

```

Let's make a new custom command for making a new note. The command will make a new note with an HTTP POST request:

```

Cypress.Commands.add('createNote', ({ content, important }) => [
  cy.request({
    url: 'http://localhost:3001/api/notes',
    method: 'POST',
    body: { content, important },
    headers: {
      'Authorization': `Bearer ${JSON.parse(localStorage.getItem('loggedNoteappUser')).token}`
    }
  }),

  cy.visit('http://localhost:5173')
])

```

The command expects the user to be logged in and the user's details to be saved to localStorage.

Now the note beforeEach block becomes:

```

describe('Note app', function() {
  // ...

  describe('when logged in', function() {
    it('a new note can be created', function() {
      // ...
    })

    describe('and a note exists', function () {
      beforeEach(function () {
        cy.createNote({ content: 'another note cypress', important: true })
      })

      it('it can be made important', function () {
        // ...
      })
    })
  })
})

```

There is one more annoying feature in our tests. The application address `http://localhost:5173` is hardcoded in many places.

Let's define the `baseUrl` for the application in the Cypress pre-generated configuration file `cypress.config.js`:

```
const { defineConfig } = require("cypress")

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
      },
      baseUrl: 'http://localhost:5173',
    }
})
```

All the commands in the tests use the address of the application

```
cy.visit('http://localhost:5173')
```

can be transformed into

```
cy.visit('')
```

The backend's hardcoded address `http://localhost:3001` is still in the tests. Cypress documentation recommends defining other addresses used by the tests as environment variables.

Let's expand the configuration file `cypress.config.js` as follows:

```
const { defineConfig } = require("cypress")

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
      },
      baseUrl: 'http://localhost:5173',
      env: {
        BACKEND: 'http://localhost:3001/api'
      },
    }
})
```

Let's replace all the backend addresses from the tests in the following way

```
describe('Note ', function() {
  beforeEach(function() {

    cy.request('POST', `${Cypress.env('BACKEND')}/testing/reset`)

    const user = {
      name: 'Matti Luukkainen',
      username: 'mluukkai',
      password: 'secret'
    })
  })
})
```

```
    })
    cy.request('POST', `${Cypress.env('BACKEND')}/users`, user)      cy.visit('')

  })
  // ...
})
```

Changing the importance of a note

Lastly, let's take a look at the test we did for changing the importance of a note. First, we'll change the `beforeEach` block so that it creates three notes instead of one:

```
describe('when logged in', function() {
  describe('and several notes exist', function () {
    beforeEach(function () {
      cy.login({ username: 'mluukkai', password: 'salainen' })      cy.createNote(
        { content: 'first note', important: false })      cy.createNote({ content: 'sec
ond note', important: false })      cy.createNote({ content: 'third note', impor
tant: false })
    })

    it('one of those can be made important', function () {
      cy.contains('second note')
        .contains('make important')
        .click()

      cy.contains('second note')
        .contains('make not important')
    })
  })
})
```

How does the `cy.contains` command actually work?

When we click the `cy.contains('second note')` command in Cypress [Test Runner](#), we see that the command searches for the element containing the text *second note*:

The screenshot shows the Cypress Test Runner interface on the left and a browser window on the right. The browser displays a notes application titled 'Notes app'. The notes list contains three items: 'first note', 'second note', and 'third note'. The 'second note' item has a 'make important' button next to it, which is highlighted with a yellow background. The test runner shows the following code:

```

    cy.contains('second note')
      .contains('make important')
      .click()
  
```

By clicking the next line `.contains('make important')` we see that the test uses the 'make important' button corresponding to the *second note*:

The screenshot shows the Cypress Test Runner interface on the left and a browser window on the right. The browser displays a notes application titled 'Notes app'. The notes list contains three items: 'first note', 'second note', and 'third note'. The 'second note' item has a 'make important' button next to it, which is highlighted with a yellow background. The test runner shows the following code:

```

    cy.contains('second note')
      .contains('make important')
      .click()
  
```

When chained, the second *contains* command *continues* the search from within the component found by the first command.

If we had not chained the commands, and instead write:

```

cy.contains('second note')
cy.contains('make important').click()
  
```

the result would have been entirely different. The second line of the test would click the button of a

wrong note:

The screenshot shows the Cypress test runner interface. On the left, the test file 'note_app.cy.js' is open, displaying a failing test. The test body contains the following code:

```
1 -contains second note
(xhr) GET 200 /api/notes
2   -contains make important
3     -click
(xhr) PUT 200
/api/notes/63d0d80a124fee74005fa415
4 -contains second note
5 -contains make not important
```

An assertion error is shown at line 5:

```
! Assertion
Timed out retrying after 4000ms: Expected to find content: 'make not important' within the element: <li.note> but never did.
```

The right side of the screenshot shows the application's UI. It has a header 'Notes app' and a message 'Matti Luukkainen logged in'. Below that is a list of notes:

- first note make important
- second note make important
- third note make important

At the bottom, there is a footer: 'Note app, Department of Computer Science, University of Helsinki 2023'.

When coding tests, you should check in the test runner that the tests use the right components!

Let's change the *Note* component so that the text of the note is rendered to a *span*.

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important' : 'make important'

  return [
    <li className='note'>
      <span>{note.content}</span>           <button onClick={toggleImportance}>{label}</button>
    </li>
  ]
}
```

Our tests break! As the test runner reveals, `cy.contains('second note')` now returns the component containing the text, and the button is not in it.

The screenshot shows the Cypress Test Runner interface. On the left, the test file `note_app.cy.js` is open, displaying a failing test. The test code includes a `visit` command, a `request` command, and a `visit` command again. The `TEST BODY` section contains a `-contains` assertion for the text "second note". A screenshot of the browser window on the right shows a notes application with three notes. Each note has a "make important" button next to its name. The error message in the test runner indicates that the button was not found within the expected element.

One way to fix this is the following:

```
it('one of those can be made important', function () {
  cy.contains('second note').parent().find('button').click()
  cy.contains('second note').parent().find('button')
    .should('contain', 'make not important')
})
```

In the first line, we use the `parent` command to access the parent element of the element containing *second note* and find the button from within it. Then we click the button and check that the text on it changes.

Note that we use the command `find` to search for the button. We cannot use `cy.get` here, because it always searches from the *whole* page and would return all 5 buttons on the page.

Unfortunately, we have some copy-paste in the tests now, because the code for searching for the right button is always the same.

In these kinds of situations, it is possible to use the `as` command:

```
it('one of those can be made important', function () {
  cy.contains('second note').parent().find('button').as('theButton')
  cy.get('@theButton').click()
  cy.get('@theButton').should('contain', 'make not important')
})
```

Now the first line finds the right button and uses `as` to save it as *theButton*. The following lines can use the named element with `cy.get('@theButton')`.

Running and debugging the tests

Finally, some notes on how Cypress works and debugging your tests.

Because of the form of the Cypress tests, it gives the impression that they are normal JavaScript code, and we could for example try this:

```
const button = cy.contains('log in')
button.click()
debugger
cy.contains('logout').click()
```

This won't work, however. When Cypress runs a test, it adds each `cy` command to an execution queue. When the code of the test method has been executed, Cypress will execute each command in the queue one by one.

Cypress commands always return *undefined*, so `button.click()` in the above code would cause an error. An attempt to start the debugger would not stop the code between executing the commands, but before any commands have been executed.

Cypress commands are *like promises*, so if we want to access their return values, we have to do it using the `then` command. For example, the following test would print the number of buttons in the application, and click the first button:

```
it('then example', function() {
  cy.get('button').then(buttons => {
    console.log('number of buttons', buttons.length)
    cy.wrap(buttons[0]).click()
  })
})
```

Stopping the test execution with the debugger is [possible](#). The debugger starts only if Cypress test runner's developer console is open.

The developer console is all sorts of useful when debugging your tests. You can see the HTTP requests done by the tests on the Network tab, and the console tab will show you information about your tests:

The screenshot shows the Cypress graphical test runner interface. On the left, the test file 'note_app.cy.js' is open, displaying a series of Cypress commands. On the right, a browser window shows a 'Notes app' page with a list of notes and a 'make important' button. Below the browser, the developer tools' Elements tab is active, showing the DOM structure of the note item. A red arrow points from the Cypress console output to the 'Yielded:' line, which displays the selected button element.

```

    "Console was cleared"
    Command: find
    Selector: button
      <li class="note">
        <span>second note</span>
        <button>make not important</button>
    Applied to: </li>
    Yielded: <button>make not important</button>
    Elements: 1
  
```

So far we have run our Cypress tests using the graphical test runner. It is also possible to run them [from the command line](#). We just have to add an npm script for it:

```

  "scripts": {
    "cypress:open": "cypress open",
    "test:e2e": "cypress run"
  }
  
```

Now we can run our tests from the command line with the command `npm run test:e2e`

Note

- ✓ front page can be opened (544ms)
 - ✓ login form can be opened (972ms)
 - ✓ login fails with wrong password (792ms)
 - ✓ then example (366ms)
- when logged in
- ✓ a new note can be created (983ms)
 - and several notes exist
 - ✓ one of those can be made important (974ms)

6 passing (5s)

(Results)

```
Tests:      6
Passing:    6
Failing:   0
Pending:   0
Skipped:   0
Screenshots: 0
Video:     true
Duration:  4 seconds
Spec Ran:  note_app.cy.js
```

(Video)

- Started processing: Compressing to 32 CRF
- Finished processing: /Users/mluukkai/opetus/hy-fs/koodi/notes-app/frontend/cypress/videos/note_app.cy.js.mp4 (0 seconds)

Note that videos of the test execution will be saved to `cypress/videos/`, so you should probably git ignore this directory. It is also possible to [turn off](#) the making of videos.

Tests are found in [GitHub](#).

Final version of the frontend code can be found on the [GitHub](#) branch `part5-9`.

Exercises 5.17.-5.23.

In the last exercises of this part, we will do some E2E tests for our blog application. The material of this part should be enough to complete the exercises. You [must check out the Cypress documentation](#). It is probably the best documentation I have ever seen for an open-source project.

I especially recommend reading [Introduction to Cypress](#), which states

This is the single most important guide for understanding how to test with Cypress. Read it. Understand it.

5.17: Blog List End To End Testing, step 1

Configure Cypress for your project. Make a test for checking that the application displays the login form by default.

The structure of the test must be as follows:

```
describe('Blog app', function() {
  beforeEach(function() {
    cy.visit('http://localhost:5173')
  })

  it('Login form is shown', function() {
    // ...
  })
})
```

5.18: Blog List End To End Testing, step 2

Make tests for logging in. Test both successful and unsuccessful login attempts. Make a new user in the *beforeEach* block for the tests.

The test structure extends like so:

```
describe('Blog app', function() {
  beforeEach(function() {
    // empty the db here
    // create a user for the backend here
    cy.visit('http://localhost:5173')
  })

  it('Login form is shown', function() {
    // ...
  })
})

describe('Login', function() {
  it('succeeds with correct credentials', function() {
    // ...
  })

  it('fails with wrong credentials', function() {
    // ...
  })
})
```

The *beforeEach* block must empty the database using, for example, the *reset* method we used in the [material](#).

Optional bonus exercise: Check that the notification shown with unsuccessful login is displayed red.

5.19: Blog List End To End Testing, step 3

Make a test that verifies a logged-in user can create a new blog. The structure of the test could be as follows:

```

describe('Blog app', function() {
  // ...

  describe('When logged in', function() {
    beforeEach(function() {
      // ...
    })

    it('A blog can be created', function() {
      // ...
    })
  })
})

```

The test has to ensure that a new blog is added to the list of all blogs.

5.20: Blog List End To End Testing, step 4

Make a test that confirms users can like a blog.

5.21: Blog List End To End Testing, step 5

Make a test for ensuring that the user who created a blog can delete it.

5.22: Blog List End To End Testing, step 6

Make a test for ensuring that only the creator can see the delete button of a blog, not anyone else.

5.23: Blog List End To End Testing, step 7

Make a test that checks that the blogs are ordered by likes, with the most liked blog being first.

This exercise is quite a bit trickier than the previous ones. One solution is to add a certain class for the element which wraps the blog's content and use the `eq` method to get the blog element in a specific index:

```

cy.get('.blog').eq(0).should('contain', 'The title with the most likes')
cy.get('.blog').eq(1).should('contain', 'The title with the second most likes')

```

Note that you might end up having problems if you click a like button many times in a row. It might be that cypress does the clicking so fast that it does not have time to update the app state in between the clicks. One remedy for this is to wait for the number of likes to update in between all clicks.

This was the last exercise of this part, and it's time to push your code to GitHub and mark the exercises you completed in the [exercise submission system](#).

6: Advanced state management

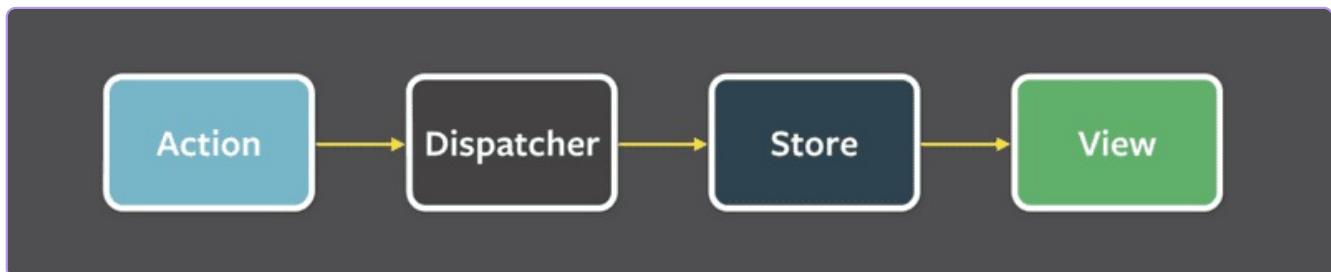
6a: Flux-architecture and Redux

So far, we have followed the state management conventions recommended by React. We have placed the state and the functions for handling it in the [higher level](#) of the component structure of the application. Quite often most of the app state and state altering functions reside directly in the root component. The state and its handler methods have then been passed to other components with props. This works up to a certain point, but when applications grow larger, state management becomes challenging.

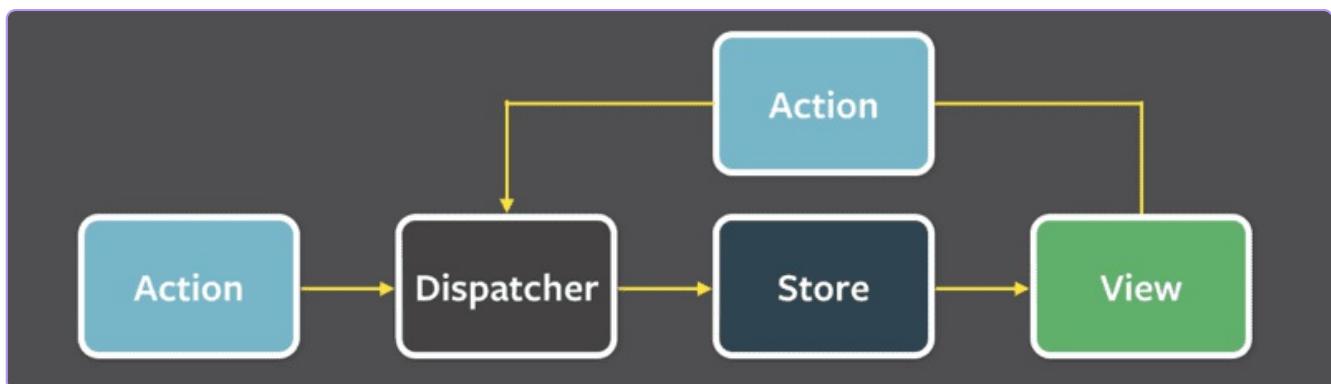
Flux-architecture

Already years ago Facebook developed the [Flux](#)-architecture to make state management of React apps easier. In Flux, the state is separated from the React components and into its own *stores*. State in the store is not changed directly, but with different *actions*.

When an action changes the state of the store, the views are rerendered:



If some action on the application, for example pushing a button, causes the need to change the state, the change is made with an action. This causes re-rendering the view again:

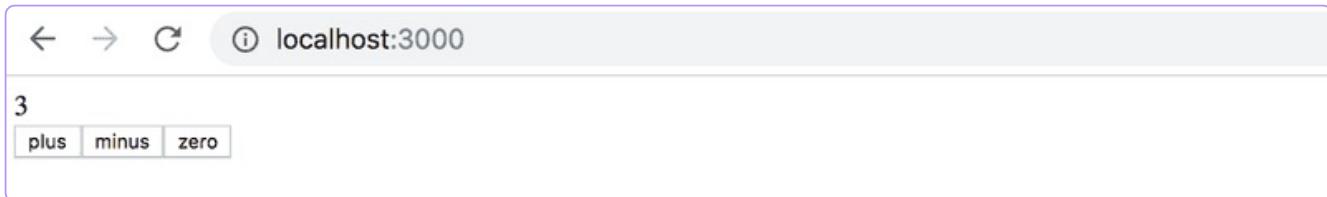


Flux offers a standard way for how and where the application's state is kept and how it is modified.

Redux

Facebook has an implementation for Flux, but we will be using the [Redux](#) library. It works with the same principle but is a bit simpler. Facebook also uses Redux now instead of their original Flux.

We will get to know Redux by implementing a counter application yet again:



Create a new Vite application and install redux with the command

```
npm install redux
```

As in Flux, in Redux the state is also stored in a [store](#).

The whole state of the application is stored in *one* JavaScript object in the store. Because our application only needs the value of the counter, we will save it straight to the store. If the state was more complicated, different things in the state would be saved as separate fields of the object.

The state of the store is changed with [actions](#). Actions are objects, which have at least a field determining the *type* of the action. Our application needs for example the following action:

```
{  
  type: 'INCREMENT'  
}
```

If there is data involved with the action, other fields can be declared as needed. However, our counting app is so simple that the actions are fine with just the type field.

The impact of the action to the state of the application is defined using a [reducer](#). In practice, a reducer is a function that is given the current state and an action as parameters. It *returns* a new state.

Let's now define a reducer for our application:

```
const counterReducer = (state, action) => {  
  if (action.type === 'INCREMENT') {  
    return state + 1  
  } else if (action.type === 'DECREMENT') {  
    return state - 1  
  } else if (action.type === 'ZERO') {  
    return 0  
  }  
  
  return state  
}
```

The first parameter is the *state* in the store. The reducer returns a *new state* based on the *action type*. So, e.g. when the type of Action is *INCREMENT*, the state gets the old value plus one. If the type of Action is *ZERO* the new value of state is zero.

Let's change the code a bit. We have used if-else statements to respond to an action and change the state. However, the [switch](#) statement is the most common approach to writing a reducer.

Let's also define a **default value** of 0 for the parameter *state*. Now the reducer works even if the store state has not been primed yet.

```
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    case 'ZERO':
      return 0
    default: // if none of the above matches, code comes here
      return state
  }
}
```

The reducer is never supposed to be called directly from the application's code. It is only given as a parameter to the *createStore* function which creates the store:

```
import { createStore } from 'redux'
const counterReducer = (state = 0, action) => {
  // ...
}

const store = createStore(counterReducer)
```

The store now uses the reducer to handle *actions*, which are *dispatched* or 'sent' to the store with its **dispatch** method.

```
store.dispatch({ type: 'INCREMENT' })
```

You can find out the state of the store using the method **getState**.

For example the following code:

```
const store = createStore(counterReducer)
console.log(store.getState())
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
console.log(store.getState())
store.dispatch({ type: 'ZERO' })
store.dispatch({ type: 'DECREMENT' })
console.log(store.getState())
```

would print the following to the console

```
0
3
-1
```

because at first, the state of the store is 0. After three *INCREMENT* actions the state is 3. In the end, after the *ZERO* and *DECREMENT* actions, the state is -1.

The third important method that the store has is `subscribe`, which is used to create callback functions that the store calls whenever an action is dispatched to the store.

If, for example, we would add the following function to subscribe, *every change in the store* would be printed to the console.

```
store.subscribe(() => {
  const storeNow = store.getState()
  console.log(storeNow)
})
```

so the code

```
const store = createStore(counterReducer)

store.subscribe(() => {
  const storeNow = store.getState()
  console.log(storeNow)
})

store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'ZERO' })
store.dispatch({ type: 'DECREMENT' })
```

would cause the following to be printed

```
1
2
3
0
-1
```

The code of our counter application is the following. All of the code has been written in the same file (*main.jsx*), so *store* is directly available for the React code. We will get to know better ways to structure React/Redux code later.

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import { createStore } from 'redux'

const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'ZERO':
      return 0
    case 'DECREMENT':
      return state - 1
  }
}
```

```

        case 'DECREMENT':
            return state - 1
        case 'ZERO':
            return 0
        default:
            return state
    }
}

const store = createStore(counterReducer)

const App = () => {
    return (
        <div>
            <div>
                {store.getState()}
            </div>
            <button
                onClick={e => store.dispatch({ type: 'INCREMENT' })}
            >
                plus
            </button>
            <button
                onClick={e => store.dispatch({ type: 'DECREMENT' })}
            >
                minus
            </button>
            <button
                onClick={e => store.dispatch({ type: 'ZERO' })}
            >
                zero
            </button>
        </div>
    )
}

const root = ReactDOM.createRoot(document.getElementById('root'))

const renderApp = () => {
    root.render(<App />)
}

renderApp()
store.subscribe(renderApp)

```

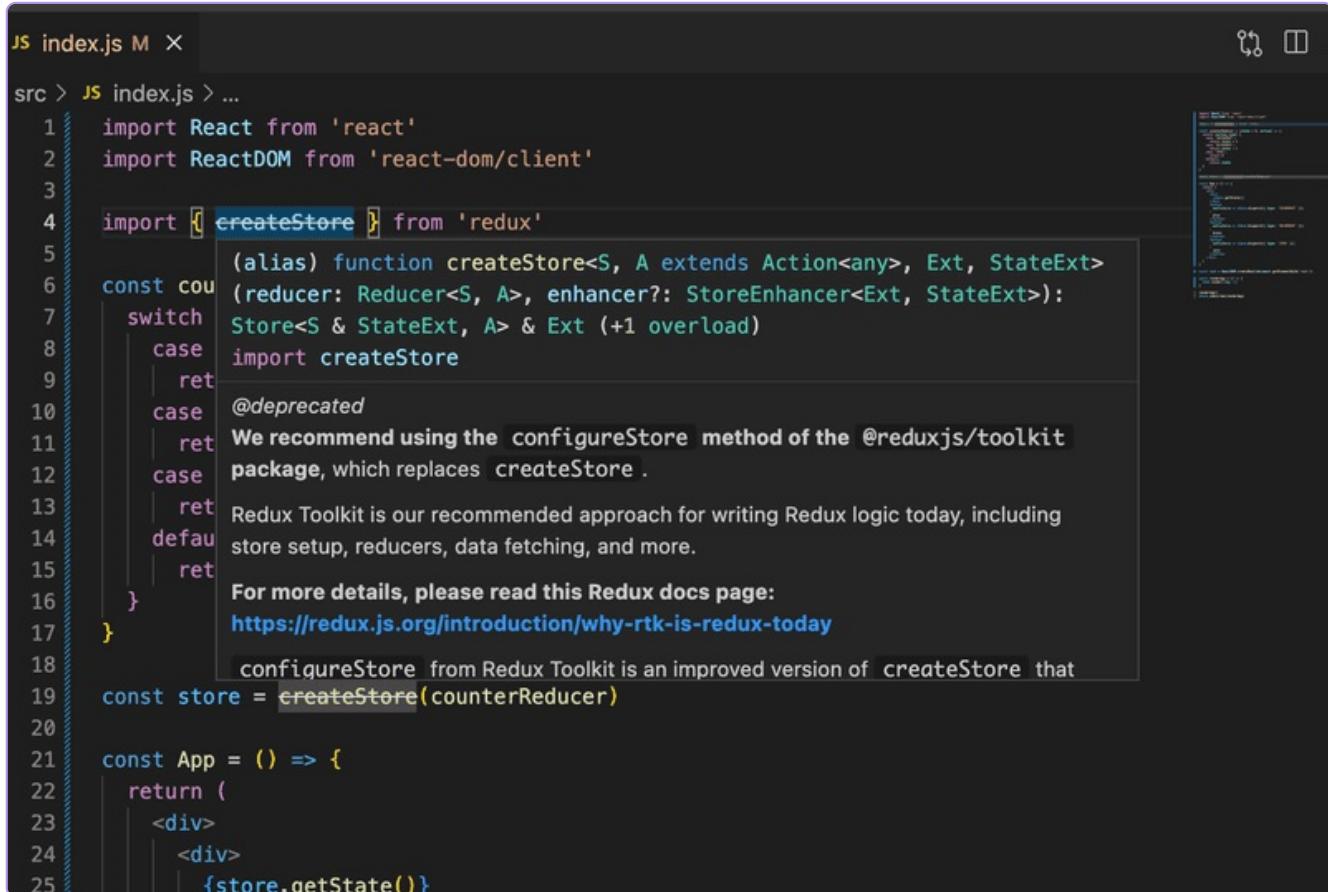
There are a few notable things in the code. *App* renders the value of the counter by asking it from the store with the method *store.getState()*. The action handlers of the buttons *dispatch* the right actions to the store.

When the state in the store is changed, React is not able to automatically re-render the application. Thus we have registered a function *renderApp*, which renders the whole app, to listen for changes in the store with the *store.subscribe* method. Note that we have to immediately call the *renderApp*

method. Without the call, the first rendering of the app would never happen.

A note about the use of createStore

The most observant will notice that the name of the function createStore is overlined. If you move the mouse over the name, an explanation will appear



JS index.js M X

```
src > JS index.js > ...
1 import React from 'react'
2 import ReactDOM from 'react-dom/client'

3 import { createStore } from 'redux'
4
5 const counterReducer = (state = 0, action) => {
6   switch (action.type) {
7     case 'increment':
8       return state + 1
9     case 'decrement':
10      return state - 1
11    default:
12      return state
13  }
14}
15
16 const store = createStore(counterReducer)
17
18 const App = () => {
19   return (
20     <div>
21       <div>
22         {store.getState()}
23       </div>
24     </div>
25   )
26}
```

The tooltip content is as follows:

We recommend using the `configureStore` method of the `@reduxjs/toolkit` package, which replaces `createStore`.

Redux Toolkit is our recommended approach for writing Redux logic today, including store setup, reducers, data fetching, and more.

For more details, please read this Redux docs page:
<https://redux.js.org/introduction/why-rtk-is-redux-today>

`configureStore` from Redux Toolkit is an improved version of `createStore` that simplifies setup and helps avoid common bugs.

You should not be using the `redux` core package by itself today, except for learning purposes. The `createStore` method from the core redux package will not be removed, but we encourage all users to migrate to using Redux Toolkit for all Redux code.

The full explanation is as follows

We recommend using the `configureStore` method of the `@reduxjs/toolkit` package, which replaces `createStore`.

Redux Toolkit is our recommended approach for writing Redux logic today, including store setup, reducers, data fetching, and more.

For more details, please read this Redux docs page: <https://redux.js.org/introduction/why-rtk-is-redux-today>

`configureStore` from Redux Toolkit is an improved version of `createStore` that simplifies setup and helps avoid common bugs.

You should not be using the `redux` core package by itself today, except for learning purposes. The `createStore` method from the core redux package will not be removed, but we encourage all users to migrate to using Redux Toolkit for all Redux code.

So, instead of the function `createStore`, it is recommended to use the slightly more "advanced"

function `configureStore`, and we will also use it when we have achieved the basic functionality of Redux.

Side note: `createStore` is defined as "deprecated", which usually means that the feature will be removed in some newer version of the library. The explanation above and this [discussion](#) reveal that `createStore` will not be removed, and it has been given the status *deprecated*, perhaps with slightly incorrect reasons. So the function is not obsolete, but today there is a more preferable, new way to do almost the same thing.

Redux-notes

We aim to modify our note application to use Redux for state management. However, let's first cover a few key concepts through a simplified note application.

The first version of our application is the following

```
const noteReducer = (state = [], action) => {
  if (action.type === 'NEW_NOTE') {
    state.push(action.payload)
    return state
  }

  return state
}

const store = createStore(noteReducer)

store.dispatch({
  type: 'NEW_NOTE',
  payload: [
    content: 'the app state is in redux store',
    important: true,
    id: 1
  ]
})

store.dispatch({
  type: 'NEW_NOTE',
  payload: [
    content: 'state changes are made with actions',
    important: false,
    id: 2
  ]
})

const App = () => {
  return(
    <div>
      <ul>
        {store.getState().map(note=>
          <li key={note.id}>
            {note.content} <strong>{note.important ? 'important' : ''}</strong>
          </li>
        )}
      </ul>
    </div>
  )
}
```

```

        </li>
    ) }
</ul>
</div>
)
}

```

So far the application does not have the functionality for adding new notes, although it is possible to do so by dispatching *NEW_NOTE* actions.

Now the actions have a type and a field *payload*, which contains the note to be added:

```

{
  type: 'NEW_NOTE',
  payload: [
    content: 'state changes are made with actions',
    important: false,
    id: 2
  ]
}

```

The choice of the field name is not random. The general convention is that actions have exactly two fields, *type* telling the type and *payload* containing the data included with the Action.

Pure functions, immutable

The initial version of the reducer is very simple:

```

const noteReducer = (state = [], action) => {
  if (action.type === 'NEW_NOTE') {
    state.push(action.payload)
    return state
  }

  return state
}

```

The state is now an Array. *NEW_NOTE*-type actions cause a new note to be added to the state with the `push` method.

The application seems to be working, but the reducer we have declared is bad. It breaks the [basic assumption](#) that reducers must be [pure functions](#).

Pure functions are such, that they *do not cause any side effects* and they must always return the same response when called with the same parameters.

We added a new note to the state with the method `state.push(action.payload)` which *changes* the state of the state-object. This is not allowed. The problem is easily solved by using the `concat` method, which creates a *new array*, which contains all the elements of the old array and the new element:

```
const noteReducer = (state = [], action) => {
  if (action.type === 'NEW_NOTE') {
    return state.concat(action.payload)
  }

  return state
}
```

A reducer state must be composed of **immutable** objects. If there is a change in the state, the old object is not changed, but it is *replaced with a new, changed, object*. This is exactly what we did with the new reducer: the old array is replaced with the new one.

Let's expand our reducer so that it can handle the change of a note's importance:

```
{
  type: 'TOGGLE_IMPORTANCE',
  payload: [
    {
      id: 2
    }
  ]
}
```

Since we do not have any code which uses this functionality yet, we are expanding the reducer in the 'test-driven' way. Let's start by creating a test for handling the action *NEW_NOTE*.

We have to first configure the **Jest** testing library for the project. Let us install the following dependencies:

```
npm install --save-dev jest @babel/preset-env @babel/preset-react eslint-plugin-jest
```

Next we'll create the file *.babelrc*, with the following content:

```
{
  "presets": [
    "@babel/preset-env",
    ["@babel/preset-react", { "runtime": "automatic" }]
  ]
}
```

Let us expand *package.json* with a script for running the tests:

```
{
  // ...
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview",
    "test": "jest"
  }
}
```

```
// ...
}
```

And finally, `.eslintrc.cjs` needs to be altered as follows:

```
module.exports = [
  root: true,
  env: {
    browser: true,
    es2020: true,
    "jest/globals": true
  },
  // ...
]
```

To make testing easier, we'll first move the reducer's code to its own module, to the file `src/reducers/noteReducer.js`. We'll also add the library `deep-freeze`, which can be used to ensure that the reducer has been correctly defined as an immutable function. Let's install the library as a development dependency:

```
npm install --save-dev deep-freeze
```

The test, which we define in file `src/reducers/noteReducer.test.js`, has the following content:

```
import noteReducer from './noteReducer'
import deepFreeze from 'deep-freeze'

describe('noteReducer', () => {
  test('returns new state with action NEW_NOTE', () => {
    const state = []
    const action = {
      type: 'NEW_NOTE',
      payload: {
        content: 'the app state is in redux store',
        important: true,
        id: 1
      }
    }

    deepFreeze(state)
    const newState = noteReducer(state, action)

    expect(newState).toHaveLength(1)
    expect(newState).toContainEqual(action.payload)
  })
})
```

The `deepFreeze(state)` command ensures that the reducer does not change the state of the store given to it as a parameter. If the reducer uses the `push` command to manipulate the state, the test will not pass

```

FAIL | src/reducers/noteReducer.test.js
noteReducer
  ✘ returns new state with action NEW_NOTE (16ms)

● noteReducer > returns new state with action NEW_NOTE

  TypeError: Cannot add property 0, object is not extensible
    at Array.push (<anonymous>)

      5 |
      6 |   if (action.type === 'NEW_NOTE') {
> 7 |     state.push(action.data)
      |     ^
      8 |     return state
      9 |   }
     10 |

  at push (src/reducers/noteReducer.js:7:11)
  at Object.it (src/reducers/noteReducer.test.js:17:22)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total

```

Now we'll create a test for the `TOGGLE_IMPORTANCE` action:

```

test('returns new state with action TOGGLE_IMPORTANCE', () => {
  const state = [
    {
      content: 'the app state is in redux store',
      important: true,
      id: 1
    },
    {
      content: 'state changes are made with actions',
      important: false,
      id: 2
    }
  ]

  const action = {
    type: 'TOGGLE_IMPORTANCE',
    payload: {
      id: 2
    }
  }

  deepFreeze(state)
  const newState = noteReducer(state, action)

  expect(newState).toHaveLength(2)

  expect(newState).toContainEqual(state[0])
}

```

```

expect(newState).toContainEqual([
  content: 'state changes are made with actions',
  important: true,
  id: 2
])
})
)

```

So the following action

```

{
  type: 'TOGGLE_IMPORTANCE',
  payload: [
    id: 2
  ]
}

```

has to change the importance of the note with the id 2.

The reducer is expanded as follows

```

const noteReducer = (state = [], action) => {
  switch(action.type) {
    case 'NEW_NOTE':
      return state.concat(action.payload)
    case 'TOGGLE_IMPORTANCE':
      const id = action.payload.id
      const noteToChange = state.find(n => n.id === id)
      const changedNote = [
        ...noteToChange,
        important: !noteToChange.important
      ]
      return state.map(note =>
        note.id !== id ? note : changedNote
      )
    }
    default:
      return state
  }
}

```

We create a copy of the note whose importance has changed with the syntax [familiar from part 2](#), and replace the state with a new state containing all the notes which have not changed and the copy of the changed note *changedNote*.

Let's recap what goes on in the code. First, we search for a specific note object, the importance of which we want to change:

```
const noteToChange = state.find(n => n.id === id)
```

then we create a new object, which is a *copy* of the original note, only the value of the *important* field has been changed to the opposite of what it was:

```
const changedNote = [
  ...noteToChange,
  important: !noteToChange.important
]
```

A new state is then returned. We create it by taking all of the notes from the old state except for the desired note, which we replace with its slightly altered copy:

```
state.map(note =>
  note.id !== id ? note : changedNote
)
```

Array spread syntax

Because we now have quite good tests for the reducer, we can refactor the code safely.

Adding a new note creates the state returned from the Array's *concat* function. Let's take a look at how we can achieve the same by using the JavaScript [array spread](#) syntax:

```
const noteReducer = (state = [], action) => {
  switch(action.type) {
    case 'NEW_NOTE':
      return [...state, action.payload]
    // ...
    default:
      return state
  }
}
```

The spread -syntax works as follows. If we declare

```
const numbers = [1, 2, 3]
```

`...numbers` breaks the array up into individual elements, which can be placed in another array.

```
[...numbers, 4, 5]
```

and the result is an array `[1, 2, 3, 4, 5]`.

If we would have placed the array to another array without the spread

```
[numbers, 4, 5]
```

the result would have been `[[1, 2, 3], 4, 5]`.

When we take elements from an array by [destructuring](#), a similar-looking syntax is used to *gather* the rest of the elements:

```
const numbers = [1, 2, 3, 4, 5, 6]

const [first, second, ...rest] = numbers

console.log(first)      // prints 1
console.log(second)     // prints 2
console.log(rest)       // prints [3, 4, 5, 6]
```

Exercises 6.1.-6.2.

Let's make a simplified version of the unicafe exercise from part 1. Let's handle the state management with Redux.

You can take the code from this repository <https://github.com/fullstack-hy2020/unicafe-redux> for the base of your project.

Start by removing the git configuration of the cloned repository, and by installing dependencies

```
cd unicafe-redux    // go to the directory of cloned repository
rm -rf .git
npm install
```

6.1: Unicafe Revisited, step 1

Before implementing the functionality of the UI, let's implement the functionality required by the store.

We have to save the number of each kind of feedback to the store, so the form of the state in the store is:

```
{  
  good: 5,  
  ok: 4,  
  bad: 2  
}
```

The project has the following base for a reducer:

```
const initialState = {  
  good: 0,  
  ok: 0,  
  bad: 0  
}  
  
const counterReducer = (state = initialState, action) => [  
  console.log(action)  
  switch (action.type) {
```

```

        case 'GOOD':
            return state
        case 'OK':
            return state
        case 'BAD':
            return state
        case 'ZERO':
            return state
        default: return state
    }
}

export default counterReducer

```

and a base for its tests

```

import deepFreeze from 'deep-freeze'
import counterReducer from './reducer'

describe('unicafe reducer', () => {
    const initialState = {
        good: 0,
        ok: 0,
        bad: 0
    }

    test('should return a proper initial state when called with undefined state', () => {
        const state = {}
        const action = {
            type: 'DO NOTHING'
        }

        const newState = counterReducer(undefined, action)
        expect(newState).toEqual(initialState)
    })

    test('good is incremented', () => {
        const action = {
            type: 'GOOD'
        }
        const state = initialState

        deepFreeze(state)
        const newState = counterReducer(state, action)
        expect(newState).toEqual({
            good: 1,
            ok: 0,
            bad: 0
        })
    })
})

```



Implement the reducer and its tests.

In the tests, make sure that the reducer is an *immutable function* with the *deep-freeze* library. Ensure that the provided first test passes, because Redux expects that the reducer returns the original state when it is called with a first parameter - which represents the previous *state* - with the value *undefined*.

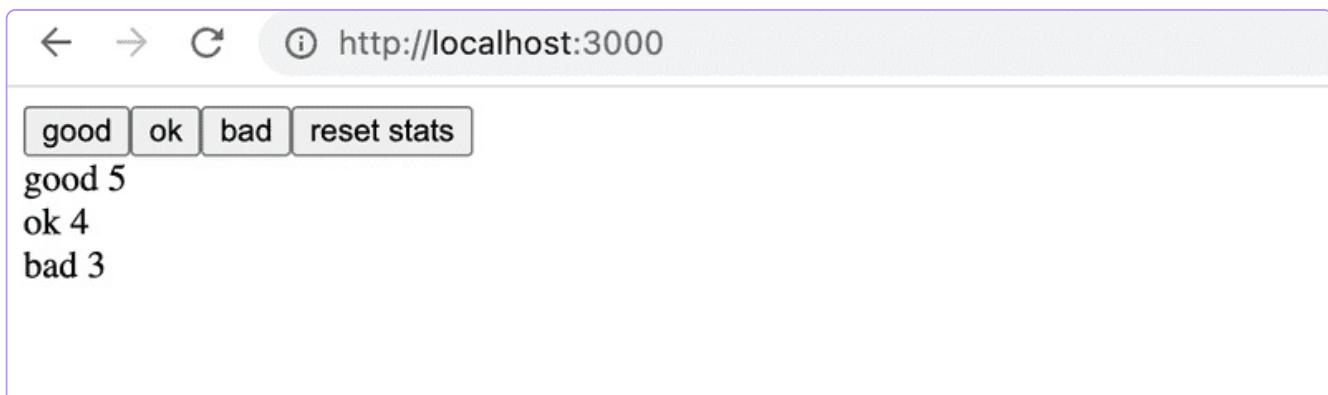
Start by expanding the reducer so that both tests pass. Then add the rest of the tests, and finally the functionality that they are testing.

A good model for the reducer is the [redux-notes](#) example above.

6.2: Unicafe Revisited, step2

Now implement the actual functionality of the application.

Your application can have a modest appearance, nothing else is needed but buttons and the number of reviews for each type:



Uncontrolled form

Let's add the functionality for adding new notes and changing their importance:

```
const generateId = () => Number((Math.random() * 1000000).toFixed(0))
const App = () => {
  const addNote = (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    store.dispatch({
      type: 'NEW_NOTE',
      payload: { content, important: false, id: generateId() }
    })
  }
  const toggleImportance = (id) => {
    store.dispatch({
      type: 'TOGGLE_IMPORTANCE',
      payload: { id }
    })
  }
  return (
    <div>
      <form onSubmit={addNote}>
        <input name="note" />
        <button type="submit">add</button>
      </form>
      <ul>
        {store.getState().map(note =>
          <li key={note.id}>
            {note.content}
            <span>
              {note.important ? 'important' : 'not important'}
              <button onClick={() => toggleImportance(note.id)}>{note.important ? 'less important' : 'more important'}</button>
            </span>
          </li>
        )}
      </ul>
    </div>
  )
}
```

```

        onClick={() => toggleImportance(note.id)} >
      {note.content} <strong>{note.important ? 'important' : ''}</strong>
    </li>
  ) []
</ul>
</div>
)
}

```

The implementation of both functionalities is straightforward. It is noteworthy that we *have not* bound the state of the form fields to the state of the *App* component like we have previously done. React calls this kind of form **uncontrolled**.

Uncontrolled forms have certain limitations (for example, dynamic error messages or disabling the submit button based on input are not possible). However they are suitable for our current needs.

You can read more about uncontrolled forms [here](#).

The method for adding new notes is simple, it just dispatches the action for adding notes:

```

addNote = (event) => {
  event.preventDefault()
  const content = event.target.note.value  event.target.note.value = ''
  store.dispatch([
    type: 'NEW_NOTE',
    payload: {
      content,
      important: false,
      id: generateId()
    }
  ])
}

```

We can get the content of the new note straight from the form field. Because the field has a name, we can access the content via the event object *event.target.note.value*.

```

<form onSubmit={addNote}>
  <input name="note" />  <button type="submit">add</button>
</form>

```

A note's importance can be changed by clicking its name. The event handler is very simple:

```

toggleImportance = (id) => [
  store.dispatch([
    type: 'TOGGLE_IMPORTANCE',
    payload: { id }
  ])
]

```

```
    })
}
```

Action creators

We begin to notice that, even in applications as simple as ours, using Redux can simplify the frontend code. However, we can do a lot better.

React components don't need to know the Redux action types and forms. Let's separate creating actions into separate functions:

```
const createNote = (content) => [
  return {
    type: 'NEW_NOTE',
    payload: {
      content,
      important: false,
      id: generateId()
    }
  }
]

const toggleImportanceOf = (id) => [
  return {
    type: 'TOGGLE_IMPORTANCE',
    payload: [ id ]
  }
]
```

Functions that create actions are called [action creators](#).

The *App* component does not have to know anything about the inner representation of the actions anymore, it just gets the right action by calling the creator function:

```
const App = () => [
  const addNote = (event) => [
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    store.dispatch(createNote(content))
  ]

  const toggleImportance = (id) => [
    store.dispatch(toggleImportanceOf(id)) 
  ]

  // ...
]
```

Forwarding Redux Store to various components

Aside from the reducer, our application is in one file. This is of course not sensible, and we should separate *App* into its module.

Now the question is, how can the *App* access the store after the move? And more broadly, when a component is composed of many smaller components, there must be a way for all of the components to access the store. There are multiple ways to share the Redux store with the components. First, we will look into the newest, and possibly the easiest way, which is using the [hooks](#) API of the [react-redux](#) library.

First, we install react-redux

```
npm install react-redux
```

Next, we move the *App* component into its own file *App.jsx*. Let's see how this affects the rest of the application files.

main.jsx becomes:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './App'
import noteReducer from './reducers/noteReducer'

const store = createStore(noteReducer)

ReactDOM.createRoot(document.getElementById('root')).render(
  <Provider store={store}>      <App />
  </Provider>
```

Note, that the application is now defined as a child of a [Provider](#) component provided by the react-redux library. The application's store is given to the Provider as its attribute *store*.

Defining the action creators has been moved to the file *reducers/noteReducer.js* where the reducer is defined. That file looks like this:

```
const noteReducer = (state = [], action) => [
  // ...
]

const generateId = () =>
  Number((Math.random() * 1000000).toFixed(0))

export const createNote = (content) => {
  return {
    type: 'NEW_NOTE',
    payload: {
      content,
      important: false,
      id: generateId()
    }
}
```

```

}
}

export const toggleImportanceOf = (id) => {
  return {
    type: 'TOGGLE_IMPORTANCE',
    payload: { id }
  }
}

export default noteReducer

```

If the application has many components which need the store, the *App* component must pass *store* as props to all of those components.

The module now has multiple *export* commands.

The reducer function is still returned with the *export default* command, so the reducer can be imported the usual way:

```
import noteReducer from './reducers/noteReducer'
```

A module can have only *one default export*, but multiple "normal" exports

```

export const createNote = (content) => {
  // ...
}

export const toggleImportanceOf = (id) => {
  // ...
}

```

Normally (not as defaults) exported functions can be imported with the curly brace syntax:

```
import { createNote } from '../../reducers/noteReducer'
```

Code for the *App* component

```

import { createNote, toggleImportanceOf } from './reducers/noteReducer'
import { useSelector, useDispatch } from 'react-redux'
const App = () => {
  const dispatch = useDispatch()  const notes = useSelector(state => state)
  const addNote = (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    dispatch(createNote(content))  }

  const toggleImportance = (id) => {
    dispatch(toggleImportanceOf(id))  }

```

```

    return [
      <div>
        <form onSubmit={addNote}>
          <input name="note" />
          <button type="submit">add</button>
        </form>
        <ul>
          {notes.map(note =>
            <li key={note.id}
                onClick={() => toggleImportance(note.id)}>
              {note.content} <strong>{note.important ? 'important' : ''}</strong>
            </li>
          )}
        </ul>
      </div>
    ]
  }

export default App

```

There are a few things to note in the code. Previously the code dispatched actions by calling the `dispatch` method of the Redux store:

```

store.dispatch({
  type: 'TOGGLE_IMPORTANCE',
  payload: { id }
})

```

Now it does it with the `dispatch` function from the `useDispatch` hook.

```

import { useSelector, useDispatch } from 'react-redux'
const App = () => {
  const dispatch = useDispatch() // ...

  const toggleImportance = (id) => {
    dispatch(toggleImportanceOf(id))
  }

  // ...
}

```

The `useDispatch` hook provides any React component access to the `dispatch` function of the Redux store defined in `main.jsx`. This allows all components to make changes to the state of the Redux store.

The component can access the notes stored in the store with the `useSelector`-hook of the `react-redux` library.

```

import { useSelector, useDispatch } from 'react-redux'
const App = () => {
  // ...

```

```
const notes = useSelector(state => state) // ...  
[]
```

`useSelector` receives a function as a parameter. The function either searches for or selects data from the Redux store. Here we need all of the notes, so our selector function returns the whole state:

```
state => state
```

which is a shorthand for:

```
(state) => [  
  return state  
]
```

Usually, selector functions are a bit more interesting and return only selected parts of the contents of the Redux store. We could for example return only notes marked as important:

```
const importantNotes = useSelector(state => state.filter(note => note.important))
```

The current version of the application can be found on [GitHub](#), branch *part6-0*.

More components

Let's separate creating a new note into a component.

```
import { useDispatch } from 'react-redux' import { createNote } from '../reducers/  
noteReducer'  
const NewNote = () => {  
  const dispatch = useDispatch()  
  const addNote = (event) => {  
    event.preventDefault()  
    const content = event.target.note.value  
    event.target.note.value = ''  
    dispatch(createNote(content))  }  
  
  return [  
    <form onSubmit={addNote}>  
      <input name="note" />  
      <button type="submit">add</button>  
    </form>  
  ]  
}  
  
export default NewNote
```

Unlike in the React code we did without Redux, the event handler for changing the state of the app (which now lives in Redux) has been moved away from the *App* to a child component. The logic for

changing the state in Redux is still neatly separated from the whole React part of the application.

We'll also separate the list of notes and displaying a single note into their own components (which will both be placed in the *Notes.jsx* file):

```
import { useDispatch, useSelector } from 'react-redux' import { toggleImportanceOf } from '../reducers/noteReducer'
const Note = ({ note, handleClick }) => {
  return (
    <li onClick={handleClick}>
      {note.content}
      <strong> {note.important ? 'important' : ''} </strong>
    </li>
  )
}

const Notes = () => {
  const dispatch = useDispatch()  const notes = useSelector(state => state)
  return (
    <ul>
      {notes.map(note =>
        <Note
          key={note.id}
          note={note}
          handleClick={() =>
            dispatch(toggleImportanceOf(note.id))
          }
        />
      )}
    </ul>
  )
}

export default Notes
```

The logic for changing the importance of a note is now in the component managing the list of notes.

There is not much code left in *App*:

```
const App = () => {

  return (
    <div>
      <NewNote />
      <Notes />
    </div>
  )
}
```

Note, responsible for rendering a single note, is very simple and is not aware that the event handler it gets as props dispatches an action. These kinds of components are called **presentational** in React

terminology.

Notes, on the other hand, is a [container](#) component, as it contains some application logic: it defines what the event handlers of the *Note* components do and coordinates the configuration of *presentational* components, that is, the *Notes*.

The code of the Redux application can be found on [GitHub](#), on the branch *part6-1*.

Exercises 6.3.-6.8.

Let's make a new version of the anecdote voting application from part 1. Take the project from this repository <https://github.com/fullstack-hy2020/redux-anecdotes> as the base of your solution.

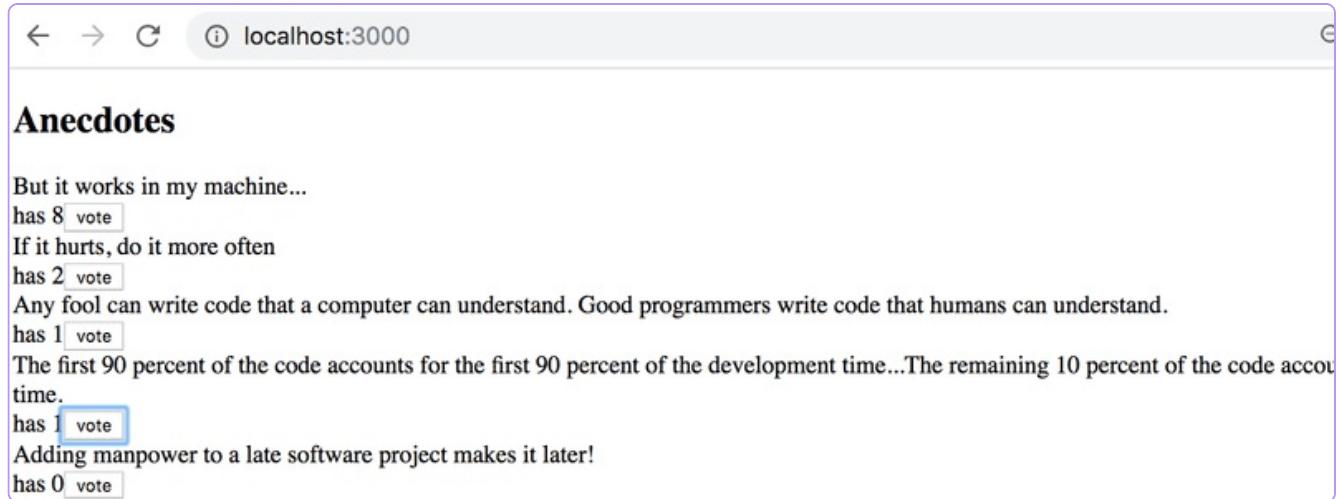
If you clone the project into an existing git repository, *remove the git configuration of the cloned application*:

```
cd redux-anecdotes // go to the cloned repository  
rm -rf .git
```

The application can be started as usual, but you have to install the dependencies first:

```
npm install  
npm run dev
```

After completing these exercises, your application should look like this:



6.3: Anecdotes, step 1

Implement the functionality for voting anecdotes. The number of votes must be saved to a Redux store.

6.4: Anecdotes, step 2

Implement the functionality for adding new anecdotes.

You can keep the form uncontrolled like we did [earlier](#).

6.5: Anecdotes, step 3

Make sure that the anecdotes are ordered by the number of votes.

6.6: Anecdotes, step 4

If you haven't done so already, separate the creation of action-objects to [action creator](#)-functions and place them in the `src/reducers/anecdoteReducer.js` file, so do what we have been doing since the chapter [action creators](#).

6.7: Anecdotes, step 5

Separate the creation of new anecdotes into a component called `AnecdoteForm`. Move all logic for creating a new anecdote into this new component.

6.8: Anecdotes, step 6

Separate the rendering of the anecdote list into a component called `AnecdoteList`. Move all logic related to voting for an anecdote to this new component.

Now the `App` component should look like this:

```
import AnecdoteForm from './components/AnecdoteForm'
import AnecdoteList from './components/AnecdoteList'

const App = () => [
  return [
    <div>
      <h2>Anecdotes</h2>
      <AnecdoteList />
      <AnecdoteForm />
    </div>
  ]
]

export default App
```

6b: Many reducers

Let's continue our work with the simplified [Redux version](#) of our notes application.

To ease our development, let's change our reducer so that the store gets initialized with a state that contains a couple of notes:

```
const initialState = [
  {
    content: 'reducer defines how redux store works',
    important: true,
    id: 1,
  },
]
```

```

        },
        content: 'state of store can contain any data',
        important: false,
        id: 2,
    },
],
]

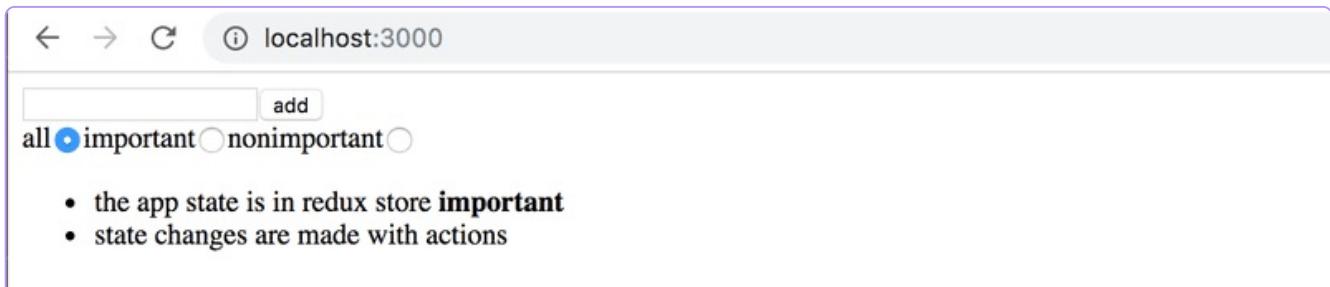
const noteReducer = (state = initialState, action) => {
    // ...
}

// ...
export default noteReducer

```

Store with complex state

Let's implement filtering for the notes that are displayed to the user. The user interface for the filters will be implemented with [radio buttons](#):



Let's start with a very simple and straightforward implementation:

```

import NewNote from './components/NewNote'
import Notes from './components/Notes'

const App = () => {
    const filterSelected = [value] => { console.log(value) }
    return (
        <div>
            <NewNote />
            <div> all <input type="radio" name="filter" onChange={() => filterSelected('ALL')} /> important <input type="radio" name="filter" onChange={() => filterSelected('IMPORTANT')} /> nonimportant <input type="radio" name="filter" onChange={() => filterSelected('NONIMPORTANT')} /> </div>
            <Notes />
        </div>
    )
}

```

Since the `name` attribute of all the radio buttons is the same, they form a *button group* where only one option can be selected.

The buttons have a change handler that currently only prints the string associated with the clicked

button to the console.

In the following section, we will implement filtering by storing both the notes as well as *the value of the filter* in the redux store. When we are finished, we would like the state of the store to look like this:

```
{  
  notes: [  
    { content: 'reducer defines how redux store works', important: true, id: 1},  
    { content: 'state of store can contain any data', important: false, id: 2}  
  ],  
  filter: 'IMPORTANT'  
}
```

Only the array of notes was stored in the state of the previous implementation of our application. In the new implementation, the state object has two properties, *notes* that contains the array of notes and *filter* that contains a string indicating which notes should be displayed to the user.

Combined reducers

We could modify our current reducer to deal with the new shape of the state. However, a better solution in this situation is to define a new separate reducer for the state of the filter:

```
const filterReducer = (state = 'ALL', action) => [  
  switch (action.type) {  
    case 'SET_FILTER':  
      return action.payload  
    default:  
      return state  
  }  
]
```

The actions for changing the state of the filter look like this:

```
{  
  type: 'SET_FILTER',  
  payload: 'IMPORTANT'  
}
```

Let's also create a new *action creator* function. We will write its code in a new *src/reducers/filterReducer.js* module:

```
const filterReducer = (state = 'ALL', action) => [  
  // ...  
]  
  
export const filterChange = filter => [  
  return [  
    type: 'SET_FILTER',  
    payload: filter,  
  ]  
]
```

```
    }
}

export default filterReducer
```

We can create the actual reducer for our application by combining the two existing reducers with the [combineReducers](#) function.

Let's define the combined reducer in the `main.jsx` file:

```
import ReactDOM from 'react-dom/client'
import { createStore, combineReducers } from 'redux' import { Provider } from 'react-redux'
import App from './App'

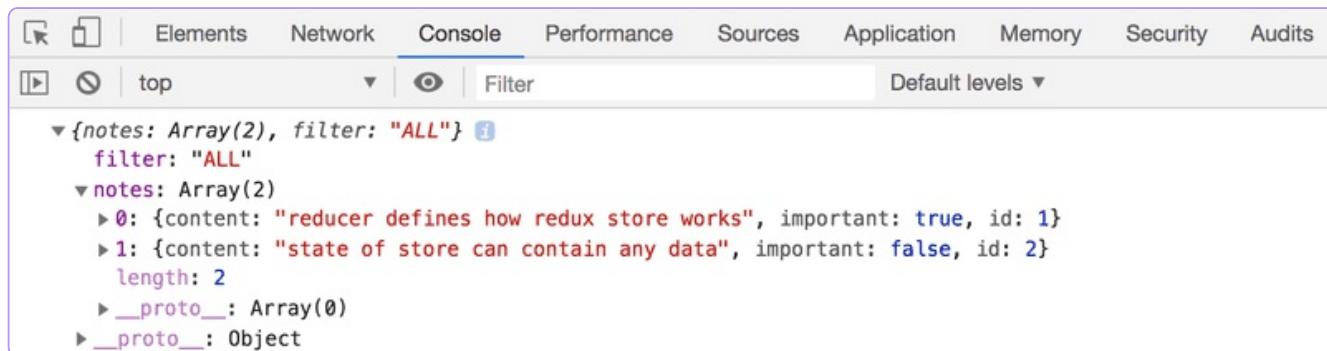
import noteReducer from './reducers/noteReducer'
import filterReducer from './reducers/filterReducer'
const reducer = combineReducers({ notes: noteReducer, filter: filterReducer })
const store = createStore(reducer)
console.log(store.getState())

/*
ReactDOM.createRoot(document.getElementById('root')).render(
  <Provider store={store}>
    <App />
  </Provider>
) */

ReactDOM.createRoot(document.getElementById('root')).render(
  <Provider store={store}>
    <div />
  </Provider>
)
```

Since our application breaks completely at this point, we render an empty `div` element instead of the `App` component.

The state of the store gets printed to the console:



As we can see from the output, the store has the exact shape we wanted it to!

Let's take a closer look at how the combined reducer is created:

```

const reducer = combineReducers([
  notes: noteReducer,
  filter: filterReducer,
])

```

The state of the store defined by the reducer above is an object with two properties: *notes* and *filter*. The value of the *notes* property is defined by the *noteReducer*, which does not have to deal with the other properties of the state. Likewise, the *filter* property is managed by the *filterReducer*.

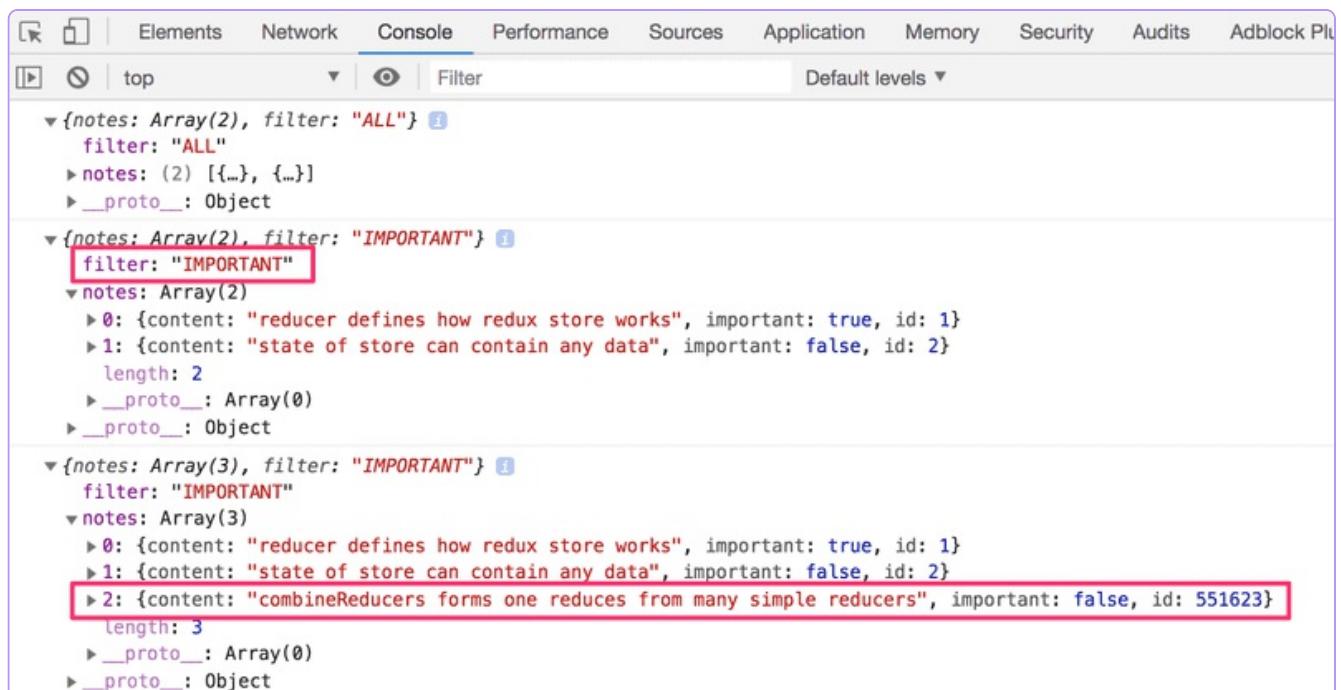
Before we make more changes to the code, let's take a look at how different actions change the state of the store defined by the combined reducer. Let's add the following to the *main.jsx* file:

```

import { createNote } from './reducers/noteReducer'
import { filterChange } from './reducers/filterReducer'
//...
store.subscribe(() => console.log(store.getState()))
store.dispatch(filterChange('IMPORTANT'))
store.dispatch(createNote('combineReducers forms one reducer from many simple reducers'))

```

By simulating the creation of a note and changing the state of the filter in this fashion, the state of the store gets logged to the console after every change that is made to the store:



At this point, it is good to become aware of a tiny but important detail. If we add a console log statement *to the beginning of both reducers*:

```

const filterReducer = (state = 'ALL', action) => {
  console.log('ACTION: ', action)
  // ...
}

```

Based on the console output one might get the impression that every action gets duplicated:

Is there a bug in our code? No. The combined reducer works in such a way that every *action* gets handled in *every* part of the combined reducer, or in other words, every reducer "listens" to all of the dispatched actions and does something with them if it has been instructed to do so. Typically only one reducer is interested in any given action, but there are situations where multiple reducers change their respective parts of the state based on the same action.

Finishing the filters

Let's finish the application so that it uses the combined reducer. We start by changing the rendering of the application and hooking up the store to the application in the `main.jsx` file:

Next, let's fix a bug that is caused by the code expecting the application store to be an array of notes:

```
✖ ▶ Uncaught TypeError: notes.map is not a function
    at Notes (Notes.js:19:14)
    at renderWithHooks (react-dom.development.js:16305:18)
    at mountIndeterminateComponent (react-dom.development.js:20074:13)
    at beginWork (react-dom.development.js:21587:16)
    at HTMLUnknownElement.callCallback2 (react-dom.development.js:4164:14)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:16)
    at invokeGuardedCallback (react-dom.development.js:4277:31)
    at beginWork$1 (react-dom.development.js:27451:7)
    at performUnitOfWork (react-dom.development.js:26557:12)
    at workLoopSync (react-dom.development.js:26466:5)
```

It's an easy fix. Because the notes are in the store's field *notes*, we only have to make a little change to the selector function:

```

        handleClick={() =>
          dispatch(toggleImportanceOf(note.id))
        }
      />
    ) []
  </ul>
)
}

```

Previously the selector function returned the whole state of the store:

```
const notes = useSelector(state => state)
```

And now it returns only its field *notes*

```
const notes = useSelector(state => state.notes)
```

Let's extract the visibility filter into its own *src/components/VisibilityFilter.jsx* component:

```

import { filterChange } from '../reducers/filterReducer'
import { useDispatch } from 'react-redux'

const VisibilityFilter = (props) => {
  const dispatch = useDispatch()

  return (
    <div>
      all
      <input
        type="radio"
        name="filter"
        onChange={() => dispatch(filterChange('ALL'))}
      />
      important
      <input
        type="radio"
        name="filter"
        onChange={() => dispatch(filterChange('IMPORTANT'))}
      />
      nonimportant
      <input
        type="radio"
        name="filter"
        onChange={() => dispatch(filterChange('NONIMPORTANT'))}
      />
    </div>
  )
}

export default VisibilityFilter

```

With the new component, *App* can be simplified as follows:

```
import Notes from './components/Notes'
import NewNote from './components/NewNote'
import VisibilityFilter from './components/VisibilityFilter'

const App = () => [
  return [
    <div>
      <NewNote />
      <VisibilityFilter />
      <Notes />
    </div>
  ]
]

export default App
```

The implementation is rather straightforward. Clicking the different radio buttons changes the state of the store's *filter* property.

Let's change the *Notes* component to incorporate the filter:

```
const Notes = () => {
  const dispatch = useDispatch()
  const notes = useSelector(state => [
    if (state.filter === 'ALL') [
      return state.notes
    ] else if (state.filter === 'IMPORTANT') [
      state.notes.filter(note => note.important)
    ] else [
      state.notes.filter(note => !note.important)
    ]
  ])
  return (
    <ul>
      {notes.map(note =>
        <Note
          key={note.id}
          note={note}
          handleClick={() =>
            dispatch(toggleImportanceOf(note.id))
          }
        />
      )}
    </ul>
  )
}
```

We only make changes to the selector function, which used to be

```
useSelector(state => state.notes)
```

Let's simplify the selector by destructuring the fields from the state it receives as a parameter:

```
const notes = useSelector(({ filter, notes }) => {
```

```

if ( filter === 'ALL' ) {
  return notes
}
return filter === 'IMPORTANT'
  ? notes.filter(note => note.important)
  : notes.filter(note => !note.important)
})

```

There is a slight cosmetic flaw in our application. Even though the filter is set to *ALL* by default, the associated radio button is not selected. Naturally, this issue can be fixed, but since this is an unpleasant but ultimately harmless bug we will save the fix for later.

The current version of the application can be found on [GitHub](#), branch *part6-2*.

Exercise 6.9

6.9 Better Anecdotes, step 7

Implement filtering for the anecdotes that are displayed to the user.

Store the state of the filter in the redux store. It is recommended to create a new reducer, action creators, and a combined reducer for the store using the *combineReducers* function.

Create a new *Filter* component for displaying the filter. You can use the following code as a template for the component:

```

const Filter = () => {
  const handleChange = (event) => {
    // input-field value is in variable event.target.value
  }
  const style = [
    marginBottom: 10
  ]

  return (
    <div style={style}>
      filter <input onChange={handleChange} />
    </div>
  )
}

```

```
    }
}

export default Filter
```

Redux Toolkit

As we have seen so far, Redux's configuration and state management implementation requires quite a lot of effort. This is manifested for example in the reducer and action creator-related code which has somewhat repetitive boilerplate code. [Redux Toolkit](#) is a library that solves these common Redux-related problems. The library for example greatly simplifies the configuration of the Redux store and offers a large variety of tools to ease state management.

Let's start using Redux Toolkit in our application by refactoring the existing code. First, we will need to install the library:

```
npm install @reduxjs/toolkit
```

Next, open the *main.jsx* file which currently creates the Redux store. Instead of Redux's *createStore* function, let's create the store using Redux Toolkit's *configureStore* function:

```
import ReactDOM from 'react-dom/client'
import { Provider } from 'react-redux'
import { configureStore } from '@reduxjs/toolkit' import App from './App'

import noteReducer from './reducers/noteReducer'
import filterReducer from './reducers/filterReducer'

const store = configureStore({ reducer: { notes: noteReducer, filter: filterReducer } })
console.log(store.getState())

ReactDOM.createRoot(document.getElementById('root')).render(
  <Provider store={store}>
    <App />
  </Provider>
)
```

We already got rid of a few lines of code, now we don't need the *combineReducers* function to create the store's reducer. We will soon see that the *configureStore* function has many additional benefits such as the effortless integration of development tools and many commonly used libraries without the need for additional configuration.

Let's move on to refactoring the reducers, which brings forth the benefits of the Redux Toolkit. With Redux Toolkit, we can easily create reducer and related action creators using the *createSlice* function. We can use the *createSlice* function to refactor the reducer and action creators in the *reducers/noteReducer.js* file in the following manner:

```
import { createSlice } from '@reduxjs/toolkit'
const initialState = [
```

```

[{
  content: 'reducer defines how redux store works',
  important: true,
  id: 1,
},
{
  content: 'state of store can contain any data',
  important: false,
  id: 2,
},
]

const generateId = () =>
  Number((Math.random() * 1000000).toFixed(0))

const noteSlice = createSlice({
  name: 'notes',
  initialState: [],
  reducers: {
    createNote(state, action) {
      const content = action.payload
      state.push({
        content,
        important: false,
        id: generateId(),
      })
    },
    toggleImportanceOf(state, action) {
      const id = action.payload
      const noteToChange = state.find(n => n.id === id)
      const changedNote = {
        ...noteToChange,
        important: !noteToChange.important
      }
      return state.map(note =>
        note.id !== id ? note : changedNote
      )
    }
  }
})

```

The `createSlice` function's `name` parameter defines the prefix which is used in the action's type values. For example, the `createNote` action defined later will have the type value of `notes/createNote`. It is a good practice to give the parameter a value which is unique among the reducers. This way there won't be unexpected collisions between the application's action type values. The `initialState` parameter defines the reducer's initial state. The `reducers` parameter takes the reducer itself as an object, of which functions handle state changes caused by certain actions. Note that the `action.payload` in the function contains the argument provided by calling the action creator:

```
dispatch(createNote('Redux Toolkit is awesome!'))
```

This dispatch call is equivalent to dispatching the following object:

```
dispatch({ type: 'notes/createNote', payload: 'Redux Toolkit is awesome!' })
```

If you followed closely, you might have noticed that inside the `createNote` action, there seems to happen something that violates the reducers' immutability principle mentioned earlier:

```

createNote(state, action) {
  const content = action.payload

  state.push({
    content,
    important: false,
    id: generateId(),
  })
}

```

We are mutating *state* argument's array by calling the *push* method instead of returning a new instance of the array. What's this all about?

Redux Toolkit utilizes the [Immer](#) library with reducers created by *createSlice* function, which makes it possible to mutate the *state* argument inside the reducer. Immer uses the mutated state to produce a new, immutable state and thus the state changes remain immutable. Note that *state* can be changed without "mutating" it, as we have done with the *toggleImportanceOf* action. In this case, the function directly *returns* the new state. Nevertheless mutating the state will often come in handy especially when a complex state needs to be updated.

The *createSlice* function returns an object containing the reducer as well as the action creators defined by the *reducers* parameter. The reducer can be accessed by the *noteSlice.reducer* property, whereas the action creators by the *noteSlice.actions* property. We can produce the file's exports in the following way:

```
const noteSlice = createSlice/* ... */  
  
export const { createNote, toggleImportanceOf } = noteSlice.actions  
export default noteSlice.reducer
```

The imports in other files will work just as they did before:

```
import noteReducer, { createNote, toggleImportanceOf } from './reducers/noteReducer'
```

We need to alter the action type names in the tests due to the conventions of ReduxToolkit:

```
import noteReducer from './noteReducer'  
import deepFreeze from 'deep-freeze'  
  
describe('noteReducer', () => [  
  test('returns new state with action notes/createNote', () => [  
    const state = []  
    const action = [  
      type: 'notes/createNote', payload: 'the app state is in redux store'  
    ]  
  
    deepFreeze(state)  
    const newState = noteReducer(state, action)  
  
    expect(newState).toHaveLength(1)  
    expect(newState.map(s => s.content)).toContainEqual(action.payload)  
  ])  
  
  test('returns new state with action notes/toggleImportanceOf', () => [  
    const state = [  
      {  
        content: 'the app state is in redux store',  
        importance: true  
      }  
    ]  
    const action = {  
      type: 'notes/toggleImportanceOf',  
      payload: { index: 0 }  
    }  
  
    const newState = noteReducer(state, action)  
  
    expect(newState[0].importance).toBeFalsy()  
  ])  
]
```

```

        important: true,
        id: 1
    ],
    [
        content: 'state changes are made with actions',
        important: false,
        id: 2
    ]
]

const action = [
    type: 'notes/toggleImportanceOf', payload: 2
]

deepFreeze(state)
const newState = noteReducer(state, action)

expect(newState).toHaveLength(2)

expect(newState).toContainEqual(state[0])

expect(newState).toContainEqual([
    content: 'state changes are made with actions',
    important: true,
    id: 2
])
]
]
}
})

```

Redux Toolkit and console.log

As we have learned, `console.log` is an extremely powerful tool; it often saves us from trouble.

Let's try to print the state of the Redux Store to the console in the middle of the reducer created with the function `createSlice`:

```

const noteSlice = createSlice({
    name: 'notes',
    initialState,
    reducers: [
        // ...
        toggleImportanceOf(state, action) {
            const id = action.payload

            const noteToChange = state.find(n => n.id === id)

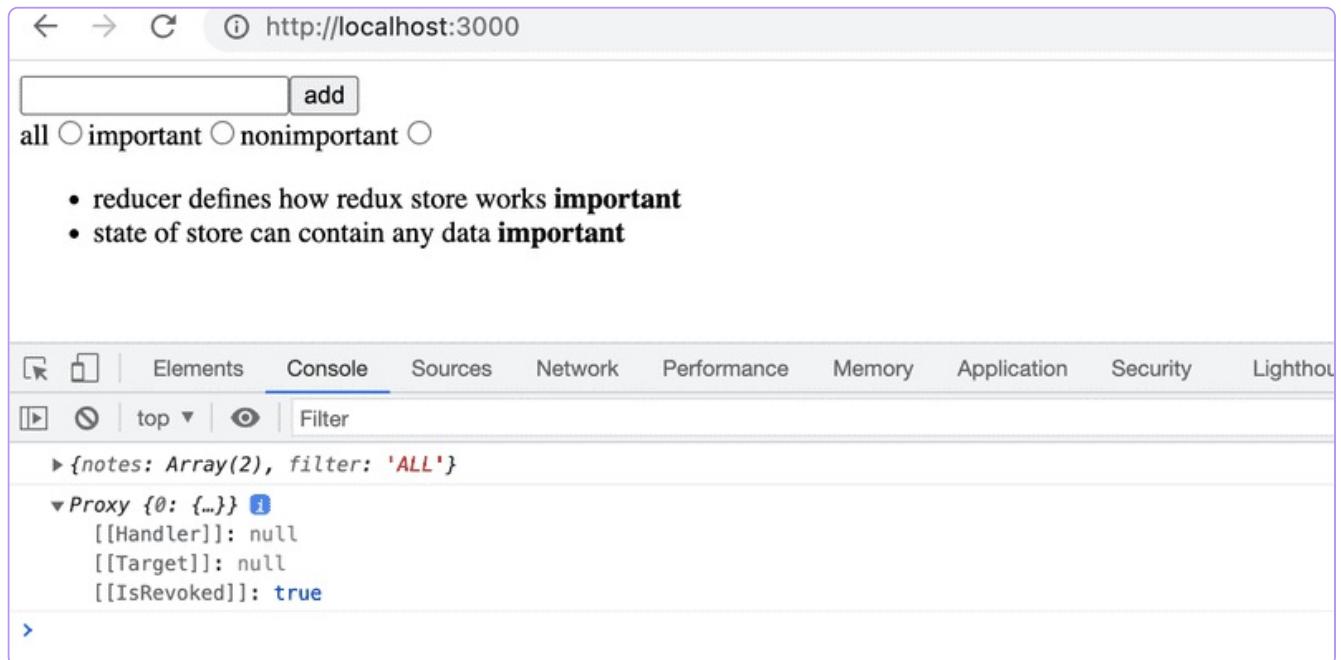
            const changedNote = [
                ...noteToChange,
                important: !noteToChange.important
            ]

            console.log(state)
            return state.map(note =>

```

```
        note.id !== id ? note : changedNote
    )
)
}
})
```

The following is printed to the console



The output is interesting but not very useful. This is about the previously mentioned Immer library used by the Redux Toolkit internally to save the state of the Store.

The status can be converted to a human-readable format by using the `current` function from the `immer` library.

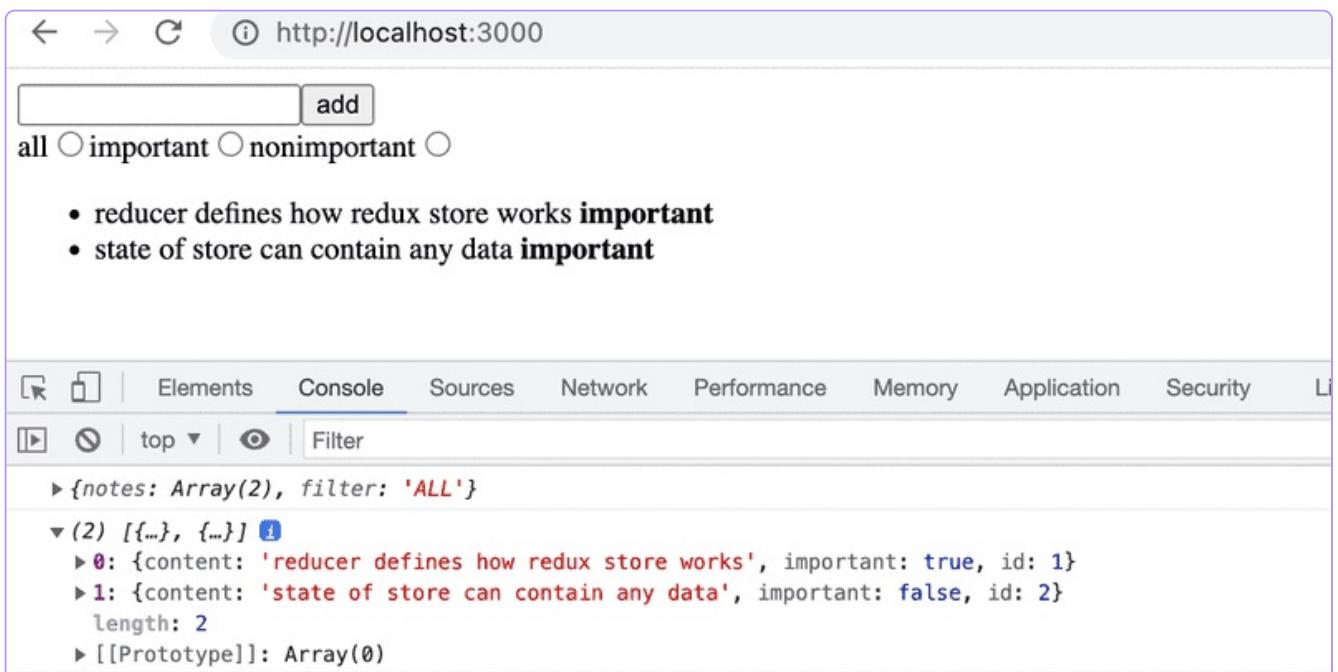
Let's update the imports to include the "current" function from the `immer` library:

```
import { createSlice, current } from '@reduxjs/toolkit'
```

Then we update the `console.log` function call:

```
console.log(current[state])
```

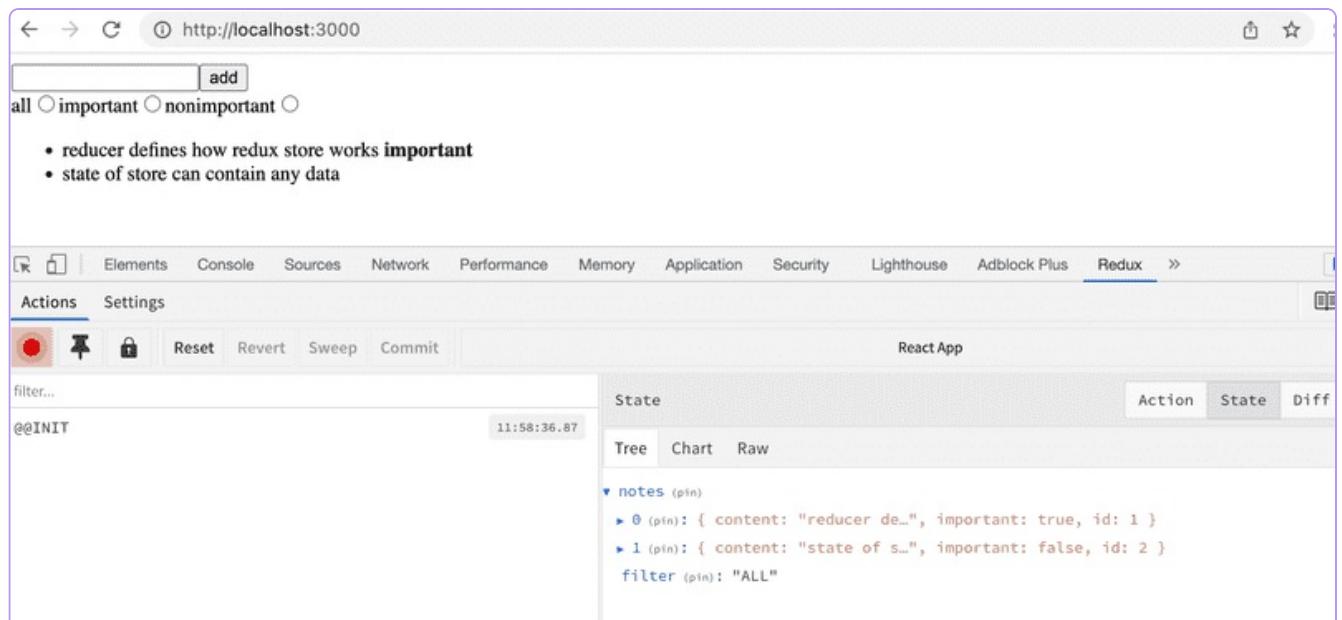
Console output is now human readable



Redux DevTools

[Redux DevTools](#) is a Chrome addon that offers useful development tools for Redux. It can be used for example to inspect the Redux store's state and dispatch actions through the browser's console. When the store is created using Redux Toolkit's `configureStore` function, no additional configuration is needed for Redux DevTools to work.

Once the addon is installed, clicking the *Redux* tab in the browser's developer tools, the Redux DevTools should open:



You can inspect how dispatching a certain action changes the state by clicking the action:

The screenshot shows the Redux DevTools extension in the browser's developer tools. The left pane displays two actions: '@INIT' at timestamp 11:58:36.87 and 'notes/createNote' at timestamp +01:49.05. The right pane shows the state tree for 'notes', which contains three items (id: 1, 2, 3) with their respective content and importance status. A red arrow points from the 'notes/createNote' action in the left pane to the state tree in the right pane.

```

notes
  0 (pin): { content: "reducer de...", important: true, id: 1 }
  1 (pin): { content: "state of s...", important: false, id: 2 }
  2 (pin): { content: "redux tool...", important: false, id: 384562 }
  filter (pin): "ALL"

```

It is also possible to dispatch actions to the store using the development tools:

The screenshot shows the Redux DevTools extension. In the left pane, a 'Custom action' is defined with the following code:

```

1: {
2:   type: 'notes/toggleImportanceOf',
3:   payload: 2
4: }

```

A red box highlights the action code, and a red arrow points to the 'Dispatch' button in the bottom right corner of the DevTools interface.

You can find the code for our current application in its entirety in the *part6-3* branch of [this GitHub repository](#).

Exercises 6.10.-6.13.

Let's continue working on the anecdote application using Redux that we started in exercise 6.3.

6.10 Better Anecdotes, step 8

Install Redux Toolkit for the project. Move the Redux store creation into the file *store.js* and use Redux Toolkit's *configureStore* to create the store.

Change the definition of the *filter reducer and action creators* to use the Redux Toolkit's *createSlice* function.

Also, start using Redux DevTools to debug the application's state easier.

6.11 Better Anecdotes, step 9

Change also the definition of the *anecdote reducer and action creators* to use the Redux Toolkit's `createSlice` function.

Implementation note: when you use the Redux Toolkit to return the initial state of anecdotes, it will be immutable, so you will need to make a copy of it to sort the anecdotes, or you will encounter the error "TypeError: Cannot assign to read only property". You can use the spread syntax to make a copy of the array. Instead of:

```
anecdotes.sort()
```

Write:

```
[...anecdotes].sort()
```

6.12 Better Anecdotes, step 10

The application has a ready-made body for the *Notification* component:

```
const Notification = () => {
  const style = [
    border: 'solid',
    padding: 10,
    borderWidth: 1
  ]
  return (
    <div style={style}>
      render here notification...
    </div>
  )
}

export default Notification
```

Extend the component so that it renders the message stored in the Redux store, making the component take the following form:

```
import { useSelector } from 'react-redux'
const Notification = () => {
  const notification = useSelector(/* something here */)
  const style = [
    border: 'solid',
    padding: 10,
    borderWidth: 1
  ]
  return (
    <div style={style}>
      {notification}
    </div>
  )
}
```

You will have to make changes to the application's existing reducer. Create a separate reducer for the new functionality by using the Redux Toolkit's `createSlice` function.

The application does not have to use the `Notification` component intelligently at this point in the exercises. It is enough for the application to display the initial value set for the message in the `notificationReducer`.

6.13 Better Anecdotes, step 11

Extend the application so that it uses the `Notification` component to display a message for five seconds when the user votes for an anecdote or creates a new anecdote:

The screenshot shows a web browser window with the URL `localhost:3000`. The page title is "Anecdotes". Below the title, there is a notification message: "you voted 'If it hurts, do it more often'". The main content area lists several anecdotes:

- If it hurts, do it more often
has 4
- Adding manpower to a late software project makes it later!
has 0
- The first 90 percent of the code accounts for the first 90 percent of the development time...The remaining 10 percent of the code accounts for the other 90 percent.
has 0
- Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
has 0
- Premature optimization is the root of all evil.
has 0
- Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, debugging it.

It's recommended to create separate `action creators` for setting and removing notifications.

6c: Communicating with server in a Redux application

Let's expand the application so that the notes are stored in the backend. We'll use `json-server`, familiar from part 2.

The initial state of the database is stored in the file `db.json`, which is placed in the root of the project:

```
[{"notes": [{"content": "the app state is in redux store", "important": true, "id": 1}, {"content": "state changes are made with actions", "important": false, "id": 2}]}]
```

We'll install json-server for the project:

```
npm install json-server --save-dev
```

and add the following line to the *scripts* part of the file *package.json*

```
"scripts": [
  "server": "json-server -p3001 --watch db.json",
  // ...
]
```

Now let's launch json-server with the command *npm run server*.

Getting data from the backend

Next, we'll create a method into the file *services/notes.js*, which uses *axios* to fetch data from the backend

```
import axios from 'axios'

const baseUrl = 'http://localhost:3001/notes'

const getAll = async () => {
  const response = await axios.get(baseUrl)
  return response.data
}

export default [ getAll ]
```

We'll add axios to the project

```
npm install axios
```

We'll change the initialization of the state in *noteReducer*, so that by default there are no notes:

```
const noteSlice = createSlice({
  name: 'notes',
  initialState: [],
  // ...
})
```

Let's also add a new action *appendNote* for adding a note object:

```
const noteSlice = createSlice({
  name: 'notes',
  initialState: [],
  reducers: [
    createNote(state, action) {
      const content = action.payload
```

```

state.push([
  content,
  important: false,
  id: generateId(),
])

// ...
toggleImportanceOf(state, action) {
  const id = action.payload

  const noteToChange = state.find(n => n.id === id)

  const changedNote = [
    ...noteToChange,
    important: !noteToChange.important
  ]

  return state.map(note =>
    note.id !== id ? note : changedNote
  )
}

// ...
appendNote(state, action) {
  state.push(action.payload)
}

}

export const { createNote, toggleImportanceOf, appendNote } = noteSlice.actions
export default noteSlice.reducer

```

A quick way to initialize the notes state based on the data received from the server is to fetch the notes in the `main.jsx` file and dispatch an action using the `appendNote` action creator for each individual note object:

```

// ...
import noteService from './services/notes' import noteReducer, { appendNote } from
'./reducers/noteReducer'
const store = configureStore({
  reducer: [
    notes: noteReducer,
    filter: filterReducer,
  ],
})

noteService.getAll().then(notes => notes.forEach(note => {
  store.dispatch(appendNote(note))
}))
// ...

```

Dispatching multiple actions seems a bit impractical. Let's add an action creator `setNotes` which can be used to directly replace the notes array. We'll get the action creator from the `createSlice` function by implementing the `setNotes` action:

```
// ...
```

```

const noteSlice = createSlice({
  name: 'notes',
  initialState: [],
  reducers: {
    createNote(state, action) {
      const content = action.payload

      state.push([
        content,
        important: false,
        id: generateId(),
      ])
    },
    toggleImportanceOf(state, action) {
      const id = action.payload

      const noteToChange = state.find(n => n.id === id)

      const changedNote = [
        ...noteToChange,
        important: !noteToChange.important
      ]

      return state.map(note =>
        note.id !== id ? note : changedNote
      )
    },
    appendNote(state, action) {
      state.push(action.payload)
    },
    setNotes(state, action) {
      return action.payload
    },
  }
}

export const [ createNote, toggleImportanceOf, appendNote, setNotes ] = noteSlice.actions
export default noteSlice.reducer

```

Now, the code in the *main.jsx* file looks a lot better:

```

// ...
import noteService from './services/notes'
import noteReducer, { setNotes } from './reducers/noteReducer'
const store = configureStore({
  reducer: [
    notes: noteReducer,
    filter: filterReducer,
  ],
})

noteService.getAll().then(notes =>

```

```
store.dispatch(setNotes(notes))
```

NB: Why didn't we use `await` in place of promises and event handlers (registered to `then`-methods)?

Await only works inside `async` functions, and the code in `main.jsx` is not inside a function, so due to the simple nature of the operation, we'll abstain from using `async` this time.

We do, however, decide to move the initialization of the notes into the `App` component, and, as usual, when fetching data from a server, we'll use the `effect hook`.

```
import { useEffect } from 'react' import NewNote from './components/NewNote'
import Notes from './components/Notes'
import VisibilityFilter from './components/VisibilityFilter'
import noteService from './services/notes' import { setNotes } from './reducers/noteReducer' import { useDispatch } from 'react-redux'
const App = () => {
  const dispatch = useDispatch()  useEffect(() => {
    noteService.getAll()
    .then(notes => dispatch(setNotes(notes)))
  }, [])
  return (
    <div>
      <NewNote />
      <VisibilityFilter />
      <Notes />
    </div>
  )
}

export default App
```

Sending data to the backend

We can do the same thing when it comes to creating a new note. Let's expand the code communicating with the server as follows:

```
const baseUrl = 'http://localhost:3001/notes'

const getAll = async () => {
  const response = await axios.get(baseUrl)
  return response.data
}

const createNew = async (content) => {
  const object = { content, important: false }
  const response = await axios.post(baseUrl, object)
  return response.data
}

export default [
  getAll,
  createNew,
```

The method `addNote` of the component `NewNote` changes slightly:

```
import { useDispatch } from 'react-redux'
import { createNote } from '../reducers/noteReducer'
import noteService from '../services/notes'
const NewNote = (props) => {
  const dispatch = useDispatch()

  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    const newNote = await noteService.createNew(content)
    dispatch(createNote(newNote))
  }

  return (
    <form onSubmit={addNote}>
      <input name="note" />
      <button type="submit">add</button>
    </form>
  )
}

export default NewNote
```

Because the backend generates ids for the notes, we'll change the action creator `createNote` in the file `noteReducer.js` accordingly:

```
const noteSlice = createSlice({
  name: 'notes',
  initialState: [],
  reducers: {
    createNote(state, action) {
      state.push(action.payload)
    },
    // ...
  },
})
```

Changing the importance of notes could be implemented using the same principle, by making an asynchronous method call to the server and then dispatching an appropriate action.

The current state of the code for the application can be found on [GitHub](#) in the branch `part6-3`.

Exercises 6.14.-6.15.

6.14 Anecdotes and the Backend, step 1

When the application launches, fetch the anecdotes from the backend implemented using json-server.

As the initial backend data, you can use, e.g. [this](#).

6.15 Anecdotes and the Backend, step 2

Modify the creation of new anecdotes, so that the anecdotes are stored in the backend.

Asynchronous actions and Redux Thunk

Our approach is quite good, but it is not great that the communication with the server happens inside the functions of the components. It would be better if the communication could be abstracted away from the components so that they don't have to do anything else but call the appropriate *action creator*. As an example, *App* would initialize the state of the application as follows:

```
const App = () => [
  const dispatch = useDispatch()

  useEffect(() => [
    dispatch(initializeNotes())
  ], [])
}

// ...
```

and *NewNote* would create a new note as follows:

```
const NewNote = () => [
  const dispatch = useDispatch()

  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    dispatch(createNote(content))
  }

  // ...
]
```

In this implementation, both components would dispatch an action without the need to know about the communication with the server that happens behind the scenes. These kinds of *async actions* can be implemented using the [Redux Thunk](#) library. The use of the library doesn't need any additional configuration or even installation when the Redux store is created using the Redux Toolkit's *configureStore* function.

With Redux Thunk it is possible to implement *action creators* which return a function instead of an object. The function receives Redux store's *dispatch* and *getState* methods as parameters. This allows for example implementations of asynchronous action creators, which first wait for the completion of a certain asynchronous operation and after that dispatch some action, which changes the store's state.

We can define an action creator *initializeNotes* which initializes the notes based on the data received from the server:

```
// ...
```

```

import noteService from '../services/notes'
const noteSlice = createSlice({/* ... */})

export const [ createNote, toggleImportanceOf, setNotes, appendNote ] = noteSlice.actions

export const initializeNotes = () => {
  return async dispatch => {
    const notes = await noteService.getAll()
    dispatch(setNotes(notes))
  }
}

export default noteSlice.reducer

```

In the inner function, meaning the *asynchronous action*, the operation first fetches all the notes from the server and then *dispatches* the *setNotes* action, which adds them to the store.

The component *App* can now be defined as follows:

```

// ...
import { initializeNotes } from './reducers/noteReducer'
const App = () => {
  const dispatch = useDispatch()

  useEffect(() => {
    dispatch(initializeNotes())
  }, [])

  return (
    <div>
      <NewNote />
      <VisibilityFilter />
      <Notes />
    </div>
  )
}

```

The solution is elegant. The initialization logic for the notes has been completely separated from the React component.

Next, let's replace the *createNote* action creator created by the *createSlice* function with an asynchronous action creator:

```

// ...
import noteService from '../services/notes'

const noteSlice = createSlice({
  name: 'notes',
  initialState: [],
  reducers: {
    toggleImportanceOf(state, action) {
      const id = action.payload

      const noteToChange = state.find(n => n.id === id)

      const changedNote = {
        ...noteToChange,
        important: !noteToChange.important
      }

      state[id] = changedNote
    }
  }
})

```

```

    }

    return state.map(note =>
      note.id !== id ? note : changedNote
    )
  },
  appendNote(state, action) {
    state.push(action.payload)
  },
  setNotes(state, action) {
    return action.payload
  }
  // createNote definition removed from here!
},
}

export const { toggleImportanceOf, appendNote, setNotes } = noteSlice.actions
export const initializeNotes = () => [
  return async dispatch => {
    const notes = await noteService.getAll()
    dispatch(setNotes(notes))
  }
]

export const createNote = content => {
  return async dispatch => {
    const newNote = await noteService.createNew(content)
    dispatch(appendNote(newNote))
  }
}

```

The principle here is the same: first, an asynchronous operation is executed, after which the action changing the state of the store is *dispatched*.

The component *NewNote* changes as follows:

```

// ...
import { createNote } from '../reducers/noteReducer'
const NewNote = () => {
  const dispatch = useDispatch()

  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    dispatch(createNote(content))
  }

  return (
    <form onSubmit={addNote}>
      <input name="note" />
      <button type="submit">add</button>
    </form>
  )
}

```

Finally, let's clean up the `main.jsx` file a bit by moving the code related to the creation of the Redux store into its own, `store.js` file:

```
import { configureStore } from '@reduxjs/toolkit'

import noteReducer from './reducers/noteReducer'
import filterReducer from './reducers/filterReducer'

const store = configureStore({
  reducer: {
    notes: noteReducer,
    filter: filterReducer
  }
})

export default store
```

After the changes, the content of the `main.jsx` is the following:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import { Provider } from 'react-redux'
import store from './store' import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(
  <Provider store={store}>
    <App />
  </Provider>
)
```

The current state of the code for the application can be found on [GitHub](#) in the branch `part6-5`.

Redux Toolkit offers a multitude of tools to simplify asynchronous state management. Suitable tools for this use case are for example the `createAsyncThunk` function and the `RTK Query` API.

Exercises 6.16.-6.19.

6.16 Anecdotes and the Backend, step 3

Modify the initialization of the Redux store to happen using asynchronous action creators, which are made possible by the Redux Thunk library.

6.17 Anecdotes and the Backend, step 4

Also modify the creation of a new anecdote to happen using asynchronous action creators, made possible by the Redux Thunk library.

6.18 Anecdotes and the Backend, step 5

Voting does not yet save changes to the backend. Fix the situation with the help of the Redux Thunk

library.

6.19 Anecdotes and the Backend, step 6

The creation of notifications is still a bit tedious since one has to do two actions and use the `setTimeout` function:

```
dispatch(setNotification(`new anecdote ${content}`))
setTimeout(() => {
  dispatch(clearNotification())
}, 5000)
```

Make an action creator, which enables one to provide the notification as follows:

```
dispatch(setNotification(`you voted ${anecdote.content}`), 10))
```

The first parameter is the text to be rendered and the second parameter is the time to display the notification given in seconds.

Implement the use of this improved notification in your application.

6d: React Query, useReducer and the context

At the end of this part, we will look at a few more different ways to manage the state of an application.

Let's continue with the note application. We will focus on communication with the server. Let's start the application from scratch. The first version is as follows:

```
const App = () => {
  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    console.log(content)
  }

  const toggleImportance = (note) => {
    console.log('toggle importance of', note.id)
  }

  const notes = []

  return (
    <div>
      <h2>Notes app</h2>
      <form onSubmit={addNote}>
        <input name="note" />
        <button type="submit">add</button>
      </form>
    </div>
  )
}
```

```

    {notes.map(note =>
      <li key={note.id} onClick={() => toggleImportance(note)}>
        {note.content}
        <strong> {note.important ? 'important' : ''}</strong>
      </li>
    )}
  </div>
}

export default App

```

The initial code is on GitHub in this [repository](#), in the branch *part6-0*.

Note: By default, cloning the repo will only give you the main branch. To get the initial code from the part6-0 branch, use the following command:

```
git clone --branch part6-0 https://github.com/fullstack-hy2020/query-notes.git
```

Managing data on the server with the React Query library

We shall now use the [React Query](#) library to store and manage data retrieved from the server. The latest version of the library is also called TanStack Query, but we stick to the familiar name.

Install the library with the command

```
npm install @tanstack/react-query
```

A few additions to the file *main.jsx* are needed to pass the library functions to the entire application:

```

import React from 'react'
import ReactDOM from 'react-dom/client'
import { QueryClient, QueryClientProvider } from '@tanstack/react-query'
import App from './App'

const queryClient = new QueryClient()
ReactDOM.createRoot(document.getElementById('root')).render(
  <QueryClientProvider client={queryClient}>    <App />
</QueryClientProvider>

```

We can now retrieve the notes in the *App* component. The code expands as follows:

```

import { useQuery } from '@tanstack/react-query'
import axios from 'axios'
const App = () => {
  // ...

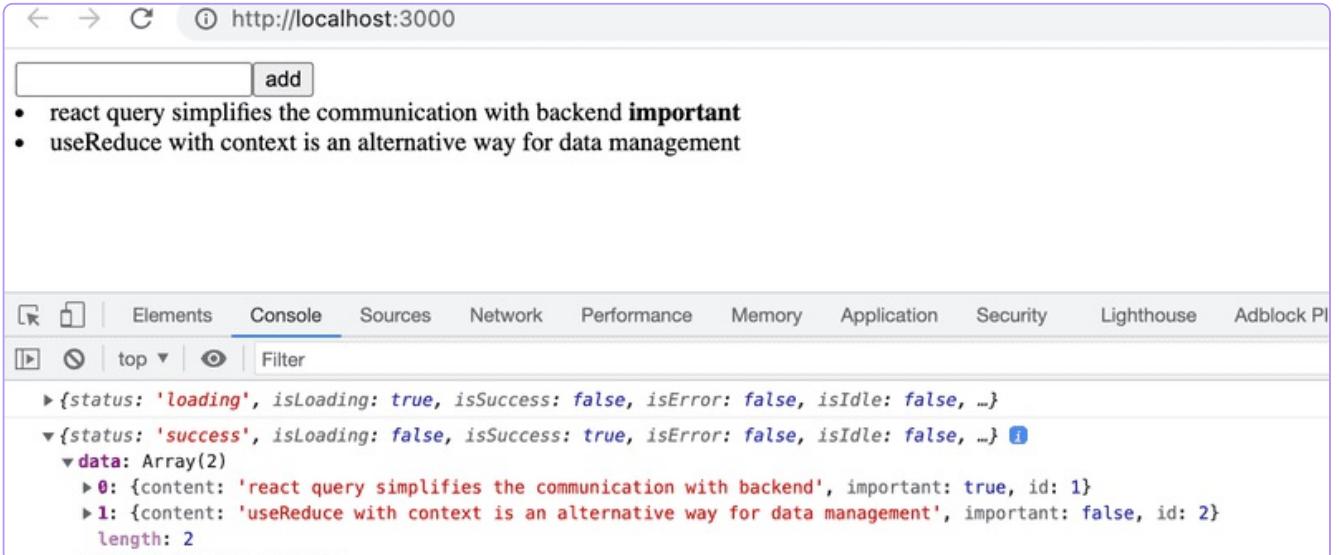
  const result = useQuery({   queryKey: ['notes'],   queryFn: () => axios.get('http://localhost:3001/notes').then(res => res.data) }, console.log(JSON.stringify(result)))
  if (result.isLoading) {    return <div>loading data...</div>  }

```

```
const notes = result.data
return [
  // ...
]
```

Retrieving data from the server is still done in a familiar way with the Axios `get` method. However, the Axios method call is now wrapped in a `query` formed with the `useQuery` function. The first parameter of the function call is a string `notes` which acts as a `key` to the query defined, i.e. the list of notes.

The return value of the `useQuery` function is an object that indicates the status of the query. The output to the console illustrates the situation:



A screenshot of a browser's developer tools console. The URL is `http://localhost:3000`. The console tab is selected. There is a button labeled "add". Below it, a list of items is shown:

- react query simplifies the communication with backend **important**
- useReduce with context is an alternative way for data management

Further down, the console shows the state of a query object:

```
▶ {status: 'loading', isLoading: true, isSuccess: false, isError: false, isIdle: false, ...}
▼ {status: 'success', isLoading: false, isSuccess: true, isError: false, isIdle: false, ...} ⓘ
  ▼ data: Array(2)
    ▶ 0: {content: 'react query simplifies the communication with backend', important: true, id: 1}
    ▶ 1: {content: 'useReduce with context is an alternative way for data management', important: false, id: 2}
    length: 2
```

That is, the first time the component is rendered, the query is still in *loading* state, i.e. the associated HTTP request is pending. At this stage, only the following is rendered:

```
<div>loading data...</div>
```

However, the HTTP request is completed so quickly that not even Max Verstappen would be able to see the text. When the request is completed, the component is rendered again. The query is in the state *success* on the second rendering, and the field `data` of the query object contains the data returned by the request, i.e. the list of notes that is rendered on the screen.

So the application retrieves data from the server and renders it on the screen without using the React hooks `useState` and `useEffect` used in chapters 2-5 at all. The data on the server is now entirely under the administration of the React Query library, and the application does not need the state defined with React's `useState` hook at all!

Let's move the function making the actual HTTP request to its own file `requests.js`

```
import axios from 'axios'

export const getNotes = () =>
  axios.get('http://localhost:3001/notes').then(res => res.data)
```

The `App` component is now slightly simplified

```
import { useQuery } from '@tanstack/react-query'
import { getNotes } from './requests'
const App = () => {
  // ...

  const result = useQuery({
    queryKey: ['notes'],
    queryFn: getNotes
  })

  // ...
}
```

The current code for the application is in [GitHub](#) in the branch `part6-1`.

Synchronizing data to the server using React Query

Data is already successfully retrieved from the server. Next, we will make sure that the added and modified data is stored on the server. Let's start by adding new notes.

Let's make a function `createNote` to the file `requests.js` for saving new notes:

```
import axios from 'axios'

const baseUrl = 'http://localhost:3001/notes'

export const getNotes = () =>
  axios.get(baseUrl).then(res => res.data)

export const createNote = newNote => axios.post(baseUrl, newNote).then(res => res.data)
```

The `App` component will change as follows

```
import { useQuery, useMutation } from '@tanstack/react-query'
import { getNotes, createNote } from './requests'
const App = () => {
  const newNoteMutation = useMutation({ mutationFn: createNote })
  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    newNoteMutation.mutate({ content, important: true })
  }
}
```

To create a new note, a `mutation` is defined using the function `useMutation`:

```
const newNoteMutation = useMutation({ mutationFn: createNote })
```

The parameter is the function we added to the file *requests.js*, which uses Axios to send a new note to the server.

The event handler *addNote* performs the mutation by calling the mutation object's function *mutate* and passing the new note as an argument:

```
newNoteMutation.mutate({ content, important: true })
```

Our solution is good. Except it doesn't work. The new note is saved on the server, but it is not updated on the screen.

In order to render a new note as well, we need to tell React Query that the old result of the query whose key is the string *notes* should be **invalidated**.

Fortunately, invalidation is easy, it can be done by defining the appropriate *onSuccess* callback function to the mutation:

```
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query'
import { getNotes, createNote } from './requests'

const App = () => {
  const queryClient = useQueryClient()
  const newNoteMutation = useMutation({
    mutationFn: createNote,
    onSuccess: () => {
      queryClient.invalidateQueries(['notes'])
    }
  })
  // ...
}
```

Now that the mutation has been successfully executed, a function call is made to

```
queryClient.invalidateQueries('notes')
```

This in turn causes React Query to automatically update a query with the key *notes*, i.e. fetch the notes from the server. As a result, the application renders the up-to-date state on the server, i.e. the added note is also rendered.

Let us also implement the change in the importance of notes. A function for updating notes is added to the file *requests.js*:

```
export const updateNote = updatedNote =>
  axios.put(`/${baseUrl}/${updatedNote.id}`, updatedNote).then(res => res.data)
```

Updating the note is also done by mutation. The *App* component expands as follows:

```

import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query'
import { getNotes, createNote, updateNote } from './requests'
const App = () => [
  // ...

  const updateNoteMutation = useMutation({
    mutationFn: updateNote,
    onSuccess: () => {
      queryClient.invalidateQueries(['notes'])
    },
  })

  const toggleImportance = (note) => {
    updateNoteMutation.mutate({ ...note, important: !note.important })
  }
]

```

So again, a mutation was created that invalidated the query *notes* so that the updated note is rendered correctly. Using mutations is easy, the method *mutate* receives a note as a parameter, the importance of which is been changed to the negation of the old value.

The current code for the application is on [GitHub](#) in the branch *part6-2*.

Optimizing the performance

The application works well, and the code is relatively simple. The ease of making changes to the list of notes is particularly surprising. For example, when we change the importance of a note, invalidating the query *notes* is enough for the application data to be updated:

```

const updateNoteMutation = useMutation({
  mutationFn: updateNote,
  onSuccess: () => {
    queryClient.invalidateQueries('notes')
  },
})

```

The consequence of this, of course, is that after the PUT request that causes the note change, the application makes a new GET request to retrieve the query data from the server:

Name	Status	Type	Initiator	Size
notes	200	xhr	bundle.js:44010	
3	200	xhr	bundle.js:44010	
notes	200	xhr	bundle.js:44010	

If the amount of data retrieved by the application is not large, it doesn't really matter. After all, from a browser-side functionality point of view, making an extra HTTP GET request doesn't really matter, but in some situations it might put a strain on the server.

If necessary, it is also possible to optimize performance [by manually updating](#) the query state maintained by React Query.

The change for the mutation adding a new note is as follows:

```
const App = () => {
  const queryClient = useQueryClient()

  const newNoteMutation = useMutation([
    mutationFn: createNote,
    onSuccess: (newNote) => {
      const notes = queryClient.getQueryData(['notes'])           queryClient.setQueryData(['notes'], notes.concat(newNote))
    }
  ])
}
```

That is, in the `onSuccess` callback, the `queryClient` object first reads the existing `notes` state of the query and updates it by adding a new note, which is obtained as a parameter of the callback function. The value of the parameter is the value returned by the function `createNote`, defined in the file `requests.js` as follows:

```
export const createNote = newNote =>
  axios.post(baseUrl, newNote).then(res => res.data)
```

It would be relatively easy to make a similar change to a mutation that changes the importance of the note, but we leave it as an optional exercise.

If we closely follow the browser's network tab, we notice that React Query retrieves all notes as soon as we move the cursor to the input field:

The screenshot shows a browser window with the URL `http://localhost:3000`. The page title is "Notes app". Below the title is a search bar with the placeholder "add" and a list of bullet points:

- react query simplifies the communication with backend
- useReduce with context is an alternative way for data management **important**
- When a query is invalidated, it is automatically redone

Below the list is a "Network" tab in the developer tools. The table shows a single request:

Name	Status	Type	Initiator	Size
notes	200	xhr	bundle.js:46254	

A red arrow points from the input field in the Notes app to the "notes" entry in the Network tab table.

What is going on? By reading the [documentation](#), we notice that the default functionality of React Query's queries is that the queries (whose status is *stale*) are updated when *window focus*, i.e. the active element of the application's user interface, changes. If we want, we can turn off the functionality by creating a query as follows:

```
const App = () => [
  // ...
  const result = useQuery({
    queryKey: ['notes'],
    queryFn: getNotes,
    refetchOnWindowFocus: false
  })

  // ...
]
```

If you put a `console.log` statement to the code, you can see from browser console how often React Query causes the application to be re-rendered. The rule of thumb is that rerendering happens at least whenever there is a need for it, i.e. when the state of the query changes. You can read more about it e.g. [here](#).

The code for the application is in [GitHub](#) in the branch *part6-3*.

React Query is a versatile library that, based on what we have already seen, simplifies the application. Does React Query make more complex state management solutions such as Redux unnecessary? No. React Query can partially replace the state of the application in some cases, but as the [documentation](#) states

- React Query is a *server-state library*, responsible for managing asynchronous operations between your server and client

- Redux, etc. are *client-state libraries* that can be used to store asynchronous data, albeit inefficiently when compared to a tool like React Query

So React Query is a library that maintains the *server state* in the frontend, i.e. acts as a cache for what is stored on the server. React Query simplifies the processing of data on the server, and can in some cases eliminate the need for data on the server to be saved in the frontend state.

Most React applications need not only a way to temporarily store the served data, but also some solution for how the rest of the frontend state (e.g. the state of forms or notifications) is handled.

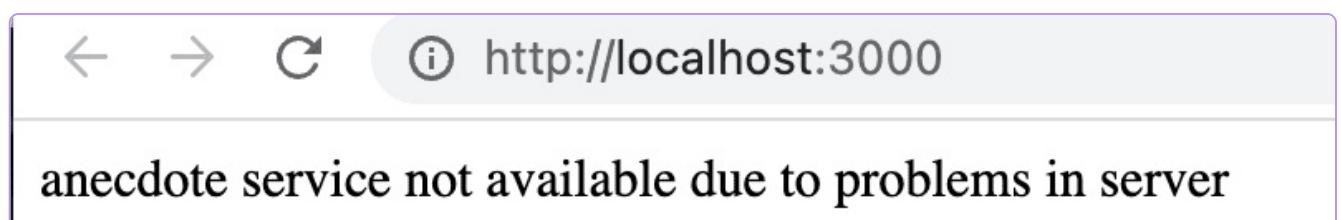
Exercises 6.20.-6.22.

Now let's make a new version of the anecdote application that uses the React Query library. Take [this project](#) as your starting point. The project has a ready-installed JSON Server, the operation of which has been slightly modified (Review the *server.js* file for more details. Make sure you're connecting to the correct *PORT*). Start the server with *npm run server*.

Exercise 6.20

Implement retrieving anecdotes from the server using React Query.

The application should work in such a way that if there are problems communicating with the server, only an error page will be displayed:



You can find [here](#) info how to detect the possible errors.

You can simulate a problem with the server by e.g. turning off the JSON Server. Please note that in a problem situation, the query is first in the state *isLoading* for a while, because if a request fails, React Query tries the request a few times before it states that the request is not successful. You can optionally specify that no retries are made:

```
const result = useQuery([
  {
    queryKey: ['anecdotes'],
    queryFn: getAnecdotes,
    retry: false
  }
])
```

or that the request is retried e.g. only once:

```
const result = useQuery([
  {
    queryKey: ['anecdotes'],
    queryFn: getAnecdotes,
    retry: 1
  }
])
```

```
    retry: 1
}
]
```

Exercise 6.21

Implement adding new anecdotes to the server using React Query. The application should render a new anecdote by default. Note that the content of the anecdote must be at least 5 characters long, otherwise the server will reject the POST request. You don't have to worry about error handling now.

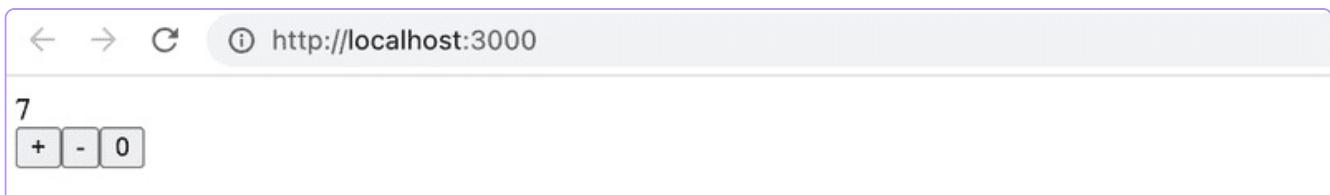
Exercise 6.22

Implement voting for anecdotes using again the React Query. The application should automatically render the increased number of votes for the voted anecdote.

useReducer

So even if the application uses React Query, some kind of solution is usually needed to manage the rest of the frontend state (for example, the state of forms). Quite often, the state created with `useState` is a sufficient solution. Using Redux is of course possible, but there are other alternatives.

Let's look at a simple counter application. The application displays the counter value, and offers three buttons to update the counter status:



We shall now implement the counter state management using a Redux-like state management mechanism provided by React's built-in `useReducer` hook. Code looks like the following:

```
import { useReducer } from 'react'

const counterReducer = (state, action) => {
  switch (action.type) {
    case "INC":
      return state + 1
    case "DEC":
      return state - 1
    case "ZERO":
      return 0
    default:
      return state
  }
}

const App = () => {
  const [counter, counterDispatch] = useReducer(counterReducer, 0)
```

```

    return [
      <div>
        <div>{counter}</div>
        <div>
          <button onClick={() => counterDispatch({ type: "INC"})}>+</button>
          <button onClick={() => counterDispatch({ type: "DEC"})}>-</button>
          <button onClick={() => counterDispatch({ type: "ZERO"})}>0</button>
        </div>
      </div>
    ]
}

export default App

```

The hook `useReducer` provides a mechanism to create a state for an application. The parameter for creating a state is the reducer function that handles state changes, and the initial value of the state:

```
const [counter, counterDispatch] = useReducer(counterReducer, 0)
```

The reducer function that handles state changes is similar to Redux's reducers, i.e. the function gets as parameters the current state and the action that changes the state. The function returns the new state updated based on the type and possible contents of the action:

```

const counterReducer = (state, action) => {
  switch (action.type) {
    case "INC":
      return state + 1
    case "DEC":
      return state - 1
    case "ZERO":
      return 0
    default:
      return state
  }
}

```

In our example, actions have nothing but a type. If the action's type is `INC`, it increases the value of the counter by one, etc. Like Redux's reducers, actions can also contain arbitrary data, which is usually put in the action's `payload` field.

The function `useReducer` returns an array that contains an element to access the current value of the state (first element of the array), and a `dispatch` function (second element of the array) to change the state:

```

const App = () => {
  const [counter, counterDispatch] = useReducer(counterReducer, 0)
  return [
    <div>
      <div>{counter}</div>      <div>
        <button onClick={() => counterDispatch({ type: "INC"})}>+</button>
        <button onClick={() => counterDispatch({ type: "DEC"})}>-</button>
      </div>
    ]
}

```

```

        <button onClick={() => counterDispatch({ type: "ZERO" })}>0</button>
    </div>
</div>
}
}

```

As can be seen the state change is done exactly as in Redux, the dispatch function is given the appropriate state-changing action as a parameter:

```
counterDispatch({ type: "INC" })
```

The current code for the application is in the repository <https://github.com/fullstack-hy2020/hook-counter> in the branch *part6-1*.

Using context for passing the state to components

If we want to split the application into several components, the value of the counter and the dispatch function used to manage it must also be passed to the other components. One solution would be to pass these as props in the usual way:

```

const Display = ({ counter }) => {
  return <div>{counter}</div>
}

const Button = ({ dispatch, type, label }) => {
  return [
    <button onClick={() => dispatch({ type })}>
      {label}
    </button>
  ]
}

const App = () => [
  const [counter, counterDispatch] = useReducer(counterReducer, 0)

  return [
    <div>
      <Display counter={counter}>/>          <div>
        <Button dispatch={counterDispatch} type='INC' label='+' />          <Button
        dispatch={counterDispatch} type='DEC' label='-' />          <Button dispatch={co
        unterDispatch} type='ZERO' label='0' />      </div>
    </div>
  ]
]

```

The solution works, but is not optimal. If the component structure gets complicated, e.g. the dispatcher should be forwarded using props through many components to the components that need it, even though the components in between in the component tree do not need the dispatcher. This phenomenon is called *prop drilling*.

React's built-in [Context API](#) provides a solution for us. React's context is a kind of global state of the

application, to which it is possible to give direct access to any component app.

Let us now create a context in the application that stores the state management of the counter.

The context is created with React's hook `createContext`. Let's create a context in the file `CounterContext.jsx`:

```
import { createContext } from 'react'

const CounterContext = createContext()

export default CounterContext
```

The `App` component can now *provide* a context to its child components as follows:

```
import CounterContext from './CounterContext'
const App = () => {
  const [counter, counterDispatch] = useReducer(counterReducer, 0)

  return (
    <CounterContext.Provider value={[counter, counterDispatch]}>      <Display />

    <div>
      <Button type='INC' label='+' />
      <Button type='DEC' label='-' />
      <Button type='ZERO' label='0' />
    </div>
  </CounterContext.Provider>  )
}
```

As can be seen, providing the context is done by wrapping the child components inside the `CounterContext.Provider` component and setting a suitable value for the context.

The context value is now set to be an array containing the value of the counter, and the *dispatch* function.

Other components now access the context using the `useContext` hook:

```
import { useContext } from 'react' import CounterContext from './CounterContext'

const Display = () => {
  const [counter] = useContext(CounterContext)  return <div>
    {counter}
  </div>
}

const Button = ({ type, label }) => {
  const [counter, dispatch] = useContext(CounterContext)  return (
    <button onClick={() => dispatch({ type })}>
      {label}
    </button>
  )
}
```

```
</button>
}
}
```

The current code for the application is in [GitHub](#) in the branch *part6-2*.

Defining the counter context in a separate file

Our application has an annoying feature, that the functionality of the counter state management is partly defined in the *App* component. Now let's move everything related to the counter to *CounterContext.jsx*:

```
import { createContext, useReducer } from 'react'

const counterReducer = (state, action) => {
  switch (action.type) {
    case "INC":
      return state + 1
    case "DEC":
      return state - 1
    case "ZERO":
      return 0
    default:
      return state
  }
}

const CounterContext = createContext()

export const CounterContextProvider = (props) => {
  const [counter, counterDispatch] = useReducer(counterReducer, 0)

  return (
    <CounterContext.Provider value={[counter, counterDispatch]}>
      {props.children}
    </CounterContext.Provider>
  )
}

export default CounterContext
```

The file now exports, in addition to the *CounterContext* object corresponding to the context, the *CounterContextProvider* component, which is practically a context provider whose value is a counter and a dispatcher used for its state management.

Let's enable the context provider by making a change in *main.jsx*:

```
import ReactDOM from 'react-dom/client'
import App from './App'
import { CounterContextProvider } from './CounterContext'
ReactDOM.createRoot(document.getElementById('root')).render(  
  <CounterContextProvider>
```

```
<CounterContextProvider>    <App />
</CounterContextProvider>
```

Now the context defining the value and functionality of the counter is available to *all* components of the application.

The *App* component is simplified to the following form:

```
import Display from './components/Display'
import Button from './components/Button'

const App = () => [
  return [
    <div>
      <Display />
      <div>
        <Button type='INC' label='+' />
        <Button type='DEC' label='-' />
        <Button type='ZERO' label='0' />
      </div>
    </div>
  ]
}

export default App
```

The context is still used in the same way, e.g. the component *Button* is defined as follows:

```
import { useContext } from 'react'
import CounterContext from '../CounterContext'

const Button = ({ type, label }) => {
  const [counter, dispatch] = useContext(CounterContext)
  return [
    <button onClick={() => dispatch({ type })}>
      {label}
    </button>
  ]
}

export default Button
```

The *Button* component only needs the *dispatch* function of the counter, but it also gets the value of the counter from the context using the function *useContext*:

```
const [counter, dispatch] = useContext(CounterContext)
```

This is not a big problem, but it is possible to make the code a bit more pleasant and expressive by defining a couple of helper functions in the *CounterContext* file:

```

import { createContext, useReducer, useContext } from 'react'
const CounterContext = createContext()

// ...

export const useCounterValue = () => {
  const counterAndDispatch = useContext(CounterContext)
  return counterAndDispatch[0]
}

export const useCounterDispatch = () => {
  const counterAndDispatch = useContext(CounterContext)
  return counterAndDispatch[1]
}

// ...

```

With the help of these helper functions, it is possible for the components that use the context to get hold of the part of the context that they need. The *Display* component changes as follows:

```

import { useCounterValue } from '../CounterContext'
const Display = () => {
  const counter = useCounterValue()  return <div>
    {counter}
  </div>
}

export default Display

```

Component *Button* becomes:

```

import { useCounterDispatch } from '../CounterContext'
const Button = ({ type, label }) => {
  const dispatch = useCounterDispatch()  return (
    <button onClick={() => dispatch({ type })}>
      {label}
    </button>
  )
}

export default Button

```

The solution is quite elegant. The entire state of the application, i.e. the value of the counter and the code for managing it, is now isolated in the file *CounterContext*, which provides components with well-named and easy-to-use auxiliary functions for managing the state.

The final code for the application is in [GitHub](#) in the branch *part6-3*.

As a technical detail, it should be noted that the helper functions *useCounterValue* and *useCounterDispatch* are defined as [custom hooks](#), because calling the hook function *useContext* is

possible only from React components or custom hooks. Custom hooks are JavaScript functions whose name must start with the word *use*. We will return to custom hooks in a little more detail in part 7 of the course.

Exercises 6.23.-6.24.

Exercise 6.23.

The application has a *Notification* component for displaying notifications to the user.

Implement the application's notification state management using the `useReducer` hook and context. The notification should tell the user when a new anecdote is created or an anecdote is voted on:

← → ⌂ http://localhost:3000

Anecdote app

anecdote 'If it hurts, do it more often' voted

create new

create

If it hurts, do it more often
has 6 **vote**

The notification is displayed for five seconds.

Exercise 6.24.

As stated in exercise 6.21, the server requires that the content of the anecdote to be added is at least 5 characters long. Now implement error handling for the insertion. In practice, it is sufficient to display a notification to the user in case of a failed POST request:

← → ⌂ http://localhost:3000

Anecdote app

too short anecdote, must have length 5 or more

create new

create

If it hurts, do it more often
has 8 **vote**

Adding manpower to a late software project makes it later!
has 4 **vote**

The first 90 percent of the code accounts for the first 90 percent of the development time...The remaining 10 percent of the code

The error condition should be handled in the callback function registered for it, see [here](#) how to register a function.

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

Which state management solution to choose?

In chapters 1-5, all state management of the application was done using React's hook `useState`. Asynchronous calls to the backend required the use of the `useEffect` hook in some situations. In principle, nothing else is needed.

A subtle problem with a solution based on a state created with the `useState` hook is that if some part of the application's state is needed by multiple components of the application, the state and the functions for manipulating it must be passed via props to all components that handle the state. Sometimes props need to be passed through multiple components, and the components along the way may not even be interested in the state in any way. This somewhat unpleasant phenomenon is called *prop drilling*.

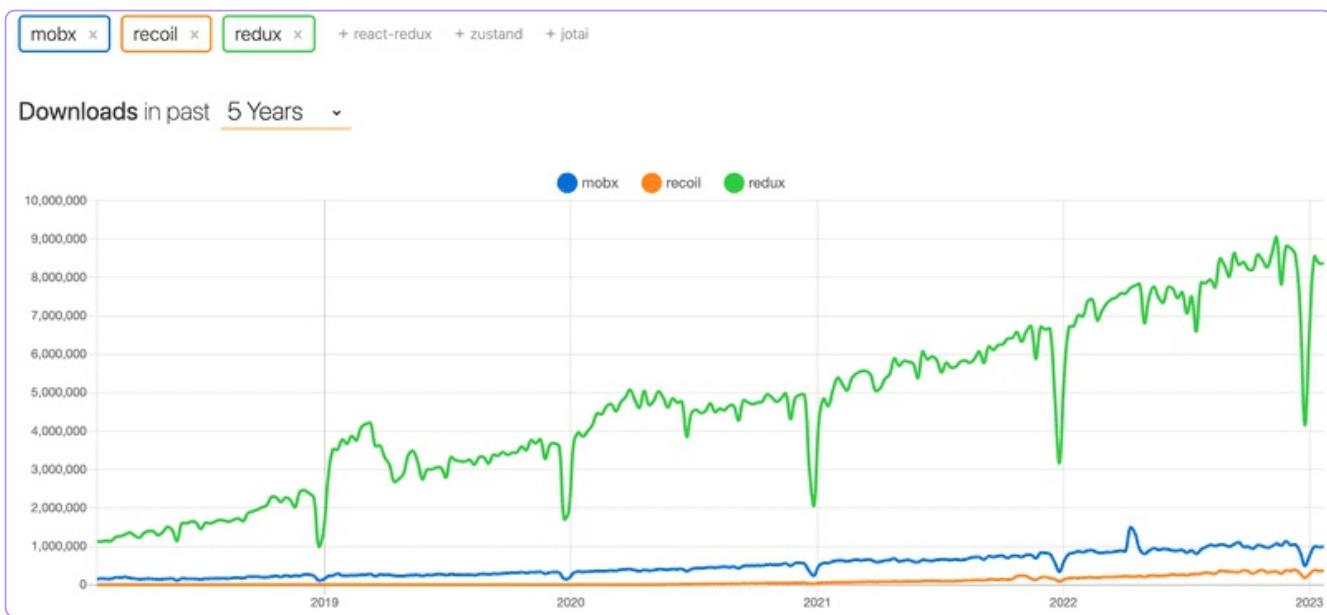
Over the years, several alternative solutions have been developed for state management of React applications, which can be used to ease problematic situations (e.g. prop drilling). However, no solution has been "final", all have their own pros and cons, and new solutions are being developed all the time.

The situation may confuse a beginner and even an experienced web developer. Which solution should be used?

For a simple application, `useState` is certainly a good starting point. If the application is communicating with the server, the communication can be handled in the same way as in chapters 1-5, using the state of the application itself. Recently, however, it has become more common to move the communication and associated state management at least partially under the control of React Query (or some other similar library). If you are concerned about `useState` and the prop drilling it entails, using context may be a good option. There are also situations where it may make sense to handle some of the state with `useState` and some with contexts.

The most comprehensive and robust state management solution is Redux, which is a way to implement the so-called [Flux](#) architecture. Redux is slightly older than the solutions presented in this section. The rigidity of Redux has been the motivation for many new state management solutions, such as React's `useReducer`. Some of the criticisms of Redux's rigidity have already become obsolete thanks to the [Redux Toolkit](#).

Over the years, there have also been other state management libraries developed that are similar to Redux, such as the newer entrant [Recoil](#) and the slightly older [MobX](#). However, according to [Npm trends](#), Redux still clearly dominates, and in fact seems to be increasing its lead:



Also, Redux does not have to be used in its entirety in an application. It may make sense, for example, to manage the form state outside of Redux, especially in situations where the state of a form does not affect the rest of the application. It is also perfectly possible to use Redux and React Query together in the same application.

The question of which state management solution should be used is not at all straightforward. It is impossible to give a single correct answer. It is also likely that the selected state management solution may turn out to be suboptimal as the application grows to such an extent that the solution has to be changed even if the application has already been put into production use.

7: React router, custom hooks, styling app with CSS and webpack

7a: React Router

The exercises in this seventh part of the course differ a bit from the ones before. In this and the next chapter, as usual, there are [exercises related to the theory of the chapter](#).

In addition to the exercises in this and the next chapter, there are a series of exercises in which we'll be revising what we've learned during the whole course, by expanding the BlogList application, in which we worked on during parts 4 and 5.

Application navigation structure

Following part 6, we return to React without Redux.

It is very common for web applications to have a navigation bar, which enables switching the view of the application.

Our app could have a main page

The screenshot shows a web browser window with the URL `localhost:3000`. The navigation bar includes links for `home`, `notes`, and `users`. The `home` link is highlighted with a red box. The main content area displays the heading **TKTL notes app** followed by a paragraph of Lorem Ipsum text. Below the text is a note: *Note app, Department of Computer Science 2020*.

and separate pages for showing information on notes and users:

The screenshot shows a web browser window with the URL `localhost:3000`. The navigation bar includes links for `home`, `notes`, and `users`. The `notes` link is highlighted with a red box. The main content area displays the heading **Notes** and a bulleted list: • HTML is easy, • Browser can execute javascript, • Browser talks to server with HTML. Below the list is a note: *Note app, Department of Computer Science 2020*.

In an [old school web app](#), changing the page shown by the application would be accomplished by the browser making an HTTP GET request to the server and rendering the HTML representing the view that was returned.

In single-page apps, we are, in reality, always on the same page. The Javascript code run by the browser creates an illusion of different "pages". If HTTP requests are made when switching views, they are only for fetching JSON-formatted data, which the new view might require for it to be shown.

The navigation bar and an application containing multiple views are very easy to implement using React.

Here is one way:

```
import { useState } from 'react'
import ReactDOM from 'react-dom/client'

const Home = () => [
  <div> <h2>TKTL notes app</h2> </div>
]

const Notes = () => [
  <div> <h2>Notes</h2> </div>
]

const Users = () => [
```

```

<div> <h2>Users</h2> </div>
}

const App = () => [
  const [page, setPage] = useState('home')

  const toPage = [page] => [event] => [
    event.preventDefault()
    setPage(page)
  ]

  const content = () => [
    if (page === 'home') [
      return <Home />
    ] else if (page === 'notes') [
      return <Notes />
    ] else if (page === 'users') [
      return <Users />
    ]
  ]
]

const padding = [
  padding: 5
]

return [
  <div>
    <div>
      <a href="" onClick={toPage('home')} style={padding}>
        home
      </a>
      <a href="" onClick={toPage('notes')} style={padding}>
        notes
      </a>
      <a href="" onClick={toPage('users')} style={padding}>
        users
      </a>
    </div>
    {content()}
  </div>
]
}

ReactDOM.createRoot(document.getElementById('root')).render(<App />)

```

Each view is implemented as its own component. We store the view component information in the application state called *page*. This information tells us which component, representing a view, should be shown below the menu bar.

However, the method is not very optimal. As we can see from the pictures, the address stays the same even though at times we are in different views. Each view should preferably have its own

address, e.g. to make bookmarking possible. The *back* button doesn't work as expected for our application either, meaning that *back* doesn't move you to the previously displayed view of the application, but somewhere completely different. If the application were to grow even bigger and we wanted to, for example, add separate views for each user and note, then this self-made *routing*, which means the navigation management of the application, would get overly complicated.

React Router

Luckily, React has the [React Router](#) library which provides an excellent solution for managing navigation in a React application.

Let's change the above application to use React Router. First, we install React Router with the command:

```
npm install react-router-dom
```

The routing provided by React Router is enabled by changing the application as follows:

```
import {
  BrowserRouter as Router,
  Routes, Route, Link
} from 'react-router-dom'

const App = () => {

  const padding = [
    padding: 5
  ]

  return (
    <Router>
      <div>
        <Link style={padding} to="/">home</Link>
        <Link style={padding} to="/notes">notes</Link>
        <Link style={padding} to="/users">users</Link>
      </div>

      <Routes>
        <Route path="/notes" element={<Notes />} />
        <Route path="/users" element={<Users />} />
        <Route path="/" element={<Home />} />
      </Routes>

      <div>
        <i>Note app, Department of Computer Science 2024</i>
      </div>
    </Router>
  )
}
```

Routing, or the conditional rendering of components *based on the URL* in the browser, is used by

placing components as children of the *Router* component, meaning inside *Router* tags.

Notice that, even though the component is referred to by the name *Router*, we are talking about [BrowserRouter](#), because here the import happens by renaming the imported object:

```
import {  
  BrowserRouter as Router, Routes, Route, Link  
} from 'react-router-dom'
```

According to the [v5 docs](#):

BrowserRouter is a *Router* that uses the HTML5 history API (pushState, replaceState and the popState event) to keep your UI in sync with the URL.

Normally the browser loads a new page when the URL in the address bar changes. However, with the help of the [HTML5 history API](#), *BrowserRouter* enables us to use the URL in the address bar of the browser for internal "routing" in a React application. So, even if the URL in the address bar changes, the content of the page is only manipulated using Javascript, and the browser will not load new content from the server. Using the back and forward actions, as well as making bookmarks, is still logical like on a traditional web page.

Inside the router, we define *links* that modify the address bar with the help of the [Link](#) component. For example:

```
<Link to="/notes">notes</Link>
```

creates a link in the application with the text *notes*, which when clicked changes the URL in the address bar to */notes*.

Components rendered based on the URL of the browser are defined with the help of the component [Route](#). For example,

```
<Route path="/notes" element={<Notes />} />
```

defines that, if the browser address is */notes*, we render the *Notes* component.

We wrap the components to be rendered based on the URL with a [Routes](#) component

```
<Routes>  
  <Route path="/notes" element={<Notes />} />  
  <Route path="/users" element={<Users />} />  
  <Route path="/" element={<Home />} />  
</Routes>
```

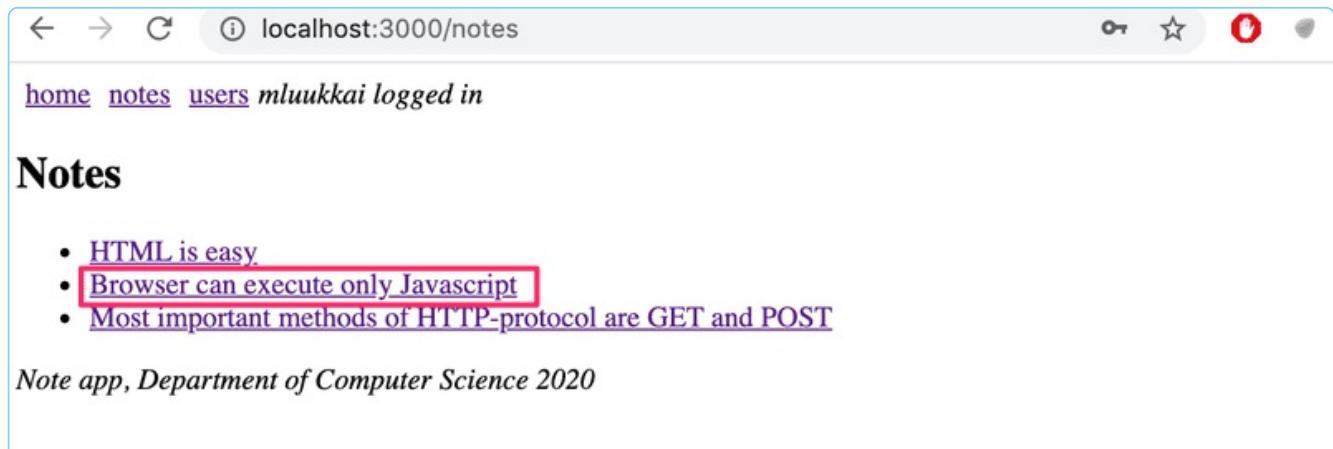
The *Routes* works by rendering the first component whose *path* matches the URL in the browser's address bar.

Parameterized route

Let's examine a slightly modified version from the previous example. The complete code for the updated example can be found [here](#).

The application now contains five different views whose display is controlled by the router. In addition to the components from the previous example (*Home*, *Notes* and *Users*), we have *Login* representing the login view and *Note* representing the view of a single note.

Home and *Users* are unchanged from the previous exercise. *Notes* is a bit more complicated. It renders the list of notes passed to it as props in such a way that the name of each note is clickable.



localhost:3000/notes

home notes users mluukkai logged in

Notes

- [HTML is easy](#)
- [Browser can execute only Javascript](#)
- [Most important methods of HTTP-protocol are GET and POST](#)

Note app, Department of Computer Science 2020

The ability to click a name is implemented with the component *Link*, and clicking the name of a note whose id is 3 would trigger an event that changes the address of the browser into *notes/3*:

```
const Notes = ({notes}) => {
  <div>
    <h2>Notes</h2>
    <ul>
      {notes.map(note =>
        <li key={note.id}>
          <Link to={`/notes/${note.id}`}>{note.content}</Link>
        </li>
      )}
    </ul>
  </div>
}
```

We define parameterized URLs in the routing of the *App* component as follows:

```
<Router>
  // ...

  <Routes>
    <Route path="/notes/:id" element={<Note notes={notes} />} />      <Route path="/notes" element={<Notes notes={notes} />} />
    <Route path="/users" element={user ? <Users /> : <Navigate replace to="/login" />} />
    <Route path="/login" element={<Login onLogin={login} />} />
    <Route path="/" element={<Home />} />
```

```
</Routes>
</Router>
```

We define the route rendering a specific note "express style" by marking the parameter with a colon - `:id`

```
<Route path="/notes/:id" element={<Note notes={notes} />} />
```

When a browser navigates to the URL for a specific note, for example, `/notes/3`, we render the `Note` component:

```
import {
  ...
  useParams from 'react-router-dom'

const Note = ({ notes }) => {
  const id = useParams().id  const note = notes.find(n => n.id === Number(id))
  return (
    <div>
      <h2>{note.content}</h2>
      <div>{note.user}</div>
      <div><strong>{note.important ? 'important' : ''}</strong></div>
    </div>
  )
}
```

The `Note` component receives all of the notes as props `notes`, and it can access the URL parameter (the id of the note to be displayed) with the `useParams` function of the React Router.

useNavigate

We have also implemented a simple login function in our application. If a user is logged in, information about a logged-in user is saved to the `user` field of the state of the `App` component.

The option to navigate to the `Login` view is rendered conditionally in the menu.

```
<Router>
  <div>
    <Link style={{padding}} to="/">home</Link>
    <Link style={{padding}} to="/notes">notes</Link>
    <Link style={{padding}} to="/users">users</Link>
    {user ? <em>{user} logged in</em> : <Link style={{padding}} to="/login">login</Link>}
  </div>
  // ...
</Router>
```

So if the user is already logged in, instead of displaying the link `Login`, we show its username:

[home](#) [notes](#) [users](#) mluukkai logged in

TKTL notes app

Lore ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

The code of the component handling the login functionality is as follows:

```
import {  
  // ...  
  useNavigate } from 'react-router-dom'  
  
const Login = (props) => [  
  const navigate = useNavigate()  
  const onSubmit = (event) => [  
    event.preventDefault()  
    props.onLogin('mluukkai')  
    navigate('/') ]  
  
  return [  
    <div>  
      <h2>login</h2>  
      <form onSubmit={onSubmit}>  
        <div>  
          username: <input />  
        </div>  
        <div>  
          password: <input type='password' />  
        </div>  
        <button type="submit">login</button>  
      </form>  
    </div>  
  ]  
]
```

What is interesting about this component is the use of the `useNavigate` function of the React Router. With this function, the browser's URL can be changed programmatically.

With user login, we call `navigate('/')` which causes the browser's URL to change to `/` and the application renders the corresponding component `Home`.

Both `useParams` and `useNavigate` are hook functions, just like `useState` and `useEffect` which we have used many times now. As you remember from part 1, there are some `rules` to using hook functions.

Redirect

There is one more interesting detail about the `Users` route:

```
<Route path="/users" element={user ? <Users /> : <Navigate replace to="/login" />} />
```

If a user isn't logged in, the *Users* component is not rendered. Instead, the user is *redirected* using the component `Navigate` to the login view:

```
<Navigate replace to="/login" />
```

In reality, it would perhaps be better to not even show links in the navigation bar requiring login if the user is not logged into the application.

Here is the *App* component in its entirety:

```
const App = () => {
  const [notes, setNotes] = useState([])
  // ...
}

const [user, setUser] = useState(null)

const login = (user) => {
  setUser(user)
}

const padding = [
  padding: 5
]

return [
  <div>
    <Router>
      <div>
        <Link style={padding} to="/">home</Link>
        <Link style={padding} to="/notes">notes</Link>
        <Link style={padding} to="/users">users</Link>
        {user
          ? <em>{user} logged in</em>
          : <Link style={padding} to="/login">login</Link>
        }
      </div>
    <Routes>
      <Route path="/notes/:id" element={<Note notes={notes} />} />
      <Route path="/notes" element={<Notes notes={notes} />} />
      <Route path="/users" element={user ? <Users /> : <Navigate replace to="/login" />} />
      <Route path="/login" element={<Login onLogin={login} />} />
      <Route path="/" element={<Home />} />
    </Routes>
  </Router>
</div>
```

```

    <footer>
      <br />
      <em>Note app, Department of Computer Science 2024</em>
    </footer>
  </div>
}

}

```

We define an element common for modern web apps called *footer*, which defines the part at the bottom of the screen, outside of the *Router*, so that it is shown regardless of the component shown in the routed part of the application.

Parameterized route revisited

Our application has a flaw. The *Note* component receives all of the notes, even though it only displays the one whose id matches the URL parameter:

```

const Note = ({ notes }) => {
  const id = useParams().id
  const note = notes.find(n => n.id === Number(id))
  // ...
}

```

Would it be possible to modify the application so that the *Note* component receives only the note that it should display?

```

const Note = ({ note }) => {
  return (
    <div>
      <h2>{note.content}</h2>
      <div>{note.user}</div>
      <div><strong>{note.important ? 'important' : ''}</strong></div>
    </div>
  )
}

```

One way to do this would be to use React Router's [useMatch](#) hook to figure out the id of the note to be displayed in the *App* component.

It is not possible to use the *useMatch* hook in the component which defines the routed part of the application. Let's move the use of the *Router* components from *App*:

```

ReactDOM.createRoot(document.getElementById('root')).render(
  <Router>    <App />
  </Router>]

```

The *App* component becomes:

```

import {
  // ...
}

```

```

useMatch} from 'react-router-dom'

const App = () => [
  // ...

  const match = useMatch('/notes/:id')  const note = match      ? notes.find(note
=> note.id === Number(match.params.id))    : null
  return [
    <div>
      <div>
        <Link style={padding} to="/">home</Link>
        // ...
      </div>

      <Routes>
        <Route path="/notes/:id" element={<Note note={note} />} />           <Route
path="/notes" element={<Notes notes={notes} />} />
        <Route path="/users" element={user ? <Users /> : <Navigate replace to="/
login" />} />
        <Route path="/login" element={<Login onLogin={login} />} />
        <Route path="/" element={<Home />} />
      </Routes>

      <div>
        <em>Note app, Department of Computer Science 2024</em>
      </div>
    </div>
  ]
]

```

Every time the component is rendered, so practically every time the browser's URL changes, the following command is executed:

```
const match = useMatch('/notes/:id')
```

If the URL matches `/notes/:id`, the `match` variable will contain an object from which we can access the parameterized part of the path, the id of the note to be displayed, and we can then fetch the correct note to display.

```
const note = match
? notes.find(note => note.id === Number(match.params.id))
: null
```

The completed code can be found [here](#).

Exercises 7.1.-7.3.

Let's return to working with anecdotes. Use the redux-free anecdote app found in the repository <https://github.com/fullstack-hy2020/routed-anecdotes> as the starting point for the exercises.

If you clone the project into an existing git repository, remember to *delete the git configuration of the cloned application*:

```
cd routed-anecdotes // go first to directory of the cloned repository  
rm -rf .git
```

The application starts the usual way, but first, you need to install its dependencies:

```
npm install  
npm run dev
```

7.1: Routed Anecdotes, step 1

Add React Router to the application so that by clicking links in the *Menu* component the view can be changed.

At the root of the application, meaning the path `/`, show the list of anecdotes:

localhost:3000

Software anecdotes

[anecdotes](#) [create](#) [new](#) [about](#)

Anecdotes

- If it hurts, do it more often
- Premature optimization is the root of all evil

Anecdote app for [Full Stack -sovelluskehitys](#). See <https://github.com/mluukkai/routed-anecdotes> for the source code.

The *Footer* component should always be visible at the bottom.

The creation of a new anecdote should happen e.g. in the path `create`:

localhost:3000/create

Software anecdotes

[anecdotes](#) [create](#) [new](#) [about](#)

create a new anecdote

content

author

url for more info

Anecdote app for [Full Stack -sovelluskehitys](#). See <https://github.com/mluukkai/routed-anecdotes> for the source code.

7.2: Routed Anecdotes, step 2

Implement a view for showing a single anecdote:

The screenshot shows a web browser window with the URL `localhost:3000/anecdotes/1` highlighted with a red box. The page title is "Software anecdotes". Below it is a section titled "If it hurts, do it more often by Jez Humble". It says "has 0 votes" and provides a link for more info: <https://martinfowler.com/bliki/FrequencyReducesDifficulty.html>. At the bottom, it mentions an anecdote app for Full Stack -sovelluskehitys and provides a link to the source code: <https://github.com/mluukkai/routed-anecdotes>.

Navigating to the page showing the single anecdote is done by clicking the name of that anecdote:

The screenshot shows a web browser window with the URL `localhost:3000`. The page title is "Software anecdotes". Below it is a section titled "Anecdotes". A list of anecdotes is shown in a bulleted list, with the first item "[If it hurts, do it more often](#)" highlighted with a red box. At the bottom, it mentions an anecdote app for Full Stack -sovelluskehitys and provides a link to the source code: <https://github.com/mluukkai/routed-anecdotes>.

7.3: Routed Anecdotes, step3

The default functionality of the creation form is quite confusing because nothing seems to be happening after creating a new anecdote using the form.

Improve the functionality such that after creating a new anecdote the application transitions automatically to showing the view for all anecdotes *and* the user is shown a notification informing them of this successful creation for the next five seconds:

The screenshot shows a browser window with the URL 'localhost:3000'. The page title is 'Software anecdotes'. Below the title, there is a success message: 'anecdotes create new about' and 'a new anecdote Goto statement considered harmful created!'. A red border highlights this message. Below the message, there is a section titled 'Anecdotes' with a list of three items: 'If it hurts, do it more often', 'Premature optimization is the root of all evil', and 'Goto statement considered harmful'. At the bottom of the page, there is a note: 'Anecdote app for Full Stack -sovelluskehitys. See <https://github.com/mluukkai/routed-anecdotes> for the source code.'

7b: Custom hooks

Hooks

React offers 15 different [built-in hooks](#), of which the most popular ones are the `useState` and `useEffect` hooks that we have already been using extensively.

In [part 5](#) we used the `useImperativeHandle` hook which allows components to provide their functions to other components. In [part 6](#) we used `useReducer` and `useContext` to implement a Redux like state management.

Within the last couple of years, many React libraries have begun to offer hook-based APIs. In [part 6](#) we used the `useSelector` and `useDispatch` hooks from the react-redux library to share our redux-store and dispatch function to our components.

The [React Router's](#) API we introduced in the [previous part](#) is also partially hook-based. Its hooks can be used to access URL parameters and the `navigation` object, which allows for manipulating the browser URL programmatically.

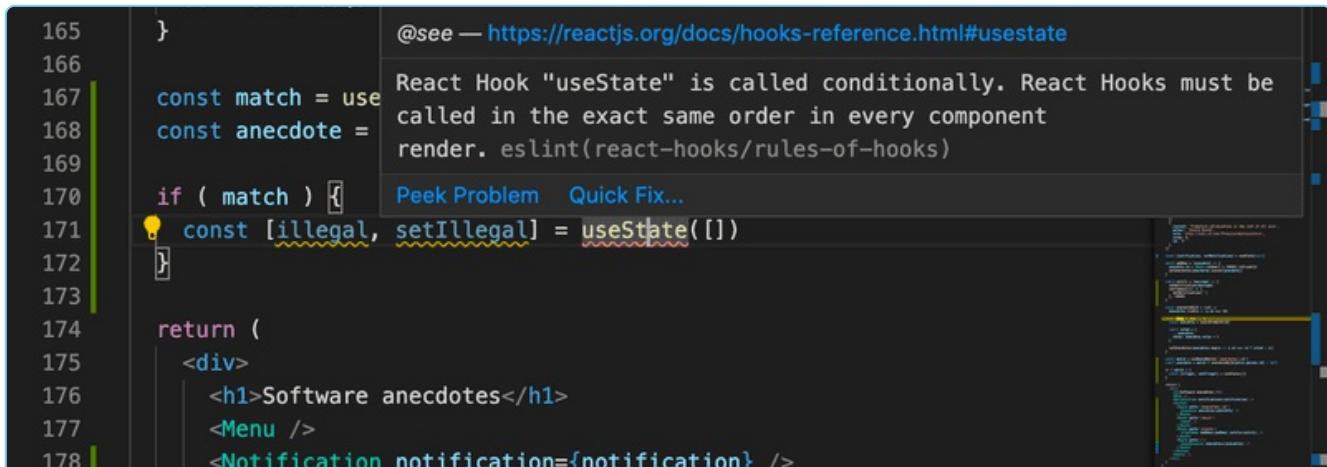
As mentioned in [part 1](#), hooks are not normal functions, and when using those we have to adhere to certain [rules or limitations](#). Let's recap the rules of using hooks, copied verbatim from the official React documentation:

Don't call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function.

You can only call Hooks while React is rendering a function component:

- Call them at the top level in the body of a function component.
- Call them at the top level in the body of a custom Hook.

There's an existing [ESlint plugin](#) that can be used to verify that the application uses hooks correctly:



```
165 } @see — https://reactjs.org/docs/hooks-reference.html#usestate
166
167 const match = useState()
168 const anecdote =
169
170 if ( match ) [
171     const [illegal, setIllegal] = useState([])
172 ]
173
174 return (
175     <div>
176         <h1>Software anecdotes</h1>
177         <Menu />
178         <Notification notification={notification} />
```

Custom hooks

React offers the option to create [custom hooks](#). According to React, the primary purpose of custom hooks is to facilitate the reuse of the logic used in components.

Building your own Hooks lets you extract component logic into reusable functions.

Custom hooks are regular JavaScript functions that can use any other hooks, as long as they adhere to the [rules of hooks](#). Additionally, the name of custom hooks must start with the word `use`.

We implemented a counter application in [part 1](#) that can have its value incremented, decremented, or reset. The code of the application is as follows:

```
import { useState } from 'react'
const App = () => {
    const [counter, setCounter] = useState(0)

    return [
        <div>
            <div>{counter}</div>
            <button onClick={() => setCounter(counter + 1)}>
                plus
            </button>
            <button onClick={() => setCounter(counter - 1)}>
                minus
            </button>
            <button onClick={() => setCounter(0)}>
                zero
            </button>
        </div>
    ]
}
```

Let's extract the counter logic into a custom hook. The code for the hook is as follows:

```

const useCounter = () => {
  const [value, setValue] = useState(0)

  const increase = () => {
    setValue(value + 1)
  }

  const decrease = () => {
    setValue(value - 1)
  }

  const zero = () => {
    setValue(0)
  }

  return {
    value,
    increase,
    decrease,
    zero
  }
}

```

Our custom hook uses the `useState` hook internally to create its state. The hook returns an object, the properties of which include the value of the counter as well as functions for manipulating the value.

React components can use the hook as shown below:

```

const App = () => {
  const counter = useCounter()

  return (
    <div>
      <div>{counter.value}</div>
      <button onClick={counter.increase}>
        plus
      </button>
      <button onClick={counter.decrease}>
        minus
      </button>
      <button onClick={counter.zero}>
        zero
      </button>
    </div>
  )
}

```

By doing this we can extract the state of the `App` component and its manipulation entirely into the `useCounter` hook. Managing the counter state and logic is now the responsibility of the custom hook.

The same hook could be *reused* in the application that was keeping track of the number of clicks

made to the left and right buttons:

```
const App = () => {
  const left = useCounter()
  const right = useCounter()

  return [
    <div>
      {left.value}
      <button onClick={left.increase}>
        left
      </button>
      <button onClick={right.increase}>
        right
      </button>
      {right.value}
    </div>
  ]
}
```

The application creates *two* completely separate counters. The first one is assigned to the variable *left* and the other to the variable *right*.

Dealing with forms in React is somewhat tricky. The following application presents the user with a form that requires him to input their name, birthday, and height:

```
const App = () => {
  const [name, setName] = useState('')
  const [born, setBorn] = useState('')
  const [height, setHeight] = useState('')

  return [
    <div>
      <form>
        name:
        <input
          type='text'
          value={name}
          onChange={(event) => setName(event.target.value)}
        />
        <br/>
        birthdate:
        <input
          type='date'
          value={born}
          onChange={(event) => setBorn(event.target.value)}
        />
        <br />
        height:
        <input
          type='number'
          value={height}
        </input>
      </form>
    </div>
  ]
}
```

```

        onChange={(event) => setHeight(event.target.value)}
      />
    </form>
    <div>
      {name} {born} {height}
    </div>
  </div>
}

}

```

Every field of the form has its own state. To keep the state of the form synchronized with the data provided by the user, we have to register an appropriate *onChange* handler for each of the *input* elements.

Let's define our own custom *useField* hook that simplifies the state management of the form:

```

const useField = (type) => {
  const [value, setValue] = useState('')

  const onChange = (event) => [
    setValue(event.target.value)
  ]

  return [
    type,
    value,
    onChange
  ]
}

```

The hook function receives the type of the input field as a parameter. It returns all of the attributes required by the *input*: its type, value and the *onChange* handler.

The hook can be used in the following way:

```

const App = () => [
  const name = useField('text')
  // ...

  return [
    <div>
      <form>
        <input
          type={name.type}
          value={name.value}
          onChange={name.onChange}
        />
        // ...
      </form>
    </div>
]

```

Spread attributes

We could simplify things a bit further. Since the `name` object has exactly all of the attributes that the `input` element expects to receive as props, we can pass the props to the element using the [spread syntax](#) in the following way:

```
<input {...name} />
```

As the [example](#) in the React documentation states, the following two ways of passing props to a component achieve the exact same result:

```
<Greeting firstName='Arto' lastName='Hellas' />

const person = {
  firstName: 'Arto',
  lastName: 'Hellas'
}

<Greeting {...person} />
```

The application gets simplified into the following format:

```
const App = () => {
  const name = useField('text')
  const born = useField('date')
  const height = useField('number')

  return [
    <div>
      <form>
        name:
        <input {...name} />
        <br/>
        birthdate:
        <input {...born} />
        <br />
        height:
        <input {...height} />
      </form>
      <div>
        {name.value} {born.value} {height.value}
      </div>
    </div>
  ]
}
```

Dealing with forms is greatly simplified when the unpleasant nitty-gritty details related to synchronizing the state of the form are encapsulated inside our custom hook.

Custom hooks are not only a tool for reusing code; they also provide a better way for dividing it into smaller modular parts.

More about hooks

The internet is starting to fill up with more and more helpful material related to hooks. The following sources are worth checking out:

- [Awesome React Hooks Resources](#)
- [Easy to understand React Hook recipes by Gabe Ragland](#)
- [Why Do React Hooks Rely on Call Order?](#)

Exercises 7.4.-7.8.

We'll continue with the app from the [exercises](#) of the [react router](#) chapter.

7.4: Anecdotes and Hooks step 1

Simplify the anecdote creation form of your application with the `useField` custom hook we defined earlier.

One natural place to save the custom hooks of your application is in the `/src/hooks/index.js` file.

If you use the [named export](#) instead of the default export:

```
import { useState } from 'react'

export const useField = (type) => {
  const [value, setValue] = useState('')

  const onChange = (event) => [
    setValue(event.target.value)
  ]

  return [
    type,
    value,
    onChange
  ]
}

// modules can have several named exports
export const useAnotherHook = () => { // ...
}
```

Then [importing](#) happens in the following way:

```
import { useField } from './hooks'

const App = () => [
  // ...
  const username = useField('text')
```



7.5: Anecdotes and Hooks step 2

Add a button to the form that you can use to clear all the input fields:

localhost:3000/create

Software anecdotes

[anecdotes](#) [create new](#) [about](#)

create a new anecdote

content

author Jhonny Doe

url for more info

Anecdote app for [Full Stack -websovelluskehitys](https://github.com/fullstack-hy2020/routed-anecdotes/blob/master/src/App.js). See <https://github.com/fullstack-hy2020/routed-anecdotes/blob/master/src/App.js>

Expand the functionality of the `useField` hook so that it offers a new `reset` operation for clearing the field.

Depending on your solution, you may see the following warning in your console:

[HMR] Waiting for update signal from WDS...

✖ Warning: Invalid value for prop `reset` on <input> tag. Either remove it from the element, or pass a string or number value to ke
https://fb.me/react-attribute-behavior

in input (at App.js:96)
in div (at App.js:94)
in form (at App.js:93)
in div (at App.js:91)
in CreateNew (at App.js:185)
in Route (at App.js:184)
in Switch (at App.js:177)
in div (at App.js:173)
in App (at App.js:198)
in Router (created by BrowserRouter)
in BrowserRouter (at App.js:197)

We will return to this warning in the next exercise.

7.6: Anecdotes and Hooks step 3

If your solution did not cause a warning to appear in the console, you have already finished this exercise.

If you see the *Invalid value for prop 'reset' on <input> tag* warning in the console, make the necessary changes to get rid of it.

The reason for this warning is that after making the changes to your application, the following expression:

```
<input {...content} />
```

Essentially, is the same as this:

```
<input  
  value={content.value}  
  type={content.type}  
  onChange={content.onChange}  
  reset={content.reset} />
```

The *input* element should not be given a *reset* attribute.

One simple fix would be to not use the spread syntax and write all of the forms like this:

```
<input  
  value={username.value}  
  type={username.type}  
  onChange={username.onChange}  
/>
```

If we were to do this, we would lose much of the benefit provided by the *useField* hook. Instead, come up with a solution that fixes the issue, but is still easy to use with the spread syntax.

7.7: Country hook

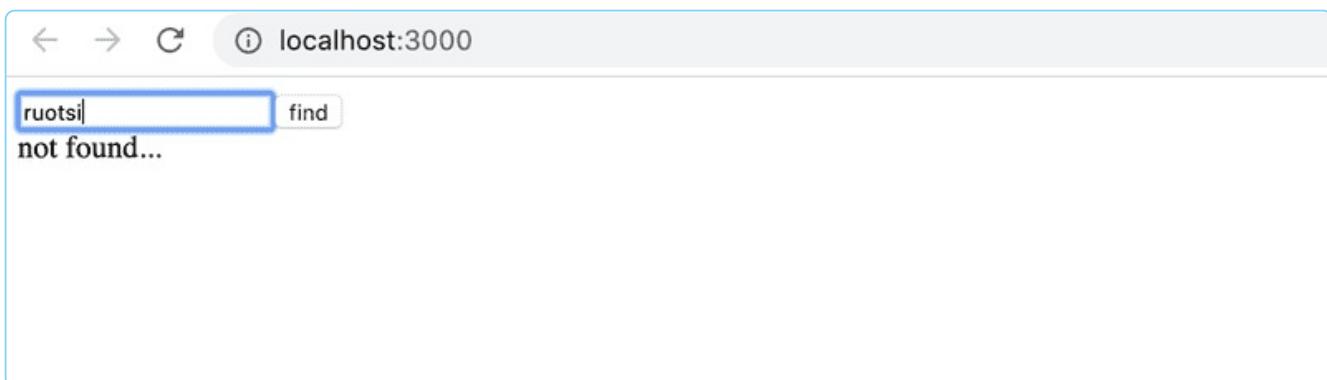
Let's return to exercises [2.18-2.20](#).

Use the code from <https://github.com/fullstack-hy2020/country-hook> as your starting point.

The application can be used to search for a country's details from the service in <https://studies.cs.helsinki.fi/restcountries/>. If a country is found, its details are displayed:

A screenshot of a web browser window titled "localhost:3000". In the address bar, the URL "localhost:3000" is visible. Below the address bar is a search input field containing the text "nepal". To the right of the input field is a button labeled "find". The main content area displays the following information for Nepal:
Nepal
capital Kathmandu
population 28431500
An image of the flag of Nepal is shown, featuring a white Ashoka Chakra in the center, surrounded by a red border, with blue triangles at the hoist and fly ends.

If no country is found, a message is displayed to the user:



The application is otherwise complete, but in this exercise, you have to implement a custom hook `useCountry`, which can be used to search for the details of the country given to the hook as a parameter.

Use the API endpoint `name` to fetch a country's details in a `useEffect` hook within your custom hook.

Note that in this exercise it is essential to use `useEffect`'s `second parameter` array to control when the effect function is executed. See the course [part 2](#) for more info how the second parameter could be used.

7.8: Ultimate Hooks

The code of the application responsible for communicating with the backend of the note application of the previous parts looks like this:

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

let token = null

const setToken = newToken => {
  token = `bearer ${newToken}`
}

const getAll = async () => {
  const response = await axios.get(baseUrl)
  return response.data
}

const create = async newObject => {
  const config = [
    headers: { Authorization: token },
  ]

  const response = await axios.post(baseUrl, newObject, config)
  return response.data
}

const update = async (id, newObject) => {
  const response = await axios.put(`/${baseUrl}/${id}`, newObject)
  return response.data
}
```

```
}

export default [ getAll, create, update, setToken ]
```

We notice that the code is in no way specific to the fact that our application deals with notes. Excluding the value of the `baseUrl` variable, the same code could be reused in the blog post application for dealing with the communication with the backend.

Extract the code for communicating with the backend into its own `useResource` hook. It is sufficient to implement fetching all resources and creating a new resource.

You can do the exercise in the project found in the <https://github.com/fullstack-hy2020/ultimate-hooks> repository. The `App` component for the project is the following:

```
const App = () => [
  const content = useField('text')
  const name = useField('text')
  const number = useField('text')

  const [notes, noteService] = useResource('http://localhost:3005/notes')
  const [persons, personService] = useResource('http://localhost:3005/persons')

  const handleNoteSubmit = (event) => {
    event.preventDefault()
    noteService.create([ content: content.value ])
  }

  const handlePersonSubmit = (event) => {
    event.preventDefault()
    personService.create([ name: name.value, number: number.value ])
  }

  return (
    <div>
      <h2>notes</h2>
      <form onSubmit={handleNoteSubmit}>
        <input {...content} />
        <button>create</button>
      </form>
      {notes.map(n => <p key={n.id}>{n.content}</p>)}

      <h2>persons</h2>
      <form onSubmit={handlePersonSubmit}>
        name <input {...name} /> <br/>
        number <input {...number} />
        <button>create</button>
      </form>
      {persons.map(n => <p key={n.id}>{n.name} {n.number}</p>)}
    </div>
  )
]
```

The `useResource` custom hook returns an array of two items just like the state hooks. The first item of the array contains all of the individual resources and the second item of the array is an object that can be used for manipulating the resource collection, like creating new ones.

If you implement the hook correctly, it can be used for both notes and phone numbers (start the server with the `npm run server` command at port 3005).

notes

custom-hooks are awesome!

best feature ever <3

persons

name: jessica

number: 91-123-4342432

mluukkai 040-5483923

hellas 012-123123

7c: More about styles

In part 2, we examined two different ways of adding styles to our application: the old-school [single CSS file](#) and [inline styles](#). In this part, we will take a look at a few other ways.

Ready-made UI libraries

One approach to defining styles for an application is to use a ready-made "UI framework".

One of the first widely popular UI frameworks was the [Bootstrap](#) toolkit created by Twitter which may still be the most popular. Recently, there has been an explosion in the number of new UI frameworks that have entered the arena. The selection is so vast that there is little hope of creating an exhaustive list of options.

Many UI frameworks provide developers of web applications with ready-made themes and "components" like buttons, menus, and tables. We write components in quotes because, in this context, we are not talking about React components. Usually, UI frameworks are used by including the CSS stylesheets and JavaScript code of the framework in the application.

Many UI frameworks have React-friendly versions where the framework's "components" have been transformed into React components. There are a few different React versions of Bootstrap like [reactstrap](#) and [react-bootstrap](#).

Next, we will take a closer look at two UI frameworks, Bootstrap and [MaterialUI](#). We will use both frameworks to add similar styles to the application we made in the [React Router](#) section of the course material.

React Bootstrap

Let's start by taking a look at Bootstrap with the help of the [react-bootstrap](#) package.

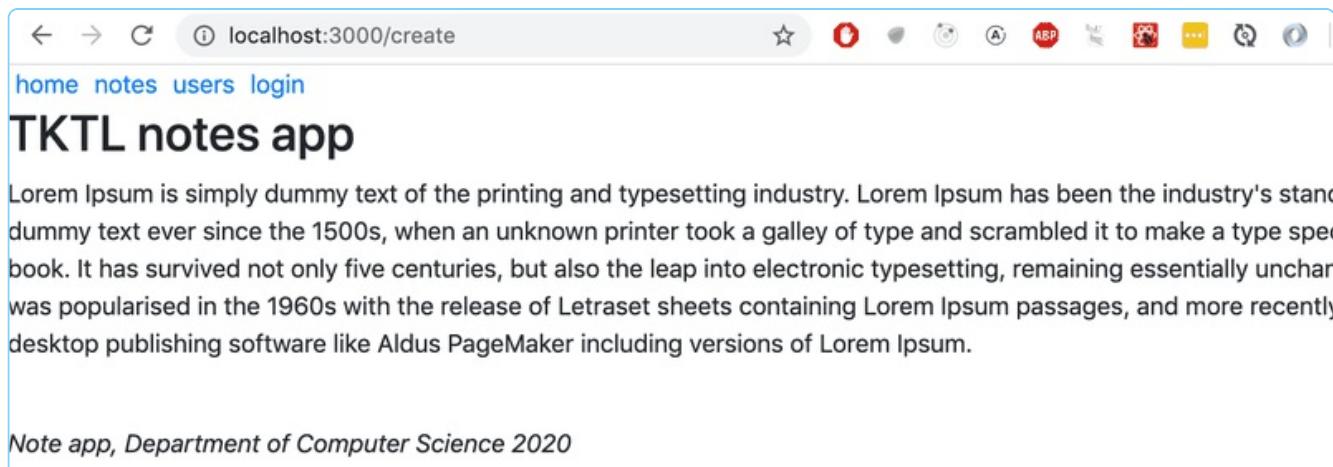
Let's install the package with the command:

```
npm install react-bootstrap
```

Then let's add a [link for loading the CSS stylesheet](#) for Bootstrap inside of the `head` tag in the `public/index.html` file of the application:

```
<head>
  <link
    rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
    integrity="sha384-9ndCyUaIbzAi2FUVXJi0CjmCapSm07SnpJef0486qhLnuZ2cdeRh002iuK6FUUVM"
    crossorigin="anonymous"
  />
  // ...
</head>
```

When we reload the application, we notice that it already looks a bit more stylish:

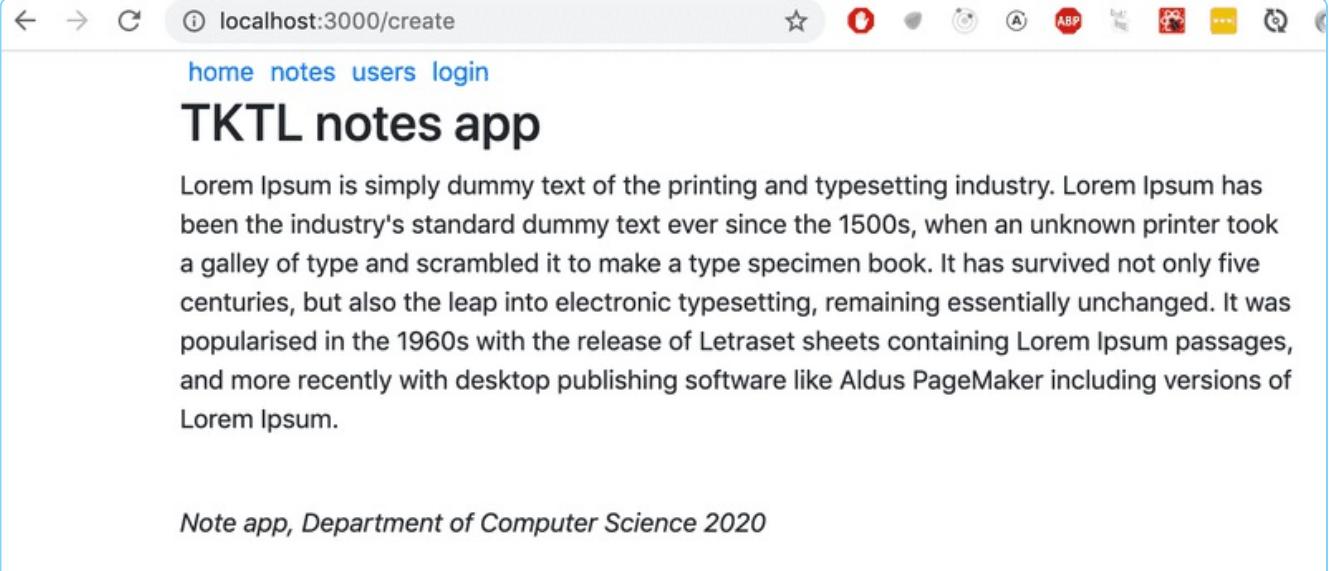


In Bootstrap, all of the contents of the application are typically rendered inside a [container](#). In practice this is accomplished by giving the root `div` element of the application the `container` class attribute:

```
const App = () => [
  // ...

  return (
    <div className="container">      // ...
    </div>
  )
]
```

We notice that this already affected the appearance of the application. The content is no longer as close to the edges of the browser as it was earlier:



The screenshot shows a web browser window with the URL `localhost:3000/create`. The page title is "TKTL notes app". The navigation bar includes links for "home", "notes", "users", and "login". The main content area contains a large block of placeholder text: "Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum." Below this text is a note: "Note app, Department of Computer Science 2020". The browser interface shows various icons in the top right corner.

Tables

Next, let's make some changes to the `Notes` component so that it renders the list of notes as a [table](#). React Bootstrap provides a built-in `Table` component for this purpose, so there is no need to define CSS classes separately.

```
const Notes = ({ notes }) => [
  <div>
    <h2>Notes</h2>
    <Table striped>      <tbody>
      {notes.map(note =>
        <tr key={note.id}>
          <td>
            <Link to={`/notes/${note.id}`}>
              {note.content}
            </Link>
          </td>
          <td>
            {note.user}
          </td>
        </tr>
      )}
    </tbody>
  </Table>
</div>
]
```

The appearance of the application is quite stylish:

The screenshot shows a web browser window with the URL `localhost:3000/notes`. The page has a header with links for `home`, `notes`, `users`, and `login`. The main content is titled "Notes". There are three cards displayed:

- HTML is easy** by Matti Luukkainen
- Browser can execute only Javascript** by Matti Luukkainen
- The most important methods of HTTP protocol are ovat GET are POST** by Arto Hellas

Notice that the React Bootstrap components have to be imported separately from the library as shown below:

```
import { Table } from 'react-bootstrap'
```

Forms

Let's improve the form in the `Login` view with the help of Bootstrap [forms](#).

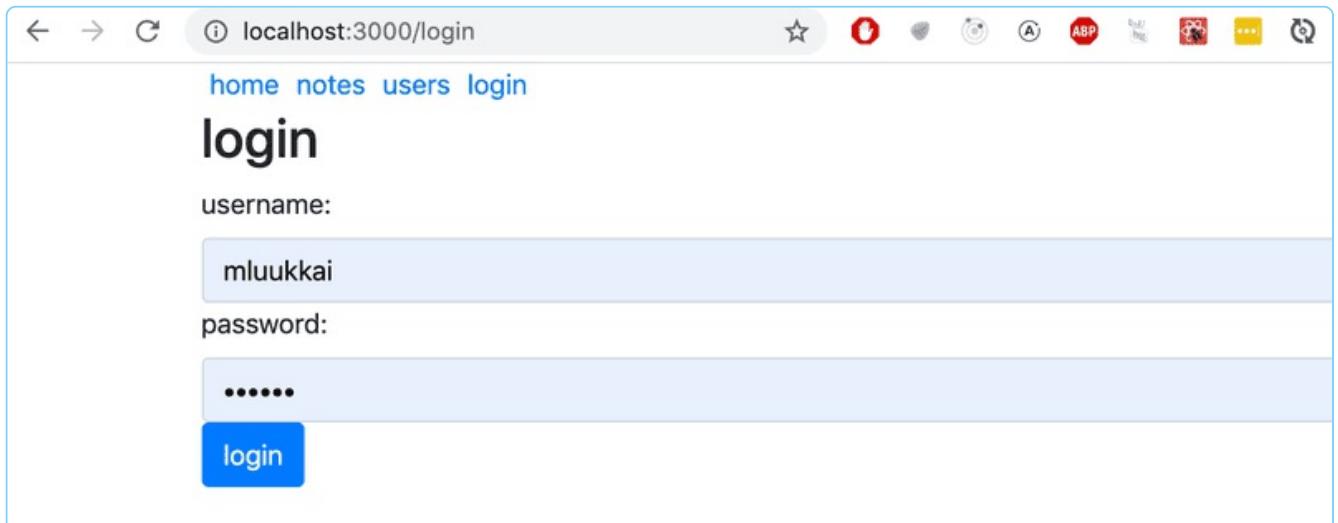
React Bootstrap provides built-in [components](#) for creating forms (although the documentation for them is slightly lacking):

```
let Login = (props) => [
  // ...
  return (
    <div>
      <h2>login</h2>
      <Form onSubmit={onSubmit}>
        <Form.Group>
          <Form.Label>username:</Form.Label>
          <Form.Control
            type="text"
            name="username"
          />
        </Form.Group>
        <Form.Group>
          <Form.Label>password:</Form.Label>
          <Form.Control
            type="password"
          />
        </Form.Group>
        <Button variant="primary" type="submit">
          login
        </Button>
      </Form>
    </div>
  )
}
```

The number of components we need to import increases:

```
import { Table, Form, Button } from 'react-bootstrap'
```

After switching over to the Bootstrap form, our improved application looks like this:



home notes users login

login

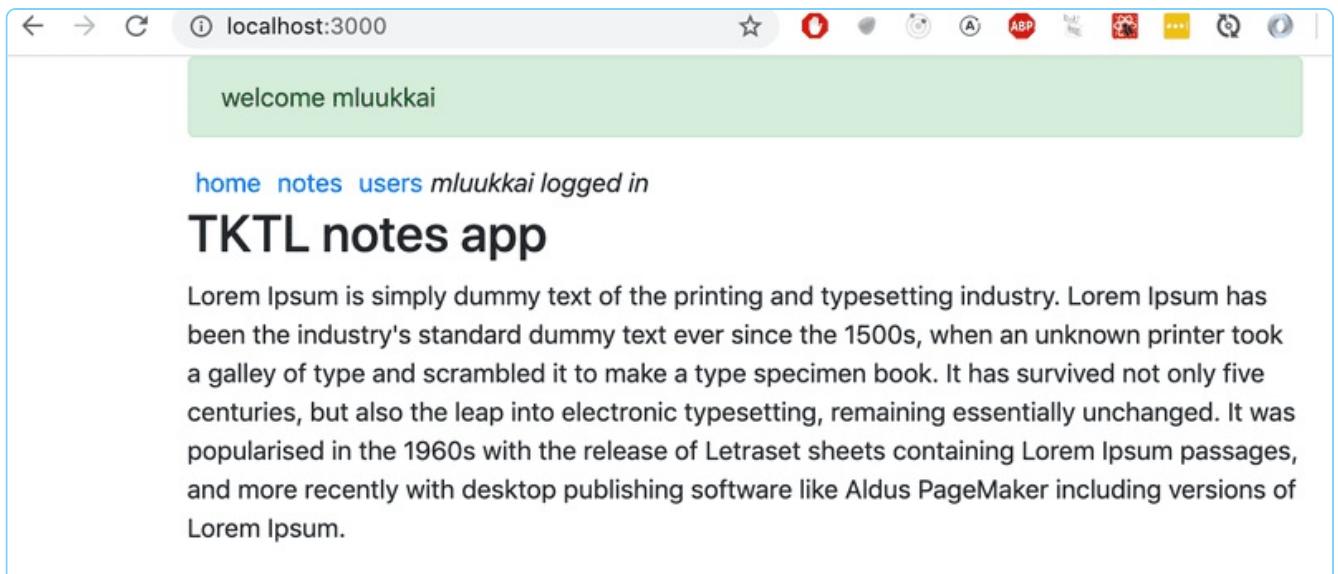
username:

password:

login

Notification

Now that the login form is in better shape, let's take a look at improving our application's notifications:



welcome mluukkai

home notes users mluukkai logged in

TKTL notes app

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Let's add a message for the notification when a user logs into the application. We will store it in the `message` variable in the `App` component's state:

```
const App = () => {
  const [notes, setNotes] = useState([])
  // ...
}

const [user, setUser] = useState(null)
```

```
const [message, setMessage] = useState(null)
const login = (user) => {
  setUser(user)
  setMessage(`welcome ${user}`)
  setTimeout(() => {
    setMessage(null)
  }, 10000)
  // ...
}
```

We will render the message as a Bootstrap [Alert](#) component. Once again, the React Bootstrap library provides us with a matching [React component](#):

```
<div className="container">
  {message && <Alert variant="success"> {message} </Alert>} // .
  ..
</div>
```

Navigation structure

Lastly, let's alter the application's navigation menu to use Bootstrap's [Navbar](#) component. The React Bootstrap library provides us with [matching built-in components](#). Through trial and error, we end up with a working solution despite the cryptic documentation:

```
<Navbar collapseOnSelect expand="lg" bg="dark" variant="dark">
  <Navbar.Toggle aria-controls="responsive-navbar-nav" />
  <Navbar.Collapse id="responsive-navbar-nav">
    <Nav className="me-auto">
      <Nav.Link href="#" as="span">
        <Link style={padding} to="/">home</Link>
      </Nav.Link>
      <Nav.Link href="#" as="span">
        <Link style={padding} to="/notes">notes</Link>
      </Nav.Link>
      <Nav.Link href="#" as="span">
        <Link style={padding} to="/users">users</Link>
      </Nav.Link>
      <Nav.Link href="#" as="span">
        {user
          ? <em style={padding}>{user} logged in</em>
          : <Link style={padding} to="/login">login</Link>
        }
      </Nav.Link>
    </Nav>
  </Navbar.Collapse>
</Navbar>
```

The resulting layout has a very clean and pleasing appearance:

localhost:3000

home notes users login

TKTL notes app

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Note app, Department of Computer Science 2020

If the viewport of the browser is narrowed, we notice that the menu "collapses" and it can be expanded by clicking the "hamburger" button:

localhost:3000

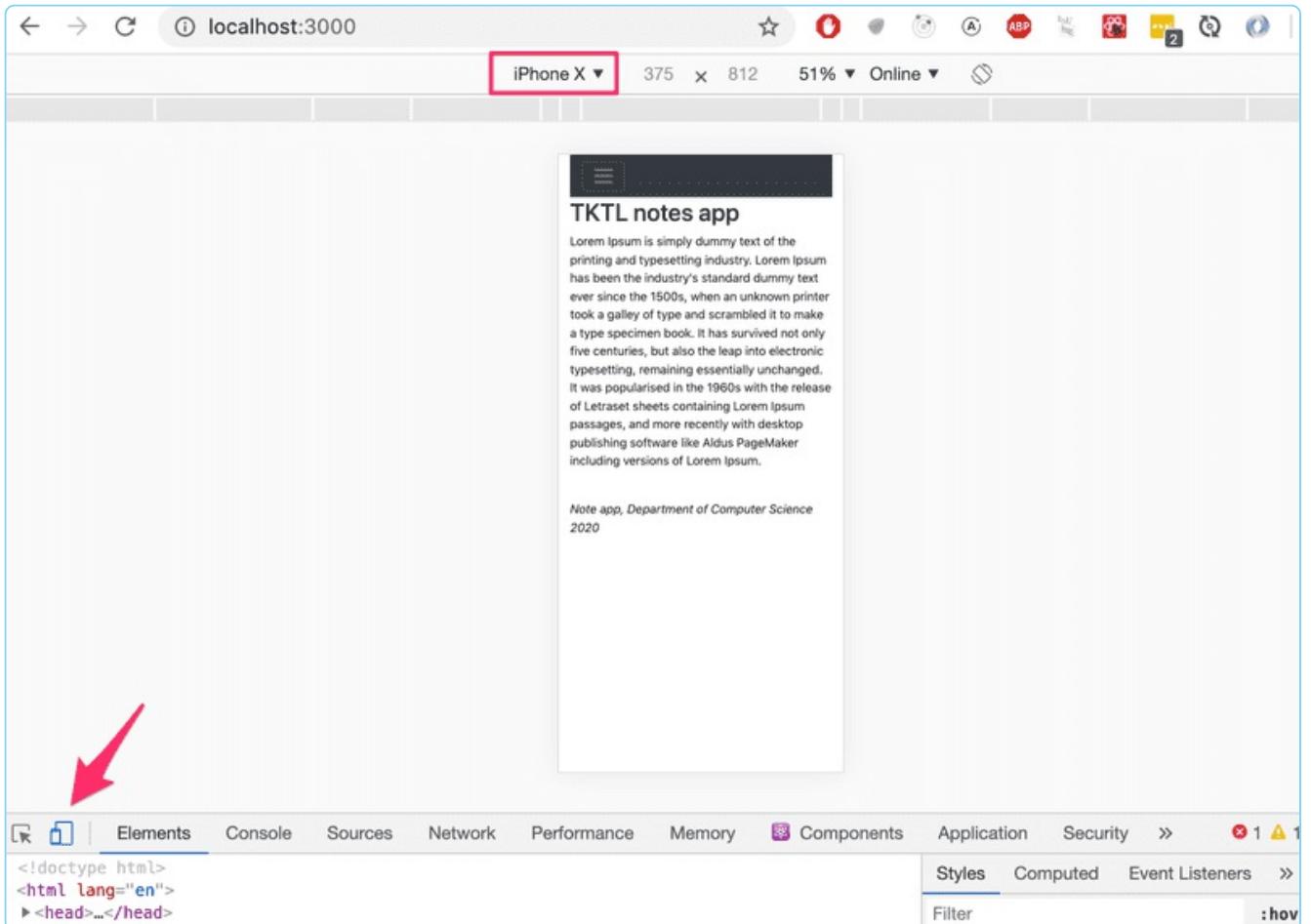
≡

TKTL notes app

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Bootstrap and a large majority of existing UI frameworks produce **responsive** designs, meaning that the resulting applications render well on a variety of different screen sizes.

Chrome's developer tools make it possible to simulate using our application in the browser of different mobile clients:



You can find the complete code for the application [here](#).

Material UI

As our second example, we will look into the [MaterialUI](#) React library, which implements the [Material Design](#) visual language developed by Google.

Install the library with the command

```
npm install @mui/material @emotion/react @emotion/styled
```

Now let's use MaterialUI to do the same modifications to the code we did earlier with Bootstrap.

Render the contents of the whole application within a [Container](#):

```
import { Container } from '@mui/material'

const App = () => [
  // ...
  return [
    <Container>
      // ...
    </Container>
  ]
]
```

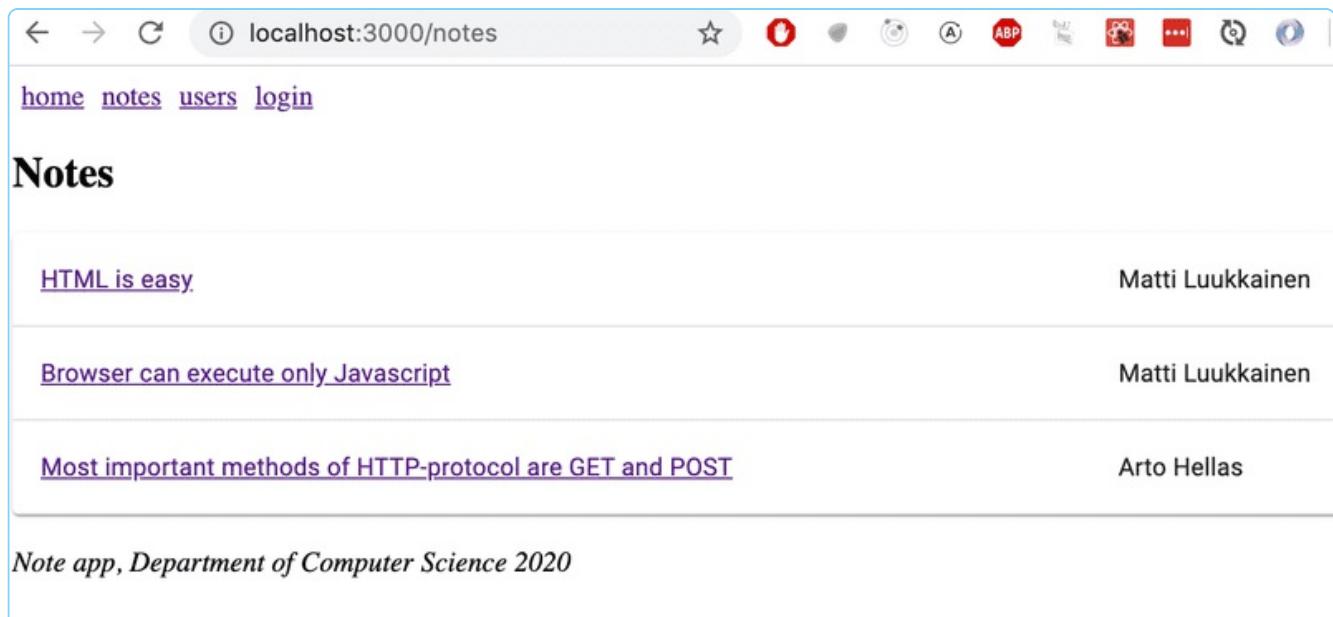
Table

Let's start with the *Notes* component. We'll render the list of notes as a `table`:

```
const Notes = ({ notes }) => [
  <div>
    <h2>Notes</h2>

    <TableContainer component={Paper}>
      <Table>
        <TableBody>
          {notes.map(note => [
            <TableRow key={note.id}>
              <TableCell>
                <Link to={`/notes/${note.id}`}>{note.content}</Link>
              </TableCell>
              <TableCell>
                {note.user}
              </TableCell>
            </TableRow>
          )]}
        </TableBody>
      </Table>
    </TableContainer>
  </div>
]
```

The table looks like so:



A screenshot of a web browser window displaying a list of notes. The address bar shows "localhost:3000/notes". The page title is "Notes". Below the title, there is a list of three notes, each consisting of a link to the note content and the user who created it.

Note Content	User
HTML is easy	Matti Luukkainen
Browser can execute only Javascript	Matti Luukkainen
Most important methods of HTTP-protocol are GET and POST	Arto Hellas

Note app, Department of Computer Science 2020

One less pleasant feature of Material UI is that each component has to be imported separately. The import list for the notes page is quite long:

```
import {
  Container,
```

```
Table,
TableBody,
TableCell,
TableContainer,
TableRow,
Paper,
] from '@mui/material'
```

Form

Next, let's make the login form in the *Login* view better using the **TextField** and **Button** components:

```
const Login = (props) => [
  const navigate = useNavigate()

  const onSubmit = (event) => [
    event.preventDefault()
    props.onLogin('mluukkai')
    navigate('/')

  return [
    <div>
      <h2>login</h2>
      <form onSubmit={onSubmit}>
        <div>
          <TextField label="username" />
        </div>
        <div>
          <TextField label="password" type='password' />
        </div>
        <div>
          <Button variant="contained" color="primary" type="submit">
            login
          </Button>
        </div>
      </form>
    </div>
  ]
]
```

The result is:

localhost:3000/login

[home](#) [notes](#) [users](#) [login](#)

login

username
mluukkai

password

LOGIN

Note app, Department of Computer Science 2020

MaterialUI, unlike Bootstrap, does not provide a component for the form itself. The form here is an ordinary HTML `form` element.

Remember to import all the components used in the form.

Notification

The notification displayed on login can be done using the `Alert` component, which is quite similar to Bootstrap's equivalent component:

```
<div>
  {(message && <Alert severity="success"> {message} </Alert> )}</div>
```

Alert is quite stylish:

localhost:3000

[home](#) [notes](#) [users](#) *mluukkai logged in*

TKTL notes app

>Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Note app, Department of Computer Science 2020

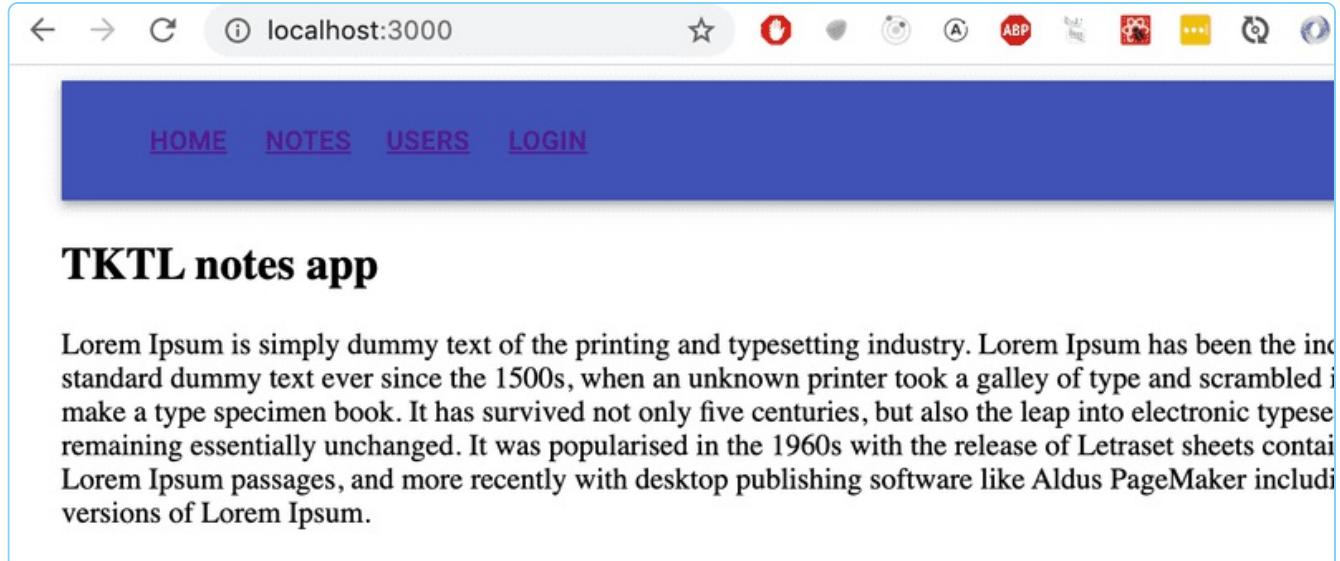
Navigation structure

We can implement navigation using the `AppBar` component.

If we use the example code from the documentation

```
<AppBar position="static">
  <Toolbar>
    <IconButton edge="start" color="inherit" aria-label="menu">
      </IconButton>
    <Button color="inherit">
      <Link to="/">home</Link>
    </Button>
    <Button color="inherit">
      <Link to="/notes">notes</Link>
    </Button>
    <Button color="inherit">
      <Link to="/users">users</Link>
    </Button>
    <Button color="inherit">
      {user
        ? <em>{user} logged in</em>
        : <Link to="/login">login</Link>
      }
    </Button>
  </Toolbar>
</AppBar>
```

we do get working navigation, but it could look better



We can find a better way in the [documentation](#). We can use `component props` to define how the root element of a MaterialUI component is rendered.

By defining

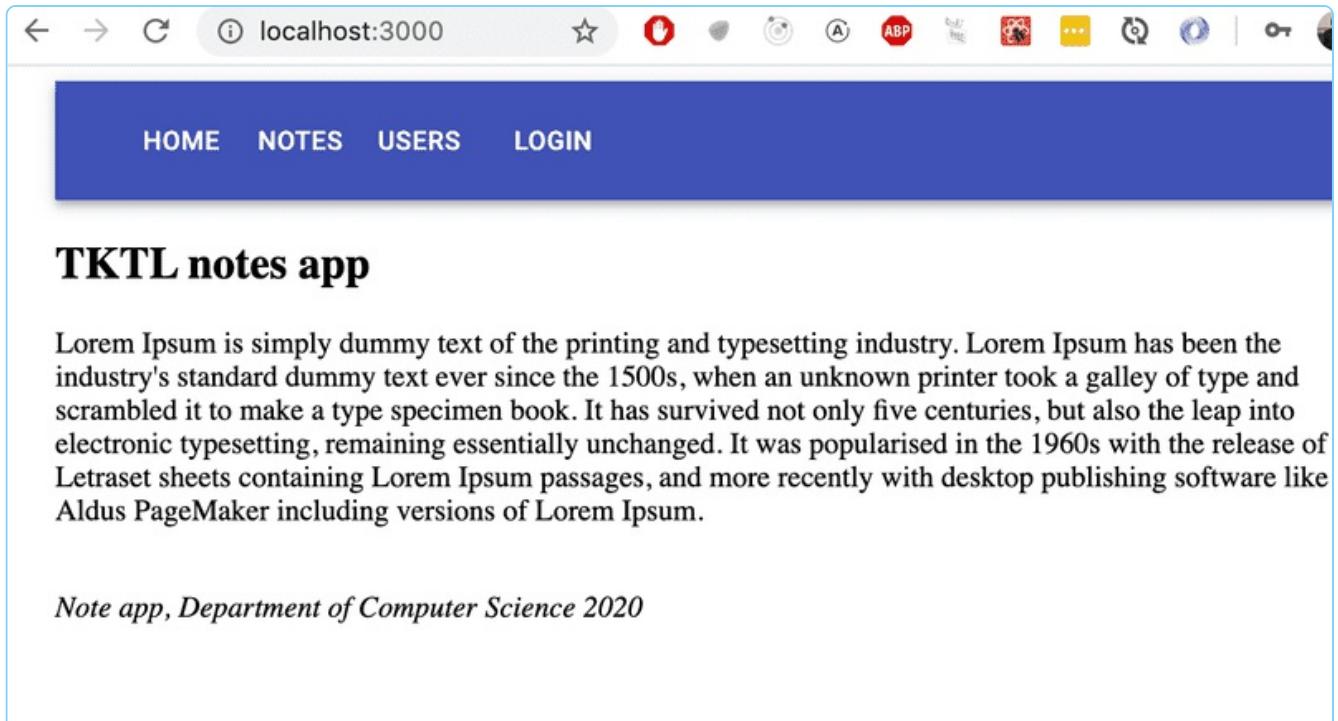
```
<Button color="inherit" component={Link} to="/">
  home
</Button>
```

the *Button* component is rendered so that its root component is react-router-dom's *Link*, which receives its path as the prop field *to*.

The code for the navigation bar is the following:

```
<AppBar position="static">
  <Toolbar>
    <Button color="inherit" component={Link} to="/">
      home
    </Button>
    <Button color="inherit" component={Link} to="/notes">
      notes
    </Button>
    <Button color="inherit" component={Link} to="/users">
      users
    </Button>
    {user
      ? <em>{user} logged in</em>
      : <Button color="inherit" component={Link} to="/login">
        login
      </Button>
    }
  </Toolbar>
</AppBar>
```

and it looks like we want it to:

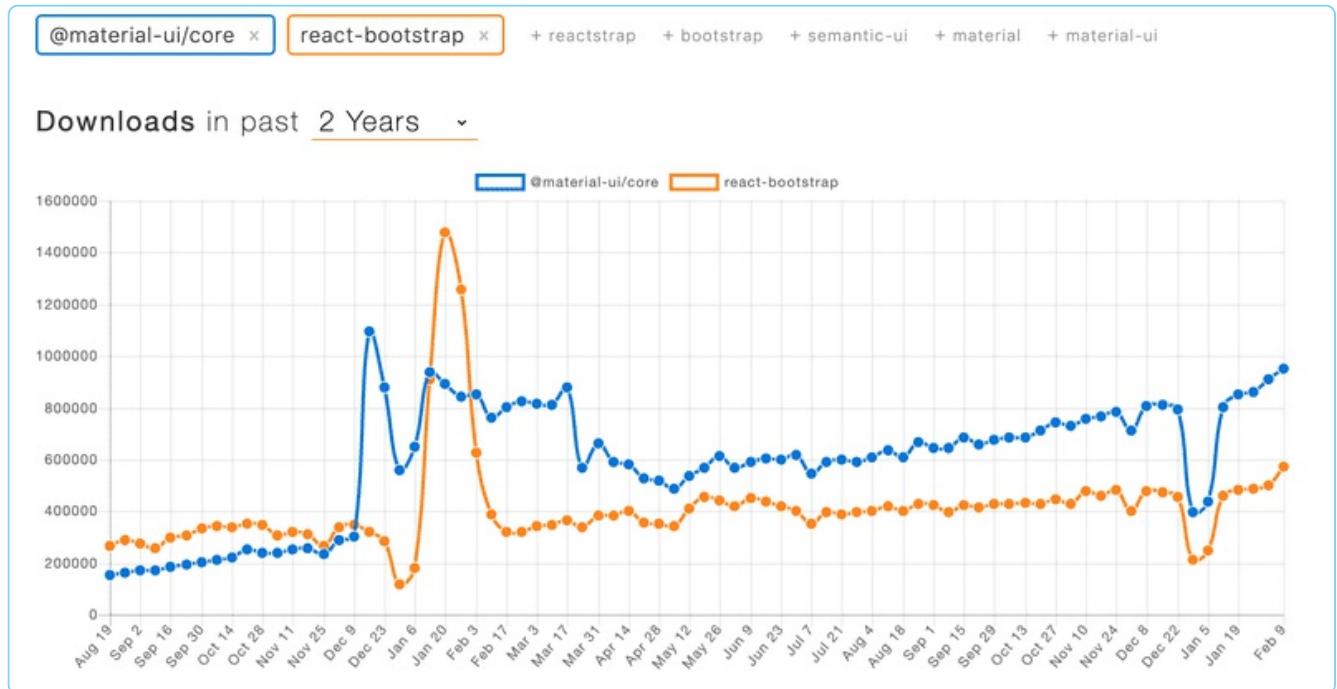


The code of the application can be found [here](#).

Closing thoughts

The difference between react-bootstrap and MaterialUI is not big. It's up to you which one you find

better looking. I have not used MaterialUI a lot, but my first impressions are positive. Its documentation is a bit better than react-bootstrap's. According to <https://www.npmtrends.com/> which tracks the popularity of different npm-libraries, MaterialUI passed react-bootstrap in popularity at the end of 2018:



In the two previous examples, we used the UI frameworks with the help of React-integration libraries.

Instead of using the [React Bootstrap](#) library, we could have just as well used Bootstrap directly by defining CSS classes for our application's HTML elements. Instead of defining the table with the *Table* component:

```
<Table striped>
  // ...
</Table>
```

We could have used a regular HTML *table* and added the required CSS class:

```
<table className="table striped">
  // ...
</table>
```

The benefit of using the React Bootstrap library is not that evident from this example.

In addition to making the frontend code more compact and readable, another benefit of using React UI framework libraries is that they include the JavaScript that is needed to make specific components work. Some Bootstrap components require a few unpleasant [JavaScript dependencies](#) that we would prefer not to include in our React applications.

Some potential downsides to using UI frameworks through integration libraries instead of using them "directly" are that integration libraries may have unstable APIs and poor documentation. The situation with [Semantic UI React](#) is a lot better than with many other UI frameworks, as it is an official React integration library.

There is also the question of whether or not UI framework libraries should be used in the first place. It is up to everyone to form their own opinion, but for people lacking knowledge in CSS and web design, they are very useful tools.

Other UI frameworks

Here are some other UI frameworks for your consideration. If you do not see your favorite UI framework in the list, please make a pull request to the course material for adding it.

- <https://bulma.io/>
- <https://ant.design/>
- <https://get.foundation/>
- <https://chakra-ui.com/>
- <https://tailwindcss.com/>
- <https://semantic-ui.com/>
- <https://mantine.dev/>
- <https://react.fluentui.dev/>
- <https://storybook.js.org>
- <https://www.primefaces.org/primereact/>
- <https://v2.grommet.io>
- <https://blueprintjs.com>
- <https://evergreen.segment.com>
- <https://www.radix-ui.com/>
- <https://react-spectrum.adobe.com/react-aria/index.html>
- <https://master.co/>
- <https://nextui.org/>
- <https://daisyui.com/>
- <https://ui.shadcn.com/>
- <https://www.tremor.so/>
- <https://headlessui.com/>

Styled components

There are also [other ways](#) of styling React applications that we have not yet taken a look at.

The [styled components](#) library offers an interesting approach for defining styles through [tagged template literals](#) that were introduced in ES6.

Let's make a few changes to the styles of our application with the help of styled components. First, install the package with the command:

```
npm install styled-components
```

Then let's define two components with styles:

```
import styled from 'styled-components'

const Button = styled.button`  
background: Bisque;  
font-size: 1em;  
margin: 1em;  
padding: 0.25em 1em;
```

```

border: 2px solid Chocolate;
border-radius: 3px;
`
```

```

const Input = styled.input`
margin: 0.25em;
`
```

The code above creates styled versions of the *button* and *input* HTML elements and then assigns them to the *Button* and *Input* variables.

The syntax for defining the styles is quite interesting, as the CSS rules are defined inside of backticks.

The styled components that we defined work exactly like regular *button* and *input* elements, and they can be used in the same way:

```

const Login = (props) => [
// ...
return [
<div>
  <h2>login</h2>
  <form onSubmit={onSubmit}>
    <div>
      username:
      <Input />          </div>
    <div>
      password:
      <Input type='password' />      </div>
    <Button type="submit" primary=''>login</Button>      </form>
  </div>
]
]
```

Let's create a few more components for styling this application which will be styled versions of *div* elements:

```

const Page = styled.div`
padding: 1em;
background: papayawhip;
`
```

```

const Navigation = styled.div`
background: BurlyWood;
padding: 1em;
`
```

```

const Footer = styled.div`
background: Chocolate;
padding: 1em;
margin-top: 1em;
`
```

Let's use the components in our application:

```
const App = () => {
  // ...

  return [
    <Page>      <Navigation>      <Link style={padding} to="/">home</Link>
    <Link style={padding} to="/notes">notes</Link>
    <Link style={padding} to="/users">users</Link>
    <user
      ? <em>{user} logged in</em>
      : <Link style={padding} to="/login">login</Link>
    >
  </Navigation>
  <Routes>
    <Route path="/notes/:id" element={<Note note={note} />} />
    <Route path="/notes" element={<Notes notes={notes} />} />
    <Route path="/users" element={user ? <Users /> : <Navigate replace to="/login" />} />
    <Route path="/login" element={<Login onLogin={login} />} />
    <Route path="/" element={<Home />} />
  </Routes>

  <Footer>      <em>Note app, Department of Computer Science 2022</em>
  </Footer>    </Page>  ]
}
```

The appearance of the resulting application is shown below:



Styled components have seen consistent growth in popularity in recent times, and quite a lot of people consider it to be the best way of defining styles in React applications.

Exercises

The exercises related to the topics presented here can be found at the end of this course material section in the exercise set [for extending the blog list application](#).

7d: Webpack

In the early days, React was somewhat famous for being very difficult to configure the tools required for application development. To make the situation easier, [Create React App](#) was developed, which eliminated configuration-related problems. [Vite](#), which is also used in the course, has recently replaced Create React App in new applications.

Both Vite and Create React App use *bundlers* to do the actual work. We will now familiarize ourselves with the bundler called [Webpack](#) used by Create React App. Webpack was by far the most popular bundler for years. Recently, however, there have been several new generation bundlers such as [esbuild](#) used by Vite, which are significantly faster and easier to use than Webpack. However, e.g. esbuild still lacks some useful features (such as hot reload of the code in the browser), so next we will get to know the old ruler of bundlers, Webpack.

Bundling

We have implemented our applications by dividing our code into separate modules that have been *imported* to places that require them. Even though ES6 modules are defined in the ECMAScript standard, the older browsers do not know how to handle code that is divided into modules.

For this reason, code that is divided into modules must be *bundled* for browsers, meaning that all of the source code files are transformed into a single file that contains all of the application code. When we deployed our React frontend to production in [part 3](#), we performed the bundling of our application with the `npm run build` command. Under the hood, the npm script bundles the source, and this produces the following collection of files in the `dist` directory:

```
|- assets
  |- index-d526a0c5.css
  |- index-e92ae01e.js
  |- react-35ef61ed.svg
  index.html
  vite.svg
```

The `index.html` file located at the root of the `dist` directory is the "main file" of the application which loads the bundled JavaScript file with a `script` tag:

```
<!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-e92ae01e.js"></script>
    <link rel="stylesheet" href="/assets/index-d526a0c5.css" />
  </head>
  <body>
    <div id="root"></div>
```

```
</body>  
</html>
```

As we can see from the example application that was created with Vite, the build script also bundles the application's CSS files into a single `/assets/index-d526a0c5.css` file.

In practice, bundling is done so that we define an entry point for the application, which typically is the `index.js` file. When webpack bundles the code, it includes not only the code from the entry point but also the code that is imported by the entry point, as well as the code imported by its import statements, and so on.

Since part of the imported files are packages like React, Redux, and Axios, the bundled JavaScript file will also contain the contents of each of these libraries.

The old way of dividing the application's code into multiple files was based on the fact that the `index.html` file loaded all of the separate JavaScript files of the application with the help of script tags. This resulted in decreased performance, since the loading of each separate file results in some overhead. For this reason, these days the preferred method is to bundle the code into a single file.

Next, we will create a webpack configuration by hand, suitable for a new React application.

Let's create a new directory for the project with the following subdirectories (`build` and `src`) and files:

```
└── build  
└── package.json  
└── src  
    └── index.js  
└── webpack.config.js
```

The contents of the `package.json` file can e.g. be the following:

```
{  
  "name": "webpack-part7",  
  "version": "0.0.1",  
  "description": "practising webpack",  
  "scripts": {},  
  "license": "MIT"  
}
```

Let's install webpack with the command:

```
npm install --save-dev webpack webpack-cli
```

We define the functionality of webpack in the `webpack.config.js` file, which we initialize with the following content:

```

const path = require('path')

const config = () => {
  return {
    entry: './src/index.js',
    output: [
      path.resolve(__dirname, 'build'),
      filename: 'main.js'
    ]
  }
}

module.exports = config

```

Note: it would be possible to make the definition directly as an object instead of a function:

```

const path = require('path')

const config = {
  entry: './src/index.js',
  output: [
    path.resolve(__dirname, 'build'),
    filename: 'main.js'
  ]
}

module.exports = config

```

An object will suffice in many situations, but we will later need certain features that require the definition to be done as a function.

We will then define a new npm script called *build* that will execute the bundling with webpack:

```

// ...
"scripts": {
  "build": "webpack --mode=development"
}
// ...

```

Let's add some more code to the *src/index.js* file:

```

const hello = name => {
  console.log(`hello ${name}`)
}

```

When we execute the *npm run build* command, our application code will be bundled by webpack. The operation will produce a new *main.js* file that is added under the *build* directory:

```

→ webpack npm run build

> webpack-osa7@0.0.1 build
> webpack --mode=development

asset main.js 1.24 KiB [compared for emit] (name: main)
./src/index.js 56 bytes [built] [code generated]
webpack 5.68.0 compiled successfully in 110 ms
→ webpack cat build/main.js
/*
 * ATTENTION: The "eval" devtool has been used (maybe by default in mode: "development").
 * This devtool is neither made for production nor for readable output files.
 * It uses "eval()" calls to create a separate source file in the browser devtools.
 * If you are trying to read the output file, select a different devtool (https://webpack.js.org/configuration/devtool/)
 * or disable the default devtool with "devtool: false".
 * If you are looking for production-ready output files, see mode: "production"
(https://webpack.js.org/configuration/mode/).
 */
/***** () => { // webpackBootstrap

```

The file contains a lot of stuff that looks quite interesting. We can also see the code we wrote earlier at the end of the file:

```

eval("const hello = name => {\n  console.log(`hello ${name}`)\n}\n\n//# sourceMappingURL=webpack://webpack-osa7./src/index.js?")

```

Let's add an *App.js* file under the *src* directory with the following content:

```

const App = () => {
  return null
}

export default App

```

Let's import and use the *App* module in the *index.js* file:

```

import App from './App'

const hello = name => {
  console.log(`hello ${name}`)
}

App()

```

When we bundle the application again with the *npm run build* command, we notice that webpack has acknowledged both files:

```
→ webpack npm run build

> webpack-osa7@0.0.1 build
> webpack --mode=development

asset main.js 4.21 KiB [emitted] (name: main)
  runtime modules 670 bytes 3 modules
  cacheable modules 144 bytes
    ./src/index.js 89 bytes [built] [code generated]
    ./src/App.js 55 bytes [built] [code generated]
webpack 5.68.0 compiled successfully in 114 ms
```

Our application code can be found at the end of the bundle file in a rather obscure format:

```
eval("__webpack_require__.r(__webpack_exports__);/* harmony import */ var _App
__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(/*! ./App */ "./src/App.js");
\n\n\nconst hello = name => {\n  console.log(`hello ${name}`)\n}\n\n;(0,_App_
__WEBPACK_IMPORTED_MODULE_0__["default"])((){}\n// sourceURL=webpack://webpack-
osa7./src/index.js?");
```

Configuration file

Let's take a closer look at the contents of our current `webpack.config.js` file:

```
const path = require('path')

const config = () => {
  return {
    entry: './src/index.js',
    output: [
      path.resolve(__dirname, 'build'),
      filename: 'main.js'
    ]
  }
}

module.exports = config
```

The configuration file has been written in JavaScript and the function returning the configuration object is exported using Node's module syntax.

Our minimal configuration definition almost explains itself. The `entry` property of the configuration object specifies the file that will serve as the entry point for bundling the application.

The `output` property defines the location where the bundled code will be stored. The target directory must be defined as an *absolute path*, which is easy to create with the `path.resolve` method. We also use `__dirname` which is a variable in Node that stores the path to the current directory.

Bundling React

Next, let's transform our application into a minimal React application. Let's install the required libraries:

```
npm install react react-dom
```

And let's turn our application into a React application by adding the familiar definitions in the *index.js* file:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

We will also make the following changes to the *App.js* file:

```
import React from 'react' // we need this now also in component files

const App = () => {
  return [
    <div>
      hello webpack
    </div>
  ]
}

export default App
```

We still need the *build/index.html* file that will serve as the "main page" of our application, which will load our bundled JavaScript code with a *script* tag:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/javascript" src="./main.js"></script>
  </body>
</html>
```

When we bundle our application, we run into the following problem:

```

ERROR in ./src/App.js 2:2
Module parse failed: Unexpected token (2:2)
You may need an appropriate loader to handle this file type, currently no loaders
are configured to process this file. See https://webpack.js.org/concepts#loaders
| const App = () => (
|   <div>hello webpack</div>
| )
|
@ ./src/index.js 1:0-24 7:0-3

webpack 5.68.0 compiled with 1 error in 121 ms

```

Loaders

The error message from webpack states that we may need an appropriate *loader* to bundle the *App.js* file correctly. By default, webpack only knows how to deal with plain JavaScript. Although we may have become unaware of it, we are using [JSX](#) for rendering our views in React. To illustrate this, the following code is not regular JavaScript:

```

const App = () => {
  return (
    <div>
      hello webpack
    </div>
  )
}

```

The syntax used above comes from JSX and it provides us with an alternative way of defining a React element for an HTML *div* tag.

We can use [loaders](#) to inform webpack of the files that need to be processed before they are bundled.

Let's configure a loader to our application that transforms the JSX code into regular JavaScript:

```

const path = require('path')

const config = () => {
  return {
    entry: './src/index.js',
    output: [
      path.resolve(__dirname, 'build'),
      filename: 'main.js'
    ],
    module: [
      rules: [
        {
          test: /\.js$/,
          loader: 'babel-loader',
          options: [
            {
              presets: ['@babel/preset-react']
            }
          ]
        }
      ]
    ]
  }
}

module.exports = config

```

Loaders are defined under the *module* property in the *rules* array.

The definition of a single loader consists of three parts:

```
{  
  test: /\.js$/,  
  loader: 'babel-loader',  
  options: [  
    presets: ['@babel/preset-react']  
  ]  
}
```

The *test* property specifies that the loader is for files that have names ending with *.js*. The *loader* property specifies that the processing for those files will be done with [babel-loader](#). The *options* property is used for specifying parameters for the loader, which configure its functionality.

Let's install the loader and its required packages as a *development dependency*:

```
npm install @babel/core babel-loader @babel/preset-react --save-dev
```

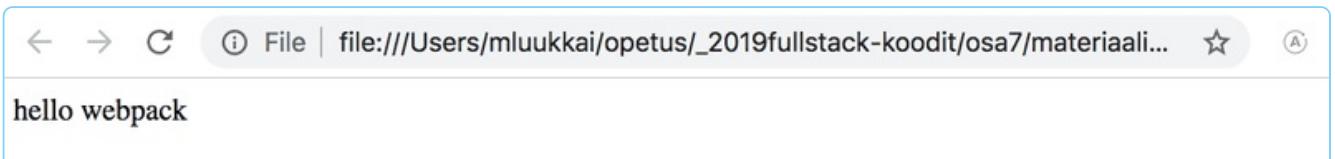
Bundling the application will now succeed.

If we make some changes to the *App* component and take a look at the bundled code, we notice that the bundled version of the component looks like this:

```
const App = () =>  
  react__WEBPACK_IMPORTED_MODULE_0___default.a.createElement(  
    'div',  
    null,  
    'hello webpack'  
  )
```

As we can see from the example above, the React elements that were written in JSX are now created with regular JavaScript by using React's [createElement](#) function.

You can test the bundled application by opening the *build/index.html* file with the *open file* functionality of your browser:



It's worth noting that if the bundled application's source code uses *async/await*, the browser will not render anything on some browsers. [Googling the error message in the console](#) will shed some light on the issue. With the [previous solution](#) being deprecated we now have to install two more missing dependencies, that is [core-js](#) and [regenerator-runtime](#):

```
npm install core-js regenerator-runtime
```

You need to import these dependencies at the top of the `index.js` file:

```
import 'core-js/stable/index.js'  
import 'regenerator-runtime/runtime.js'
```

Our configuration contains nearly everything that we need for React development.

Transpilers

The process of transforming code from one form of JavaScript to another is called [transpiling](#). The general definition of the term is to compile source code by transforming it from one language to another.

By using the configuration from the previous section, we are *transpiling* the code containing JSX into regular JavaScript with the help of [babel](#), which is currently the most popular tool for the job.

As mentioned in part 1, most browsers do not support the latest features that were introduced in ES6 and ES7, and for this reason, the code is usually transpiled to a version of JavaScript that implements the ES5 standard.

The compilation process that is executed by Babel is defined with [plugins](#). In practice, most developers use ready-made [presets](#) that are groups of pre-configured plugins.

Currently, we are using the [@babel/preset-react](#) preset for transpiling the source code of our application:

```
{  
  test: /\.js$/,  
  loader: 'babel-loader',  
  options: [  
    presets: ['@babel/preset-react']  
  ]  
}
```

Let's add the [@babel/preset-env](#) plugin that contains everything needed to take code using all of the latest features and to transpile it to code that is compatible with the ES5 standard:

```
{  
  test: /\.js$/,  
  loader: 'babel-loader',  
  options: [  
    presets: ['@babel/preset-env', '@babel/preset-react']  
  ]  
}
```

Let's install the preset with the command:

```
npm install @babel/preset-env --save-dev
```

When we transpile the code, it gets transformed into old-school JavaScript. The definition of the transformed `App` component looks like this:

```
var App = function App() {
  return _react2.default.createElement('div', null, 'hello webpack')
};
```

As we can see, variables are declared with the `var` keyword, as ES5 JavaScript does not understand the `const` keyword. Arrow functions are also not used, which is why the function definition used the `function` keyword.

CSS

Let's add some CSS to our application. Let's create a new `src/index.css` file:

```
.container {
  margin: 10px;
  background-color: #dee8e4;
}
```

Then let's use the style in the `App` component:

```
const App = () => {
  return [
    <div className="container">
      hello webpack
    </div>
  ]
};
```

And we import the style in the `index.js` file:

```
import './index.css'
```

This will cause the transpilation process to break:

```
ERROR in ./src/index.css 1:0
Module parse failed: Unexpected token (1:0)
You may need an appropriate loader to handle this file type, currently no loaders
are configured to process this file. See https://webpack.js.org/concepts#loaders
> .container {
|   margin: 10;
|   background-color: #dee8e4;
@ ./src/index.js 4:0-21
```

When using CSS, we have to use `css` and `style` loaders:

```
{
  rules: [
```

```

    [
      test: /\.js$/,
      loader: 'babel-loader',
      options: {
        presets: ['@babel/preset-react', '@babel/preset-env'],
      },
    ],
    [
      test: /\.css$/,
      use: ['style-loader', 'css-loader'],
    ],
  ],
}

```

The job of the [css loader](#) is to load the *CSS* files and the job of the [style loader](#) is to generate and inject a *style* element that contains all of the styles of the application.

With this configuration, the *CSS* definitions are included in the *main.js* file of the application. For this reason, there is no need to separately import the *CSS* styles in the main *index.html* file.

If needed, the application's *CSS* can also be generated into its own separate file, by using the [mini-css-extract-plugin](#).

When we install the loaders:

```
npm install style-loader css-loader --save-dev
```

The bundling will succeed once again and the application gets new styles.

Webpack-dev-server

The current configuration makes it possible to develop our application but the workflow is awful (to the point where it resembles the development workflow with Java). Every time we make a change to the code, we have to bundle it and refresh the browser to test it.

The [webpack-dev-server](#) offers a solution to our problems. Let's install it with the command:

```
npm install --save-dev webpack-dev-server
```

Let's define an npm script for starting the dev server:

```

{
  // ...
  "scripts": {
    "build": "webpack --mode=development",
    "start": "webpack serve --mode=development"
  }
}

```

Let's also add a new *devServer* property to the configuration object in the *webpack.config.js* file:

```
const config = [
  entry: './src/index.js',
  output: [

```

```

    path: path.resolve(__dirname, 'build'),
    filename: 'main.js',
  },
  devServer: {
    static: path.resolve(__dirname, 'build'),
    compress: true,
    port: 3000,
  },
}

```

The `npm start` command will now start the dev-server at port 3000, meaning that our application will be available by visiting <http://localhost:3000> in the browser. When we make changes to the code, the browser will automatically refresh the page.

The process for updating the code is fast. When we use the dev-server, the code is not bundled the usual way into the `main.js` file. The result of the bundling exists only in memory.

Let's extend the code by changing the definition of the `App` component as shown below:

```

import React, { useState } from 'react'
import './index.css'

const App = () => {
  const [counter, setCounter] = useState(0)

  return (
    <div className="container">
      hello webpack {counter} clicks
      <button onClick={() => setCounter(counter + 1)}>
        press
      </button>
    </div>
  )
}

export default App

```

The application works nicely and the development workflow is quite smooth.

Source maps

Let's extract the click handler into its own function and store the previous value of the counter in its own `values` state:

```

const App = () => {
  const [counter, setCounter] = useState(0)
  const [values, setValues] = useState()
  const handleClick = () => {
    setCounter(counter + 1)
    setValues({ ...values, current: counter })
  }
  return (
    <div className="container">
      hello webpack {counter} clicks
      <button onClick={handleClick}> press
    </button>
  )
}

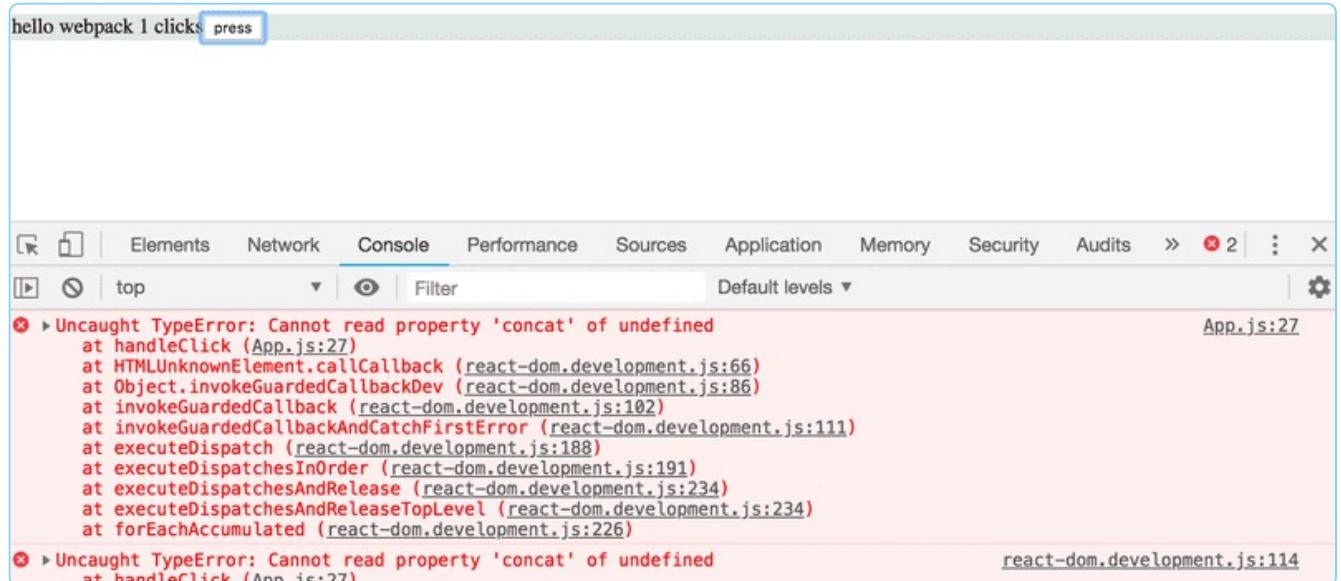
```

```

        </div>
    }
}

```

The application no longer works and the console will display the following error:



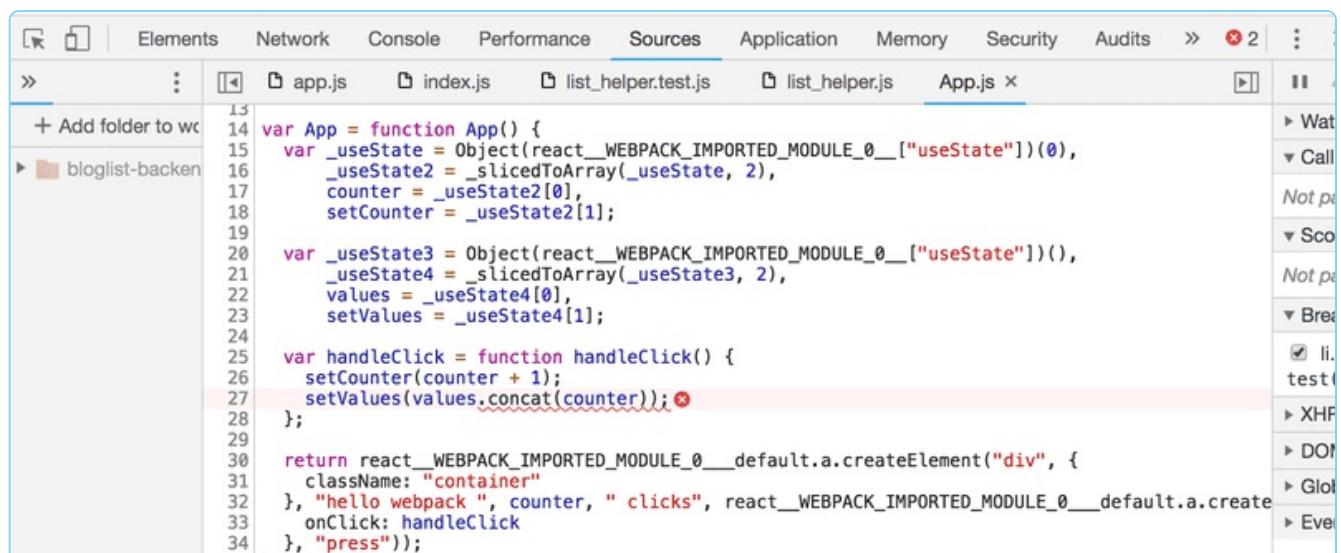
We know that the error is in the onClick method, but if the application was any larger the error message would be quite difficult to track down:

```

App.js:27 Uncaught TypeError: Cannot read property 'concat' of undefined
    at handleClick (App.js:27)

```

The location of the error indicated in the message does not match the actual location of the error in our source code. If we click the error message, we notice that the displayed source code does not resemble our application code:



Of course, we want to see our actual source code in the error message.

Luckily, fixing this error message is quite easy. We will ask webpack to generate a so-called [source map](#).

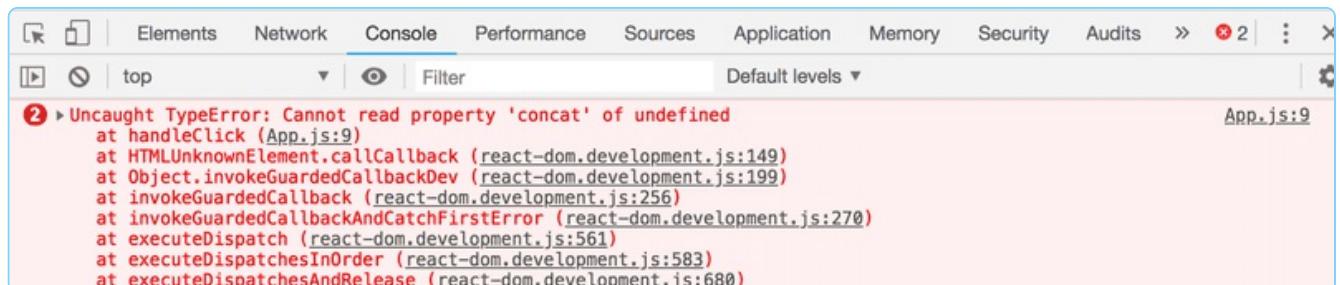
`map` for the bundle, which makes it possible to *map errors* that occur during the execution of the bundle to the corresponding part in the original source code.

The source map can be generated by adding a new `devtool` property to the configuration object with the value 'source-map':

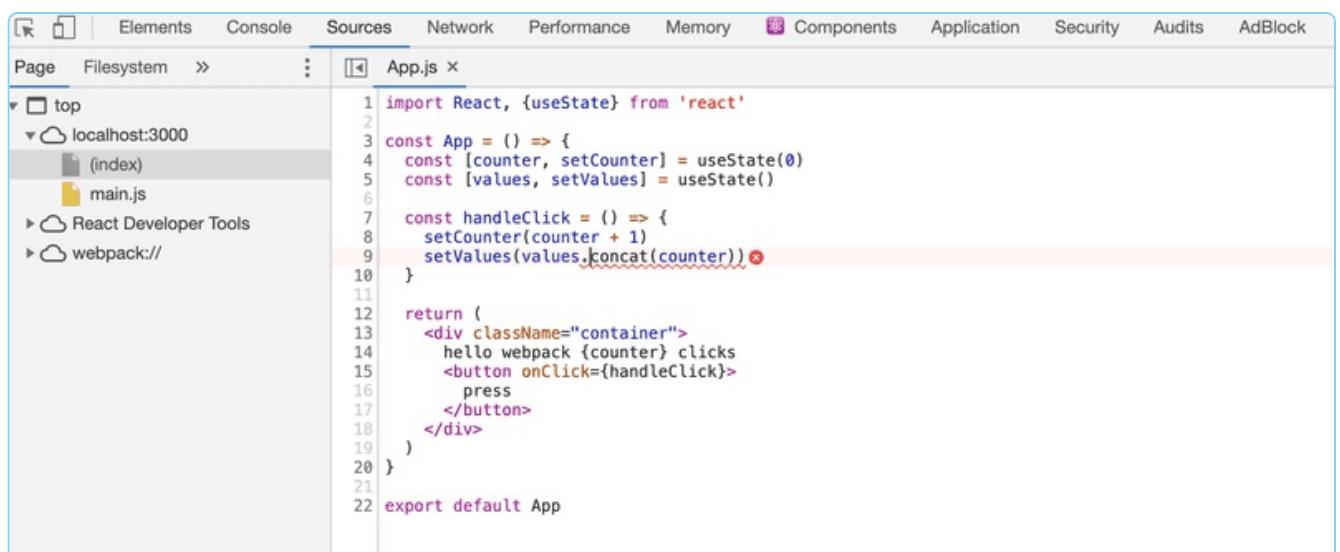
```
const config = {
  entry: './src/index.js',
  output: [
    // ...
  ],
  devServer: {
    // ...
  },
  devtool: 'source-map', // ...
};
```

Webpack has to be restarted when we make changes to its configuration. It is also possible to make webpack watch for changes made to itself, but we will not do that this time.

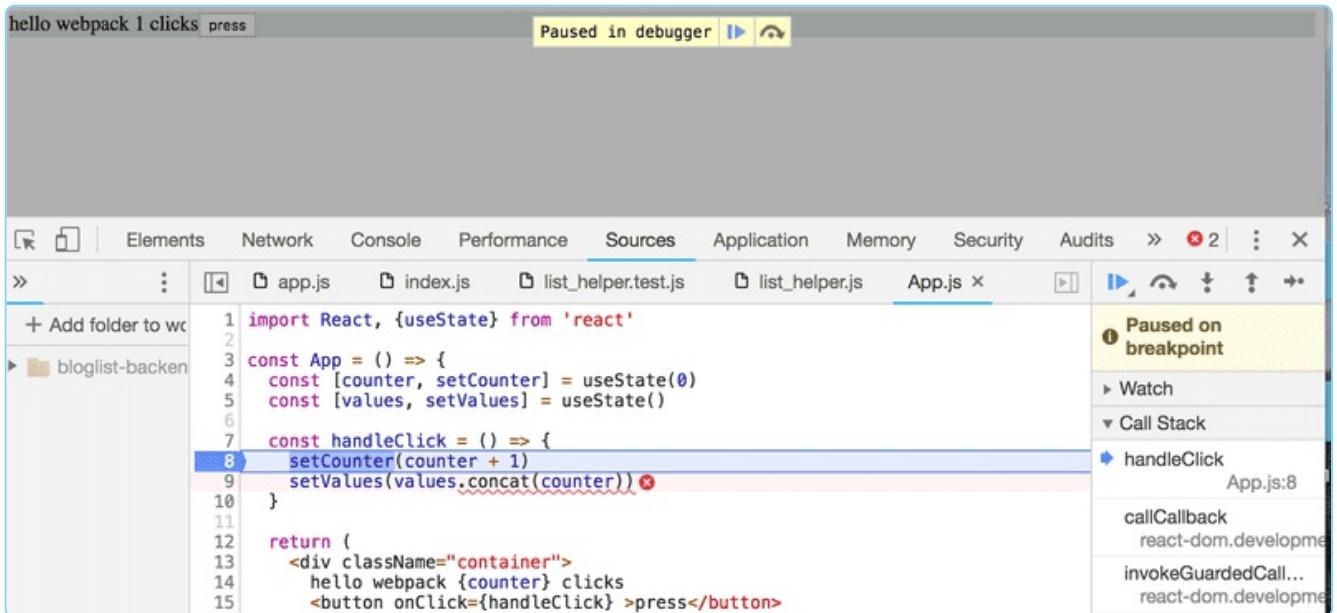
The error message is now a lot better



since it refers to the code we wrote:



Generating the source map also makes it possible to use the Chrome debugger:



Let's fix the bug by initializing the state of `values` as an empty array:

```
const App = () => {
  const [counter, setCounter] = useState(0)
  const [values, setValues] = useState([])
  // ...
}
```

Minifying the code

When we deploy the application to production, we are using the `main.js` code bundle that is generated by webpack. The size of the `main.js` file is 1009487 bytes even though our application only contains a few lines of our code. The large file size is because the bundle also contains the source code for the entire React library. The size of the bundled code matters since the browser has to load the code when the application is first used. With high-speed internet connections, 1009487 bytes is not an issue, but if we were to keep adding more external dependencies, loading speeds could become an issue, particularly for mobile users.

If we inspect the contents of the bundle file, we notice that it could be greatly optimized in terms of file size by removing all of the comments. There's no point in manually optimizing these files, as there are many existing tools for the job.

The optimization process for JavaScript files is called *minification*. One of the leading tools intended for this purpose is [UglifyJS](#).

Starting from version 4 of webpack, the minification plugin does not require additional configuration to be used. It is enough to modify the npm script in the `package.json` file to specify that webpack will execute the bundling of the code in *production* mode:

```
{
  "name": "webpack-part7",
  "version": "0.0.1",
  "description": "practising webpack",
  "scripts": {
```

```

  "build": "webpack --mode=production",    "start": "webpack serve --mode=development"
},
"license": "MIT",
"dependencies": [
  // ...
],
"devDependencies": {
  // ...
}
}

```

When we bundle the application again, the size of the resulting *main.js* decreases substantially:

```
$ ls -l build/main.js
-rw-r--r-- 1 mluukkai  ATKK\hyad-all  227651 Feb  7 15:58 build/main.js
```

The output of the minification process resembles old-school C code; all of the comments and even unnecessary whitespace and newline characters have been removed, variable names have been replaced with a single character.

```

function h(){if(!d){var e=u(p);d=!0;for(var t=c.length;t;)for(s=c,c=[],++f<t;)s
&&s[f].run();f=-1,t=c.length;s=null,d=!1,function(e){if(o==clearTimeout) return
clearTimeout(e);if((o==null||!o)&&clearTimeout) return o=clearTimeout,clearTimeout(e);
try{o(e)}catch(t){try{return o.call(null,e)}catch(t){return o.call(this,e)}}}(e)
}a.nextTick=function(e){var t=new Array(arguments.length-1);if(arguments.length>1)

```

Development and production configuration

Next, let's add a backend to our application by repurposing the now-familiar note application backend.

Let's store the following content in the *db.json* file:

```
{
  "notes": [
    [
      {
        "important": true,
        "content": "HTML is easy",
        "id": "5a3b8481bb01f9cb00ccb4a9"
      },
      {
        "important": false,
        "content": "Mongo can save js objects",
        "id": "5a3b920a61e8c8d3f484bdd0"
      }
    ]
}
```

Our goal is to configure the application with webpack in such a way that, when used locally, the application uses the json-server available in port 3001 as its backend.

The bundled file will then be configured to use the backend available at the <https://notes2023.fly.dev/api/notes> URL.

We will install *axios*, start the json-server, and then make the necessary changes to the application. For the sake of changing things up, we will fetch the notes from the backend with our [custom hook](#) called *useNotes*:

```
import React, { useState, useEffect } from 'react' import axios from 'axios' const useNotes = (url) => {  const [notes, setNotes] = useState([])  useEffect(() => {    axios.get(url).then(response => {      setNotes(response.data)    })  }, [url])  return notes}const App = () => {  const [counter, setCounter] = useState(0)  const [values, setValues] = useState([])  const url = 'https://notes2023.fly.dev/api/notes'  const notes = useNotes(url)  const handleClick = () => {    setCounter(counter + 1)    setValues(values.concat(counter))  }  return (    <div className="container">      hello webpack {counter} clicks      <button onClick={handleClick}>press</button>      <div>{notes.length} notes on server {url}</div>    </div>  )}export default App
```

The address of the backend server is currently hardcoded in the application code. How can we change the address in a controlled fashion to point to the production backend server when the code is bundled for production?

Webpack's configuration function has two parameters, *env* and *argv*. We can use the latter to find out the *mode* defined in the npm script:

```
const path = require('path')const config = [env, argv] => {  console.log('argv.mode:', argv.mode)  return {    // ...  }}module.exports = config
```

Now, if we want, we can set Webpack to work differently depending on whether the application's

operating environment, or *mode*, is set to production or development.

We can also use webpack's `DefinePlugin` for defining *global default constants* that can be used in the bundled code. Let's define a new global constant `BACKEND_URL` that gets a different value depending on the environment that the code is being bundled for:

```
const path = require('path')
const webpack = require('webpack')
const config = [env, argv] => [
  console.log('argv', argv.mode)

  const backend_url = argv.mode === 'production' ? 'https://notes2023.fly.dev'
  /api/notes' : 'http://localhost:3001/notes'
  return [
    entry: './src/index.js',
    output: [
      path: path.resolve(__dirname, 'build'),
      filename: 'main.js'
    ],
    devServer: [
      static: path.resolve(__dirname, 'build'),
      compress: true,
      port: 3000,
    ],
    devtool: 'source-map',
    module: [
      // ...
    ],
    plugins: [
      new webpack.DefinePlugin([
        BACKEND_URL: JSON.stringify(
          backend_url
        )
      ])
    ]
  ]

module.exports = config
```

The global constant is used in the following way in the code:

```
const App = () => [
  const [counter, setCounter] = useState(0)
  const [values, setValues] = useState([])
  const notes = useNotes(BACKEND_URL)
  // ...
  return [
    <div className="container">
      hello webpack {counter} clicks
      <button onClick={handleClick}>press</button>
      <div>{notes.length} notes on server {BACKEND_URL}</div>
    </div>
  ]
]
```

If the configuration for development and production differs a lot, it may be a good idea to [separate](#)

the configuration of the two into their own files.

Now, if the application is started with the command `npm start` in development mode, it fetches the notes from the address <http://localhost:3001/notes>. The version bundled with the command `npm run build` uses the address <https://notes2023.fly.dev/api/notes> to get the list of notes.

We can inspect the bundled production version of the application locally by executing the following command in the `build` directory:

```
npx static-server
```

By default, the bundled application will be available at <http://localhost:9080>.

Polyfill

Our application is finished and works with all relatively recent versions of modern browsers, except for Internet Explorer. The reason for this is that, because of `axios`, our code uses [Promises](#), and no existing version of IE supports them:

Browser compatibility

Update compatibility data on GitHub

Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js
Basic support													
32	Yes	29	No	19	8	4.4.3	32	Yes	29	Yes	8	Yes	0.12
Promise() constructor													
32	Yes	29 *	No	19	8 *	4.4.3	32	Yes	29 *	Yes	8 *	Yes	0.12 *
all													
32	Yes	29	No	19	8	4.4.3	32	Yes	29	Yes	8	Yes	0.12
prototype													
32	Yes	29	No	19	8	4.4.3	32	Yes	29	Yes	8	Yes	0.12

There are many other things in the standard that IE does not support. Something as harmless as the `find` method of JavaScript arrays exceeds the capabilities of IE:

Browser compatibility

[Update compatibility data on GitHub](#)

Desktop						Mobile						Node.js	
Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js
Basic support													
45	Yes	25	No	32	8	Yes	Yes	Yes	4	Yes	8	Yes	4.0.0

In these situations, it is not enough to transpile the code, as transpilation simply transforms the code from a newer version of JavaScript to an older one with wider browser support. IE understands Promises syntactically but it simply has not implemented their functionality. The *find* property of arrays in IE is simply *undefined*.

If we want the application to be IE-compatible, we need to add a [polyfill](#), which is code that adds the missing functionality to older browsers.

Polyfills can be added with the help of [webpack and Babel](#) or by installing one of many existing polyfill libraries.

The polyfill provided by the [promise-polyfill](#) library is easy to use. We simply have to add the following to our existing application code:

```
import PromisePolyfill from 'promise-polyfill'

if (!window.Promise) {
  window.Promise = PromisePolyfill
}
```

If the global *Promise* object does not exist, meaning that the browser does not support Promises, the polyfilled *Promise* is stored in the global variable. If the polyfilled *Promise* is implemented well enough, the rest of the code should work without issues.

One exhaustive list of existing polyfills can be found [here](#).

The browser compatibility of different APIs can be checked by visiting <https://caniuse.com> or [Mozilla's website](#).

7e: Class components, Miscellaneous

Class Components

During the course, we have only used React components having been defined as Javascript functions. This was not possible without the [hook](#) functionality that came with version 16.8 of React. Before, when defining a component that uses state, one had to define it using Javascript's [Class](#) syntax.

It is beneficial to at least be familiar with Class Components to some extent since the world contains a lot of old React code, which will probably never be completely rewritten using the updated syntax.

Let's get to know the main features of Class Components by producing yet another very familiar anecdote application. We store the anecdotes in the file *db.json* using *json-server*. The contents of the file are lifted from [here](#).

The initial version of the Class Component looks like this

```
import React from 'react'

class App extends React.Component {
  constructor(props) {
    super(props)
  }

  render() {
    return (
      <div>
        <h1>anecdote of the day</h1>
      </div>
    )
  }
}

export default App
```

The component now has a [constructor](#), in which nothing happens at the moment, and contains the method [render](#). As one might guess, render defines how and what is rendered to the screen.

Let's define a state for the list of anecdotes and the currently-visible anecdote. In contrast to when using the [useState](#) hook, Class Components only contain one state. So if the state is made up of multiple "parts", they should be stored as properties of the state. The state is initialized in the constructor:

```
class App extends React.Component {
  constructor(props) {
    super(props)

    this.state = { anecdotest: [], current: 0 }
  }

  render() {
    if (this.state.anecdotest.length === 0) {
      return <div>no anecdotes...</div>
    }
    return (
      <div>
        <h1>anecdote of the day</h1>
        <div>{this.state.anecdotest[this.state.current].content}</div>
      </div>
    )
  }
}
```

```
</div>      <button>next</button>      </div>
}
}
}
```

The component state is in the instance variable `this.state`. The state is an object having two properties. `this.state.anecdotes` is the list of anecdotes and `this.state.current` is the index of the currently-shown anecdote.

In Functional components, the right place for fetching data from a server is inside an [effect hook](#), which is executed when a component renders or less frequently if necessary, e.g. only in combination with the first render.

The [lifecycle methods](#) of Class Components offer corresponding functionality. The correct place to trigger the fetching of data from a server is inside the lifecycle method `componentDidMount`, which is executed once right after the first time a component renders:

```
class App extends React.Component {
  constructor(props) {
    super(props)

    this.state = [
      anecdotes: [],
      current: 0
    ]
  }

  componentDidMount = () => {
    axios.get('http://localhost:3001/anecdotes')
      .then(response => {
        this.setState({ anecdotes: response.data })
      })
      // ...
  }
}
```

The callback function of the HTTP request updates the component state using the method `setState`. The method only touches the keys that have been defined in the object passed to the method as an argument. The value for the key `current` remains unchanged.

Calling the method `setState` always triggers the rerender of the Class Component, i.e. calling the method `render`.

We'll finish off the component with the ability to change the shown anecdote. The following is the code for the entire component with the addition highlighted:

```
class App extends React.Component {
  constructor(props) {
    super(props)

    this.state = [
      anecdotes: [],
      current: 0
    ]
  }

  // ...
}
```

```

componentDidMount = () => {
  axios.get('http://localhost:3001/anecdotes').then(response => {
    this.setState({ anecdotes: response.data })
  })
}

handleClick = () => {
  const current = Math.floor(Math.random() * this.state.anecdotes.length)
  this.setState({ current })
}

render() {
  if (this.state.anecdotes.length === 0) {
    return <div>no anecdotes...</div>
  }

  return (
    <div>
      <h1>anecdote of the day</h1>
      <div>{this.state.anecdotes[this.state.current].content}</div>
      <button onClick={this.handleClick}>next</button>
    </div>
  )
}

```

For comparison, here is the same application as a Functional component:

```

const App = () => {
  const [anecdotes, setAnecdotes] = useState([])
  const [current, setCurrent] = useState(0)

  useEffect(() => {
    axios.get('http://localhost:3001/anecdotes').then(response => {
      setAnecdotes(response.data)
    })
  }, [])

  const handleClick = () => {
    setCurrent(Math.round(Math.random() * (anecdotes.length - 1)))
  }

  if (anecdotes.length === 0) {
    return <div>no anecdotes...</div>
  }

  return (
    <div>
      <h1>anecdote of the day</h1>
      <div>{anecdotes[current].content}</div>
      <button onClick={handleClick}>next</button>
    </div>
  )
}

```

In the case of our example, the differences were minor. The biggest difference between Functional components and Class components is mainly that the state of a Class component is a single object, and that the state is updated using the method `setState`, while in Functional components the state can consist of multiple different variables, with all of them having their own update function.

In some more advanced use cases, the effect hook offers a considerably better mechanism for controlling side effects compared to the lifecycle methods of Class Components.

A notable benefit of using Functional components is not having to deal with the self-referencing *this* reference of the Javascript class.

In my opinion, and the opinion of many others, Class Components offer little benefit over Functional components enhanced with hooks, except for the so-called [error boundary](#) mechanism, which currently (15th February 2021) isn't yet in use by functional components.

When writing fresh code, [there is no rational reason to use Class Components](#) if the project is using React with a version number 16.8 or greater. On the other hand, [there is currently no need to rewrite all old React code](#) as Functional components.

Organization of code in React application

In most applications, we followed the principle by which components were placed in the directory *components*, reducers were placed in the directory *reducers*, and the code responsible for communicating with the server was placed in the directory *services*. This way of organizing fits a smaller application just fine, but as the amount of components increases, better solutions are needed. There is no one correct way to organize a project. The article [The 100% correct way to structure a React app \(or why there's no such thing\)](#) provides some perspective on the issue.

Frontend and backend in the same repository

During the course, we have created the frontend and backend into separate repositories. This is a very typical approach. However, we did the deployment by [copying](#) the bundled frontend code into the backend repository. A possibly better approach would have been to deploy the frontend code separately.

Sometimes, there may be a situation where the entire application is to be put into a single repository. In this case, a common approach is to put the *package.json* and *webpack.config.js* in the root directory, as well as place the frontend and backend code into their own directories, e.g. *client* and *server*.

Changes on the server

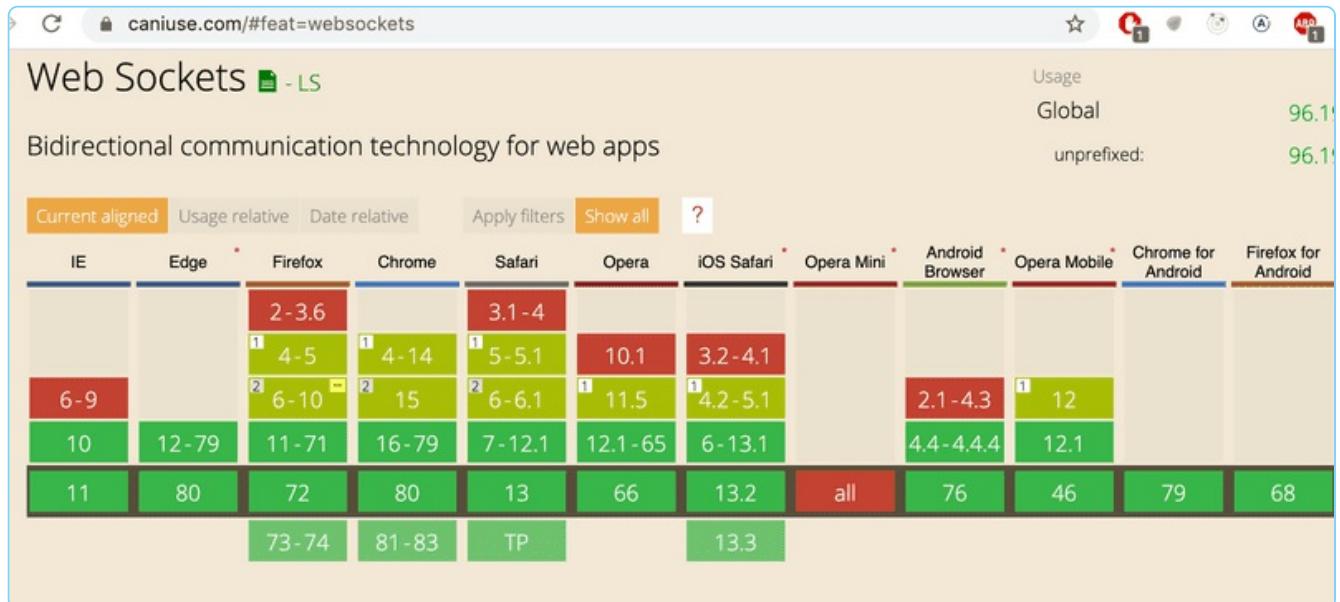
If there are changes in the state on the server, e.g. when new blogs are added by other users to the bloglist service, the React frontend we implemented during this course will not notice these changes until the page reloads. A similar situation arises when the frontend triggers a time-consuming computation in the backend. How do we reflect the results of the computation to the frontend?

One way is to execute [polling](#) on the frontend, meaning repeated requests to the backend API e.g. using the [setInterval](#) command.

A more sophisticated way is to use [WebSockets](#) which allow for establishing a two-way communication channel between the browser and the server. In this case, the browser does not need

to poll the backend, and instead only has to define callback functions for situations when the server sends data about updating state using a WebSocket.

WebSockets is an API provided by the browser, which is not yet fully supported on all browsers:



Instead of directly using the WebSocket API, it is advisable to use the [Socket.io](#) library, which provides various *fallback* options in case the browser does not have full support for WebSockets.

In [part 8](#), our topic is GraphQL, which provides a nice mechanism for notifying clients when there are changes in the backend data.

Virtual DOM

The concept of the Virtual DOM often comes up when discussing React. What is it all about? As mentioned in [part 0](#), browsers provide a [DOM API](#) through which the JavaScript running in the browser can modify the elements defining the appearance of the page.

When a software developer uses React, they rarely or never directly manipulate the DOM. The function defining the React component returns a set of [React elements](#). Although some of the elements look like normal HTML elements

```
const element = <h1>Hello, world</h1>
```

they are also just JavaScript-based React elements at their core.

The React elements defining the appearance of the components of the application make up the [Virtual DOM](#), which is stored in system memory during runtime.

With the help of the [ReactDOM](#) library, the virtual DOM defined by the components is rendered to a real DOM that can be shown by the browser using the DOM API:

```
ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

When the state of the application changes, a *new virtual DOM* gets defined by the components. React has the previous version of the virtual DOM in memory and instead of directly rendering the

new virtual DOM using the DOM API, React computes the optimal way to update the DOM (remove, add or modify elements in the DOM) such that the DOM reflects the new virtual DOM.

On the role of React in applications

In the material, we may not have put enough emphasis on the fact that React is primarily a library for managing the creation of views for an application. If we look at the traditional [Model View Controller](#) pattern, then the domain of React would be *View*. React has a more narrow area of application than e.g. [Angular](#), which is an all-encompassing Frontend MVC framework. Therefore, React is not called a *framework*, but a *library*.

In small applications, data handled by the application is stored in the state of the React components, so in this scenario, the state of the components can be thought of as *models* of an MVC architecture.

However, MVC architecture is not usually mentioned when talking about React applications. Furthermore, if we are using Redux, then the applications follow the [Flux](#) architecture and the role of React is even more focused on creating the views. The business logic of the application is handled using the Redux state and action creators. If we're using [Redux Thunk](#) familiar from part 6, then the business logic can be almost completely separated from the React code.

Because both React and [Flux](#) were created at Facebook, one could say that using React only as a UI library is the intended use case. Following the Flux architecture adds some overhead to the application, and if we're talking about a small application or prototype, it might be a good idea to use React "wrong", since [over-engineering](#) rarely yields an optimal result.

Part 6 [last chapter](#) covers the newer trends of state management in React. React's hook functions [useReducer](#) and [useContext](#) provide a kind of lightweight version of Redux. [React Query](#), on the other hand, is a library that solves many of the problems associated with handling state on the server, eliminating the need for a React application to store data retrieved from the server directly in frontend state.

React/node-application security

So far during the course, we have not touched on information security much. We do not have much time for this now either, but fortunately, University of Helsinki has a MOOC course [Securing Software](#) for this important topic.

We will, however, take a look at some things specific to this course.

The Open Web Application Security Project, otherwise known as [OWASP](#), publishes an annual list of the most common security risks in Web applications. The most recent list can be found [here](#). The same risks can be found from one year to another.

At the top of the list, we find *injection*, which means that e.g. text sent using a form in an application is interpreted completely differently than the software developer had intended. The most famous type of injection is probably [SQL injection](#).

For example, imagine that the following SQL query is executed in a vulnerable application:

```
let query = "SELECT * FROM Users WHERE name = '" + userName + "';"
```

Now let's assume that a malicious user *Arto Hellas* would define their name as

```
Arto Hell-as'; DROP TABLE Users; --
```

so that the name would contain a single quote ', which is the beginning and end character of a SQL string. As a result of this, two SQL operations would be executed, the second of which would destroy the database table *Users*:

```
SELECT * FROM Users WHERE name = 'Arto Hell-as'; DROP TABLE Users; --'
```

SQL injections are prevented using [parameterized queries](#). With them, user input isn't mixed with the SQL query, but the database itself inserts the input values at placeholders in the query (usually ?):

```
execute("SELECT * FROM Users WHERE name = ?", [userName])
```

Injection attacks are also possible in NoSQL databases. However, mongoose prevents them by [sanitizing](#) the queries. More on the topic can be found e.g. [here](#).

Cross-site scripting (XSS) is an attack where it is possible to inject malicious JavaScript code into a legitimate web application. The malicious code would then be executed in the browser of the victim. If we try to inject the following into e.g. the notes application:

```
<script>
  alert('Evil XSS attack')
</script>
```

the code is not executed, but is only rendered as 'text' on the page:



since React [takes care of sanitizing data in variables](#). Some versions of React [have been vulnerable](#) to XSS attacks. The security holes have of course been patched, but there is no guarantee that there couldn't be any more.

One needs to remain vigilant when using libraries; if there are security updates to those libraries, it is advisable to update those libraries in one's applications. Security updates for Express are found in the [library's documentation](#) and the ones for Node are found in [this blog](#).

You can check how up-to-date your dependencies are using the command

```
npm outdated --depth 0
```

The one-year-old project that is used in [part 9](#) of this course already has quite a few outdated dependencies:

Package	Current	Wanted	Latest	Location	Depended by
@types/jest	26.0.20	26.0.20	27.4.0	node_modules/@types/jest	patientor
@types/node	12.11.7	12.11.7	17.0.15	node_modules/@types/node	patientor
@types/react	17.0.2	17.0.39	17.0.39	node_modules/@types/react	patientor
@types/react-dom	17.0.1	17.0.11	17.0.11	node_modules/@types/react-dom	patientor
@types/react-router-dom	5.1.7	5.3.3	5.3.3	node_modules/@types/react-router-dom	patientor
@typescript-eslint/eslint-plugin	4.16.1	4.33.0	5.10.2	node_modules/@typescript-eslint/eslint-plugin	patientor
@typescript-eslint/parser	4.16.1	4.33.0	5.10.2	node_modules/@typescript-eslint/parser	patientor
axios	0.21.1	0.21.4	0.25.0	node_modules/axios	patientor
formik	2.2.6	2.2.9	2.2.9	node_modules/formik	patientor
react	17.0.1	17.0.2	17.0.2	node_modules/react	patientor
react-dom	17.0.1	17.0.2	17.0.2	node_modules/react-dom	patientor
react-router-dom	5.2.0	5.3.0	6.2.1	node_modules/react-router-dom	patientor
react-scripts	4.0.3	4.0.3	5.0.0	node_modules/react-scripts	patientor
semantic-ui-react	2.0.3	2.1.1	2.1.1	node_modules/semantic-ui-react	patientor
typescript	4.2.2	4.5.5	4.5.5	node_modules/typescript	patientor

The dependencies can be brought up to date by updating the file *package.json*. The best way to do that is by using a tool called *npm-check-updates*. It can be installed globally by running the command:

```
npm install -g npm-check-updates
```

Using this tool, the up-to-dateness of dependencies is checked in the following way:

```
$ npm-check-updates
Checking ...\\ultimate-hooks\\package.json
[=====] 9/9 100%

@testing-library/react      ^13.0.0 → ^13.1.1
@testing-library/user-event  ^14.0.4 → ^14.1.1
react-scripts                5.0.0 → 5.0.1

Run ncu -u to upgrade package.json
```

The file *package.json* is brought up to date by running the command *ncu -u*.

```
$ ncu -u
Upgrading ...\\ultimate-hooks\\package.json
[=====] 9/9 100%

@testing-library/react      ^13.0.0 → ^13.1.1
@testing-library/user-event  ^14.0.4 → ^14.1.1
react-scripts                5.0.0 → 5.0.1

Run npm install to install new versions.
```

Then it is time to update the dependencies by running the command *npm install*. However, old versions of the dependencies are not necessarily a security risk.

The npm [audit](#) command can be used to check the security of dependencies. It compares the version numbers of the dependencies in your application to a list of the version numbers of dependencies containing known security threats in a centralized error database.

Running `npm audit` on the same project, it prints a long list of complaints and suggested fixes. Below is a part of the report:

```
$ patientor npm audit

... many lines removed ...

url-parse <1.5.2
Severity: moderate
Open redirect in url-parse - https://github.com/advisories/GHSA-hh27-ffr2-f2jc
fix available via `npm audit fix`
node_modules/url-parse

ws 6.0.0 - 6.2.1 || 7.0.0 - 7.4.5
Severity: moderate
ReDoS in Sec-WebSocket-Protocol header - https://github.com/advisories/GHSA-6fc8-
4gx4-v693
ReDoS in Sec-WebSocket-Protocol header - https://github.com/advisories/GHSA-6fc8-
4gx4-v693
fix available via `npm audit fix`
node_modules/webpack-dev-server/node_modules/ws
node_modules/ws

120 vulnerabilities [102 moderate, 16 high, 2 critical]

To address issues that do not require attention, run:
  npm audit fix

To address all issues [including breaking changes], run:
  npm audit fix --force
```

After only one year, the code is full of small security threats. Luckily, there are only 2 critical threats. Let's run `npm audit fix` as the report suggests:

```
$ npm audit fix

+ mongoose@5.9.1
added 19 packages from 8 contributors, removed 8 packages and updated 15 package
s in 7.325s
fixed 354 of 416 vulnerabilities in 20047 scanned packages
  1 package update for 62 vulns involved breaking changes
  (use `npm audit fix --force` to install breaking changes, or refer to `npm aud
it` for steps to fix these manually)
```

62 threats remain because, by default, `audit fix` does not update dependencies if their *major* version number has increased. Updating these dependencies could lead to the whole application breaking down.

The source for the critical bug is the library [immer](#)

```
immer <9.0.6
Severity: critical
Prototype Pollution in immer - https://github.com/advisories/GHSA-33f9-j839-rf8h
fix available via `npm audit fix --force`
Will install react-scripts@5.0.0, which is a breaking change
```

Running `npm audit fix --force` would upgrade the library version but would also upgrade the library `react-scripts` and that would potentially break down the development environment. So we will leave the library upgrades for later...

One of the threats mentioned in the list from OWASP is *Broken Authentication* and the related *Broken Access Control*. The token-based authentication we have been using is fairly robust if the application is being used on the traffic-encrypting HTTPS protocol. When implementing access control, one should e.g. remember to not only check a user's identity in the browser but also on the server. Bad security would be to prevent some actions to be taken only by hiding the execution options in the code of the browser.

On Mozilla's MDN, there is a very good [Website security guide](#), which brings up this very important topic:

! Important: The single most important lesson you can learn about website security is to **never trust data from the browser**. This includes, but is not limited to data in URL parameters of GET requests, POST requests, HTTP headers and cookies, and user-uploaded files. Always check and sanitize all incoming data. Always assume the worst.

The documentation for Express includes a section on security: [Production Best Practices: Security](#), which is worth a read. It is also recommended to add a library called [Helmet](#) to the backend. It includes a set of middleware that eliminates some security vulnerabilities in Express applications.

Using the ESLint [security-plugin](#) is also worth doing.

Current trends

Finally, let's take a look at some technology of tomorrow (or, actually, already today), and the directions in which Web development is heading.

Typed versions of JavaScript

Sometimes, the [dynamic typing](#) of JavaScript variables creates annoying bugs. In part 5, we talked briefly about [PropTypes](#): a mechanism which enables one to enforce type-checking for props passed to React components.

Lately, there has been a notable uplift in the interest in [static type checking](#). At the moment, the most popular typed version of Javascript is [TypeScript](#) which has been developed by Microsoft. Typescript is covered in [part 9](#).

Server-side rendering, isomorphic applications and universal code

The browser is not the only domain where components defined using React can be rendered. The rendering can also be done on the [server](#). This kind of approach is increasingly being used, such that, when accessing the application for the first time, the server serves a pre-rendered page made with React. From here onwards, the operation of the application continues, as usual, meaning the browser executes React, which manipulates the DOM shown by the browser. The rendering that is done on the server goes by the name: *server-side rendering*.

One motivation for server-side rendering is Search Engine Optimization (SEO). Search engines have traditionally been very bad at recognizing JavaScript-rendered content. However, the tide might be turning, e.g. take a look at [this](#) and [this](#).

Of course, server-side rendering is not anything specific to React or even JavaScript. Using the same programming language throughout the stack in theory simplifies the execution of the concept because the same code can be run on both the front- and backend.

Along with server-side rendering, there has been talk of so-called *isomorphic applications* and *universal code*, although there has been some debate about their definitions. According to some [definitions](#), an isomorphic web application performs rendering on both frontend and backend. On the other hand, universal code is code that can be executed in most environments, meaning both frontend and backend.

React and Node provide a desirable option for implementing an isomorphic application as universal code.

Writing universal code directly using React is currently still pretty cumbersome. Lately, a library called [Next.js](#), which is implemented on top of React, has garnered much attention and is a good option for making universal applications.

Progressive web apps

Lately, people have started using the term [progressive web app](#) (PWA) launched by Google.

In short, we are talking about web applications working as well as possible on every platform and taking advantage of the best parts of those platforms. The smaller screen of mobile devices must not hamper the usability of the application. PWAs should also work flawlessly in offline mode or with a slow internet connection. On mobile devices, they must be installable just like any other application. All the network traffic in a PWA should be encrypted.

Microservice architecture

During this course, we have only scratched the surface of the server end of things. In our applications, we had a *monolithic* backend, meaning one application making up a whole and running on a single server, serving only a few API endpoints.

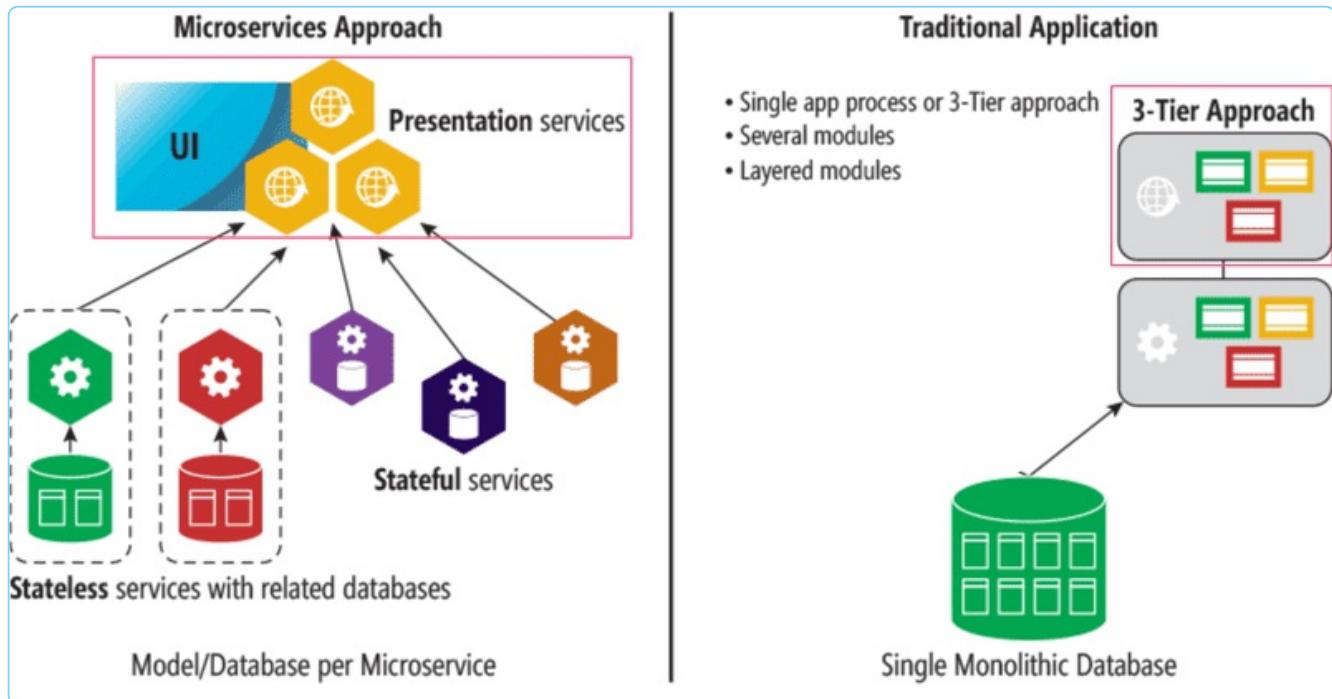
As the application grows, the monolithic backend approach starts turning problematic both in terms of performance and maintainability.

A [microservice architecture](#) (microservices) is a way of composing the backend of an application from many separate, independent services, which communicate with each other over the network. An individual microservice's purpose is to take care of a particular logical functional whole. In a pure microservice architecture, the services do not use a shared database.

For example, the bloglist application could consist of two services: one handling the user and another

taking care of the blogs. The responsibility of the user service would be user registration and user authentication, while the blog service would take care of operations related to the blogs.

The image below visualizes the difference between the structure of an application based on a microservice architecture and one based on a more traditional monolithic structure:



The role of the frontend (enclosed by a square in the picture) does not differ much between the two models. There is often a so-called [API gateway](#) between the microservices and the frontend, which provides an illusion of a more traditional "everything on the same server" API. [Netflix](#), among others, uses this type of approach.

Microservice architectures emerged and evolved for the needs of large internet-scale applications. The trend was set by Amazon far before the appearance of the term microservice. The critical starting point was an email sent to all employees in 2002 by Amazon CEO Jeff Bezos:

All teams will henceforth expose their data and functionality through service interfaces.

Teams must communicate with each other through these interfaces.

There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

It doesn't matter what technology you use.

All service interfaces, without exception, must be designed from the ground up to be externalize-able. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world.

No exceptions.

Anyone who doesn't do this will be fired. Thank you; have a nice day!

Nowadays, one of the biggest forerunners in the use of microservices is [Netflix](#).

The use of microservices has steadily been gaining hype to be kind of a [silver bullet](#) of today, which is being offered as a solution to almost every kind of problem. However, there are several challenges when it comes to applying a microservice architecture, and it might make sense to go [monolith first](#) by initially making a traditional all-encompassing backend. Or maybe [not](#). There are a bunch of different opinions on the subject. Both links lead to Martin Fowler's site; as we can see, even the wise are not entirely sure which one of the right ways is more right.

Unfortunately, we cannot dive deeper into this important topic during this course. Even a cursory look at the topic would require at least 5 more weeks.

Serverless

After the release of Amazon's [lambda](#) service at the end of 2014, a new trend started to emerge in web application development: [serverless](#).

The main thing about lambda, and nowadays also Google's [Cloud functions](#) as well as [similar functionality in Azure](#), is that it enables *the execution of individual functions* in the cloud. Before, the smallest executable unit in the cloud was a single *process*, e.g. a runtime environment running a Node backend.

E.g. Using Amazon's [API gateway](#) it is possible to make serverless applications where the requests to the defined HTTP API get responses directly from cloud functions. Usually, the functions already operate using stored data in the databases of the cloud service.

Serverless is not about there not being a server in applications, but about how the server is defined. Software developers can shift their programming efforts to a higher level of abstraction as there is no longer a need to programmatically define the routing of HTTP requests, database relations, etc., since the cloud infrastructure provides all of this. Cloud functions also lend themselves to creating a well-scaling system, e.g. Amazon's Lambda can execute a massive amount of cloud functions per second. All of this happens automatically through the infrastructure and there is no need to initiate new servers, etc.

Useful libraries and interesting links

The JavaScript developer community has produced a large variety of useful libraries. If you are developing anything more substantial, it is worth it to check if existing solutions are already available. Below are listed some libraries recommended by trustworthy parties.

If your application has to handle complicated data, [lodash](#), which we recommended in [part 4](#), is a good library to use. If you prefer the functional programming style, you might consider using [ramda](#).

If you are handling times and dates, [date-fns](#) offers good tools for that.

If you have complex forms in your apps, have a look at whether [React Hook Form](#) would be a good fit.

If your application displays graphs, there are multiple options to choose from. Both [recharts](#) and [highcharts](#) are well-recommended.

The [Immer](#) provides immutable implementations of some data structures. The library could be of use when using Redux, since as we [remember](#) from part 6, reducers must be pure functions, meaning they must not modify the store's state but instead have to replace it with a new one when a change occurs.

[Redux-saga](#) provides an alternative way to make asynchronous actions for [Redux Thunk](#) familiar from part 6. Some embrace the hype and like it. I don't.

For single-page applications, the gathering of analytics data on the interaction between the users and the page is [more challenging](#) than for traditional web applications where the entire page is loaded. The [React Google Analytics 4](#) library offers a solution.

You can take advantage of your React know-how when developing mobile applications using Facebook's extremely popular [React Native](#) library, which is the topic of [part 10](#) of the course.

When it comes to the tools used for the management and bundling of JavaScript projects, the community has been very fickle. Best practices have changed rapidly (the years are approximations, nobody remembers that far back in the past):

- 2011 [Bower](#)
- 2012 [Grunt](#)
- 2013-14 [Gulp](#)
- 2012-14 [Browserify](#)
- 2015-2023 [Webpack](#)
- 2023- [esbuild](#)

Hipsters seemed to have lost their interest in tool development after webpack started to dominate the markets. A few years ago, [Parcel](#) started to make the rounds marketing itself as simple (which Webpack is not) and faster than Webpack. However, after a promising start, Parcel has not gathered any steam. But recently, [esbuild](#) has been on a high rise and is already replacing Webpack.

The site <https://reactpatterns.com/> provides a concise list of best practices for React, some of which are already familiar from this course. Another similar list is [react bits](#).

[Reactiflux](#) is a big chat community of React developers on Discord. It could be one possible place to get support after the course has concluded. For example, numerous libraries have their own channels.

If you know some recommendable links or libraries, make a pull request!

7f: Exercises: extending the bloglist

In addition to the eight exercises in the [React router](#) and [custom hooks](#) sections of this seventh part of the course material, 13 exercises continue our work on the BlogList application that we worked on in parts four and five of the course material. Some of the following exercises are "features" that are independent of one another, meaning that there is no need to finish them in any particular order. You are free to skip over a part of the exercises if you wish to do so. Quite many of them are about applying the advanced state management technique (Redux, React Query and context) covered in [part 6](#).

If you do not want to use your BlogList application, you are free to use the code from the model solution as a starting point for these exercises.

Many of the exercises in this part of the course material will require the refactoring of existing code. This is a common reality of extending existing applications, meaning that refactoring is an important and necessary skill even if it may feel difficult and unpleasant at times.

One good piece of advice for both refactoring and writing new code is to take *baby steps*. Losing your sanity is almost guaranteed if you leave the application in a completely broken state for long periods while refactoring.

Exercises 7.9.-7.21.

7.9: Automatic Code Formatting

In the previous parts, we used ESLint to ensure that the code follows the defined conventions. [Prettier](#) is yet another approach for the same. According to the documentation, Prettier is *an opinionated code formatter*, that is, Prettier not only controls the code style but also formats the code according to the definition.

Prettier is easy to integrate into the code editor so that when it is saved, it is automatically formatted.

Take Prettier to use in your app and configure it to work with your editor.

State Management: Redux

There are two alternative versions to choose for exercises 7.10-7.13: you can do the state management of the application either using Redux or React Query and Context. If you want to maximize your learning, you should do both versions!

7.10: Redux, Step 1

Refactor the application to use Redux to manage the notification data.

7.11: Redux, Step 2

Note that this and the next two exercises are quite laborious but incredibly educational.

Store the information about blog posts in the Redux store. In this exercise, it is enough that you can see the blogs in the backend and create a new blog.

You are free to manage the state for logging in and creating new blog posts by using the internal state of React components.

7.12: Redux, Step 3

Expand your solution so that it is again possible to like and delete a blog.

7.13: Redux, Step 4

Store the information about the signed-in user in the Redux store.

State Management: React Query and Context

There are two alternative versions to choose for exercises 7.10-7.13: you can do the state management of the application either using Redux or React Query and Context. If you want to maximize your learning, you should do both versions!

7.10: React Query and Context step 1

Refactor the app to use the useReducer-hook and context to manage the notification data.

7.11: React Query and Context step 2

Use React Query to manage the state for blog posts. For this exercise, it is sufficient that the application displays existing blogs and that the creation of a new blog is successful.

You are free to manage the state for logging in and creating new blog posts by using the internal state of React components.

7.12: React Query and Context step 3

Expand your solution so that it is again possible to like and delete a blog.

7.13: React Query and Context step 4

Use the useReducer-hook and context to manage the data for the logged in user.

Views

The rest of the tasks are common to both the Redux and React Query versions.

7.14: Users view

Implement a view to the application that displays all of the basic information related to users:

The screenshot shows a web browser window with the URL `localhost:3000/users`. The page has a header 'blogs' and a 'logout' button. Below that is a section titled 'Users' with a table showing the number of blogs created by each user:

	blogs created
Arto Hellas	6
Matti Luukkainen	2
Venla Ruuska	0

7.15: Individual User View

Implement a view for individual users that displays all of the blog posts added by that user:

The screenshot shows a web browser window with the URL `localhost:3000/users/5c4857b1003ad1a6e6626931`. The page title is "blogs". It displays the user "Arto Hellas" who has logged in. There is a "logout" button. Below the user's name, it says "added blogs" and lists six items:

- Microservices and the First Law of Distributed Objects
- Things I Don't Know as of 2018
- You're NOT gonna need it!
- Our learnings from adopting GraphQL
- The Single Responsibility Principle
- React Testing with react-testing-library

You can access this view by clicking the name of the user in the view that lists all users:

The screenshot shows a web browser window with the URL `localhost:3000/users`. The page title is "blogs". It displays a list of users with their names and the number of blogs they have created:

User	blogs created
Arto Hellas	6
Matti Luukkainen	2
Venla Ruuska	0

NB: you will almost certainly stumble across the following error message during this exercise:

The screenshot shows a web browser window with the URL `localhost:3000/users/5e35735b8495137a875edb01`. The error message is displayed in red text: "TypeError: Cannot read property 'name' of undefined". Below the error message, it shows the stack trace: "User" and "src/components/User.js:16".

The error message will occur if you refresh the individual user page.

The cause of the issue is that, when we navigate directly to the page of an individual user, the React application has not yet received the data from the backend. One solution for this problem is to use conditional rendering:

```
const User = () => {
  const user = ...
  if (!user) { return null }
  return [
    <div>
      // ...
    </div>
```

```
    }
}
```

7.16: Blog View

Implement a separate view for blog posts. You can model the layout of your view after the following example:

The screenshot shows a browser window with the URL `localhost:3000/blogs/5c4d7da832383b0663990e9e`. The page title is "blogs". It displays a blog post by Matti Luukkainen. The post title is "The Single Responsibility Principle Robert C. Martin". Below the title is the URL `https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html`. It shows 3 likes and was added by Arto Hellas.

Users should be able to access this view by clicking the name of the blog post in the view that lists all of the blog posts.

The screenshot shows a browser window with the URL `localhost:3000`. The page title is "blogs". It displays a list of blog posts. The first post is "Microservices and the First Law of Distributed Objects Martin Fowler". Other visible post titles include "The Single Responsibility Principle Robert C. Martin", "Things I Don't Know as of 2018 Dan Abramov", "You're NOT gonna need it! Ron Jeffries", "Our learnings from adopting GraphQL Artem Statnov", and "FP vs. OO List Processing Robert C. Martin".

After you're done with this exercise, the functionality that was implemented in exercise 5.7 is no longer necessary. Clicking a blog post no longer needs to expand the item in the list and display the details of the blog post.

7.17: Navigation

Implement a navigation menu for the application:

A screenshot of a web browser window titled "localhost:3000". The header bar shows "blogs users Matti Luukkainen logged in" and a "logout" button. Below the header, the title "blog app" is displayed. A "create new" button is visible. A list of blog posts is shown in a table format:

Microservices and the First Law of Distributed Objects	Martin Fowler
The Single Responsibility Principle	Robert C. Martin
Things I Don't Know as of 2018	Dan Abramov
You're NOT gonna need it!	Ron Jeffries

7.18: Comments, step 1

Implement the functionality for commenting the blog posts:

A screenshot of a web browser window titled "localhost:3000/blogs/5c4d7da832383b0663990e9e". The header bar shows "blogs users Matti Luukkainen logged in" and a "logout" button. Below the header, the title "blog app" is displayed. The main content is a blog post titled "The Single Responsibility Principle Robert C. Martin" with the URL "<https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html>". It has "3 likes" and was "added by Arto Hellas". A "comments" section contains two bullet points:

- awesome article
- a must read for every developer

Comments should be anonymous, meaning that they are not associated with the user who left the comment.

In this exercise, it is enough for the frontend to only display the comments that the application receives from the backend.

An appropriate mechanism for adding comments to a blog post would be an HTTP POST request to the `api/blogs/:id/comments` endpoint.

7.19: Comments, step 2

Extend your application so that users can add comments to blog posts from the frontend:

The screenshot shows a web browser window with the URL localhost:3000/blogs/5c4d7da832383b0663990e9e. The page title is "blog app". The main content is a post by Robert C. Martin titled "The Single Responsibility Principle". The post has a link to its original source: <https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleResponsibilityPrinciple.html>. It has 3 likes and was added by Arto Hellas. Below the post, there is a section for comments with a placeholder "haven't read this yet.." and a "add comment" button. There are two bullet points under the comments section: "awesome article" and "a must read for every developer".

7.20: Styles, step 1

Improve the appearance of your application by applying one of the methods shown in the course material.

7.21: Styles, step 2

You can mark this exercise as finished if you use an hour or more for styling your application.

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

8: GraphQL

8a: GraphQL-server

REST, familiar to us from the previous parts of the course, has long been the most prevalent way to implement the interfaces servers offer for browsers, and in general the integration between different applications on the web.

In recent years, [GraphQL](#), developed by Facebook, has become popular for communication between web applications and servers.

The GraphQL philosophy is very different from REST. REST is *resource-based*. Every resource, for example a *user*, has its own address which identifies it, for example `/users/10`. All operations done to the resource are done with HTTP requests to its URL. The action depends on the HTTP method used.

The resource-basedness of REST works well in most situations. However, it can be a bit awkward sometimes.

Let's consider the following example: our bloglist application contains some kind of social media functionality, and we would like to show a list of all the blogs that were added by users who have commented on any of the blogs we follow.

If the server implemented a REST API, we would probably have to do multiple HTTP requests from the browser before we had all the data we wanted. The requests would also return a lot of unnecessary data, and the code on the browser would probably be quite complicated.

If this was an often-used functionality, there could be a REST endpoint for it. If there were a lot of these kinds of scenarios however, it would become very laborious to implement REST endpoints for all of them.

A GraphQL server is well-suited for these kinds of situations.

The main principle of GraphQL is that the code on the browser forms a *query* describing the data wanted, and sends it to the API with an HTTP POST request. Unlike REST, all GraphQL queries are sent to the same address, and their type is POST.

The data described in the above scenario could be fetched with (roughly) the following query:

```
query FetchBlogsQuery {
  user(username: "mluukkai") {
    followedUsers {
      blogs {
        comments {
          user {
            blogs {
              title
            }
          }
        }
      }
    }
  }
}
```

The content of the `FetchBlogsQuery` can be roughly interpreted as: find a user named `"mluukkai"` and for each of his `followedUsers`, find all their `blogs`, and for each blog, all its `comments`, and for each `user` who wrote each comment, find their `blogs`, and return the `title` of each of them.

The server's response would be about the following JSON object:

```
{
  "data": [
    "followedUsers": [
      {
        "blogs": [
          {
            "comments": [
              {
                "user": [
                  "blogs": [
                    [
                      {
                        "title": "Goto considered harmful"
                      },
                      {

```

```

        "title": "End to End Testing with Cypress is most enjoyable"
      },
      [
        {
          "title": "Navigating your transition to GraphQL"
        },
        [
          {
            "title": "From REST to GraphQL"
          }
        ]
      ],
      [
        [
          [
            [
              [
                [
                  [
                    [
                      [
                        [
                          [
                            [
                              [
                                [
                                  [
                                    [
                                      [
                                        [
                                          [
                                            [
                                              [
                                                [
                                                  [
                                                    [
                                                      [
                                                        [
                                                          [
                                                            [
                                                              [
                                                                [
                                                                  [
                                                                    [
                                                                      [
                                                                        [
                                                                          [
                                                                            [
                                                                              [
                                                                                [
                                                                                  [
                                                                                    [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
................................................................

```

The application logic stays simple, and the code on the browser gets exactly the data it needs with a single query.

Schemas and queries

We will get to know the basics of GraphQL by implementing a GraphQL version of the phonebook application from parts 2 and 3.

In the heart of all GraphQL applications is a [schema](#), which describes the data sent between the client and the server. The initial schema for our phonebook is as follows:

```

type Person {
  name: String!
  phone: String
  street: String!
  city: String!
  id: ID!
}

type Query {
  personCount: Int!
  allPersons: [Person!]!
  findPerson(name: String!): Person
}

```

The schema describes two [types](#). The first type, *Person*, determines that persons have five fields. Four of the fields are type *String*, which is one of the [scalar types](#) of GraphQL. All of the String fields, except *phone*, must be given a value. This is marked by the exclamation mark on the schema. The type of the field *id* is *ID*. *ID* fields are strings, but GraphQL ensures they are unique.

The second type is a [Query](#). Practically every GraphQL schema describes a Query, which tells what kind of queries can be made to the API.

The phonebook describes three different queries. *personCount* returns an integer, *allPersons* returns a list of *Person* objects and *findPerson* is given a string parameter and it returns a *Person* object.

Again, exclamation marks are used to mark which return values and parameters are *Non-Null*. *personCount* will, for sure, return an integer. The query *findPerson* must be given a string as a parameter. The query returns a *Person*-object or *null*. *allPersons* returns a list of *Person* objects, and the list does not contain any *null* values.

So the schema describes what queries the client can send to the server, what kind of parameters the queries can have, and what kind of data the queries return.

The simplest of the queries, *personCount*, looks as follows:

```
query {
  personCount
}
```

Assuming our application has saved the information of three people, the response would look like this:

```
{
  "data": [
    "personCount": 3
  ]
}
```

The query fetching the information of all of the people, *allPersons*, is a bit more complicated. Because the query returns a list of *Person* objects, the query must describe which *fields* of the objects the query returns:

```
query {
  allPersons {
    name
    phone
  }
}
```

The response could look like this:

```
{
  "data": [
    "allPersons": [
      {
        "name": "Arto Hellas",
        "phone": "040-123543"
      },
      {
        "name": "Matti Luukkainen",
        "phone": "040-432342"
      }
    ]
}
```

```
[  
  {  
    "name": "Venla Ruuska",  
    "phone": null  
  },  
  {  
    "name": "Arto Hellas",  
    "phone": "040-123456",  
    "city": "Espoo",  
    "street": "Tapiolankatu 5 A",  
    "id": "3d594650-3436-11e9-bc57-8b80ba54c431"  
  }  
]
```

A query can be made to return any field described in the schema. For example, the following would also be possible:

```
query {  
  allPersons{  
    name  
    city  
    street  
  }  
}
```

The last example shows a query which requires a parameter, and returns the details of one person.

```
query {  
  findPerson(name: "Arto Hellas") {  
    phone  
    city  
    street  
    id  
  }  
}
```

So, first, the parameter is described in round brackets, and then the fields of the return value object are listed in curly brackets.

The response is like this:

```
{  
  "data": [  
    "findPerson": {  
      "phone": "040-123543",  
      "city": "Espoo",  
      "street": "Tapiolankatu 5 A",  
      "id": "3d594650-3436-11e9-bc57-8b80ba54c431"  
    }  
  ]  
}
```

The return value was marked as nullable, so if we search for the details of an unknown

```
query {  
  findPerson(name: "Joe Biden") {
```

```
  phone
```

```
  }
```

```
}
```

the return value is *null*.

```
{
  "data": [
    "findPerson": null
  ]
}
```

As you can see, there is a direct link between a GraphQL query and the returned JSON object. One can think that the query describes what kind of data it wants as a response. The difference to REST queries is stark. With REST, the URL and the type of the request have nothing to do with the form of the returned data.

GraphQL query describes only the data moving between a server and the client. On the server, the data can be organized and saved any way we like.

Despite its name, GraphQL does not actually have anything to do with databases. It does not care how the data is saved. The data a GraphQL API uses can be saved into a relational database, document database, or to other servers which a GraphQL server can access with for example REST.

Apollo Server

Let's implement a GraphQL server with today's leading library: [Apollo Server](#).

Create a new npm project with *npm init* and install the required dependencies.

```
npm install @apollo/server graphql
```

Also create a `index.js` file in your project's root directory.

The initial code is as follows:

```
const { ApolloServer } = require('@apollo/server')
const { startStandaloneServer } = require('@apollo/server/standalone')

let persons = [
  {
    name: "Arto Hellas",
    phone: "040-123543",
    street: "Tapiolankatu 5 A",
    city: "Espoo",
    id: "3d594650-3436-11e9-bc57-8b80ba54c431"
  },
  {
    name: "Matti Luukkainen",
    phone: "040-432342",
    street: "Malminkaari 10 A",
    id: "4d594650-3436-11e9-bc57-8b80ba54c432"
  }
]
```

```

    city: "Helsinki",
    id: '3d599470-3436-11e9-bc57-8b80ba54c431',
  },
  [
    {
      name: "Venla Ruuska",
      street: "Nallemäentie 22 C",
      city: "Helsinki",
      id: '3d599471-3436-11e9-bc57-8b80ba54c431'
    }
  ]
]

const typeDefs = `

type Person {
  name: String!
  phone: String
  street: String!
  city: String!
  id: ID!
}

type Query {
  personCount: Int!
  allPersons: [Person!]!
  findPerson(name: String!): Person
}

`


const resolvers = [
  {
    Query: {
      personCount: () => persons.length,
      allPersons: () => persons,
      findPerson: (root, args) =>
        persons.find(p => p.name === args.name)
    }
  }
]

const server = new ApolloServer([
  typeDefs,
  resolvers,
])

startStandaloneServer(server, [
  listen: { port: 4000 },
]).then(({ url }) => {
  console.log(`Server ready at ${url}`)
})

```

The heart of the code is an [ApolloServer](#), which is given two parameters:

```

const server = new ApolloServer([
  typeDefs,
  resolvers,
])

```

```
}}
```

The first parameter, `typeDefs`, contains the GraphQL schema.

The second parameter is an object, which contains the `resolvers` of the server. These are the code, which defines *how* GraphQL queries are responded to.

The code of the resolvers is the following:

```
const resolvers = {
  Query: {
    personCount: () => persons.length,
    allPersons: () => persons,
    findPerson: (root, args) =>
      persons.find(p => p.name === args.name)
  }
}
```

As you can see, the resolvers correspond to the queries described in the schema.

```
type Query {
  personCount: Int!
  allPersons: [Person!]!
  findPerson(name: String!): Person
}
```

So there is a field under `Query` for every query described in the schema.

The query

```
query {
  personCount
}
```

Has the resolver

```
() => persons.length
```

So the response to the query is the length of the array `persons`.

The query which fetches all persons

```
query {
  allPersons {
    name
  }
}
```

has a resolver which returns *all* objects from the `persons` array.

```
() => persons
```

Start the server by running `node index.js` in the terminal.

Apollo Studio Explorer

When Apollo server is run in development mode the page <http://localhost:4000> has a button *Query your server* that takes us to [Apollo Studio Explorer](#). This is very useful for a developer, and can be used to make queries to the server.

Let's try it out:

The screenshot shows the Apollo Studio Explorer interface. On the left, the 'Documentation' panel displays the schema for a 'Query' type, showing fields like 'personCount', 'allPersons', and 'findPerson'. In the center, the 'Operation' panel contains a GraphQL query for 'ExampleQuery':

```
query ExampleQuery {
  allPersons {
    name
    phone
  }
}
```

On the right, the 'Response' panel shows the JSON result of the query:

```
{"data": {"allPersons": [{"name": "Arto Hellas", "phone": "040-123543"}, {"name": "Matti Luukainen", "phone": "040-432342"}, {"name": "Venla Ruuska", "phone": null}]}}
```

The status bar at the top right indicates STATUS 200 | 13.7ms | 156B.

At the left side Explorer shows the API-documentation that it has automatically generated based on the schema.

Parameters of a resolver

The query fetching a single person

```
query {
  findPerson(name: "Arto Hellas") {
    phone
    city
    street
  }
}
```

has a resolver which differs from the previous ones because it is given *two parameters*:

```
[root, args] => persons.find(p => p.name === args.name)
```

The second parameter, *args*, contains the parameters of the query. The resolver then returns from the array *persons* the person whose name is the same as the value of *args.name*. The resolver does not need the first parameter *root*.

In fact, all resolver functions are given [four parameters](#). With JavaScript, the parameters don't have to be defined if they are not needed. We will be using the first and the third parameter of a resolver later in this part.

The default resolver

When we do a query, for example

```
query {
  findPerson(name: "Arto Hellas") {
    phone
    city
    street
  }
}
```

the server knows to send back exactly the fields required by the query. How does that happen?

A GraphQL server must define resolvers for *each* field of each type in the schema. We have so far only defined resolvers for fields of the type *Query*, so for each query of the application.

Because we did not define resolvers for the fields of the type *Person*, Apollo has defined [default resolvers](#) for them. They work like the one shown below:

```
const resolvers = [
  Query: {
    personCount: () => persons.length,
    allPersons: () => persons,
    findPerson: [root, args] => persons.find(p => p.name === args.name)
  },
  Person: {
    name: [root] => root.name,
    phone: [root] => root.phone,
    street: [root] => root.street,
    city: [root] => root.city,
    id: [root] => root.id
  }
]
```

The default resolver returns the value of the corresponding field of the object. The object itself can be accessed through the first parameter of the resolver, *root*.

If the functionality of the default resolver is enough, you don't need to define your own. It is also possible to define resolvers for only some fields of a type, and let the default resolvers handle the rest.

We could for example define that the address of all persons is *Manhattan New York* by hard-coding the following to the resolvers of the street and city fields of the type *Person*:

```
Person: {
  street: [root] => "Manhattan",
  city: [root] => "New York"
}
```

Object within an object

Let's modify the schema a bit

```
type Address { street: String! city: String! }
type Person {
  name: String!
  phone: String
  address: Address!
  id: ID!
}

type Query {
  personCount: Int!
  allPersons: [Person!]!
  findPerson(name: String!): Person
}
```

so a person now has a field with the type *Address*, which contains the street and the city.

The queries requiring the address change into

```
query {
  findPerson(name: "Arto Hellas") {
    phone
    address {
      city
      street
    }
  }
}
```

and the response is now a person object, which *contains* an address object.

```
{
  "data": [
    "findPerson": [
      "phone": "040-123543",
      "address": [
        "city": "Espoo",
        "street": "Tapiolankatu 5 A"
      ]
    ]
  ]
}
```

We still save the persons in the server the same way we did before.

```
let persons = [
  {
    name: "Arto Hellas",
    phone: "040-123543",
    street: "Tapiolankatu 5 A",
  }
]
```

```

    city: "Espoo",
    id: "3d594650-3436-11e9-bc57-8b80ba54c431"
  },
  // ...
]

```

The person-objects saved in the server are not exactly the same as the GraphQL type *Person* objects described in the schema.

Contrary to the *Person* type, the *Address* type does not have an *id* field, because they are not saved into their own separate data structure in the server.

Because the objects saved in the array do not have an *address* field, the default resolver is not sufficient. Let's add a resolver for the *address* field of *Person* type :

```

const resolvers = [
  Query: {
    personCount: () => persons.length,
    allPersons: () => persons,
    findPerson: (root, args) =>
      persons.find(p => p.name === args.name)
  },
  Person: {
    address: (root) => {
      return {
        street: root.street,
        city: root.city
      }
    }
  }
]

```

So every time a *Person* object is returned, the fields *name*, *phone* and *id* are returned using their default resolvers, but the field *address* is formed by using a self-defined resolver. The parameter *root* of the resolver function is the person-object, so the street and the city of the address can be taken from its fields.

The current code of the application can be found on [Github](#), branch *part8-1*.

Mutations

Let's add a functionality for adding new persons to the phonebook. In GraphQL, all operations which cause a change are done with [mutations](#). Mutations are described in the schema as the keys of type *Mutation*.

The schema for a mutation for adding a new person looks as follows:

```

type Mutation {
  addPerson(
    name: String!
    phone: String
    street: String!
    city: String!
  ): Person
}

```

The Mutation is given the details of the person as parameters. The parameter *phone* is the only one which is nullable. The Mutation also has a return value. The return value is type *Person*, the idea

being that the details of the added person are returned if the operation is successful and if not, null. Value for the field *id* is not given as a parameter. Generating an id is better left for the server.

Mutations also require a resolver:

```
const { v1: uuid } = require('uuid')

// ...

const resolvers = [
  // ...
  Mutation: {
    addPerson: (root, args) => {
      const person = { ...args, id: uuid() }
      persons = persons.concat(person)
      return person
    }
  }
]
```

The mutation adds the object given to it as a parameter *args* to the array *persons*, and returns the object it added to the array.

The *id* field is given a unique value using the [uuid](#) library.

A new person can be added with the following mutation

```
mutation {
  addPerson(
    name: "Pekka Mikkola"
    phone: "045-2374321"
    street: "Vilppulantie 25"
    city: "Helsinki"
  ) {
    name
    phone
    address {
      city
      street
    }
    id
  }
}
```

Note that the person is saved to the *persons* array as

```
{
  name: "Pekka Mikkola",
  phone: "045-2374321",
  street: "Vilppulantie 25",
  city: "Helsinki",
```

```
id: "2b24e0b0-343c-11e9-8c2a-cb57c2bf804f"
```

But the response to the mutation is

```
{
  "data": [
    "addPerson": [
      {
        "name": "Pekka Mikkola",
        "phone": "045-2374321",
        "address": [
          {
            "city": "Helsinki",
            "street": "Vilppulantie 25"
          }
        ],
        "id": "2b24e0b0-343c-11e9-8c2a-cb57c2bf804f"
      }
    ]
}
```

So the resolver of the `address` field of the `Person` type formats the response object to the right form.

Error handling

If we try to create a new person, but the parameters do not correspond with the schema description, the server gives an error message:

The screenshot shows the GraphQL playground interface. On the left, under 'Operation', is the following GraphQL query:

```
mutation {
  addPerson(
    name: "Donald Trump"
  ) {
    name
    phone
    address{
      city
      street
    }
    id
  }
}
```

On the right, under 'Response', is the JSON response from the server:

```
{
  "errors": [
    {
      "message": "Field \"addPerson\" argument \"street\" of type \"String!\" is required, but it was not provided.",
      "extensions": {
        "code": "GRAPHQL_VALIDATION_FAILED",
        "exception": {
          "stacktrace": [
            "GraphQLError: Field \"addPerson\" argument \"street\" of type \"String!\" is required, but it was not provided.",
            "    at Object.leave (/Users/...)"
          ]
        }
      }
    }
  ],
  "status": 400,
  "duration": 34.0ms,
  "bodySize": 3.0KB
}
```

So some of the error handling can be automatically done with GraphQL [validation](#).

However, GraphQL cannot handle everything automatically. For example, stricter rules for data sent to a Mutation have to be added manually. An error could be handled by throwing `GraphQLError` with a proper [error code](#).

Let's prevent adding the same name to the phonebook multiple times:

```
const { GraphQLError } = require('graphql')
// ...
```

```

const resolvers = [
  // ...
  Mutation: [
    addPerson: (root, args) => [
      if (persons.find(p => p.name === args.name)) { throw new GraphQLERR
        or('Name must be unique', {
          extensions: {
            code: 'BAD_USER_INPUT',
            invalidArgs: args.name
          }
        })
      }
      const person = { ...args, id: uuid() }
      persons = persons.concat(person)
      return person
    ]
  ]
]

```

So if the name to be added already exists in the phonebook, throw *GraphQLError* error.

The screenshot shows the GraphQL playground interface. On the left, the mutation code is pasted:

```

1  mutation {
2    addPerson(
3      name: "Pekka Mikkola"
4      phone: "045-2374321"
5      street: "Vilppulantie 25"
6      city: "Helsinki"
7    ) {
8      name
9      phone
10     address{
11       city
12       street
13     }
14     id
15   }
16 }

```

On the right, the response shows an error message:

```

{
  "data": {
    "addPerson": null
  },
  "errors": [
    {
      "message": "Name must be unique",
      "locations": [
        {
          "line": 5,
          "column": 3
        }
      ],
      "path": [
        "addPerson"
      ],
      "extensions": {
        "code": "BAD_USER_INPUT",
        "invalidArgs": "Pekka Mikkola"
      }
    }
  ],
  "stacktrace": [
    "GraphQLError: Name must be unique"
  ]
}

```

The current code of the application can be found on [GitHub](#), branch *part8-2*.

Enum

Let's add a possibility to filter the query returning all persons with the parameter *phone* so that it returns only persons with a phone number

```

query {
  allPersons(phone: YES) {
    name
  }
}

```

```
    phone  
  }  
}  
}
```

or persons without a phone number

```
query {  
  allPersons(phone: NO) [  
    name  
  ]  
}
```

The schema changes like so:

```
enum YesNo { YES NO}  
type Query {  
  personCount: Int!  
  allPersons(phone: YesNo): [Person!]!  findPerson(name: String!): Person  
}
```

The type `YesNo` is a GraphQL [enum](#), or an enumerable, with two possible values: `YES` or `NO`. In the query `allPersons`, the parameter `phone` has the type `YesNo`, but is nullable.

The resolver changes like so:

```
Query: {  
  personCount: () => persons.length,  
  allPersons: (root, args) => [  
    if (!args.phone) { return persons }  
    const byPhone = (person) =>  
      args.phone === 'YES' ? person.phone : !person.  
      phone  
      return persons.filter(byPhone) ],  
  findPerson: (root, args) =>  
    persons.find(p => p.name === args.name)  
},
```

Changing a phone number

Let's add a mutation for changing the phone number of a person. The schema of this mutation looks as follows:

```
type Mutation {  
  addPerson(  
    name: String!  
    phone: String  
    street: String!  
    city: String!  
  ): Person  
  editNumber(  
    name: String!  
    phone: String! ): Person}
```

and is done by a resolver:

```

Mutation: [
  // ...
  editNumber: (root, args) => [
    const person = persons.find(p => p.name === args.name)
    if (!person) {
      return null
    }

    const updatedPerson = { ...person, phone: args.phone }
    persons = persons.map(p => p.name === args.name ? updatedPerson : p)
    return updatedPerson
  ]
]

```

The mutation finds the person to be updated by the field *name*.

The current code of the application can be found on [Github](#), branch *part8-3*.

More on queries

With GraphQL, it is possible to combine multiple fields of type *Query*, or "separate queries" into one query. For example, the following query returns both the amount of persons in the phonebook and their names:

```

query {
  personCount
  allPersons {
    name
  }
}

```

The response looks as follows:

```

{
  "data": {
    "personCount": 3,
    "allPersons": [
      {
        "name": "Arto Hellas"
      },
      {
        "name": "Matti Luukkainen"
      },
      {
        "name": "Venla Ruuska"
      }
    ]
  }
}

```

Combined query can also use the same query multiple times. You must however give the queries alternative names like so:

```
query {
  havePhone: allPersons(phone: YES) {
    name
  }
  phoneless: allPersons(phone: NO) {
    name
  }
}
```

The response looks like:

```
{
  "data": [
    "havePhone": [
      {
        "name": "Arto Hellas"
      },
      {
        "name": "Matti Luukkainen"
      }
    ],
    "phoneless": [
      {
        "name": "Venla Ruuska"
      }
    ]
  ]
}
```

In some cases, it might be beneficial to name the queries. This is the case especially when the queries or mutations have [parameters](#). We will get into parameters soon.

Exercises 8.1.-8.7

Through the exercises, we will implement a GraphQL backend for a small library. Start with [this file](#). Remember to *npm init* and to install dependencies!

8.1: The number of books and authors

Implement queries *bookCount* and *authorCount* which return the number of books and the number of authors.

The query

```
query {
  bookCount
  authorCount
```

should return

```
{  
  "data": [  
    {  
      "bookCount": 7,  
      "authorCount": 5  
    },  
    {  
      "bookCount": 7,  
      "authorCount": 5  
    }  
  ]  
}
```

8.2: All books

Implement query *allBooks*, which returns the details of all books.

In the end, the user should be able to do the following query:

```
query {  
  allBooks {  
    title  
    author  
    published  
    genres  
  }  
}
```

8.3: All authors

Implement query *allAuthors*, which returns the details of all authors. The response should include a field *bookCount* containing the number of books the author has written.

For example the query

```
query {  
  allAuthors {  
    name  
    bookCount  
  }  
}
```

should return

```
{  
  "data": [  
    {  
      "allAuthors": [  
        {  
          "name": "Robert Martin",  
          "bookCount": 2  
        },  
        {  
          "name": "John Smith",  
          "bookCount": 3  
        }  
      ]  
    }  
  ]  
}
```

```
[  
  {  
    "name": "Martin Fowler",  
    "bookCount": 1  
  },  
  {  
    "name": "Fyodor Dostoevsky",  
    "bookCount": 2  
  },  
  {  
    "name": "Joshua Kerievsky",  
    "bookCount": 1  
  },  
  {  
    "name": "Sandi Metz",  
    "bookCount": 1  
  }  
]
```

8.4: Books of an author

Modify the *allBooks* query so that a user can give an optional parameter *author*. The response should include only books written by that author.

For example query

```
query {  
  allBooks(author: "Robert Martin") {  
    title  
  }  
}
```

should return

```
[  
  {  
    "data": [  
      "allBooks": [  
        {  
          "title": "Clean Code"  
        },  
        {  
          "title": "Agile software development"  
        }  
      ]  
    ]  
  }  
]
```

8.5: Books by genre

Modify the query `allBooks` so that a user can give an optional parameter `genre`. The response should include only books of that genre.

For example query

```
query {  
  allBooks(genre: "refactoring") {  
    title  
    author  
  }  
}
```

should return

```
{  
  "data": [  
    "allBooks": [  
      {  
        "title": "Clean Code",  
        "author": "Robert Martin"  
      },  
      {  
        "title": "Refactoring, edition 2",  
        "author": "Martin Fowler"  
      },  
      {  
        "title": "Refactoring to patterns",  
        "author": "Joshua Kerievsky"  
      },  
      {  
        "title": "Practical Object-Oriented Design, An Agile Primer Using Ruby",  
        "author": "Sandi Metz"  
      }  
    ]  
  ]  
}
```

The query must work when both optional parameters are given:

```
query {  
  allBooks(author: "Robert Martin", genre: "refactoring") {  
    title  
    author  
  }  
}
```

8.6: Adding a book

Implement mutation `addBook`, which can be used like this:

```
mutation {
  addBook(
    title: "NoSQL Distilled",
    author: "Martin Fowler",
    published: 2012,
    genres: ["database", "nosql"]
  ) {
    title,
    author
  }
}
```

The mutation works even if the author is not already saved to the server:

```
mutation {
  addBook(
    title: "Pimeyden tango",
    author: "Reijo Mäki",
    published: 1997,
    genres: ["crime"]
  ) {
    title,
    author
  }
}
```

If the author is not yet saved to the server, a new author is added to the system. The birth years of authors are not saved to the server yet, so the query

```
query {
  allAuthors {
    name
    born
    bookCount
  }
}
```

returns

```
{
  "data": {
    "allAuthors": [
      // ...
      [
        {
          "name": "Reijo Mäki",
          "born": null,
          "bookCount": 1
        }
      ]
    ]
  }
}
```

8.7: Updating the birth year of an author

Implement mutation `editAuthor`, which can be used to set a birth year for an author. The mutation is used like so:

```
mutation {
  editAuthor(name: "Reijo Mäki", setBornTo: 1958) {
    name
    born
  }
}
```

If the correct author is found, the operation returns the edited author:

```
{
  "data": {
    "editAuthor": {
      "name": "Reijo Mäki",
      "born": 1958
    }
  }
}
```

If the author is not in the system, `null` is returned:

```
{
  "data": {
    "editAuthor": null
  }
}
```

8b: React and GraphQL

We will next implement a React app that uses the GraphQL server we created.

The current code of the server can be found on [GitHub](#), branch `part8-3`.

In theory, we could use GraphQL with HTTP POST requests. The following shows an example of this with Postman:

The screenshot shows the GraphQL playground interface at <http://localhost:4000/graphql>. A POST request is being sent to the same endpoint. The 'Body' tab is selected, showing a JSON query:

```
1 {  
2   "query": "query { allPersons { name } }"  
3 }
```

The response status is 200 OK, with a response time of 26 ms and a size of 346 B. The response body is displayed in Pretty mode:

```
1 {  
2   "data": {  
3     "allPersons": [  
4       {  
5         "name": "Arto Hellas"  
6       },  
7       {  
8         "name": "Matti Luukkainen"  
9       },  
10      {  
11        "name": "Venla Ruuska"  
12      }  
13    ]  
14  }  
15 }
```

The communication works by sending HTTP POST requests to <http://localhost:4000/graphql>. The query itself is a string sent as the value of the key `query`.

We could take care of the communication between the React app and GraphQL by using Axios. However, most of the time, it is not very sensible to do so. It is a better idea to use a higher-order library capable of abstracting the unnecessary details of the communication.

At the moment, there are two good options: [Relay](#) by Facebook and [Apollo Client](#), which is the client side of the same library we used in the previous section. Apollo is absolutely the most popular of the two, and we will use it in this section as well.

Apollo client

Let us create a new React app, and can continue installing dependencies required by [Apollo client](#).

```
npm install @apollo/client graphql
```

We'll start with the following code for our application:

```
import ReactDOM from 'react-dom/client'
```

```

import App from './App'

import { ApolloClient, InMemoryCache, gql } from '@apollo/client'

const client = new ApolloClient({
  uri: 'http://localhost:4000',
  cache: new InMemoryCache(),
})

const query = gql` 
query {
  allPersons {
    name,
    phone,
    address {
      street,
      city
    }
    id
  }
}
` 

client.query({ query })
  .then((response) => {
    console.log(response.data)
  })
}

ReactDOM.createRoot(document.getElementById('root')).render(<App />)

```

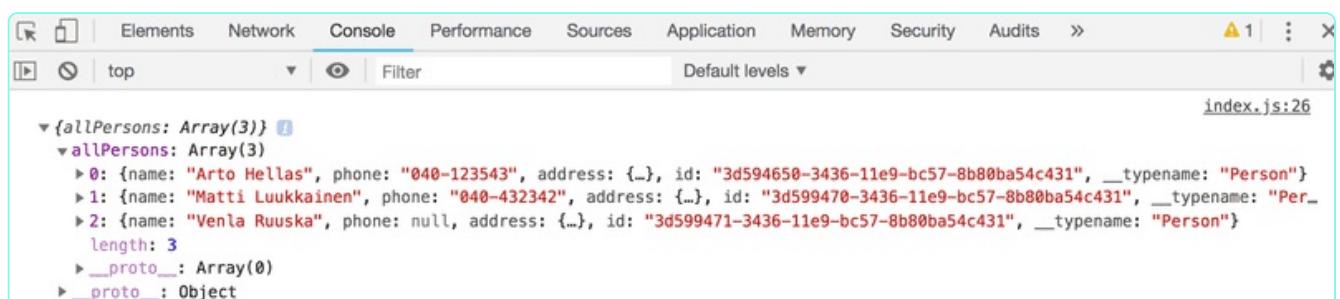
The beginning of the code creates a new `client` object, which is then used to send a query to the server:

```

client.query({ query })
  .then((response) => {
    console.log(response.data)
  })

```

The server's response is printed to the console:



The application can communicate with a GraphQL server using the `client` object. The client can be made accessible for all components of the application by wrapping the `App` component with

ApolloProvider.

```
import ReactDOM from 'react-dom/client'
import App from './App'

import {
  ApolloClient,
  ApolloProvider, InMemoryCache
} from '@apollo/client'

const client = new ApolloClient([
  uri: 'http://localhost:4000',
  cache: new InMemoryCache(),
])

ReactDOM.createRoot(document.getElementById('root')).render(
  <ApolloProvider client={client}>    <App />
  </ApolloProvider>
)
```

Making queries

We are ready to implement the main view of the application, which shows a list of person's name and phone number.

Apollo Client offers a few alternatives for making [queries](#). Currently, the use of the hook function [useQuery](#) is the dominant practice.

The query is made by the *App* component, the code of which is as follows:

```
import { gql, useQuery } from '@apollo/client'

const ALL_PERSONS = gql`query {
  allPersons {
    name
    phone
    id
  }
}`

const App = () => {
  const result = useQuery(ALL_PERSONS)

  if (result.loading) {
    return <div>loading...</div>
  }

  return [
    <div>
      {result.data.allPersons.map(p => p.name).join(', ')}
    </div>
  ]
}
```

```

        </div>
    }
}

export default App

```

When called, `useQuery` makes the query it receives as a parameter. It returns an object with multiple [fields](#). The field `loading` is true if the query has not received a response yet. Then the following code gets rendered:

```

if (result.loading) {
    return <div>loading...</div>
}

```

When a response is received, the result of the `allPersons` query can be found in the `data` field, and we can render the list of names to the screen.

```

<div>
    {result.data.allPersons.map(p => p.name).join(' ')}
</div>

```

Let's separate displaying the list of persons into its own component:

```

const Persons = ({ persons }) => {
    return [
        <div>
            <h2>Persons</h2>
            {persons.map(p =>
                <div key={p.name}>
                    {p.name} {p.phone}
                </div>
            )}
        </div>
    ]
}

```

The `App` component still makes the query, and passes the result to the new component to be rendered:

```

const App = () => {
    const result = useQuery(ALL_PERSONS)

    if (result.loading) {
        return <div>loading...</div>
    }

    return [
        <Persons persons={result.data.allPersons}>
    ]
}

```

Named queries and variables

Let's implement functionality for viewing the address details of a person. The *findPerson* query is well-suited for this.

The queries we did in the last chapter had the parameter hardcoded into the query:

```
query {
  findPerson(name: "Arto Hellas") {
    phone
    city
    street
    id
  }
}
```

When we do queries programmatically, we must be able to give them parameters dynamically.

GraphQL [variables](#) are well-suited for this. To be able to use variables, we must also name our queries.

A good format for the query is this:

```
query findPersonByName($nameToSearch: String!) {
  findPerson(name: $nameToSearch) {
    name
    phone
    address {
      street
      city
    }
  }
}
```

The name of the query is *findPersonByName*, and it is given a string *\$nameToSearch* as a parameter.

It is also possible to do queries with parameters with the Apollo Explorer. The parameters are given in *Variables*:

The screenshot shows the Apollo GraphQL playground interface. At the top, there are tabs for 'addPerson', 'AllPersons', and 'findPersonByName'. The 'findPersonByName' tab is active, showing an operation with the following GraphQL code:

```

1 query findPersonByName($nameToSearch: String!) {
2   findPerson(name: $nameToSearch) {
3     name
4     phone
5     address {
6       street
7       city
8     }
9   }
10 }

```

Below the code, the 'Variables' section contains the following JSON data:

```

1 "nameToSearch": "Arto Hellas"
2
3

```

The 'Response' section shows the result of the query, which is a JSON object:

```

{
  "data": {
    "findPerson": {
      "name": "Arto Hellas",
      "phone": "121345",
      "address": {
        "street": "Tapiolankatu 5 A",
        "city": "Espoo"
      }
    }
  }
}

```

The `useQuery` hook is well-suited for situations where the query is done when the component is rendered. However, we now want to make the query only when a user wants to see the details of a specific person, so the query is done only [as required](#).

One possibility for this kind of situations is the hook function `useLazyQuery` that would make it possible to define a query which is executed *when* the user wants to see the detailed information of a person.

However, in our case we can stick to `useQuery` and use the option `skip`, which makes it possible to do the query only if a set condition is true.

The solution is as follows:

```

import { useState } from 'react'
import { gql, useQuery } from '@apollo/client'

const FIND_PERSON = gql` 
query findPersonByName($nameToSearch: String!) {
  findPerson(name: $nameToSearch) {
    name
    phone
    id
    address {
      street
      city
    }
  }
}

const Person = ({ person, onClose }) => {
  return [

```

```

<div>
  <h2>{person.name}</h2>
  <div>
    {person.address.street} {person.address.city}
  </div>
  <div>{person.phone}</div>
  <button onClick={onClose}>close</button>
</div>
}

}

const Persons = ({ persons }) => {
  const [nameToSearch, setNameToSearch] = useState(null)  const result = useQuery
(FIND_PERSON, { variables: { nameToSearch }, skip: !nameToSearch, })
  if (nameToSearch && result.data) { return (
    <Person person={result.data.findPerson} />
    )
  }

  return (
    <div>
      <h2>Persons</h2>
      {persons.map((p) => (
        <div key={p.name}>
          {p.name} {p.phone}
          <button onClick={() => setNameToSearch(p.name)}> show address
        </button>
        </div>
      ))
    )
  )
}

export default Persons

```

The code has changed quite a lot, and all of the changes are not completely apparent.

When the button *show address* of a person is pressed, the name of the person is set to state *nameToSearch*:

```

<button onClick={() => setNameToSearch(p.name)}>
  show address
</button>

```

This causes the component to re-render itself. On render the query *FIND_PERSON* that fetches the detailed information of a user is executed *if the variable nameToSearch has a value*:

```

const result = useQuery(FIND_PERSON, {
  variables: { nameToSearch },
  skip: !nameToSearch,
})

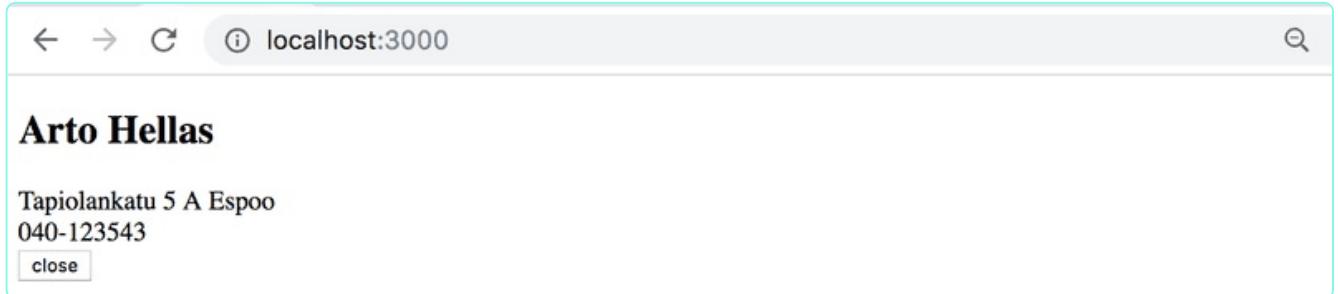
```

When the user is not interested in seeing the detailed info of any person, the state variable *nameToSearch* is null and the query is not executed.

If the state `nameToSearch` has a value and the query result is ready, the component `Person` renders the detailed info of a person:

```
if (nameToSearch && result.data) {  
  return [  
    <Person  
      person={result.data.findPerson}  
      onClose={() => setNameToSearch(null)}  
    />  
  ]  
}
```

A single-person view looks like this:



When a user wants to return to the person list, the `nameToSearch` state is set to `null`.

The current code of the application can be found on [GitHub](#) branch `part8-1`.

Cache

When we do multiple queries, for example with the address details of Arto Hellas, we notice something interesting: the query to the backend is done only the first time around. After this, despite the same query being done again by the code, the query is not sent to the backend.

The screenshot shows a browser window with the URL `localhost:3000`. At the top, there is a header with the text "Arto Hellas" and address information: "Tapiolankatu 5 A Espoo" and "040-123543". Below this is a "close" button. The main content area is a developer tools Network tab. It has a list of requests on the left and their corresponding responses on the right. One response is highlighted in red, showing the JSON data:

```
{"data": {"findPerson": {"name": "Arto Hellas", "phone": "040-123543", "address": "Tapiolankatu 5 A Espoo", "id": "3d594650-3436-11e9-bc57-8b80ba54c431"}, "allPersons": [{"name": "Arto Hellas", "id": "3d594650-3436-11e9-bc57-8b80ba54c431"}, {"name": "Matti Luukkainen", "id": "3d599470-3436-11e9-bc57-8b80ba54c431"}, {"name": "Venla Ruuska", "id": "3d599471-3436-11e9-bc57-8b80ba54c431"}]}
```

Apollo client saves the responses of queries to [cache](#). To optimize performance if the response to a query is already in the cache, the query is not sent to the server at all.

The screenshot shows a browser window with the URL `localhost:3000`. At the top, there is a header with the text "Persons" and three names: "Arto Hellas 121345", "Matti Luukkainen 040-432342", and "Venla Ruuska", each with a "show address" button. Below this is a "close" button. The main content area is the Apollo GraphQL DevTools interface. On the left, there is an "EXPLORER" section with a search bar and a list of queries. The "CACHE" section is active, showing the "ROOT_QUERY" cache entry. The cache data is displayed as a tree structure:

```
ROOT_QUERY
  __typename: "Query"
  allPersons: []
    0: {}
      __ref: "Person:3d594650-3436-11e9-bc57-8b80ba54c431"
    1: {}
      __ref: "Person:3d599470-3436-11e9-bc57-8b80ba54c431"
    2: {}
      __ref: "Person:3d599471-3436-11e9-bc57-8b80ba54c431"
  findPerson("name": "Arto Hellas"): {}
  findPerson("name": "Matti Luukkainen"): {}
```

Cache shows the detailed info of Arto Hellas after the query `findPerson`:

Doing mutations

Let's implement functionality for adding new persons.

In the previous chapter, we hardcoded the parameters for mutations. Now, we need a version of the `addPerson` mutation which uses [variables](#):

```
const CREATE_PERSON = gql`  
mutation createPerson($name: String!, $street: String!, $city: String!, $phone: String) {  
  addPerson(  
    name: $name,  
    street: $street,  
    city: $city,  
    phone: $phone  
  ) {  
    name  
    phone  
    id  
    address {  
      street  
      city  
    }  
  }  
}
```

The hook function [useMutation](#) provides the functionality for making mutations.

Let's create a new component for adding a new person to the directory:

```
import { useState } from 'react'  
import { gql, useMutation } from '@apollo/client'
```

```

const CREATE_PERSON = gql`  

  // ...
  

const PersonForm = () => {
  const [name, setName] = useState('')
  const [phone, setPhone] = useState('')
  const [street, setStreet] = useState('')
  const [city, setCity] = useState('')
  

  const [createPerson] = useMutation(CREATE_PERSON)
  const submit = (event) => {
    event.preventDefault()
  

    createPerson({ variables: { name, phone, street, city } })
    setName('')
    setPhone('')
    setStreet('')
    setCity('')
  }
}
  

  return (
    <div>
      <h2>create new</h2>
      <form onSubmit={submit}>
        <div>
          name <input value={name}>
          onChange={({ target }) => setName(target.value)}
        />
        </div>
        <div>
          phone <input value={phone}>
          onChange={({ target }) => setPhone(target.value)}
        />
        </div>
        <div>
          street <input value={street}>
          onChange={({ target }) => setStreet(target.value)}
        />
        </div>
        <div>
          city <input value={city}>
          onChange={({ target }) => setCity(target.value)}
        />
        </div>
        <button type='submit'>add!</button>
      </form>
    </div>
  )
}
  

export default PersonForm

```

The code of the form is straightforward and the interesting lines have been highlighted. We can define mutation functions using the `useMutation` hook. The hook returns an array, the first element of which contains the function to cause the mutation.

```
const [ createPerson ] = useMutation(CREATE_PERSON)
```

The query variables receive values when the query is made:

```
createPerson({ variables: { name, phone, street, city } })
```

New persons are added just fine, but the screen is not updated. This is because Apollo Client cannot automatically update the cache of an application, so it still contains the state from before the mutation. We could update the screen by reloading the page, as the cache is emptied when the page is reloaded. However, there must be a better way to do this.

Updating the cache

There are a few different solutions for this. One way is to make the query for all persons [poll](#) the server, or make the query repeatedly.

The change is small. Let's set the query to poll every two seconds:

```
const App = () => [
  const result = useQuery(ALL_PERSONS, {
    pollInterval: 2000
  })

  if (result.loading) {
    return <div>loading...</div>
  }

  return [
    <div>
      <Persons persons={result.data.allPersons}>
        <PersonForm />
      </Persons>
    </div>
  ]
}

export default App
```

The solution is simple, and every time a user adds a new person, it appears immediately on the screens of all users.

The bad side of the solution is all the pointless web traffic.

Another easy way to keep the cache in sync is to use the `useMutation` hook's [refetchQueries](#) parameter to define that the query fetching all persons is done again whenever a new person is created.

```

const ALL_PERSONS = gql` 
query {
  allPersons {
    name
    phone
    id
  }
}
` 

const PersonForm = (props) => [
// ...

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    refetchQueries: [ { query: ALL_PERSONS } ]
  })
]

```

The pros and cons of this solution are almost opposite of the previous one. There is no extra web traffic because queries are not done just in case. However, if one user now updates the state of the server, the changes do not show to other users immediately.

If you want to do multiple queries, you can pass multiple objects inside refetchQueries. This will allow you to update different parts of your app at the same time. Here is an example:

```

const [ createPerson ] = useMutation(CREATE_PERSON, {
  refetchQueries: [ { query: ALL_PERSONS }, { query: OTHER_QUERY }, { query: ... } ] // pass as many queries as you need
})

```

There are other ways to update the cache. More about those later in this part.

At the moment, queries and components are defined in the same place in our code. Let's separate the query definitions into their own file *queries.js*:

```

import { gql } from '@apollo/client'

export const ALL_PERSONS = gql` 
query {
  // ...
}
` 

export const FIND_PERSON = gql` 
query findPersonByName($nameToSearch: String!) {
  // ...
}
` 

export const CREATE_PERSON = gql` 
mutation createPerson($name: String!, $street: String!, $city: String!, $phone : String) {
  // ...
}
`

```

Each component then imports the queries it needs:

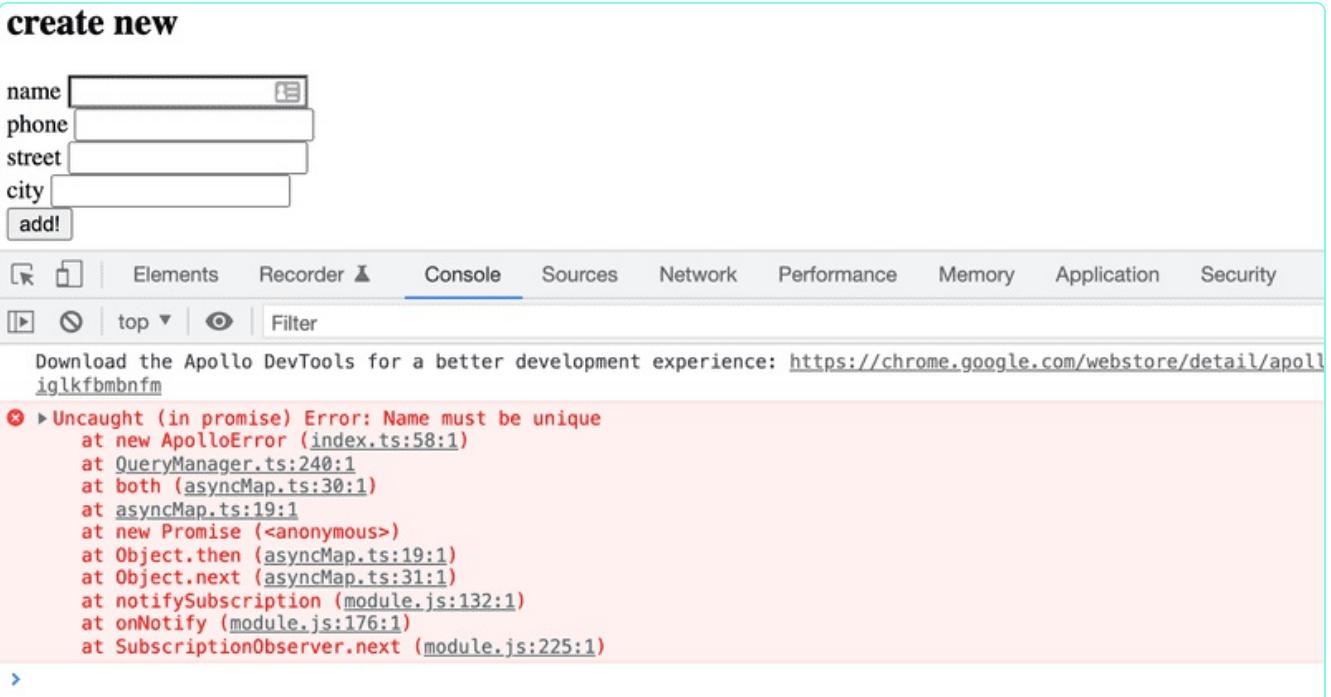
```
import { ALL_PERSONS } from './queries'

const App = () => [
  const result = useQuery(ALL_PERSONS)
  // ...
]
```

The current code of the application can be found on [GitHub](#) branch *part8-2*.

Handling mutation errors

Trying to create a person with invalid data causes an error:



We should handle the exception. We can register an error handler function to the mutation using the `useMutation` hook's `onError` option.

Let's register the mutation with an error handler that uses the `setError*` function it receives as a parameter to set an error message:

```
const PersonForm = ({ setError }) => [
  // ...

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    refetchQueries: [ { query: ALL_PERSONS } ],
    onError: (error) => {
      const messages = error.graphQLErrors.map(e => e.message).join('\n')
      setError(messages)
    }
  })
]
```

We can then render the error message on the screen as necessary:

```
const App = () => {
  const [errorMessage, setErrorMessage] = useState(null)
  const result = useQuery(ALL_PERSONS)

  if (result.loading) {
    return <div>loading...</div>
  }

  const notify = (message) => {
    setErrorMessage(message)
    setTimeout(() => {
      setErrorMessage(null)
    }, 10000)
  }

  return (
    <div>
      <Notify errorMessage={errorMessage} />      <Persons persons={result.data.allPersons} />
      <PersonForm setError={notify} />      </div>
    )
  }
}

const Notify = ({errorMessage}) => {
  if (!errorMessage) {
    return null
  }
  return (
    <div style={{color: 'red'}}> {errorMessage} </div>
  )
}
```

Now the user is informed about an error with a simple notification.

The screenshot shows a web browser window with the URL `localhost:3000`. At the top, there is an error message: **Name must be unique**. Below it, the title **Persons** is displayed. A list of existing persons is shown with their names and phone numbers: Arto Hellas (040-123543), Matti Luukainen (040-432342), and Venla Ruuska. Each person entry includes a "show address" button. Below this list is a form for adding a new person. The form has fields for name (Arto Hellas), phone, street, and city, each with a "show address" button. At the bottom of the form is a blue "add!" button.

The current code of the application can be found on [GitHub](#) branch *part8-3*.

Updating a phone number

Let's add the possibility to change the phone numbers of persons to our application. The solution is almost identical to the one we used for adding new persons.

Again, the mutation requires parameters.

```

export const EDIT_NUMBER = gql` 
  mutation editNumber($name: String!, $phone: String!) {
    editNumber(name: $name, phone: $phone) {
      name
      phone
      address {
        street
        city
      }
      id
    }
  }
`
```

The *PhoneForm* component responsible for the change is straightforward. The form has fields for the person's name and new phone number, and calls the *changeNumber* function. The function is done using the *useMutation* hook. Interesting lines on the code have been highlighted.

```

import { useState } from 'react'
import { useMutation } from '@apollo/client'

import { EDIT_NUMBER } from '../queries'

const PhoneForm = () => {
  const [name, setName] = useState('')
  const [phone, setPhone] = useState('')

  const [changeNumber] = useMutation(EDIT_NUMBER)
  const submit = (event) => {
    event.preventDefault()

    changeNumber({ variables: { name, phone } })
    setName('')
    setPhone('')
  }

  return [
    <div>
      <h2>change number</h2>

      <form onSubmit={submit}>
        <div>
          name <input
            value={name}
            onChange={({ target }) => setName(target.value)}
          />
        </div>
        <div>
          phone <input
            value={phone}
            onChange={({ target }) => setPhone(target.value)}
          />
        </div>
      </form>
    </div>
  ]
}
```

```

        />
      </div>
      <button type='submit'>change number</button>
    </form>
  </div>
}

}

export default PhoneForm

```

It looks bleak, but it works:

The screenshot shows a web application interface. At the top, a navigation bar includes a back arrow, forward arrow, refresh icon, and the URL "localhost:3000". Below the navigation, the word "Persons" is displayed in bold. Underneath "Persons", there is a list of three items: "Arto Hellas 1" with a "show address" button, "Matti Luukkainen 040-432342" with a "show address" button, and "Venla Ruuska" with a "show address" button. Below this section, the text "create new" is followed by four input fields: "name", "phone", "street", and "city", each with its own "show address" button. An "add!" button is located below these fields. Further down, the text "change number" is followed by two input fields: "name" (containing "Arto Hellas") and "phone" (containing "040-2123124"). A "change number" button is positioned below the phone field. The entire application is contained within a light blue border.

Surprisingly, when a person's number is changed, the new number automatically appears on the list of persons rendered by the *Persons* component. This happens because each person has an identifying field of type *ID*, so the person's details saved to the cache update automatically when they are changed with the mutation.

Our application still has one small flaw. If we try to change the phone number for a name which does not exist, nothing seems to happen. This happens because if a person with the given name cannot be found, the mutation response is *null*:

change number

The screenshot shows a browser's developer tools Network tab. At the top, there are input fields for 'name' and 'phone', and a button labeled 'change number'. Below the inputs, the Network tab is selected. A search bar is present. In the main area, a table lists a single entry for 'localhost'. The table has columns for Name, Headers, Preview, Response, Timing, and Initiator. The 'Response' column shows the following JSON:

Name	Headers	Preview	Response	Timing	Initiator
localhost			1 {"data": {"editNumber": null}} 2		

For GraphQL, this is not an error, so registering an *onError* error handler is not useful.

We can use the *result* field returned by the *useMutation* hook as its second parameter to generate an error message.

```
const PhoneForm = ([{ setError }] => {
  const [name, setName] = useState('')
  const [phone, setPhone] = useState('')

  const [changeNumber, result] = useMutation(EDIT_NUMBER)
  const submit = (event) => {
    // ...
  }

  useEffect(() => {
    if (result.data && result.data.editNumber === null) {
      setError('person not found')
    } else {
      [result.data]
    }
  })
})
```

If a person cannot be found, or the *result.data.editNumber* is *null*, the component uses the callback function it received as props to set a suitable error message. We want to set the error message only when the result of the mutation *result.data* changes, so we use the *useEffect* hook to control setting the error message.

The current code of the application can be found on [GitHub](#) branch *part8-4*.

Apollo Client and the applications state

In our example, management of the applications state has mostly become the responsibility of Apollo Client. This is a quite typical solution for GraphQL applications. Our example uses the state of the React components only to manage the state of a form and to show error notifications. As a result, it could be that there are no justifiable reasons to use Redux to manage application state when using GraphQL.

When necessary, Apollo enables saving the application's local state to [Apollo cache](#).

Exercises 8.8.-8.12

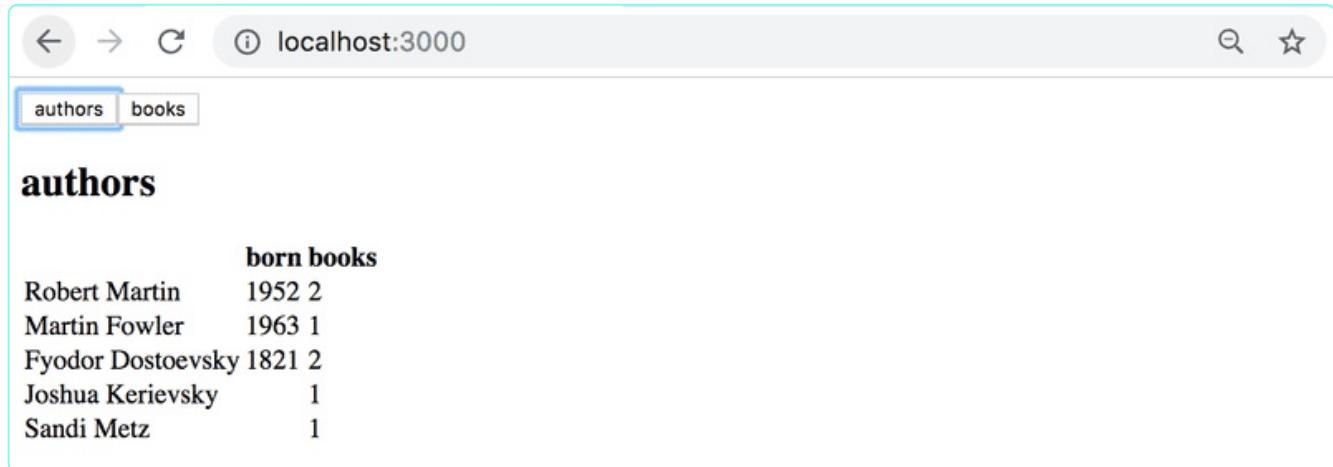
Through these exercises, we'll implement a frontend for the GraphQL library.

Take [this project](#) as a start for your application.

Note if you want, you can also use [React router](#) to implement the application's navigation!

8.8: Authors view

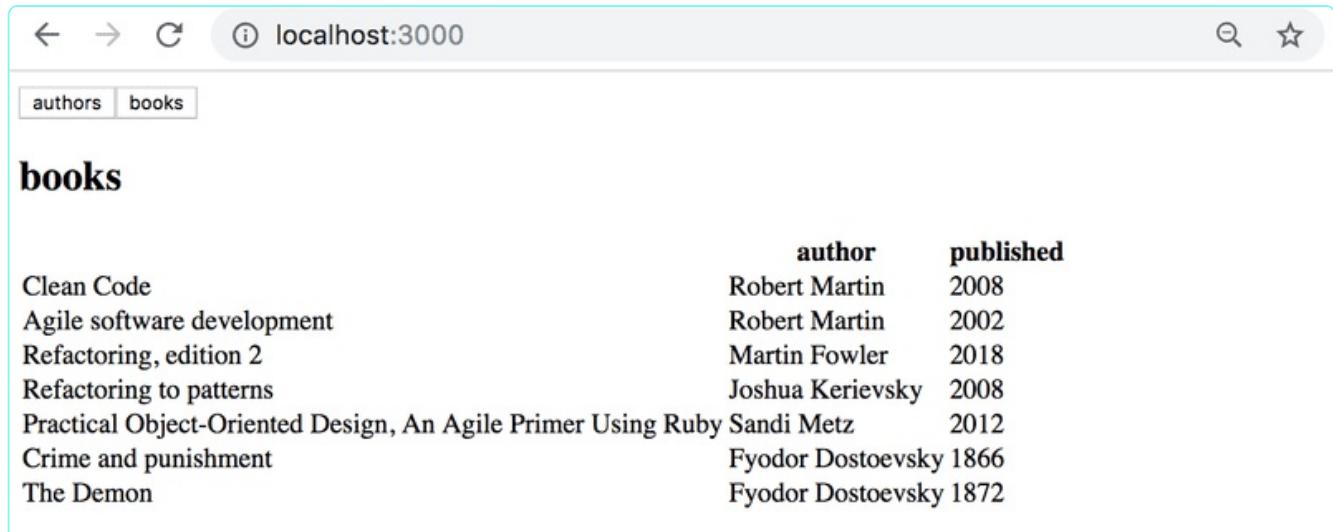
Implement an Authors view to show the details of all authors on a page as follows:



born	books
Robert Martin	1952 2
Martin Fowler	1963 1
Fyodor Dostoevsky	1821 2
Joshua Kerievsky	1
Sandi Metz	1

8.9: Books view

Implement a Books view to show on a page all other details of all books except their genres.



author	published
Robert Martin	2008
Robert Martin	2002
Martin Fowler	2018
Joshua Kerievsky	2008
Sandi Metz	2012
Fyodor Dostoevsky	1866
Fyodor Dostoevsky	1872

8.10: Adding a book

Implement a possibility to add new books to your application. The functionality can look like this:

localhost:3000

authors books add book

title Pimeyden tango

author Reijo Mäki

published 1997

genres: crime turku

create book

Make sure that the Authors and Books views are kept up to date after a new book is added.

In case of problems when making queries or mutations, check from the developer console what the server response is:

localhost:3000

Unhandled Rejection (Error): Network error: Response not successful: Received status code 400

```
new ApolloError
/Users/mluukkai/opetus/_koodi_fs/8/laskarit/frontend/ApolloError.js:26
 23 | __extends(ApolloError, _super);
 24 | function ApolloError(_a) {
 25 |   var graphQLErrors = _a.graphQLErrors, networkError = _a.networkError, errorMessage = _a.errorMessage, extraInfo = _a.extraInfo;
> 26 |   var _this = _super.call(this, errorMessage) || this;
 27 |   ^ _this.graphQLErrors = graphQLErrors || [];
 28 |   _this.networkError = networkError || null;
 29 |   if (!errorMessage) {
```

View compiled

error

nts	Console	Sources	Network	Performance	Memory	Components	Application	Security	Audits	AdBlock	Adblock Plus	Redux	Profiler	
	x	● ○ ⚡ 🔍	<input checked="" type="checkbox"/> Preserve log <input checked="" type="checkbox"/> Disable cache	Online	▼ ↑ ↓									
C	Filter	<input type="checkbox"/> Hide data URLs All <input checked="" type="checkbox"/> XHR JS CSS Img Media Font Doc WS Manifest Other												
	Name	x	Headers	Preview	Response	Timing	Initiator							
	localhost		▼ {errors: [{message: "Variable \"\$year\" of required type \"Int!\" was not provided.",...}]} ▶ errors: [{message: "Variable \"\$year\" of required type \"Int!\" was not provided.",...}]											
	0.chunk.js													
	0.chunk.js.map													

8.11: Authors birth year

Implement a possibility to set authors birth year. You can create a new view for setting the birth year, or place it on the Authors view:

authors books add book

authors

	born	books
Robert Martin	1952	2
Martin Fowler	1963	1
Fyodor Dostoevsky	1821	2
Joshua Kerievsky		1
Sandi Metz		1

Set birthyear

name Joshua Kerievsky
born 1967
update author

Make sure that the Authors view is kept up to date after setting a birth year.

8.12: Authors birth year advanced

Change the birth year form so that a birth year can be set only for an existing author. Use [select tag](#), [react select](#), or some other mechanism.

A solution using the react select library looks as follows:

authors books add book

authors

	born	books
Robert Martin	1952	2
Martin Fowler	1963	1
Fyodor Dostoevsky	1821	2
Joshua Kerievsky	1981	1
Sandi Metz		1

Set birthyear

Joshua Kerievsky
born 1967
update author

8c: Database and user administration

We will now add user management to our application, but let's first start using a database for storing data.

Mongoose and Apollo

Install Mongoose and dotenv:

```
npm install mongoose dotenv
```

We will imitate what we did in parts 3 and 4.

The person schema has been defined as follows:

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

const schema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 5
  },
  phone: {
    type: String,
    minlength: 5
  },
  street: {
    type: String,
    required: true,
    minlength: 5
  },
  city: {
    type: String,
    required: true,
    minlength: 3
  }
})

module.exports = mongoose.model('Person', schema)
```

We also included a few validations. *required: true*, which makes sure that a value exists, is actually redundant: we already ensure that the fields exist with GraphQL. However, it is good to also keep validation in the database.

We can get the application to mostly work with the following changes:

```
// ...
const mongoose = require('mongoose')
mongoose.set('strictQuery', false)
const Person = require('./models/person')

require('dotenv').config()

const MONGODB_URI = process.env.MONGODB_URI
```

```

console.log('connecting to', MONGODB_URI)

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

const typeDefs = gql` 
  ...
  `

const resolvers = [
  Query: {
    personCount: async () => Person.collection.countDocuments(),
    allPersons: async (root, args) => [
      // filters missing
      return Person.find({})
    ],
    findPerson: async (root, args) => Person.findOne({ name: args.name }),
  },
  Person: {
    address: (root) => {
      return {
        street: root.street,
        city: root.city,
      }
    }
  },
  Mutation: {
    addPerson: async (root, args) => {
      const person = new Person({ ...args })
      return person.save()
    },
    editNumber: async (root, args) => {
      const person = await Person.findOne({ name: args.name })
      person.phone = args.phone
      return person.save()
    }
  },
]

```

The changes are pretty straightforward. However, there are a few noteworthy things. As we remember, in Mongo, the identifying field of an object is called `_id` and we previously had to parse the name of the field to `id` ourselves. Now GraphQL can do this automatically.

Another noteworthy thing is that the resolver functions now return a *promise*, when they previously returned normal objects. When a resolver returns a promise, Apollo server [sends back](#) the value which the promise resolves to.

For example, if the following resolver function is executed,

```
allPersons: async (root, args) => {
  return Person.find({})
},
```

Apollo server waits for the promise to resolve, and returns the result. So Apollo works roughly like this:

```
allPersons: async (root, args) => {
  const result = await Person.find({})
  return result
},
```

Let's complete the *allPersons* resolver so it takes the optional parameter *phone* into account:

```
Query: [
  // ...
  allPersons: async (root, args) => {
    if (!args.phone) {
      return Person.find({})
    }

    return Person.find({ phone: { $exists: args.phone === 'YES' } })
  },
]
```

So if the query has not been given a parameter *phone*, all persons are returned. If the parameter has the value *YES*, the result of the query

```
Person.find({ phone: { $exists: true } })
```

is returned, so the objects in which the field *phone* has a value. If the parameter has the value *NO*, the query returns the objects in which the *phone* field has no value:

```
Person.find({ phone: { $exists: false } })
```

Validation

As well as in GraphQL, the input is now validated using the validations defined in the mongoose schema. For handling possible validation errors in the schema, we must add an error-handling *try/catch* block to the *save* method. When we end up in the catch, we throw a exception [GraphQLError](#) with error code :

```
Mutation: [
  addPerson: async (root, args) => {
    const person = new Person({ ...args })

    try {
      await person.save()
    } catch (error) {
      throw new G
```

```

    graphQLError('Saving person failed', {
      extensions: {
        code: 'B'
        AD_USER_INPUT: {
          invalidArgs: args.name,
          error
        }
      }
    })
    return person
  },
  editNumber: async (root, args) => {
    const person = await Person.findOne({ name: args.name })
    person.phone = args.phone

    try {
      await person.save()
    } catch (error) {
      throw new GraphQLError('Saving number failed', {
        extensions: {
          code: 'B'
          AD_USER_INPUT: {
            invalidArgs: args.name,
            error
          }
        }
      })
    }
  }
}

```

We have also added the Mongoose error and the data that caused the error to the `extensions` object that is used to convey more info about the cause of the error to the caller. The frontend can then display this information to the user, who can try the operation again with a better input.

The code of the backend can be found on [Github](#), branch `part8-4`.

User and log in

Let's add user management to our application. For simplicity's sake, let's assume that all users have the same password which is hardcoded to the system. It would be straightforward to save individual passwords for all users following the principles from [part 4](#), but because our focus is on GraphQL, we will leave out all that extra hassle this time.

The user schema is as follows:

```

const mongoose = require('mongoose')

const schema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    minlength: 3
  },
  friends: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Person'
    }
  ]
})

module.exports = mongoose.model('User', schema)

```

Every user is connected to a bunch of other persons in the system through the `friends` field. The idea is that when a user, e.g. *mluukkai*, adds a person, e.g. *Arto Hellas*, to the list, the person is added to

their *friends* list. This way, logged-in users can have their own personalized view in the application.

Logging in and identifying the user are handled the same way we used in [part 4](#) when we used REST, by using tokens.

Let's extend the schema like so:

```
type User {
  username: String!
  friends: [Person!]!
  id: ID!
}

type Token {
  value: String!
}

type Query {
  // ...
  me: User
}

type Mutation {
  // ...
  createUser(
    username: String!
  ): User
  login(
    username: String!
    password: String!
  ): Token
}
```

The query *me* returns the currently logged-in user. New users are created with the *createUser* mutation, and logging in happens with the *login* mutation.

The resolvers of the mutations are as follows:

```
const jwt = require('jsonwebtoken')

Mutation: [
  // ...
  createUser: async (root, args) => {
    const user = new User({ username: args.username })

    return user.save()
      .catch(error => {
        throw new GraphQLError('Creating the user failed', {
          extensions: {
            code: 'BAD_USER_INPUT',
            invalidArgs: args.username,
            error
        })
      })
  }
]
```

```

        }
    })
},
login: async [root, args] => {
    const user = await User.findOne({ username: args.username })

    if (!user || args.password !== 'secret') {
        throw new GraphQLError('wrong credentials', [
            extensions: {
                code: 'BAD_USER_INPUT'
            }
        ])
    }

    const userForToken = [
        username: user.username,
        id: user._id,
    ]

    return { value: jwt.sign(userForToken, process.env.JWT_SECRET) }
},
}

```

The new user mutation is straightforward. The login mutation checks if the username/password pair is valid. And if it is indeed valid, it returns a jwt token familiar from [part 4](#). Note that the `JWT_SECRET` must be defined in the `.env` file.

User creation is done now as follows:

```

mutation {
  createUser(
    username: "mluukkai"
  ) {
    username
    id
  }
}

```

The mutation for logging in looks like this:

```

mutation {
  login(
    username: "mluukkai"
    password: "secret"
  ) {
    value
  }
}

```

Just like in the previous case with REST, the idea now is that a logged-in user adds a token they

receive upon login to all of their requests. And just like with REST, the token is added to GraphQL queries using the *Authorization* header.

In the Apollo Explorer, the header is added to a query like so:

The screenshot shows the Apollo Explorer interface. In the 'Operation' tab, a GraphQL query named 'findPersonByName' is defined. It has a variable '\$nameToSearch' and returns a 'findPerson' object with fields 'name', 'phone', and 'address' (which contains 'street' and 'city'). The 'Variables' tab shows a single header 'Authorization' with the value 'bearer eyJhbGciOiJIUzI1NiIsInR5cC'. The 'Response' tab shows a JSON response with a 'data' field containing the 'findPerson' object with the specified details. The status bar at the top right indicates a 200 status code, 80.3ms latency, and 134B response size.

Modify the startup of the backend by giving the function that handles the startup `startStandaloneServer` another parameter `context`

```
startStandaloneServer(server, {
  listen: { port: 4000 },
  context: async ({ req, res }) => {
    const auth = req ? req.headers.authorization : null
    if (auth && auth.startsWith('Bearer ')) {
      const decodedToken = jwt.verify(auth.substring(7), process.env.JWT_SECRET)
      const currentUser = await User.findById(decodedToken.id).populate('friends')
      return { currentUser }
    }
    return null
  }
})
```

The object returned by `context` is given to all resolvers as their *third parameter*. Context is the right place to do things which are shared by multiple resolvers, like [user identification](#).

So our code sets the object corresponding to the user who made the request to the `currentUser` field of the context. If there is no user connected to the request, the value of the field is `undefined`.

The resolver of the `me` query is very simple: it just returns the logged-in user it receives in the `currentUser` field of the third parameter of the resolver, `context`. It's worth noting that if there is no logged-in user, i.e. there is no valid token in the header attached to the request, the query returns `null`:

```
Query: {
  // ...
  me: (root, args, context) => {
    return context.currentUser
```

```
}
```

If the header has the correct value, the query returns the user information identified by the header

The screenshot shows the GraphQL playground interface. In the 'Operation' tab, a query is defined:

```
query {
  me {
    username
    id
    friends {
      name
    }
  }
}
```

In the 'Headers' tab, an 'Authorization' header is set to 'Bearer eyJhbGciOiJIUzI1Ni...'.

The 'Response' tab shows the result of the query:

```
data: {
  me: {
    "username": "mluukkai",
    "id": "63dce23c255e4f416390fc9e",
    "friends": []
  }
}
```

The status is 200, and the response size is 85B.

Friends list

Let's complete the application's backend so that adding and editing persons requires logging in, and added persons are automatically added to the friends list of the user.

Let's first remove all persons not in anyone's friends list from the database.

addPerson mutation changes like so:

```
Mutation: [
  addPerson: async [root, args, context] => {
    const person = new Person([
      ...args
    ])
    const currentUser = context.currentUser
    if (!currentUser) {
      throw new GraphQLError('not authenticated', {
        extensions: {
          code: 'BAD_USER_INPUT',
        }
      })
    }
    try {
      await person.save()
      currentUser.friends = currentUser.friends.concat(person)
      await currentUser.save()
    } catch (error) {
      throw new GraphQLError('Saving user failed', {
        extensions: {
          code: 'BAD_USER_INPUT',
          invalidArgs: args.name,
          error
        }
      })
    }
  }
]

return person
```

```
//  
// ...  
}
```

If a logged-in user cannot be found from the context, an *GraphQLError* with a proper message is thrown. Creating new persons is now done with *async/await* syntax, because if the operation is successful, the created person is added to the friends list of the user.

Let's also add functionality for adding an existing user to your friends list. The mutation is as follows:

```
type Mutation {  
    // ...  
    addAsFriend(  
        name: String!  
    ): User  
}
```

And the mutation's resolver:

```
addAsFriend: async (root, args, { currentUser }) => {  
    const isFriend = (person) =>  
        currentUser.friends.map(f => f._id.toString()).includes(person._id.toString())  
  
    if (!currentUser) {  
        throw new GraphQLError('wrong credentials', {  
            extensions: { code: 'BAD_USER_INPUT' }  
        })  
    }  
  
    const person = await Person.findOne({ name: args.name })  
    if (!isFriend(person)) {  
        currentUser.friends = currentUser.friends.concat(person)  
    }  
  
    await currentUser.save()  
  
    return currentUser  
},
```

Note how the resolver *destructures* the logged-in user from the context. So instead of saving *currentUser* to a separate variable in a function

```
addAsFriend: async (root, args, context) => {  
    const currentUser = context.currentUser
```

it is received straight in the parameter definition of the function:

```
addAsFriend: async [root, args, { currentUser }] => {
```

The following query now returns the user's friends list:

```
query {
  me {
    username
    friends {
      name
      phone
    }
  }
}
```

The code of the backend can be found on [Github](#) branch *part8-5*.

Exercises 8.13.-8.16

The following exercises are quite likely to break your frontend. Do not worry about it yet; the frontend shall be fixed and expanded in the next chapter.

8.13: Database, part 1

Change the library application so that it saves the data to a database. You can find the *mongoose schema* for books and authors from [here](#).

Let's change the book graphql schema a little

```
type Book {
  title: String!
  published: Int!
  author: Author!  genres: [String!]!
  id: ID!
}
```

so that instead of just the author's name, the book object contains all the details of the author.

You can assume that the user will not try to add faulty books or authors, so you don't have to care about validation errors.

The following things do *not* have to work just yet:

- *allBooks* query with parameters
- *bookCount* field of an author object
- *author* field of a book
- *editAuthor* mutation

Note: despite the fact that author is now an *object* within a book, the schema for adding a book can remain same, only the *name* of the author is given as a parameter

```

type Mutation {
  addBook(
    title: String!
    author: String!      published: Int!
    genres: [String!]!
  ) : Book!
  editAuthor(name: String!, setBornTo: Int!): Author
}

```

8.14: Database, part 2

Complete the program so that all queries (to get *allBooks* working with the parameter *author* and *bookCount* field of an author object is not required) and mutations work.

Regarding the *genre* parameter of the all books query, the situation is a bit more challenging. The solution is simple, but finding it can be a headache. You might benefit from [this](#).

8.15 Database, part 3

Complete the program so that database validation errors (e.g. book title or author name being too short) are handled sensibly. This means that they cause `GraphQLError` with a suitable error message to be thrown.

8.16 user and logging in

Add user management to your application. Expand the schema like so:

```

type User {
  username: String!
  favoriteGenre: String!
  id: ID!
}

type Token {
  value: String!
}

type Query {
  // ...
  me: User
}

type Mutation {
  // ...
  createUser(
    username: String!
    favoriteGenre: String!
  ) : User
  login(
    username: String!
    password: String!
  )
}

```

```
    } : Token  
}
```

Create resolvers for query *me* and the new mutations *createUser* and *login*. Like in the course material, you can assume all users have the same hardcoded password.

Make the mutations *addBook* and *editAuthor* possible only if the request includes a valid token.

(Don't worry about fixing the frontend for the moment.)

8d: Login and updating the cache

The frontend of our application shows the phone directory just fine with the updated server. However, if we want to add new persons, we have to add login functionality to the frontend.

User login

Let's add the variable *token* to the application's state. When a user is logged in, it will contain a user token. If *token* is undefined, we render the *LoginForm* component responsible for user login. The component receives an error handler and the *setToken* function as parameters:

```
const App = () => [  
  const [token, setToken] = useState(null)  
  // ...  
  
  if (!token) {  
    return (  
      <div>  
        <Notify errorMessage={errorMessage} />  
        <h2>Login</h2>  
        <LoginForm  
          setToken={setToken}  
          setError={notify}  
        />  
      </div>  
    )  
  }  
  
  return [  
    // ...  
  ]  
]
```

Next, we define a mutation for logging in:

```
export const LOGIN = gql`  
  mutation login($username: String!, $password: String!) {  
    login(username: $username, password: $password) {  
      value  
    }  
  }`
```

The `LoginForm` component works pretty much just like all the other components doing mutations that we have previously created. Interesting lines in the code have been highlighted:

```
import { useState, useEffect } from 'react'
import { useMutation } from '@apollo/client'
import { LOGIN } from '../queries'

const LoginForm = ({ setError, setToken }) => {
  const [username, setUsername] = useState('')
  const [password, setPassword] = useState('')

  const [ login, result ] = useMutation(LOGIN, {      onError: (error) => [
    setError(error.graphQLErrors[0].message)
  ]
})

  useEffect(() => {      if ( result.data ) {          const token = result.data.login.value
          setToken(token)
          localStorage.setItem('phonenumbers-user-token', token)
        }, [result.data])
  const submit = async (event) => {
    event.preventDefault()

    login({ variables: { username, password } })
  }

  return (
    <div>
      <form onSubmit={submit}>
        <div>
          username <input
            value={username}
            onChange={({ target }) => setUsername(target.value)}
          />
        </div>
        <div>
          password <input
            type='password'
            value={password}
            onChange={({ target }) => setPassword(target.value)}
          />
        </div>
        <button type='submit'>login</button>
      </form>
    </div>
  )
}

export default LoginForm
```

We are using an effect hook to save the token's value to the state of the `App` component and the local storage after the server has responded to the mutation. Use of the effect hook is necessary to avoid an endless rendering loop.

Let's also add a button which enables a logged-in user to log out. The button's `onClick` handler sets the `token` state to null, removes the token from local storage and resets the cache of the Apollo client. The last step is **important**, because some queries might have fetched data to cache, which only logged-in users should have access to.

We can reset the cache using the `resetStore` method of an Apollo `client` object. The client can be accessed with the `useApolloClient` hook:

```
const App = () => [
  const [token, setToken] = useState(null)
  const [errorMessage, setErrorMessage] = useState(null)
  const result = useQuery(ALL_PERSONS)
  const client = useApolloClient()
  if (result.loading) {
    return <div>loading...</div>
  }

  const logout = () => {
    setToken(null)
    localStorage.clear()
    client.resetStore()
    if (!token) {
      return (
        <> <Notify errorMessage={errorMessage} />
        <LoginForm setToken={setToken} setError={notify} /> </> )
    }
    return (
      <>
        <Notify errorMessage={errorMessage} />
        <button onClick={logout}>logout</button>
        <Persons persons={result.data.allPersons} />
        <PersonForm setError={notify} />
        <PhoneForm setError={notify} />
      </>
    )
  }
]
```

Adding a token to a header

After the backend changes, creating new persons requires that a valid user token is sent with the request. In order to send the token, we have to change the way we define the `ApolloClient` object in `main.jsx` a little.

```
import { ApolloClient, InMemoryCache, ApolloProvider, createHttpLink } from '@apollo/client'
import { setContext } from '@apollo/client/link/context'
const authLink = setContext(({ headers }) => {
  const token = localStorage.getItem('phonenumbers-user-token')
  return {
    headers: { ...headers, authorization: token ? `Bearer ${token}` : null },
  }
})
const httpLink = createHttpLink({
  uri: 'http://localhost:4000',
})
```

```
const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: authLink.concat(httpLink))
```

The field `uri` that was previously used when creating the `client` object has been replaced by the field `link`, which defines in a more complicated case how Apollo is connected to the server. The server url is now wrapped using the function `createHttpLink` into a suitable `httpLink` object. The link is modified by the `context` defined by the `authLink` object so that a possible token in `localStorage` is [set to header `authorization`](#) for each request to the server.

Creating new persons and changing numbers works again. There is however one remaining problem. If we try to add a person without a phone number, it is not possible.

A screenshot of a web browser window. The address bar shows "localhost:3000". The main content area displays a red error message: "Person validation failed: phone: Path `phone` (`) is shorter than the minimum allowed length (5)., s is required." Below the error message, there is a heading "Persons" and a list of two persons: "Pekka Mikkola 045-2374321" and "Arto Mikkola 045-2374321". Each person entry has a "show address" button next to it.

Validation fails, because frontend sends an empty string as the value of `phone`.

Let's change the function creating new persons so that it sets `phone` to `undefined` if user has not given a value.

```
const PersonForm = ({ setError }) => [
  // ...
  const submit = async (event) => {
    event.preventDefault()
    createPerson([
      variables: [
        name, street, city,          phone: phone.length > 0 ? phone : undefined
      ]
    ])

    // ...
  }
]

// ...
]
```

Updating cache, revisited

We have to [update](#) the cache of the Apollo client on creating new persons. We can update it using the mutation's `refetchQueries` option to define that the `ALL_PERSONS` query is done again.

```
const PersonForm = ({ setError }) => [
  // ...
```

```

const [ createPerson ] = useMutation(CREATE_PERSON, {
  refetchQueries: [ {query: ALL_PERSONS} ],
  onError: (error) => {
    const messages = error.graphQLErrors.map(e => e.message).join('\n')
    setError(messages)
  }
})

```

This approach is pretty good, the drawback being that the query is always rerun with any updates.

It is possible to optimize the solution by handling updating the cache ourselves. This is done by defining a suitable `update` callback for the mutation, which Apollo runs after the mutation:

```

const PersonForm = ({ setError }) => {
  // ...

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    onError: (error) => {
      const messages = error.graphQLErrors.map(e => e.message).join('\n')
      setError(messages)
    },
    update: (cache, response) => {
      cache.updateQuery({ query: ALL_PERSONS },
        (allPersons) => {
          return [
            ...allPersons,
            response.data.addPerson
          ]
        }
      )
    }
  })
}

```

The callback function is given a reference to the cache and the data returned by the mutation as parameters. For example, in our case, this would be the created person.

Using the function `updateQuery` the code updates the query `ALLPERSONS` in the cache by adding the new person to the cached data.

In some situations, the only sensible way to keep the cache up to date is using the `update` callback.

When necessary, it is possible to disable cache for the whole application or `single queries` by setting the field managing the use of cache, `fetchPolicy` as `no-cache`.

Be diligent with the cache. Old data in the cache can cause hard-to-find bugs. As we know, keeping the cache up to date is very challenging. According to a coder proverb:

There are only two hard things in Computer Science: cache invalidation and naming things.
Read more [here](#).

The current code of the application can be found on [Github](#), branch `part8-5`.

Exercises 8.17.-8.22

8.17 Listing books

After the backend changes, the list of books does not work anymore. Fix it.

8.18 Log in

Adding new books and changing the birth year of an author do not work because they require a user to be logged in.

Implement login functionality and fix the mutations.

It is not necessary yet to handle validation errors.

You can decide how the login looks on the user interface. One possible solution is to make the login form into a separate view which can be accessed through a navigation menu:

	author	published
Clean Code	Robert Martin	2008
Agile software development	Robert Martin	2001
Refactoring, edition 2	Martin Fowler	2018
Refactoring to patterns	Joshua Kerievsky	2008
Practical Object-Oriented Design, An Agile Primer Using Ruby	Sandi Metz	2012

The login form:

name miuukkai
password ..
login

When a user is logged in, the navigation changes to show the functionalities which can only be done by a logged-in user:

	author	published
Clean Code	Robert Martin	2008
Agile software development	Robert Martin	2001
Refactoring, edition 2	Martin Fowler	2018
Refactoring to patterns	Joshua Kerievsky	2008
Practical Object-Oriented Design, An Agile Primer Using Ruby	Sandi Metz	2012

8.19 Books by genre, part 1

Complete your application to filter the book list by genre. Your solution might look something like this:

The screenshot shows a web browser window with the URL `localhost:3000`. The page has a header with links for `authors`, `books`, `add book`, and `logout`. Below the header, the word **books** is displayed in bold. Underneath it, the text **in genre patterns** is shown. A table lists two books with columns **author** and **published**:

	author	published
Agile software development	Robert Martin	2001
Refactoring to patterns	Joshua Kerievsky	2008

Below the table is a horizontal bar with several genre buttons: `refactoring`, `agile`, `patterns` (which is highlighted), `design`, `crime`, `classic`, and `all genres`.

In this exercise, the filtering can be done using just React.

8.20 Books by genre, part 2

Implement a view which shows all the books based on the logged-in user's favourite genre.

The screenshot shows a web browser window with the URL `localhost:3000`. The page has a header with links for `authors`, `books`, `add book`, `recommend` (which is highlighted), and `logout`. Below the header, the word **recommendations** is displayed in bold. Underneath it, the text **books in your favorite genre patterns** is shown. A table lists two books with columns **author** and **published**:

	author	published
Agile software development	Robert Martin	2001
Refactoring to patterns	Joshua Kerievsky	2008

8.21 books by genre with GraphQL

In the previous two exercises, the filtering could have been done using just React. To complete this exercise, you should redo the filtering the books based on a selected genre (that was done in exercise 8.19) using a GraphQL query to the server. If you already did so then you do not have to do anything.

This and the next exercises are quite **challenging** like it should be this late in the course. You might want to complete first the easier ones in the [next part](#).

8.22 Up-to-date cache and book recommendations

If you did the previous exercise, that is, fetch the books in a genre with GraphQL, ensure somehow that the books view is kept up to date. So when a new book is added, the books view is updated **at least** when a genre selection button is pressed.

When new genre selection is not done, the view does not have to be updated.

8e: Fragments and subscriptions

We are approaching the end of this part. Let's finish by having a look at a few more details about GraphQL.

Fragments

It is pretty common in GraphQL that multiple queries return similar results. For example, the query for the details of a person

```
query {
  findPerson(name: "Pekka Mikkola") {
    name
    phone
    address {
      street
      city
    }
  }
}
```

and the query for all persons

```
query {
  allPersons {
    name
    phone
    address {
      street
      city
    }
  }
}
```

both return persons. When choosing the fields to return, both queries have to define exactly the same fields.

These kinds of situations can be simplified with the use of [fragments](#). Let's declare a fragment for selecting all fields of a person:

```
fragment PersonDetails on Person {
  name
  phone
  address {
    street
    city
  }
}
```

With the fragment, we can do the queries in a compact form:

```
query {
  allPersons {
    ...PersonDetails
  }
}

query {
```

```
findPerson(name: "Pekka Mikkola") {  
  ...PersonDetails  
}
```

The fragments **are not** defined in the GraphQL schema, but in the client. The fragments must be declared when the client uses them for queries.

In principle, we could declare the fragment with each query like so:

```
export const FIND_PERSON = gql`  
query findPersonByName($nameToSearch: String!) {  
  findPerson(name: $nameToSearch) {  
    ...PersonDetails  
  }  
}  
  
fragment PersonDetails on Person {  
  name  
  phone  
  address {  
    street  
    city  
  }  
}
```

However, it is much better to declare the fragment once and save it to a variable.

```
const PERSON_DETAILS = gql`  
fragment PersonDetails on Person {  
  id  
  name  
  phone  
  address {  
    street  
    city  
  }  
}
```

Declared like this, the fragment can be placed to any query or mutation using a [dollar sign and curly braces](#):

```
export const FIND_PERSON = gql`  
query findPersonByName($nameToSearch: String!) {  
  findPerson(name: $nameToSearch) {  
    ...PersonDetails  
  }  
}  
${PERSON_DETAILS}  
`
```

Subscriptions

Along with query and mutation types, GraphQL offers a third operation type: [subscriptions](#). With subscriptions, clients can *subscribe* to updates about changes in the server.

Subscriptions are radically different from anything we have seen in this course so far. Until now, all interaction between browser and server was due to a React application in the browser making HTTP requests to the server. GraphQL queries and mutations have also been done this way. With subscriptions, the situation is the opposite. After an application has made a subscription, it starts to listen to the server. When changes occur on the server, it sends a notification to all of its *subscribers*.

Technically speaking, the HTTP protocol is not well-suited for communication from the server to the browser. So, under the hood, Apollo uses [WebSockets](#) for server subscriber communication.

Refactoring the backend

Since version 3.0 Apollo Server does not support subscriptions out of the box, we need to do some changes before we set up subscriptions. Let us also clean the app structure a bit.

Let's start by extracting the schema definition to the file *schema.js*

```
const typeDefs = `

type User {
  username: String!
  friends: [Person!]!
  id: ID!
}

type Token {
  value: String!
}

type Address {
  street: String!
  city: String!
}

type Person {
  name: String!
  phone: String
  address: Address!
  id: ID!
}

enum YesNo {
  YES
  NO
}

type Query {
  personCount: Int!
}
```

```

    allPersons(phone: YesNo): [Person!]!
    findPerson(name: String!): Person
    me: User
}

type Mutation {
  addPerson(
    name: String!
    phone: String
    street: String!
    city: String!
  ): Person
  editNumber(name: String!, phone: String!): Person
  createUser(username: String!): User
  login(username: String!, password: String!): Token
  addAsFriend(name: String!): User
}

module.exports = typeDefs

```

The resolvers definition is moved to the file *resolvers.js*

```

const { GraphQLError } = require('graphql')
const jwt = require('jsonwebtoken')
const Person = require('../models/person')
const User = require('../models/user')

const resolvers = [
  Query: {
    personCount: async () => Person.collection.countDocuments(),
    allPersons: async (root, args, context) => {
      if (!args.phone) {
        return Person.find({})
      }

      return Person.find({ phone: { $exists: args.phone === 'YES' } })
    },
    findPerson: async (root, args) => Person.findOne({ name: args.name }),
    me: (root, args, context) => {
      return context.currentUser
    }
  },
  Person: {
    address: ({ street, city }) => {
      return {
        street,
        city
      }
    }
  },
  Mutation: {
    addPerson: async (root, args, context) => {

```

```

const person = new Person({ ...args })
const currentUser = context.currentUser

if (!currentUser) {
  throw new GraphQLError('not authenticated', {
    extensions: {
      code: 'BAD_USER_INPUT',
    }
  })
}

try {
  await person.save()
  currentUser.friends = currentUser.friends.concat(person)
  await currentUser.save()
} catch (error) {
  throw new GraphQLError('Saving user failed', {
    extensions: {
      code: 'BAD_USER_INPUT',
      invalidArgs: args.name,
      error
    }
  })
}

return person
},  

editNumber: async (root, args) => [
  const person = await Person.findOne({ name: args.name })
  person.phone = args.phone

  try {
    await person.save()
  } catch (error) {
    throw new GraphQLError('Editing number failed', {
      extensions: {
        code: 'BAD_USER_INPUT',
        invalidArgs: args.name,
        error
      }
    })
  }

  return person
},  

createUser: async (root, args) => [
  const user = new User({ username: args.username })

  return user.save()
  .catch(error => [
    throw new GraphQLError('Creating the user failed', {
      extensions: {
        code: 'BAD_USER_INPUT'
      }
    })
  ])
]

```

```

        invalidArgs: args.username,
        error
    }
}
}

},
login: async [root, args] => {
    const user = await User.findOne({ username: args.username })

    if (!user || args.password !== 'secret') {
        throw new GraphQLError('wrong credentials', [
            extensions: { code: 'BAD_USER_INPUT' }
        ])
    }

    const userForToken = {
        username: user.username,
        id: user._id,
    }

    return { value: jwt.sign(userForToken, process.env.JWT_SECRET) }
},
addAsFriend: async [root, args, { currentUser }] => {
    const nonFriendAlready = [person] =>
        !currentUser.friends.map(f => f._id.toString()).includes(person._id.toString())

    if (!currentUser) {
        throw new GraphQLError('wrong credentials', [
            extensions: { code: 'BAD_USER_INPUT' }
        ])
    }

    const person = await Person.findOne({ name: args.name })
    if (nonFriendAlready(person)) {
        currentUser.friends = currentUser.friends.concat(person)
    }

    await currentUser.save()

    return currentUser
}
}

module.exports = resolvers

```

So far, we have started the application with the easy-to-use function `startStandaloneServer`, thanks to which the application has not had to be configured that much:

```
const { startStandaloneServer } = require('@apollo/server/standalone')
```

```

// ...

const server = new ApolloServer([
  typeDefs,
  resolvers,
])

startStandaloneServer(server, [
  listen: { port: 4000 },
  context: async ({ req, res }) => {
    // ...
  },
]).then(({ url }) => {
  console.log(`Server ready at ${url}`)
})

```

Unfortunately, `startStandaloneServer` does not allow adding subscriptions to the application, so let's switch to the more robust `expressMiddleware` function. As the name of the function already suggests, it is an Express middleware, which means that Express must also be configured for the application, with the GraphQL server acting as middleware.

Let us install Express

```
npm install express cors
```

and the file `index.js` changes to:

```

const { ApolloServer } = require('@apollo/server')
const { expressMiddleware } = require('@apollo/server/express4') const { ApolloServerPluginDrainHttpServer } = require('@apollo/server/plugin/drainHttpServer') const { makeExecutableSchema } = require('@graphql-tools/schema') const express = require('express') const cors = require('cors') const http = require('http') const jwt = require('jsonwebtoken')

const mongoose = require('mongoose')

const User = require('../models/user')

const typeDefs = require('./schema')
const resolvers = require('./resolvers')

const MONGODB_URI = 'mongodb+srv://databaseurlhere'

console.log('connecting to', MONGODB_URI)

mongoose
  .connect(MONGODB_URI)
  .then(() => {
    console.log('connected to MongoDB')
  })
  .catch((error) => {

```

```

        console.log(`error connection to MongoDB: ${ error.message }`)
    }

    // setup is now within a function
    const start = async () => {
        const app = express()
        const httpServer = http.createServer(app)

        const server = new ApolloServer({
            schema: makeExecutableSchema({ typeDefs, resolvers }),
            plugins: [ApolloServerPluginDrainHttpServer({ httpServer })],
        })

        await server.start()

        app.use([
            '/',
            cors(),
            express.json(),
            expressMiddleware(server, {
                context: async ({ req }) => {
                    const auth = req ? req.headers.authorization : null
                    if (auth && auth.startsWith('Bearer ')) {
                        const decodedToken = jwt.verify(auth.substring(7), process.env.JWT_SECRET)
                        const currentUser = await User.findById(decodedToken.id).populate(
                            'friends'
                        )
                        return { currentUser }
                    }
                },
            }),
        ])
    }

    const PORT = 4000

    httpServer.listen(PORT, () =>
        console.log(`Server is now running on http://localhost:${PORT}`)
    )
}

start()

```

There are several changes to the code. [ApolloServerPluginDrainHttpServer](#) has now been added to the configuration of the GraphQL server according to the recommendations of the documentation:

We highly recommend using this plugin to ensure your server shuts down gracefully.

The GraphQL server in the `server` variable is now connected to listen to the root of the server, i.e. to

the / route, using the `expressMiddleware` object. Information about the logged-in user is set in the context using the function we defined earlier. Since it is an Express server, the middlewares `express-json` and `cors` are also needed so that the data included in the requests is correctly parsed and so that CORS problems do not appear.

Since the GraphQL server must be started before the Express application can start listening to the specified port, the entire initialization has had to be placed in an *async function*, which allows waiting for the GraphQL server to start.

The backend code can be found on [GitHub](#), branch *part8-6*.

Subscriptions on the server

Let's implement subscriptions for subscribing for notifications about new persons added.

The schema changes like so:

```
type Subscription {
  personAdded: Person!
}
```

So when a new person is added, all of its details are sent to all subscribers.

First, we have to install two packages for adding subscriptions to GraphQL and a Node.js WebSocket library:

```
npm install graphql-ws ws @graphql-tools/schema
```

The file `index.js` is changed to:

```
const { WebSocketServer } = require('ws')
const { useServer } = require('graphql-ws/lib/use/ws')
// ...

const start = async () => {
  const app = express()
  const httpServer = http.createServer(app)

  const wsServer = new WebSocketServer({
    server: httpServer,
    path: '/ws'
  })

  const schema = makeExecutableSchema({ typeDefs, resolvers })
  const serverCleanup = useServer({ schema, wsServer })
  const server = new ApolloServer({
    schema,
    plugins: [
      ApolloServerPluginDrainHttpServer({ httpServer }),
      {
        async serverWillStart() {
          return {
            async drainServer() {
              await serverCleanup.dispose()
            }
          }
        }
      }
    ]
  })
}

start()
```

```

await server.start()

app.use([
  '/',
  cors(),
  express.json(),
  expressMiddleware(server, [
    context: async ({ req }) => {
      const auth = req ? req.headers.authorization : null
      if (auth && auth.startsWith('Bearer ')) {
        const decodedToken = jwt.verify(auth.substring(7), process.env.JWT_SECRET)
        const currentUser = await User.findById(decodedToken.id).populate('friends')
        return { currentUser }
      }
    },
  ]),
])

const PORT = 4000

httpServer.listen(PORT, () =>
  console.log(`Server is now running on http://localhost:${PORT}`)
)

start()

```

When queries and mutations are used, GraphQL uses the HTTP protocol in the communication. In case of subscriptions, the communication between client and server happens with [WebSockets](#).

The above code registers a `WebSocketServer` object to listen the `WebSocket` connections, besides the usual `HTTP` connections that the server listens to. The second part of the definition registers a function that closes the `WebSocket` connection on server shutdown. If you're interested in more details about configurations, Apollo's [documentation](#) explains in relative detail what each line of code does.

`WebSockets` are a perfect match for communication in the case of GraphQL subscriptions since when `WebSockets` are used, also the server can initiate the communication.

The subscription `personAdded` needs a resolver. The `addPerson` resolver also has to be modified so that it sends a notification to subscribers.

The required changes are as follows:

```

const { PubSub } = require('graphql-subscriptions')
const pubsub = new PubSub()
// ...

const resolvers = [
  // ...

```

```

Mutation: [
  addPerson: async (root, args, context) => {
    const person = new Person({ ...args })
    const currentUser = context.currentUser

    if (!currentUser) {
      throw new GraphQLError('not authenticated', {
        extensions: {
          code: 'BAD_USER_INPUT',
        }
      })
    }

    try {
      await person.save()
      currentUser.friends = currentUser.friends.concat(person)
      await currentUser.save()
    } catch (error) {
      throw new GraphQLError('Saving user failed', {
        extensions: {
          code: 'BAD_USER_INPUT',
          invalidArgs: args.name,
          error
        }
      })
    }
  }

  pubsub.publish('PERSON_ADDED', { personAdded: person })
  return person
},
Subscription: { personAdded: { subscribe: () => pubsub.asyncIterator('PERSON_ADDED') } }
]

```

The following library needs to be installed:

```
npm install graphql-subscriptions
```

With subscriptions, the communication happens using the [publish-subscribe](#) principle utilizing the object [PubSub](#).

There are only a few lines of code added, but quite a lot is happening under the hood. The resolver of the *personAdded* subscription registers and saves info about all the clients that do the subscription. The clients are saved to an "[iterator object](#)" called *PERSON_ADDED* thanks to the following code:

```

Subscription: {
  personAdded: {
    subscribe: () => pubsub.asyncIterator('PERSON_ADDED')
  }
}

```

The iterator name is an arbitrary string, but to follow the convention, it is the subscription name written in capital letters.

Adding a new person *publishes* a notification about the operation to all subscribers with PubSub's method *publish*:

```
pubsub.publish('PERSON_ADDED', { personAdded: person })
```

Execution of this line sends a WebSocket message about the added person to all the clients registered in the iterator *PERSON_ADDED*.

It's possible to test the subscriptions with the Apollo Explorer like this:

The screenshot shows the Apollo GraphQL Explorer interface. On the left, under 'Operation', there is a code editor with the following GraphQL subscription query:

```
1 subscription PersonAdded {
2   personAdded {
3     name
4     phone
5   }
6 }
```

A blue button labeled 'PersonAdded' is highlighted. Below the code editor, there are sections for 'Variables' and 'Headers'. Under 'Variables', there is a single variable '1'. Under 'Headers', there is a dropdown menu. To the right, under 'Response', there is a panel titled 'Subscriptions' with two log entries:

```
// Response received at 15:23:08
{
  "data": {
    "personAdded": {
      "name": "Pekka Mikkola",
      "phone": "040-1113333"
    }
  }
}

// Response received at 15:22:26
{
  "data": {
    "personAdded": {
      "name": "Leevi Hellas",
      "phone": null
    }
  }
}
```

When the blue button *PersonAdded* is pressed, Explorer starts to wait for a new person to be added. On addition (that you need to do from another browser window), the info of the added person appears on the right side of the Explorer.

If the subscription does not work, check that you have the correct connection settings:

The screenshot shows the Apollo GraphQL Studio interface. On the left, there's a sidebar with tabs like 'Documentation', 'Operation', and 'Fields'. The 'Operation' tab is active, displaying a GraphQL query. On the right, a modal window titled 'Connection settings' is open. It has sections for 'Auto Update' (ON) and 'Endpoint' (set to 'http://localhost:4000/'). Under 'Subscriptions', it shows 'ws://localhost:4000/' and 'graphql-ws' as the implementation. A note at the bottom cautions against using ws:// due to browser restrictions.

The backend code can be found on [GitHub](#), branch *part8-7*.

Implementing subscriptions involves a lot of configurations. You will be able to cope with the few exercises of this course without worrying much about the details. If you are planning to use subscriptions in an production use application, you should definitely read Apollo's [documentation on subscriptions](#) carefully.

Subscriptions on the client

In order to use subscriptions in our React application, we have to do some changes, especially to its [configuration](#). The configuration in *main.jsx* has to be modified like so:

```
import {
  ApolloClient, InMemoryCache, ApolloProvider, createHttpLink,
  split } from '@apollo/client'
import { setContext } from '@apollo/client/link/context'

import { getMainDefinition } from '@apollo/client/utilities' import { GraphQLWsLink }
  from '@apollo/client/link/subscriptions' import { createClient } from 'graphql-ws'

const authLink = setContext(({_, {headers}}) => {
  const token = localStorage.getItem('phonenumbers-user-token')
  return {
    headers: {
      ...headers,
      authorization: token ? `Bearer ${token}` : null,
    }
  }
})

const httpLink = createHttpLink({ uri: 'http://localhost:4000' })

const wsLink = new GraphQLWsLink( createClient({ url: 'ws://localhost:4000' }) )
const splitLink = split( ({query}) => {   const definition = getMainDefinition(query)
  if (definition.kind === 'Subscription') {
    return { link: wsLink, condition: true }
  } else {
    return { link: httpLink, condition: true }
  }
})
```

```

on(query)    return (      definition.kind === 'OperationDefinition' &&      def
initiation.operation === 'subscription'    )    ||, wsLink, authLink.concat(httpLink)
)

const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: splitLink
}

ReactDOM.createRoot(document.getElementById('root')).render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>
)

```

For this to work, we have to install a dependency:

```
npm install graphql-ws
```

The new configuration is due to the fact that the application must have an HTTP connection as well as a WebSocket connection to the GraphQL server.

```

const httpLink = createHttpLink({ uri: 'http://localhost:4000' })

const wsLink = new GraphQLWsLink(
  createClient({
    url: 'ws://localhost:4000',
  })
)

```

The subscriptions are done using the [useSubscription](#) hook function.

Let's make the following changes to the code. Add the code defining the subscription to the file `queries.js`:

```
export const PERSON_ADDED = gql`  subscription {    personAdded {      ...Person
Details    }  } ${PERSON_DETAILS}`
```

and do the subscription in the App component:

```

import { useQuery, useMutation, useSubscription } from '@apollo/client'

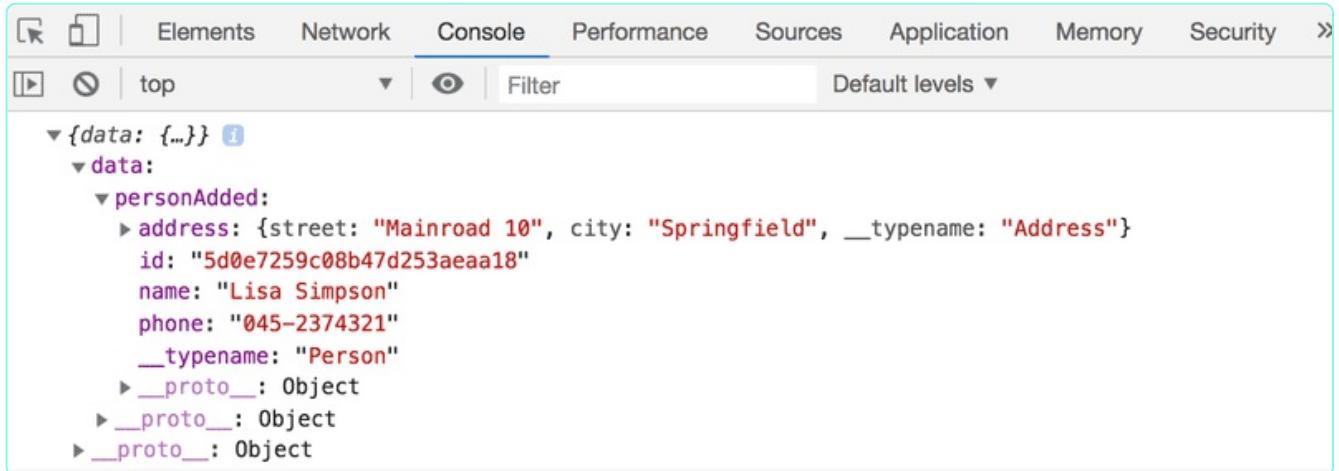
const App = () => [
  // ...

  useSubscription(PERSON_ADDED, [
    onData: ({ data }) => {
      console.log(data)
    }
  ])
]

```

```
// ...
```

When a new person is now added to the phonebook, no matter where it's done, the details of the new person are printed to the client's console:



When a new person is added, the server sends a notification to the client, and the callback function defined in the `onData` attribute is called and given the details of the new person as parameters.

Let's extend our solution so that when the details of a new person are received, the person is added to the Apollo cache, so it is rendered to the screen immediately.

```
const App = () => [
  // ...

  useSubscription(PERSON_ADDED, [
    onData: ({ data, client }) => {
      const addedPerson = data.data.personAdded
      notify(`#${addedPerson.name} added`)

      client.cache.updateQuery({ query: ALL_PERSONS }, ({ allPersons }) => {
        return {
          allPersons: allPersons.concat(addedPerson),
        }
      })
    }
  ])
  // ...
]
```

Our solution has a small problem: a person is added to the cache and also rendered twice since the component `PersonForm` is adding it to the cache as well.

Let us now fix the problem by ensuring that a person is not added twice in the cache:

```
// function that takes care of manipulating cache
export const updateCache = [cache, query, addedPerson] => [
  // helper that is used to eliminate saving same person twice
  const uniqByName = (a) => [
    let seen = new Set()
    return a.filter(item) => {
      let k = item.name
      return seen.has(k) ? false : seen.add(k)
    }
  ]
]
```

```

k}    })
  cache_updateQuery(query, ({ allPersons }) => {
    return { ...allPersons: un
iqByName(allPersons.concat(addedPerson)), ... })
}
const App = () => {
  const result = useQuery(ALL_PERSONS)
  const [errorMessage, setErrorMessage] = useState(null)
  const [token, setToken] = useState(null)
  const client = useApolloClient()

  useSubscription(PERSON_ADDED, [
    onData: ({ data, client }) => {
      const addedPerson = data.data.personAdded
      notify(`#${addedPerson.name} added`)
      updateCache(client.cache, { query: ALL_PERSONS }, addedPerson)
    },
  ])
}

// ...
}

```

The function `updateCache` can also be used in `PersonForm` for the cache update:

```

import { updateCache } from '../App'
const PersonForm = ({ setError }) => {
  // ...

  const [createPerson] = useMutation(CREATE_PERSON, [
    onError: (error) => {
      setError(error.graphQLErrors[0].message)
    },
    update: (cache, response) => {
      updateCache(cache, { query: ALL_PERSONS }, response.data.addPerson)
    }
  ])

  // ...
}

```

The final code of the client can be found on [GitHub](#), branch *part8-6*.

n+1 problem

First of all, you'll need to enable a debugging option via `mongoose` in your backend project directory, by adding a line of code as shown below:

```

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

```

```
mongoose.set('debug', true);
```

Let's add some things to the backend. Let's modify the schema so that a *Person* type has a *friendOf* field, which tells whose friends list the person is on.

```
type Person {  
    name: String!  
    phone: String  
    address: Address!  
    friendOf: [User!]!  
    id: ID!  
}
```

The application should support the following query:

```
query {  
    findPerson(name: "Leevi Hellas") {  
        friendOf {  
            username  
        }  
    }  
}
```

Because *friendOf* is not a field of *Person* objects on the database, we have to create a resolver for it, which can solve this issue. Let's first create a resolver that returns an empty list:

```
Person: {  
    address: (root) => {  
        return [  
            street: root.street,  
            city: root.city  
        ]  
    },  
    friendOf: (root) => [] // return list of users  
    return []  
},
```

The parameter *root* is the person object for which a friends list is being created, so we search from all *User* objects the ones which have *root._id* in their friends list:

```
Person: {  
    // ...  
    friendOf: async (root) => {  
        const friends = await User.find({  
            friends: {  
                $in: [root._id]  
            }  
        })  
  
        return friends  
    }  
},
```

},

Now the application works.

We can immediately do even more complicated queries. It is possible for example to find the friends of all users:

```
query {  
  allPersons {  
    name  
    friendOf {  
      username  
    }  
  }  
}
```

There is however one issue with our solution: it does an unreasonable amount of queries to the database. If we log every query to the database, just like this for example,

```
Query: {  
  allPersons: (root, args) => [  
    console.log('Person.find') if (!args.phone) { return Person.find({}) }  
    return Person.find({ phone: { $exists: args.phone === 'YES' } })  
  ]  
}  
  
// ..  
  
}  
  
// ..  
  
friendOf: async (root) => [  
  const friends = await User.find({ friends: { $in: [root._id] } })  
  console.log("User.find")  
  return friends  
],
```

and considering we have 5 persons saved, and we query *allPersons* without *phone* as argument, we see an absurd amount of queries like below.

```
Person.find  
User.find  
User.find  
User.find  
User.find  
User.find
```

So even though we primarily do one query for all persons, every person causes one more query in their resolver.

This is a manifestation of the famous [n+1 problem](#), which appears every once in a while in different

contexts, and sometimes sneaks up on developers without them noticing.

The right solution for the n+1 problem depends on the situation. Often, it requires using some kind of a join query instead of multiple separate queries.

In our situation, the easiest solution would be to save whose friends list they are on each *Person* object:

```
const schema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 5
  },
  phone: {
    type: String,
    minlength: 5
  },
  street: {
    type: String,
    required: true,
    minlength: 5
  },
  city: {
    type: String,
    required: true,
    minlength: 5
  },
  friendOf: [ { type: mongoose.Schema.Types.ObjectId, ref: 'User' } ]
})
```

Then we could do a "join query", or populate the *friendOf* fields of persons when we fetch the *Person* objects:

```
Query: {
  allPersons: (root, args) => [
    console.log('Person.find')
    if (!args.phone) {
      return Person.find({}).populate('friendOf')
    }

    return Person.find({ phone: { $exists: args.phone === 'YES' } })
      .populate('friendOf'),
    // ...
  ]
}
```

After the change, we would not need a separate resolver for the *friendOf* field.

The *allPersons* query *does not cause* an n+1 problem, if we only fetch the name and the phone number:

```
query: {
```

```
allPersons {  
    name  
    phone  
}  
}
```

If we modify `allPersons` to do a join query because it sometimes causes an $n+1$ problem, it becomes heavier when we don't need the information on related persons. By using the [fourth parameter](#) of resolver functions, we could optimize the query even further. The fourth parameter can be used to inspect the query itself, so we could do the join query only in cases with a predicted threat of $n+1$ problems. However, we should not jump into this level of optimization before we are sure it's worth it.

In the words of Donald Knuth:

*Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil.***

GraphQL Foundation's [DataLoader](#) library offers a good solution for the $n+1$ problem among other issues. More about using DataLoader with Apollo server [here](#) and [here](#).

Epilogue

The application we created in this part is not optimally structured: we did some cleanups but much would still need to be done. Examples for better structuring of GraphQL applications can be found on the internet. For example, for the server [here](#) and the client [here](#).

GraphQL is already a pretty old technology, having been used by Facebook since 2012, so we can see it as "battle-tested" already. Since Facebook published GraphQL in 2015, it has slowly gotten more and more attention, and might in the near future threaten the dominance of REST. The death of REST has also already been [predicted](#). Even though that will not happen quite yet, GraphQL is absolutely worth [learning](#).

Exercises 8.23.-8.26

8.23: Subscriptions - server

Do a backend implementation for subscription `bookAdded`, which returns the details of all new books to its subscribers.

8.24: Subscriptions - client, part 1

Start using subscriptions in the client, and subscribe to `bookAdded`. When new books are added, notify the user. Any method works. For example, you can use the `window.alert` function.

8.25: Subscriptions - client, part 2

Keep the application's book view updated when the server notifies about new books (you can ignore the author view!). You can test your implementation by opening the app in two browser tabs and adding a new book in one tab. Adding the new book should update the view in both tabs.

8.26: n+1

Solve the n+1 problem of the following query using any method you like.

```
query {
  allAuthors {
    name
    bookCount
  }
}
```

Submitting exercises and getting the credits

Exercises of this part are submitted via [the submissions system](#) just like in the previous parts, but unlike previous parts, the submission goes to different "course instance". Remember that you have to finish at least 22 exercises to pass this part!

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:

My submissions

part	exercises	hours	github	comment	solutio
1	22	29	https://github.com/Kaltsoon/fs-cicd		show
total	22	29			

credits 1 based on exercises

Certificate

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

Note that you need a registration to the corresponding course part for getting the credits registered, see [here](#) for more information.

You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the certificate's language.

9: TypeScript

9a: Background and introduction

TypeScript is a programming language designed for large-scale JavaScript development created by Microsoft. For example, Microsoft's *Azure Management Portal* (1,2 million lines of code) and *Visual Studio Code*.

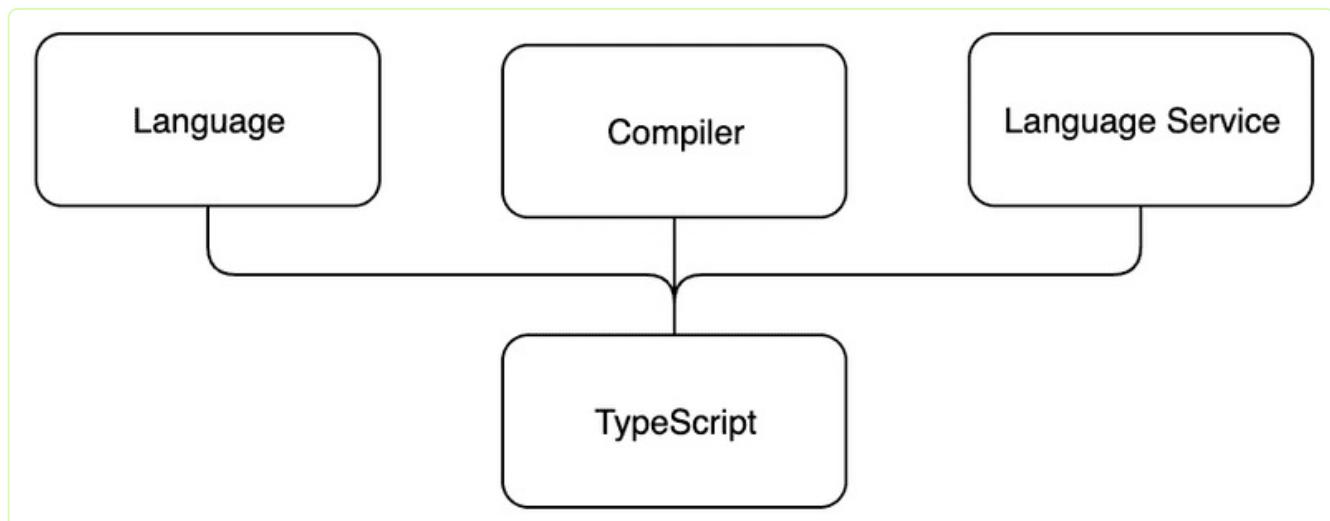
Studio Code (300 000 lines of code) have both been written in TypeScript. To support building large-scale JavaScript applications, TypeScript offers features such as better development-time tooling, static code analysis, compile-time type checking and code-level documentation.

Main principle

TypeScript is a typed superset of JavaScript, and eventually, it's compiled into plain JavaScript code. The programmer is even able to decide the version of the generated code, as long as it's ECMAScript 3 or newer. TypeScript being a superset of JavaScript means that it includes all the features of JavaScript and its additional features as well. In other words, all existing JavaScript code is valid TypeScript.

TypeScript consists of three separate, but mutually fulfilling parts:

- The language
- The compiler
- The language service



The *language* consists of syntax, keywords and type annotations. The syntax is similar to but not the same as JavaScript syntax. From the three parts of TypeScript, programmers have the most direct contact with the language.

The *compiler* is responsible for type information erasure (i.e. removing the typing information) and for code transformations. The code transformations enable TypeScript code to be transpiled into executable JavaScript. Everything related to the types is removed at compile-time, so TypeScript isn't genuine statically-typed code.

Traditionally, *compiling* means that code is transformed from a human-readable format to a machine-readable format. In TypeScript, human-readable source code is transformed into another human-readable source code, so the correct term would be *transpiling*. However, compiling has been the most commonly-used term in this context, so we will continue to use it.

The compiler also performs a static code analysis. It can emit warnings or errors if it finds a reason to do so, and it can be set to perform additional tasks such as combining the generated code into a single file.

The *language service* collects type information from the source code. Development tools can use the type information for providing intellisense, type hints and possible refactoring alternatives.

TypeScript key language features

In this section, we will describe some of the key features of the TypeScript language. The intent is to provide you with a basic understanding of TypeScript's key features to help you understand more of what is to come during this course.

Type annotations

Type annotations in TypeScript are a lightweight way to record the intended *contract* of a function or a variable. In the example below, we have defined a *birthdayGreeter* function that accepts two arguments: one of type string and one of type number. The function will return a string.

```
const birthdayGreeter = (name: string, age: number): string => {
    return `Happy birthday ${name}, you are now ${age} years old!`;
}

const birthdayHero = "Jane User";
const age = 22;

console.log(birthdayGreeter(birthdayHero, age));
```

Structural typing

TypeScript is a structurally-typed language. In structural typing, two elements are considered to be compatible with one another if, for each feature within the type of the first element, a corresponding and identical feature exists within the type of the second element. Two types are considered to be identical if they are compatible with each other.

Type inference

The TypeScript compiler can attempt to infer the type information if no type has been specified. Variables' type can be inferred based on their assigned value and their usage. The type inference takes place when initializing variables and members, setting parameter default values, and determining function return types.

For example, consider the function *add*:

```
const add = (a: number, b: number) => {
    /* The return value is used to determine
       the return type of the function */
    return a + b;
}
```

The type of the function's return value is inferred by retracing the code back to the return expression. The return expression performs an addition of the parameters *a* and *b*. We can see that *a* and *b* are numbers based on their types. Thus, we can infer the return value to be of type *number*.

Type erasure

TypeScript removes all type system constructs during compilation.

Input:

```
let x: SomeType;
```

Output:

```
let x;
```

This means that no type information remains at runtime; nothing says that some variable x was declared as being of type *SomeType*.

The lack of runtime type information can be surprising for programmers who are used to extensively using reflection or other metadata systems.

Why should one use TypeScript?

On different forums, you may stumble upon a lot of different arguments either for or against TypeScript. The truth is probably as vague as: it depends on your needs and the use of the functions that TypeScript offers. Anyway, here are some of our reasons behind why we think that the use of TypeScript may have some advantages.

First of all, TypeScript offers *type checking and static code analysis*. We can require values to be of a certain type, and have the compiler warn about using them incorrectly. This can reduce runtime errors, and you might even be able to reduce the number of required unit tests in a project, at least concerning pure-type tests. The static code analysis doesn't only warn about wrongful type usage, but also other mistakes such as misspelling a variable or function name or trying to use a variable beyond its scope.

The second advantage of TypeScript is that the type annotations in the code can function as a kind of *code-level documentation*. It's easy to check from a function signature what kind of arguments the function can consume and what type of data it will return. This form of type annotation-bound documentation will always be up to date and it makes it easier for new programmers to start working on an existing project. It is also helpful when returning to work on an old project.

Types can be reused all around the code base, and a change to a type definition will automatically be reflected everywhere the type is used. One might argue that you can achieve similar code-level documentation with e.g. [JSDoc](#), but it is not connected to the code as tightly as TypeScript's types, and may thus get out of sync more easily, and is also more verbose.

The third advantage of TypeScript is that IDEs can provide more *specific and smarter IntelliSense* when they know exactly what types of data you are processing.

All of these features are extremely helpful when you need to refactor your code. The static code analysis warns you about any errors in your code, and IntelliSense can give you hints about available properties and even possible refactoring options. The code-level documentation helps you understand the existing code. With the help of TypeScript, it is also very easy to start using the newest JavaScript language features at an early stage just by altering its configuration.

What does TypeScript not fix?

As mentioned above, TypeScript's type annotations and type checking exist only at compile time and no longer at runtime. Even if the compiler does not throw any errors, runtime errors are still possible.

These runtime errors are especially common when handling external input, such as data received from a network request.

Lastly, below, we list some issues many have with TypeScript, which might be good to be aware of:

Incomplete, invalid or missing types in external libraries

When using external libraries, you may find that some have either missing or in some way invalid type declarations. Most often, this is due to the library not being written in TypeScript, and the person adding the type declarations manually not doing such a good job with it. In these cases, you might need to define the type declarations yourself. However, there is a good chance someone has already added typings for the package you are using. Always check the DefinitelyTyped [GitHub page](#) first. It is probably the most popular source for type declaration files. Otherwise, you might want to start by getting acquainted with TypeScript's [documentation](#) regarding type declarations.

Sometimes, type inference needs assistance

The type inference in TypeScript is pretty good but not quite perfect. Sometimes, you may feel like you have declared your types perfectly, but the compiler still tells you that the property does not exist or that this kind of usage is not allowed. In these cases, you might need to help the compiler out by doing something like an "extra" type check. One should be careful with type casting (that is quite often called type assertion) or type guards: when using those, you are giving your word to the compiler that the value *is* of the type that you declare. You might want to check out TypeScript's documentation regarding [type assertions](#) and [type guarding/narrowing](#).

Mysterious type errors

The errors given by the type system may sometimes be quite hard to understand, especially if you use complex types. As a rule of thumb, the TypeScript error messages have the most useful information at the end of the message. When running into long confusing messages, start reading them from the end.

9b: First steps with TypeScript

After the brief introduction to the main principles of TypeScript, we are now ready to start our journey toward becoming FullStack TypeScript developers. Rather than giving you a thorough introduction to all aspects of TypeScript, we will focus in this part on the most common issues that arise when developing Express backends or React frontends with TypeScript. In addition to language features, we will also have a strong emphasis on tooling.

Setting things up

Install TypeScript support to your editor of choice. [Visual Studio Code](#) works natively with TypeScript.

As mentioned earlier, TypeScript code is not executable by itself. It has to be first compiled into executable JavaScript. When TypeScript is compiled into JavaScript, the code becomes subject to type erasure. This means that type annotations, interfaces, type aliases, and other type system constructs are removed and the result is pure ready-to-run JavaScript.

In a production environment, the need for compilation often means that you have to set up a "build step." During the build step, all TypeScript code is compiled into JavaScript in a separate folder, and the production environment then runs the code from that folder. In a development environment, it is

often easier to make use of real-time compilation and auto-reloading so one can see the resulting changes more quickly.

Let's start writing our first TypeScript app. To keep things simple, let's start by using the npm package `ts-node`. It compiles and executes the specified TypeScript file immediately so that there is no need for a separate compilation step.

You can install both `ts-node` and the official `typescript` package globally by running:

```
npm install --location=global ts-node typescript
```

If you can't or don't want to install global packages, you can create an npm project which has the required dependencies and run your scripts in it. We will also take this approach.

As we recall from [part 3](#), an npm project is set by running the command `npm init` in an empty directory. Then we can install the dependencies by running

```
npm install --save-dev ts-node typescript
```

and setting up `scripts` within the `package.json`:

```
{  
  // ...  
  "scripts": {  
    "ts-node": "ts-node"  },  
  // ...  
}
```

You can now use `ts-node` within this directory by running `npm run ts-node`. Note that if you are using `ts-node` through `package.json`, command-line arguments that include short or long-form options for the `npm run script` need to be prefixed with `--`. So if you want to run `file.ts` with `ts-node` and options `-s` and `--someoption`, the whole command is:

```
npm run ts-node file.ts -- -s --someoption
```

It is worth mentioning that TypeScript also provides an online playground, where you can quickly try out TypeScript code and instantly see the resulting JavaScript and possible compilation errors. You can access TypeScript's official playground [here](#).

NB: The playground might contain different `tsconfig` rules (which will be introduced later) than your local environment, which is why you might see different warnings there compared to your local environment. The playground's `tsconfig` is modifiable through the config dropdown menu.

A note about the coding style

JavaScript is a quite relaxed language in itself, and things can often be done in multiple different ways. For example, we have named vs anonymous functions, using `const` and `let` or `var`, and the optional use of `semicolons`. This part of the course differs from the rest by using semicolons. It is not a TypeScript-specific pattern but a general coding style decision taken when creating any kind of JavaScript project. Whether to use them or not is usually in the hands of the programmer, but since it

is expected to adapt one's coding habits to the existing codebase, you are expected to use semicolons and adjust to the coding style in the exercises for this part. This part has some other coding style differences compared to the rest of the course as well, e.g. in the directory naming conventions.

Let us add a configuration file `tsconfig.json` to the project with the following content:

```
[{"compilerOptions": {"noImplicitAny": false}}
```

The `tsconfig.json` file is used to define how the TypeScript compiler should interpret the code, how strictly the compiler should work, which files to watch or ignore, and [much more](#). For now, we will only use the compiler option `noImplicitAny`, which does not require having types for all variables used.

Let's start by creating a simple Multiplier. It looks exactly as it would in JavaScript.

```
const multiplicator = (a, b, printText) => {
  console.log(printText, a * b);
}

multiplicator(2, 4, 'Multiplied numbers 2 and 4, the result is:');
```

As you can see, this is still ordinary basic JavaScript with no additional TS features. It compiles and runs nicely with `npm run ts-node -- multiplier.ts`, as it would with Node.

But what happens if we end up passing the wrong *types* of arguments to the multiplicator function?

Let's try it out!

```
const multiplicator = (a, b, printText) => {
  console.log(printText, a * b);
}

multiplicator('how about a string?', 4, 'Multiplied a string and 4, the result is:');
```

Now when we run the code, the output is: *Multiplied a string and 4, the result is: NaN*.

Wouldn't it be nice if the language itself could prevent us from ending up in situations like this? This is where we see the first benefits of TypeScript. Let's add types to the parameters and see where it takes us.

TypeScript natively supports multiple types including `number`, `string` and `Array`. See the comprehensive list [here](#). More complex custom types can also be created.

The first two parameters of our function are of type `number` and the last one is of type `string`, both types are [primitives](#):

```

const multiplicator = (a: number, b: number, printText: string) => {
  console.log(printText, a * b);
}

multiplicator('how about a string?', 4, 'Multiplied a string and 4, the result is: ');

```

Now the code is no longer valid JavaScript but in fact TypeScript. When we try to run the code, we notice that it does not compile:

```

$ partb ts-node multiplier.ts
/Users/mluukkai/.nvm/versions/node/v10.18.0/lib/node_modules/ts-node/src/index.ts:427
    return new TSError(diagnosticText, diagnosticCodes)
    ^
TSError: ✘ Unable to compile TypeScript:
multiplier.ts:5:15 - error TS2345: Argument of type '"how about a string?"' is not assignable to parameter of type 'number'

5   multiplicator('how about a string?', 4, 'Multiplied a string and 4, the result is:');

```

One of the best things about TypeScript's editor support is that you don't necessarily need to even run the code to see the issues. VSCode is so efficient, that it informs you immediately when you are trying to use an incorrect type:

```

const multiplicator = (a: number, b: number, printText: string) => {
  console.log(`Multiplying ${a} and ${b} with ${printText}`);
}
multiplicator('can we multiply by a string?', 4, 'Multiplied a string and four, the result is:');

```

Creating your first own types

Let's expand our multiplicator into a slightly more versatile calculator that also supports addition and division. The calculator should accept three arguments: two numbers and the operation, either *multiply*, *add* or *divide*, which tells it what to do with the numbers.

In JavaScript, the code would require additional validation to make sure the last argument is indeed a string. TypeScript offers a way to define specific types for inputs, which describe exactly what type of input is acceptable. On top of that, TypeScript can also show the info on the accepted values already at the editor level.

We can create a *type* using the TypeScript native keyword *type*. Let's describe our type *Operation*:

```
type Operation = 'multiply' | 'add' | 'divide';
```

Now the *Operation* type accepts only three kinds of values; exactly the three strings we wanted. Using the OR operator / we can define a variable to accept multiple values by creating a [union type](#). In this case, we used exact strings (that, in technical terms, are called [string literal types](#)) but with unions, you could also make the compiler accept for example both string and number: *string* / *number*.

The *type* keyword defines a new name for a type: [a type alias](#). Since the defined type is a union of three possible values, it is handy to give it an alias that has a representative name.

Let's look at our calculator now:

```
type Operation = 'multiply' | 'add' | 'divide';

const calculator = (a: number, b: number, op: Operation) => {
  if (op === 'multiply') {
    return a * b;
  } else if (op === 'add') {
    return a + b;
  } else if (op === 'divide') {
    if (b === 0) return 'can\'t divide by 0!';
    return a / b;
  }
}
```

Now, when we hover on top of the *Operation* type in the calculator function, we can immediately see suggestions on what to do with it:

```
type Operation = "multiply" | "add" | "divide"
number, b: number, operation: Operation) : Result => {
```

And if we try to use a value that is not within the *Operation* type, we get the familiar red warning signal and extra info from our editor:

```
s index.ts > ...
1  type Operation = 'multiply' | 'add' | 'divide';
2
3  const calculator = (a: number, b: number, op: Operation) => {
4    if (op === 'multiply') {
5      return a * b;
6    } else if (op === 'add') {
7      return a + b;
8    } else if (op === 'divide') {
9      if (b === 0) return 'can\'t divide by 0!';
10     return a / b;
11   }
12 }
13 calculator(1, 2, 'yolo')
14
```

Argument of type '"yolo"' is not assignable to parameter of type 'Operation'. ts(2345)

View Problem No quick fixes available

This is already pretty nice, but one thing we haven't touched yet is typing the return value of a function. Usually, you want to know what a function returns, and it would be nice to have a guarantee that it returns what it says it does. Let's add a return value *number* to the calculator function:

```
type Operation = 'multiply' | 'add' | 'divide';

const calculator = (a: number, b: number, op: Operation): number => {
  if (op === 'multiply') {
    return a * b;
  } else if (op === 'add') {
    return a + b;
  } else if (op === 'divide') {
```

```

    if (b === 0) return 'this cannot be done';
    return a / b;
}

```

The compiler complains straight away because, in one case, the function returns a string. There are a couple of ways to fix this:

We could extend the return type to allow string values, like so:

```

const calculator = (a: number, b: number, op: Operation): number | string => {
    // ...
}

```

Or we could create a return type, which includes both possible types, much like our `Operation` type:

```

type Result = string | number;

const calculator = (a: number, b: number, op: Operation): Result => {
    // ...
}

```

But now the question is if it's *really* okay for the function to return a string?

When your code can end up in a situation where something is divided by 0, something has probably gone terribly wrong and an error should be thrown and handled where the function was called. When you are deciding to return values you weren't originally expecting, the warnings you see from TypeScript prevent you from making rushed decisions and help you to keep your code working as expected.

One more thing to consider is, that even though we have defined types for our parameters, the generated JavaScript used at runtime does not contain the type checks. So if, for example, the `Operation` parameter's value comes from an external interface, there is no definite guarantee that it will be one of the allowed values. Therefore, it's still better to include error handling and be prepared for the unexpected to happen. In this case, when there are multiple possible accepted values and all unexpected ones should result in an error, the `switch...case` statement suits better than `if...else` in our code.

The code of our calculator should look something like this:

```

type Operation = 'multiply' | 'add' | 'divide';

const calculator = (a: number, b: number, op: Operation): number => {
    switch(op) {
        case 'multiply':
            return a * b;
        case 'divide':
            if (b === 0) throw new Error('Can\'t divide by 0!');           return a / b;
        case 'add':
            return a + b;
        default:
    }
}

```

```
    throw new Error('Operation is not multiply, add or divide!'); }
}

try {
  console.log(calculator[1, 5, 'divide']);
} catch (error: unknown) {
  let errorMessage = 'Something went wrong: ';
  if (error instanceof Error) {
    errorMessage += error.message;
  }
  console.log(errorMessage);
}
```

Type narrowing

The default type of the catch block parameter `error` is `unknown`. The `unknown` is a kind of top type that was introduced in TypeScript version 3 to be the type-safe counterpart of `any`. Anything is assignable to `unknown`, but `unknown` isn't assignable to anything but itself and `any` without a type assertion or a control flow-based narrowing. Likewise, no operations are permitted on an `unknown` without first asserting or narrowing it to a more specific type.

Both the possible causes of exception (wrong operator or division by zero) will throw an `Error` object with an error message, that our program prints to the user.

If our code would be JavaScript, we could print the error message by just referring to the field `message` of the object `error` as follows:

```
try {
  console.log(calculator[1, 5, 'divide']);
} catch (error) {
  console.log('Something went wrong: ' + error.message);}
```

Since the default type of the `error` object in TypeScript is `unknown`, we have to `narrow` the type to access the field:

```
try {
  console.log(calculator[1, 5, 'divide']);
} catch (error: unknown) {
  let errorMessage = 'Something went wrong: ';
  // here we can not use error.message
  if (error instanceof Error) { // the type is narrowed and we can refer to error.message
    errorMessage += error.message;
  }
  // here we can not use error.message

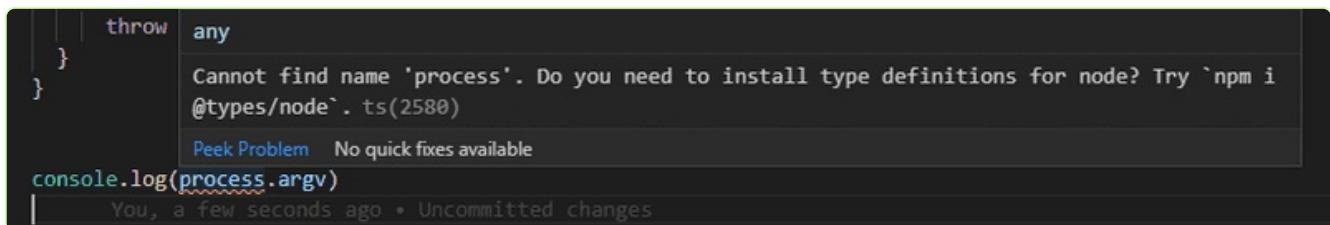
  console.log(errorMessage);
}
```

Here the narrowing was done with the `instanceof` type guard, that is just one of the many ways to narrow a type. We shall see many others later in this part.

Accessing command line arguments

The programs we have written are alright, but it sure would be better if we could use command-line arguments instead of always having to change the code to calculate different things.

Let's try it out, as we would in a regular Node application, by accessing `process.argv`. If you are using a recent npm-version (7.0 or later), there are no problems, but with an older setup something is not right:



A screenshot of a code editor showing a TypeScript error. The code contains a call to `console.log(process.argv)`. A tooltip appears over the word `process`, stating: "Cannot find name 'process'. Do you need to install type definitions for node? Try `npm i @types/node`." The status bar at the bottom shows "You, a few seconds ago • Uncommitted changes".

So what is the problem with older setups?

@types/{npm_package}

Let's return to the basic idea of TypeScript. TypeScript expects all globally-used code to be typed, as it does for your code when your project has a reasonable configuration. The TypeScript library itself contains only typings for the code of the TypeScript package. It is possible to write your own typings for a library, but that is rarely needed - since the TypeScript community has done it for us!

As with npm, the TypeScript world also celebrates open-source code. The community is active and continuously reacting to updates and changes in commonly used npm packages. You can almost always find the typings for npm packages, so you don't have to create types for all of your thousands of dependencies alone.

Usually, types for existing packages can be found from the `@types` organization within npm, and you can add the relevant types to your project by installing an npm package with the name of your package with a `@types/` prefix. For example:

```
npm install --save-dev @types/react @types/express @types/lodash @types/jest @types/mongoose
```

and so on and so on. The `@types/` are maintained by [Definitely typed](#), a community project to maintain types of everything in one place.

Sometimes, an npm package can also include its types within the code and, in that case, installing the corresponding `@types/` is not necessary.

NB: Since the typings are only used before compilation, the typings are not needed in the production build and they should *always* be in the `devDependencies` of the package.json.

Since the global variable `process` is defined by Node itself, we get its typings from the package `@types/node`.

Since version 10.0 *ts-node* has defined *@types/node* as a [peer dependency](#). If the version of npm is at least 7.0, the peer dependencies of a project are automatically installed by npm. If you have an older npm, the peer dependency must be installed explicitly:

```
npm install --save-dev @types/node
```

When the package *@types/node* is installed, the compiler does not complain about the variable *process*. Note that there is no need to require the types to the code, the installation of the package is enough!

Improving the project

Next, let's add npm scripts to run our two programs *multiplier* and *calculator*:

```
{
  "name": "fs-open",
  "version": "1.0.0",
  "description": "",
  "main": "index.ts",
  "scripts": {
    "ts-node": "ts-node",
    "multiply": "ts-node multiplier.ts",      "calculate": "ts-node calculator.ts"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "ts-node": "^10.5.0",
    "typescript": "^4.5.5"
  }
}
```

We can get the multiplier to work with command-line parameters with the following changes:

```
const multiplicator = (a: number, b: number, printText: string) => {
  console.log(printText, a * b)
}

const a: number = Number(process.argv[2])
const b: number = Number(process.argv[3])
multiplicator(a, b, `Multiplied ${a} and ${b}, the result is:`)
```

And we can run it with:

```
npm run multiply 5 2
```

If the program is run with parameters that are not of the right type, e.g.

```
npm run multiply 5 lol
```

it "works" but gives us the answer:

```
Multiplied 5 and NaN, the result is: NaN
```

The reason for this is, that `Number('lol')` returns `NaN`, which is actually of type `number`, so TypeScript has no power to rescue us from this kind of situation.

To prevent this kind of behavior, we have to validate the data given to us from the command line.

The improved version of the multiplicator looks like this:

```
interface MultiplyValues {
  value1: number;
  value2: number;
}

const parseArguments = (args: string[]): MultiplyValues => {
  if (args.length < 4) throw new Error('Not enough arguments');
  if (args.length > 4) throw new Error('Too many arguments');

  if (!isNaN(Number(args[2])) && !isNaN(Number(args[3]))) {
    return [
      value1: Number(args[2]),
      value2: Number(args[3])
    ];
  } else {
    throw new Error('Provided values were not numbers!');
  }
}

const multiplicator = (a: number, b: number, printText: string) => {
  console.log(printText, a * b);
}

try {
  const [value1, value2] = parseArguments(process.argv);
  multiplicator(value1, value2, `Multiplied ${value1} and ${value2}, the result is:`);
} catch (error: unknown) {
  let errorMessage = 'Something bad happened.';
  if (error instanceof Error) {
    errorMessage += ` Error: ${error.message}`;
  }
  console.log(errorMessage);
}
```

When we now run the program:

```
npm run multiply 1 lol
```

we get a proper error message:

```
Something bad happened. Error: Provided values were not numbers!
```

There is quite a lot going on in the code. The most important addition is the function *parseArguments* which ensures that the parameters given to *multiplicator* are of the right type. If not, an exception is thrown with a descriptive error message.

The definition of the function has a couple of interesting things:

```
const parseArguments = [args: string[]]: MultiplyValues => {
  // ...
}
```

Firstly, the parameter *args* is an [array](#) of strings.

The return value of the function has the type *MultiplyValues*, which is defined as follows:

```
interface MultiplyValues {
  value1: number;
  value2: number;
}
```

The definition utilizes TypeScript's [Interface](#) keyword, which is one way to define the "shape" an object should have. In our case, it is quite obvious that the return value should be an object with the two properties *value1* and *value2*, which should both be of type number.

The alternative array syntax

Note that there is also an alternative syntax for [arrays](#) in TypeScript. Instead of writing

```
let values: number[];
```

we could use the "generics syntax" and write

```
let values: Array<number>;
```

In this course we shall mostly be following the convention enforced by the Eslint rule [array-simple](#) that suggests writing the simple arrays with the [] syntax and using the <> syntax for the more complex ones, see [here](#) for examples.

Exercises 9.1-9.3

setup

Exercises 9.1-9.7. will all be made in the same node project. Create the project in an empty directory with *npm init* and install the *ts-node* and *typescript* packages. Also, create the file *tsconfig.json* in the directory with the following content:

```
{
```

```
"compilerOptions": {  
    "noImplicitAny": true,  
}  
}
```

The compiler option `noImplicitAny` makes it mandatory to have types for all variables used. This option is currently a default, but it lets us define it explicitly.

9.1 Body mass index

Create the code of this exercise in the file `bmiCalculator.ts`.

Write a function `calculateBmi` that calculates a **BMI** based on a given height (in centimeters) and weight (in kilograms) and then returns a message that suits the results.

Call the function in the same file with hard-coded parameters and print out the result. The code

```
console.log(calculateBmi(180, 74))
```

should print the following message:

```
Normal (healthy weight)
```

Create an npm script for running the program with the command `npm run calculateBmi`.

9.2 Exercise calculator

Create the code of this exercise in file `exerciseCalculator.ts`.

Write a function `calculateExercises` that calculates the average time of *daily exercise hours* and compares it to the *target amount* of daily hours and returns an object that includes the following values:

- the number of days
- the number of training days
- the original target value
- the calculated average time
- boolean value describing if the target was reached
- a rating between the numbers 1-3 that tells how well the hours are met. You can decide on the metric on your own.
- a text value explaining the rating, you can come up with the explanations

The daily exercise hours are given to the function as an `array` that contains the number of exercise hours for each day in the training period. Eg. a week with 3 hours of training on Monday, none on Tuesday, 2 hours on Wednesday, 4.5 hours on Thursday and so on would be represented by the following array:

```
[3, 0, 2, 4.5, 0, 3, 1]
```

For the Result object, you should create an [interface](#).

If you call the function with parameters `[3, 0, 2, 4.5, 0, 3, 1]` and 2, it should return:

```
{ periodLength: 7,
  trainingDays: 5,
  success: false,
  rating: 2,
  ratingDescription: 'not too bad but could be better',
  target: 2,
  average: 1.9285714285714286
```

Create an npm script, `npm run calculateExercises`, to call the function with hard-coded values.

9.3 Command line

Change the previous exercises so that you can give the parameters of `bmiCalculator` and `exerciseCalculator` as command-line arguments.

Your program could work eg. as follows:

```
$ npm run calculateBmi 180 91
Overweight
```

and:

```
$ npm run calculateExercises 2 1 0 2 4.5 0 3 1 0 4
{ periodLength: 9,
  trainingDays: 6,
  success: false,
  rating: 2,
  ratingDescription: 'not too bad but could be better',
  target: 2,
  average: 1.722222222222223
}
```

In the example, the *first argument* is the target value.

Handle exceptions and errors appropriately. The `exerciseCalculator` should accept inputs of varied lengths. Determine by yourself how you manage to collect all needed input.

A couple of things to notice:

If you define helper functions in other modules, you should use the [JavaScript module system](#), that is, the one we have used with React where importing is done with

```
import { isNotNumber } from "./utils"
```

and exporting

```

export const isNotNumber = (argument: any): boolean =>
  isNaN(Number(argument));

export default "this is the default..."

```

Another note: somehow surprisingly TypeScript does not allow to define the same variable in many files at a "block-scope", that is, outside functions (or classes):

The screenshot shows the VS Code interface with a tree view on the left and a code editor on the right. The tree view shows a folder named 'BMI' containing several files: node_modules, bmiCalculator.ts, exerciseCalculator.ts, file.ts, index.ts, package-lock.json, package.json, tsconfig.json, and utils.ts. The 'index.ts' file is selected. In the code editor, there is a line of code: 'const x = 1;'. A tooltip appears over the 'x' in 'const x = 1;' with the message 'Cannot redeclare block-scoped variable 'x''. Below the tooltip, it says 'file.ts(1, 7): 'x' was also declared here.' There is also a 'View Problem (F8)' link and a note 'No quick fixes available'.

This is actually not quite true. This rule applies only to files that are treated as "scripts". A file is a script if it does not contain any export or import statements. If a file has those, then the file is treated as a [module](#), and the variables do not get defined in the block scope.

More about tsconfig

We have so far used only one tsconfig rule [noImplicitAny](#). It's a good place to start, but now it's time to look into the config file a little deeper.

As mentioned, the [tsconfig.json](#) file contains all your core configurations on how you want TypeScript to work in your project.

Let's specify the following configurations in our [tsconfig.json](#) file:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "noImplicitAny": true,      "esModuleInterop": true,
    "moduleResolution": "node"
  }
}
```

Do not worry too much about the *compilerOptions*; they will be under closer inspection later on.

You can find explanations for each of the configurations from the TypeScript documentation or from the really handy [tsconfig page](#), or from the tsconfig [schema definition](#), which unfortunately is

formatted a little worse than the first two options.

Adding Express to the mix

Right now, we are in a pretty good place. Our project is set up and we have two executable calculators in it. However, since we aim to learn FullStack development, it is time to start working with some HTTP requests.

Let us start by installing Express:

```
npm install express
```

and then add the *start* script to package.json:

```
{
  // ...
  "scripts": {
    "ts-node": "ts-node",
    "multiply": "ts-node multiplier.ts",
    "calculate": "ts-node calculator.ts",
    "start": "ts-node index.ts"
  },
  // ...
}
```

Now we can create the file *index.ts*, and write the HTTP GET *ping* endpoint to it:

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

app.get('/ping', (req, res) => [
  res.send('pong')
]);

const PORT = 3003;

app.listen(PORT, () => [
  console.log(`Server running on port ${PORT}`)
]);
```

Everything else seems to be working just fine but, as you'd expect, the *req* and *res* parameters of *app.get* need typing. If you look carefully, VSCode is also complaining about the importing of Express. You can see a short yellow line of dots under *require*. Let's hover over the problem:

```

var require: NodeJS.Require
(id: string) => any

'require' call may be converted to an import. ts(80005)
Quick Fix...
const express = require('express');

const app = express();      You, a few seconds ago • Uncommitted changes

app.get('/ping', (req, res) => {
|   res.send('pong');
});

const PORT = 3003

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

```

The complaint is that the '*require*' call may be converted to an import. Let us follow the advice and write the import as follows:

```
import express from 'express';
```

NB: VSCode offers you the possibility to fix the issues automatically by clicking the *Quick Fix...* button. Keep your eyes open for these helpers/quick fixes; listening to your editor usually makes your code better and easier to read. The automatic fixes for issues can be a major time saver as well.

Now we run into another problem, the compiler complains about the import statement. Once again, the editor is our best friend when trying to find out what the issue is:

```

Could not find a declaration file for module 'express'.
'c:/Users/k036281/software/typescript/material/sandboxes/my-calculators-
express/node_modules/express/index.js' implicitly has an 'any' type.
  Try `npm install @types/express` if it exists or add a new declaration (.d.ts) file
  containing `declare module 'express';` ts(7016)
Peek Problem Quick Fix...
import express from 'express';
import _ from 'lodash';

const app = express();

app.get('/ping', (req, res) => {
|   res.send('pong');
});

const PORT = 3000

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

```

We haven't installed types for *express*. Let's do what the suggestion says and run:

```
npm install --save-dev @types/express
```

And no more errors! Let's take a look at what changed.

When we hover over the `require` statement, we can see that the compiler interprets everything express-related to be of type `any`.

The screenshot shows a code editor window for a file named `index.ts`. The code is as follows:

```
ts index.ts ×
ts index.ts > ⚏ app.get('/ping') callback
1 //import express from 'express';
2 const express = require('express');
3 const app = express();
4
5 app.get('/ping', (req, res) => {
6   res.send('pong');
7 });
8
9 const PORT = 3003;
10
11 app.listen(PORT, () => {
12   console.log(`Server running on port ${PORT}`);
13});
```

A tooltip is displayed over the `res` parameter in the `app.get` call, showing the error message: "Parameter 'res' implicitly has an 'any' type. ts(7006)". Below the message are "Peek Problem" and "Quick Fix..." buttons.

Whereas when we use `import`, the editor knows the actual types:

The screenshot shows a code editor window for `index.ts` with the following code:

```
ts index.ts > ⚏ app.get('/ping') callback
1 import express from 'express';
2 const app = express();
3
4 app.get('/ping', (req, res) => {
5   req.
6   res. [?] Symbol interface Symbolvar Symbol: SymbolConstru...
7 );
```

A tooltip is shown over the `res` variable, listing properties such as `_construct?`, `_destroy`, `_read`, etc. The `Symbol` interface is also listed.

Which import statement to use depends on the export method used in the imported package.

A good rule of thumb is to try importing a module using the `import` statement first. We will always use this method in the *frontend*. If `import` does not work, try a combined method: `import ... = require(...)`.

We strongly suggest you read more about TypeScript modules [here](#).

There is one more problem with the code:

A screenshot of the VS Code interface showing a tooltip for an unused parameter 'req'. The tooltip text is: '(parameter) req: Request<ParamsDictionary, any, any> 'req' is declared but its value is never read. ts(6133)'. It includes 'Peek Problem' and 'Quick Fix...' buttons.

```
ts index.ts
1 import express from 'express';
2 const app = express();
3 app.get('/ping', [req, res] => {
4   res.send('pong');
5 });
6
7
8 const PORT = 3003;
```

This is because we banned unused parameters in our `tsconfig.json`:

A screenshot of the VS Code interface showing the contents of `tsconfig.json`. The file contains compiler options like target ES2022, strict mode, and various rules for unused variables and parameters.

```
{
  "compilerOptions": {
    "target": "ES2022",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,      "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "noImplicitAny": true,
    "esModuleInterop": true,
    "moduleResolution": "node"
  }
}
```

This configuration might create problems if you have library-wide predefined functions that require declaring a variable even if it's not used at all, as is the case here. Fortunately, this issue has already been solved on the configuration level. Once again hovering over the issue gives us a solution. This time we can just click the quick fix button:

A screenshot of the VS Code interface showing the 'Quick Fix' option being selected for the unused 'req' parameter. The tooltip text is: '(parameter) req: Request<ParamsDictionary, any, any> 'req' is declared but its value is never read. ts(6133)'. The 'Prefix 'req' with an underscore' button is highlighted.

```
ts index.ts
1 import express from 'express';
2 const app = express();
3 app.get('/ping', [req, res] => {
4   res.send('pong');
5 });
6
7
8 const PORT = 3003;
```

If it is absolutely impossible to get rid of an unused variable, you can prefix it with an underscore to inform the compiler you have thought about it and there is nothing you can do.

Let's rename the `req` variable to `_req`. Finally, we are ready to start the application. It seems to work fine:

A screenshot of a browser window showing the result of the application run. The URL is `localhost:3003/ping`, and the response is `pong`.

To simplify the development, we should enable *auto-reloading* to improve our workflow. In this

course, you have already used *nodemon*, but ts-node has an alternative called *ts-node-dev*. It is meant to be used only with a development environment that takes care of recompilation on every change, so restarting the application won't be necessary.

Let's install *ts-node-dev* to our development dependencies:

```
npm install --save-dev ts-node-dev
```

Add a script to *package.json*:

```
{  
  // ...  
  "scripts": {  
    // ...  
    "dev": "ts-node-dev index.ts",  
    // ...  
  }  
}
```

And now, by running *npm run dev*, we have a working, auto-reloading development environment for our project!

Exercises 9.4-9.5

9.4 Express

Add Express to your dependencies and create an HTTP GET endpoint *hello* that answers 'Hello Full Stack!'

The web app should be started with the commands *npm start* in production mode and *npm run dev* in development mode. The latter should also use *ts-node-dev* to run the app.

Replace also your existing *tsconfig.json* file with the following content:

```
{  
  "compilerOptions": {  
    "noImplicitAny": true,  
    "noImplicitReturns": true,  
    "strictNullChecks": true,  
    "strictPropertyInitialization": true,  
    "strictBindCallApply": true,  
    "noUnusedLocals": true,  
    "noUnusedParameters": true,  
    "noImplicitThis": true,  
    "alwaysStrict": true,  
    "esModuleInterop": true,  
    "declaration": true,  
  },  
}
```

Make sure there aren't any errors!

9.5 WebBMI

Add an endpoint for the BMI calculator that can be used by doing an HTTP GET request to the endpoint `bmi` and specifying the input with [query string parameters](#). For example, to get the BMI of a person with a height of 180 and a weight of 72, the URL is <http://localhost:3003/bmi?height=180&weight=72>.

The response is a JSON of the form:

```
{  
  weight: 72,  
  height: 180,  
  bmi: "Normal (healthy weight)"  
}
```

See the [Express documentation](#) for info on how to access the query parameters.

If the query parameters of the request are of the wrong type or missing, a response with proper status code and an error message is given:

```
{  
  error: "malformatted parameters"  
}
```

Do not copy the calculator code to file `index.ts`; instead, make it a [TypeScript module](#) that can be imported into `index.ts`.

The horrors of `any`

Now that we have our first endpoints completed, you might notice that we have used barely any TypeScript in these small examples. When examining the code a bit closer, we can see a few dangers lurking there.

Let's add the HTTP POST endpoint `calculate` to our app:

```
import { calculator } from './calculator'  
  
app.use(express.json());  
  
// ...  
  
app.post('/calculate', (req, res) => [  
  const [ value1, value2, op ] = req.body;  
  
  const result = calculator(value1, value2, op);  
  res.send({ result });  
]) ;
```

To get this working, we must add an `export` to the function `calculator`:

```
export const calculator = (a: number, b: number, op: Operation) : number => [
```

When you hover over the *calculate* function, you can see the typing of the *calculator* even though the code itself does not contain any typings:

```
6 app.get('/ping', (_req, res) => {
7   res.send('pong2');
8 });
9
10 app.post('/calculate', (req, res) => {
11   const { value1, value2, op } = req.query;
12   const result = calculator(value1, value2, op);
13   res.send(result);
14 });
15
16 const PORT = 3003;
17
18 app.listen(PORT, () => {
19   console.log(`Server running on port ${PORT}`);
20 });
21
```

But if you hover over the values parsed from the request, an issue arises:

```
6 app.get('/ping', (_req, res) => {
7   res.send('pong2');
8 });
9
10 app.post('/calculate', (req, res) => {
11   const { value1, value2, op } = req.body;
12   const value2: any
13   const result = calculator(value1, value2, op);
14   res.send(result);
15 });
16
17 const PORT = 3003;
18
19 app.listen(PORT, () => {
20   console.log(`Server running on port ${PORT}`);
21 });
```

All of the variables have the type *any*. It is not all that surprising, as no one has given them a type yet. There are a couple of ways to fix this, but first, we have to consider why this is accepted and where the type *any* came from.

In TypeScript, every untyped variable whose type cannot be inferred implicitly becomes of type [any](#). Any is a kind of "wild card" type, which stands for *whatever* type. Things become implicitly any type quite often when one forgets to type functions.

We can also explicitly type things *any*. The only difference between the implicit and explicit any type is how the code looks; the compiler does not care about the difference.

Programmers however see the code differently when *any* is explicitly enforced than when it is implicitly inferred. Implicit *any* typings are usually considered problematic since it is quite often due

to the coder forgetting to assign types (or being too lazy to do it), and it also means that the full power of TypeScript is not properly exploited.

This is why the configuration rule [noImplicitAny](#) exists on the compiler level, and it is highly recommended to keep it on at all times. In the rare occasions when you truly cannot know what the type of a variable is, you should explicitly state that in the code:

```
const a : any = /* no clue what the type will be! */.
```

We already have [noImplicitAny: true](#) configured in our example, so why does the compiler not complain about the implicit *any* types? The reason is that the *body* field of an Express [Request](#) object is explicitly typed *any*. The same is true for the *request.query* field that Express uses for the query parameters.

What if we would like to restrict developers from using the *any* type? Fortunately, we have methods other than *tsconfig.json* to enforce a coding style. What we can do is use *ESlint* to manage our code. Let's install ESlint and its TypeScript extensions:

```
npm install --save-dev eslint @typescript-eslint/eslint-plugin @typescript-eslint/parser
```

We will configure ESlint to [disallow explicit any](#). Write the following rules to *.eslintrc.json*:

```
{  
  "parser": "@typescript-eslint/parser",  
  "parserOptions": {  
    "ecmaVersion": 11,  
    "sourceType": "module"  
  },  
  "plugins": ["@typescript-eslint"],  
  "rules": {  
    "@typescript-eslint/no-explicit-any": 2  
  }  
}
```

(Newer versions of ESlint have this rule on by default, so you don't necessarily need to add it separately.)

Let us also set up a *lint* npm script to inspect the files with *.ts* extension by modifying the *package.json* file:

```
{  
  // ...  
  "scripts": {  
    "start": "ts-node index.ts",  
    "dev": "ts-node-dev index.ts",  
    "lint": "eslint --ext .ts ." // ...  
  },  
  // ...
}
```

Now lint will complain if we try to define a variable of type *any*:

The screenshot shows a code editor with the following TypeScript code:

```
8  });
9
10 app.get('/calc' | Unexpected any. Specify a different type. eslint(@typescript-eslint/no-explicit-any)
11   | const { value1, value2, op } = req.query;
12   | const result = calculator(value1, value2, op);
13   | res.send(result);
14 );
15
16 );
```

An ESLint error is displayed for line 10, column 11, indicating an 'Unexpected any. Specify a different type.' violation. A tooltip provides options to 'Peek Problem' or 'Quick Fix...'.

[@typescript-eslint](#) has a lot of TypeScript-specific ESLint rules, but you can also use all basic ESLint rules in TypeScript projects. For now, we should probably go with the recommended settings, and we will modify the rules as we go along whenever we find something we want to change the behavior of.

On top of the recommended settings, we should try to get familiar with the coding style required in this part and *set the semicolon at the end of each line of code to be required*.

So we will use the following `.eslintrc.json`

```
{
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended-requiring-type-checking"
  ],
  "plugins": ["@typescript-eslint"],
  "env": {
    "node": true,
    "es6": true
  },
  "rules": {
    "@typescript-eslint/semi": ["error"],
    "@typescript-eslint/explicit-function-return-type": "off",
    "@typescript-eslint/explicit-module-boundary-types": "off",
    "@typescript-eslint/restrict-template-expressions": "off",
    "@typescript-eslint/restrict-plus-operands": "off",
    "@typescript-eslint/no-unused-vars": [
      "error",
      { "argsIgnorePattern": "^_" }
    ],
    "no-case-declarations": "off"
  },
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "project": "./tsconfig.json"
  }
}
```

Quite a few semicolons are missing, but those are easy to add. We also have to solve the ESLint issues concerning the *any* type:

```

1 import express from 'express';
2 const app = express();
3
4 import { calculator } from './calculator';
5
6 app.get( const value2: any
7   res.se );;
8 );;
9 Unsafe assignment of an `any` value. eslint(@typescript-eslint/no-unsafe-
10 assignment)
11 app.post View Problem Quick Fix... (⌘.)
12
13 const [ value1, value2, op ] = req.body;
14
15 const result = calculator(Number(value1), Number(value2), op);
16 res.send(result);
17 );
18
19 const PORT = 3003;
20
21 app.listen(PORT, () => {
22   console.log(`Server running on port ${PORT}`);
23 });

```

We could and probably should disable some ESLint rules to get the data from the request body.

Disabling `@typescript-eslint/no-unsafe-assignment` for the destructuring assignment and calling the `Number` constructor to values is nearly enough:

```

app.post('/calculate', (req, res) => [
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment  const [ v
  alue1, value2, op ] = req.body;

  const result = calculator(Number(value1), Number(value2), op); res.send({ res
  ult });
]

```

However this still leaves one problem to deal with, the last parameter in the function call is not safe:

```

app.get('/ping', (_req, res) => {
  res.send('pong!');
});

app.post('/calculate', (req, res) => {
  // eslint-disable-next-line @typescript-eslint/no-unsafe-a
  const { value1, value2, op } = req.body;
  const result = calculator(Number(value1), Number(value2), op); res.send(result);
});

```

`const op: any`

Unsafe argument of type `any` assigned to a
parameter of type
'Operation'. eslint(@typescript-eslint/no-
unsafe-argument)
View Problem Quick Fix... (⌘.)

We can just disable another ESLint rule to get rid of that:

```

app.post('/calculate', (req, res) => [
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const [ value1, value2, op ] = req.body;
]

```

```
// eslint-disable-next-line @typescript-eslint/no-unsafe-argument const result = calculator(Number(value1), Number(value2), op);
res.send({ result });
});
```

We now have ESLint silenced but we are totally at the mercy of the user. We most definitively should do some validation to the post data and give a proper error message if the data is invalid:

```
app.post('/calculate', (req, res) => {
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const { value1, value2, op } = req.body;

  if (!value1 || isNaN(Number(value1))) { return res.status(400).send({ error: '...' }); }
  // more validations here...

  // eslint-disable-next-line @typescript-eslint/no-unsafe-argument
  const result = calculator(Number(value1), Number(value2), op);
  return res.send({ result });
});
```

We shall see later in this part some techniques on how the *any* typed data (eg. the input an app receives from the user) can be *narrowed* to a more specific type (such as number). With a proper narrowing of types, there is no more need to silence the ESLint rules.

Type assertion

Using a **type assertion** is another "dirty trick" that can be done to keep TypeScript compiler and Eslint quiet. Let us export the type Operation in *calculator.ts*:

```
export type Operation = 'multiply' | 'add' | 'divide';
```

Now we can import the type and use a *type assertion* to tell the TypeScript compiler what type a variable has:

```
import { calculator, Operation } from './calculator';
app.post('/calculate', (req, res) => {
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const { value1, value2, op } = req.body;

  // validate the data here

  // assert the type
  const operation = op as Operation;
  const result = calculator(Number(value1), Number(value2), operation);
  return res.send({ result });
});
```

The defined constant *operation* has now the type *Operation* and the compiler is perfectly happy, no

quieting of the Eslint rule is needed on the following function call. The new variable is actually not needed, the type assertion can be done when an argument is passed to the function:

```
app.post('/calculate', (req, res) => [
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const [ value1, value2, op ] = req.body;

  // validate the data here

  const result = calculator(
    Number(value1), Number(value2), op as Operation );
  return res.send({ result });
]);
```

Using a type assertion (or quieting an Eslint rule) is always a bit risky thing. It leaves the TypeScript compiler off the hook, the compiler just trusts that we as developers know what we are doing. If the asserted type does *not* have the right kind of value, the result will be a runtime error, so one must be pretty careful when validating the data if a type assertion is used.

In the next chapter, we shall have a look at [type narrowing](#) which will provide a much more safe way of giving a stricter type for data that is coming from an external source.

Exercises 9.6-9.7

9.6 Eslint

Configure your project to use the above ESLint settings and fix all the warnings.

9.7 WebExercises

Add an endpoint to your app for the exercise calculator. It should be used by doing a HTTP POST request to the endpoint <http://localhost:3003/exercises> with the following input in the request body:

```
{  
  "daily_exercises": [1, 0, 2, 0, 3, 0, 2.5],  
  "target": 2.5  
}
```

The response is a JSON of the following form:

```
{  
  "periodLength": 7,  
  "trainingDays": 4,  
  "success": false,  
  "rating": 1,  
  "ratingDescription": "bad",  
  "target": 2.5,  
  "average": 1.2142857142857142  
}
```

If the body of the request is not in the right form, a response with the proper status code and an error message are given. The error message is either

```
{  
  error: "parameters missing"  
}
```

or

```
{  
  error: "malformatted parameters"  
}
```

depending on the error. The latter happens if the input values do not have the right type, i.e. they are not numbers or convertible to numbers.

In this exercise, you might find it beneficial to use the *explicit any* type when handling the data in the request body. Our ESLint configuration is preventing this but you may unset this rule for a particular line by inserting the following comment as the previous line:

```
// eslint-disable-next-line @typescript-eslint/no-explicit-any
```

You might also get in trouble with rules *no-unsafe-member-access* and *no-unsafe-assignment*. These rules may be ignored in this exercise.

Note that you need to have a correct setup to get the request body; see [part 3](#).

9c: Typing an Express app

Now that we have a basic understanding of how TypeScript works and how to create small projects with it, it's time to start creating something useful. We are now going to create a new project that will introduce use cases that are a little more realistic.

One major change from the previous part is that *we're not going to use ts-node anymore*. It is a handy tool that helps you get started, but in the long run, it is advisable to use the official TypeScript compiler that comes with the *typescript* npm-package. The official compiler generates and packages JavaScript files from the .ts files so that the built *production version* won't contain any TypeScript code anymore. This is the exact outcome we are aiming for since TypeScript itself is not executable by browsers or Node.

Setting up the project

We will create a project for Ilari, who loves flying small planes but has a difficult time managing his flight history. He is a coder himself, so he doesn't necessarily need a user interface, but he'd like to use some custom software with HTTP requests and retain the possibility of later adding a web-based user interface to the application.

Let's start by creating our first real project: *Ilari's flight diaries*. As usual, run *npm init* and install the *typescript* package as a dev dependency.

```
npm install typescript --save-dev
```

TypeScript's Native Compiler (*tsc*) can help us initialize our project by generating our *tsconfig.json* file. First, we need to add the *tsc* command to the list of executable scripts in *package.json* (unless you have installed *typescript* globally). Even if you installed TypeScript globally, you should always add it as a dev dependency to your project.

The npm script for running *tsc* is set as follows:

```
{  
  // ...  
  "scripts": {  
    "tsc": "tsc"  },  
  // ...  
}
```

The bare *tsc* command is often added to *scripts* so that other scripts can use it, hence don't be surprised to find it set up within the project like this.

We can now initialize our *tsconfig.json* settings by running:

```
npm run tsc -- --init
```

Note the extra -- before the actual argument! Arguments before -- are interpreted as being for the *npm* command, while the ones after that are meant for the command that is run through the script (i.e. *tsc* in this case).

The *tsconfig.json* file we just created contains a lengthy list of every configuration available to us. However, most of them are commented out. Studying this file can help you find some configuration options you might need. It is also completely okay to keep the commented lines, in case you might need them someday.

At the moment, we want the following to be active:

```
{  
  "compilerOptions": {  
    "target": "ES6",  
    "outDir": "./build/",  
    "module": "commonjs",  
    "strict": true,  
    "noUnusedLocals": true,  
    "noUnusedParameters": true,  
    "noImplicitReturns": true,  
    "noFallthroughCasesInSwitch": true,  
    "esModuleInterop": true  
  }  
}
```

Let's go through each configuration:

The `target` configuration tells the compiler which *ECMAScript* version to use when generating JavaScript. ES6 is supported by most browsers, so it is a good and safe option.

`outDir` tells where the compiled code should be placed.

`module` tells the compiler that we want to use *CommonJS* modules in the compiled code. This means we can use the old `require` syntax instead of the `import` one, which is not supported in older versions of *Node*.

`strict` is a shorthand for multiple separate options: `noImplicitAny`, `noImplicitThis`, `alwaysStrict`, `strictBindCallApply`, `strictNullChecks`, `strictFunctionTypes` and `strictPropertyInitialization`. They guide our coding style to use the TypeScript features more strictly. For us, perhaps the most important is the already-familiar `noImplicitAny`. It prevents implicitly setting type `any`, which can for example happen if you don't type the parameters of a function. Details about the rest of the configurations can be found in the [tsconfig documentation](#). Using `strict` is suggested by the official documentation.

`noUnusedLocals` prevents having unused local variables, and `noUnusedParameters` throws an error if a function has unused parameters.

`noImplicitReturns` checks all code paths in a function to ensure they return a value.

`noFallthroughCasesInSwitch` ensures that, in a `switch case`, each case ends either with a `return` or a `break` statement.

`esModuleInterop` allows interoperability between CommonJS and ES Modules; see more in the [documentation](#).

Now that we have set our configuration, we can continue by installing `express` and, of course, also `@types/express`. Also, since this is a real project, which is intended to be grown over time, we will use ESlint from the very beginning:

```
npm install express
npm install --save-dev eslint @types/express @typescript-eslint/eslint-plugin @typescript-eslint/parser
```

Now our `package.json` should look like this:

```
{
  "name": "flight-diary",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "tsc": "tsc"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "@types/express": "^4.17.18",
    "@typescript-eslint/eslint-plugin": "^6.7.3",
    "eslint": "^7.14.0"
  }
}
```

```
"@typescript-eslint/parser": "^6.7.3",
"eslint": "^8.50.0",
"typescript": "^5.2.2"
},
}
```

We also create a `.eslintrc` file with the following content:

```
{
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended-requiring-type-checking"
  ],
  "plugins": ["@typescript-eslint"],
  "env": {
    "browser": true,
    "es6": true,
    "node": true
  },
  "rules": [
    "@typescript-eslint/semi": ["error"],
    "@typescript-eslint/explicit-function-return-type": "off",
    "@typescript-eslint/explicit-module-boundary-types": "off",
    "@typescript-eslint/restrict-template-expressions": "off",
    "@typescript-eslint/restrict-plus-operands": "off",
    "@typescript-eslint/no-unsafe-member-access": "off",
    "@typescript-eslint/no-unused-vars": [
      "error",
      { "argsIgnorePattern": "^_" }
    ],
    "no-case-declarations": "off"
  ],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "project": "./tsconfig.json"
  }
}
```

Now we just need to set up our development environment, and we are ready to start writing some serious code. There are many different options for this. One option could be to use the familiar `nodemon` with `ts-node`. However, as we saw earlier, `ts-node-dev` does the same thing, so we will use that instead. So, let's install `ts-node-dev`:

```
npm install --save-dev ts-node-dev
```

We finally define a few more npm scripts, and voilà, we are ready to begin:

```
{
  // ...
```

```

"scripts": [
  "tsc": "tsc",
  "dev": "ts-node-dev index.ts" ,    "lint": "eslint --ext .ts ." },
  // ...
]

```

As you can see, there is a lot of stuff to go through before beginning the actual coding. When you are working on a real project, careful preparations support your development process. Take the time needed to create a good setup for yourself and your team, so that everything runs smoothly in the long run.

Let there be code

Now we can finally start coding! As always, we start by creating a ping endpoint, just to make sure everything is working.

The contents of the *index.ts* file:

```

import express from 'express';
const app = express();
app.use(express.json());

const PORT = 3000;

app.get('/ping', (_req, res) => [
  console.log('someone pinged here'),
  res.send('pong')
]);

app.listen(PORT, () => [
  console.log(`Server running on port ${PORT}`),
]);

```

Now, if we run the app with *npm run dev*, we can verify that a request to <http://localhost:3000/ping> gives the response *pong*, so our configuration is set!

When starting the app with *npm run dev*, it runs in development mode. The development mode is not suitable at all when we later operate the app in production.

Let's try to create a *production build* by running the TypeScript compiler. Since we have defined the *outdir* in our *tsconfig.json*, nothing's left but to run the script *npm run tsc*.

Just like magic, a native runnable JavaScript production build of the Express backend is created in file *index.js* inside the directory *build*. The compiled code looks like this

```

"use strict";
var __importDefault = (this && this.__importDefault) || function (mod) {
  return (mod && mod.__esModule) ? mod : { "default": mod };
};
Object.defineProperty(exports, "__esModule", { value: true });
const express_1 = __importDefault(require("express"));
const app = (0, express_1.default)();

```

```

app.use(express_1.default.json());
const PORT = 3000;
app.get('/ping', [req, res] => [
    console.log('someone pinged here'),
    res.send('pong')
]);
app.listen(PORT, () => [
    console.log(`Server running on port ${PORT}`)
]);

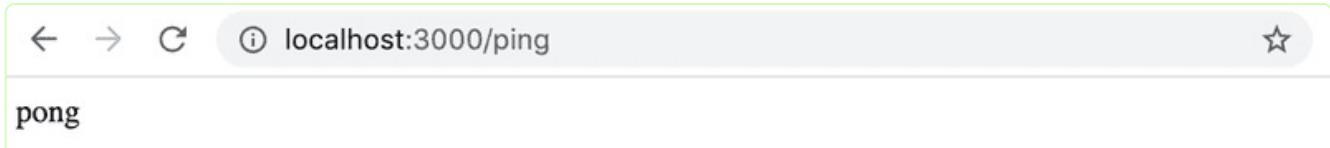
```

Currently, if we run ESLint it will also interpret the files in the *build* directory. We don't want that, since the code there is compiler-generated. We can prevent this by creating a *.eslintignore* file that lists the content we want ESLint to ignore, just like we do with git and *.gitignore*.

Let's add an npm script for running the application in production mode:

```
{
  // ...
  "scripts": {
    "tsc": "tsc",
    "dev": "ts-node-dev index.ts",
    "lint": "eslint --ext .ts .",
    "start": "node build/index.js"
  }
}
```

When we run the app with *npm start*, we can verify that the production build also works:



Now we have a minimal working pipeline for developing our project. With the help of our compiler and ESLint, we ensure that good code quality is maintained. With this base, we can start creating an app that we could, later on, deploy into a production environment.

Exercises 9.8-9.9

Before you start the exercises

For this set of exercises, you will be developing a backend for an existing project called **Patientor**, which is a simple medical record application for doctors who handle diagnoses and basic health information of their patients.

The **frontend** has already been built by outsider experts and your task is to create a backend to support the existing code.

WARNING

Quite often VS code loses track of what is really happening in the code and it shows type or style

related warnings despite the code having been fixed. If this happens (to me it has happened quite often), close and open the file that is giving you trouble or just restart the editor. It is also good to doublecheck that everything really works by running the compiler and the ESLint from the command line with commands:

```
npm run tsc  
npm run lint
```

When run in command line you get the "real result" for sure. So, never trust the editor too much!

9.8: Patientor backend, step1

Initialize a new backend project that will work with the frontend. Configure ESLint and tsconfig with the same configurations as proposed in the material. Define an endpoint that answers HTTP GET requests for route `/api/ping`.

The project should be runnable with npm scripts, both in development mode and, as compiled code, in production mode.

9.9: Patientor backend, step2

Fork and clone the project [patientor](#). Start the project with the help of the README file.

You should be able to use the frontend without a functioning backend.

Ensure that the backend answers the ping request that the *frontend* has made on startup. Check the developer tools to make sure it works:

The screenshot shows a browser window with the URL `localhost:3000/ping` in the address bar. The main content area displays the Patientor logo and a blue "Home" button. Below the browser window is the developer tools interface, specifically the Network tab. The Network tab shows a table of network requests. One row for the "ping" endpoint is highlighted with a red border, indicating it failed. The table includes columns for Name, Status, Type, Initiator, and Size. Other rows visible include "0.chunk.js" (Status 200, Type fetch, Initiator VM450:1, Size 849 KB) and "index.html" (Status 200, Type fetch, Initiator VM450:1, Size 600 KB).

Name	Status	Type	Initiator	Size
ping	(failed)	xhr	VM450:1	0 B
0.chunk.js	200	fetch	VM450:1	849 KB
index.html	200	fetch	VM450:1	600 KB

You might also want to have a look at the *console* tab. If something fails, [part 3](#) of the course shows how the problem can be solved.

Implementing the functionality

Finally, we are ready to start writing some code.

Let's start from the basics. Ilari wants to be able to keep track of his experiences on his flight journeys.

He wants to be able to save *diary entries*, which contain:

- The date of the entry
- Weather conditions (sunny, windy, cloudy, rainy or stormy)
- Visibility (great, good, ok or poor)
- Free text detailing the experience

We have obtained some sample data, which we will use as a base to build on. The data is saved in JSON format and can be found [here](#).

The data looks like the following:

```
[  
  {  
    "id": 1,  
    "date": "2017-01-01",  
    "weather": "rainy",  
    "visibility": "poor",  
    "comment": "Pretty scary flight, I'm glad I'm alive"  
  },  
  {  
    "id": 2,  
    "date": "2017-04-01",  
    "weather": "sunny",  
    "visibility": "good",  
    "comment": "Everything went better than expected, I'm learning much"  
  },  
  // ...  
]
```

Let's start by creating an endpoint that returns all flight diary entries.

First, we need to make some decisions on how to structure our source code. It is better to place all source code under *src* directory, so source code is not mixed with configuration files. We will move *index.ts* there and make the necessary changes to the npm scripts.

We will place all *routers* and modules which are responsible for handling a set of specific resources such as *diaries*, under the directory *src/routes*. This is a bit different than what we did in [part 4](#), where we used the directory *src/controllers*.

The router taking care of all diary endpoints is in *src/routes/diaries.ts* and looks like this:

```
import express from 'express';  
  
const router = express.Router();  
  
router.get('/', (_req, res) => {  
  res.send('Fetching all diaries!');  
});
```

```

router.post('/', (_req, res) => [
  res.send('Saving a diary!'),
]);

export default router;

```

We'll route all requests to prefix `/api/diaries` to that specific router in `index.ts`

```

import express from 'express';
import diaryRouter from './routes/diaries';const app = express();
app.use(express.json());

const PORT = 3000;

app.get('/ping', (_req, res) => [
  console.log('someone pinged here'),
  res.send('pong'),
]);

app.use('/api/diaries', diaryRouter);

app.listen(PORT, () => [
  console.log(`Server running on port ${PORT}`),
]);

```

And now, if we make an HTTP GET request to <http://localhost:3000/api/diaries>, we should see the message: *Fetching all diaries!*

Next, we need to start serving the seed data (found [here](#)) from the app. We will fetch the data and save it to `data/entries.json`.

We won't be writing the code for the actual data manipulations in the router. We will create a *service* that takes care of the data manipulation instead. It is quite a common practice to separate the "business logic" from the router code into modules, which are quite often called *services*. The name service originates from [Domain-driven design](#) and was made popular by the [Spring](#) framework.

Let's create a `src/services` directory and place the `diaryService.ts` file in it. The file contains two functions for fetching and saving diary entries:

```

import diaryData from '../../../../../data/entries.json';

const getEntries = () => [
  return diaryData,
];

const addDiary = () => [
  return null,
];

export default [
  getEntries,
  addDiary
];

```

But something is not right:

The screenshot shows a code editor with several tabs at the top: package.json, diaries.ts, diaryService.ts (which is currently active), diaries.json, and tsconfig.json. The diaryService.ts tab contains the following code:

```

1 import diaryData from './data/diaries.json'
2
3 const getEntries = () => {
4     return diaryData;
5 }
6
7 const addEntry = () => {
8     return null;
9 }
10
11 export default {
12     getEntries,

```

A red squiggle underline is under the line `import diaryData from './data/diaries.json'`. A tooltip message appears: "Cannot find module './data/diaries.json'. Consider using '--resolveJsonModule' to import module with '.json' extension ts(2732)". Below the tooltip are two buttons: "Peek Problem" and "Quick Fix...".

The hint says we might want to use *resolveJsonModule*. Let's add it to our tsconfig:

The screenshot shows the tsconfig.json file with the following content:

```
{
  "compilerOptions": {
    "target": "ES6",
    "outDir": "./build/",
    "module": "commonjs",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true,
    "resolveJsonModule": true
  }
}
```

And our problem is solved.

NB: For some reason, VSCode sometimes complains that it cannot find the file `./data/entries.json` from the service despite the file existing. That is a bug in the editor, and goes away when the editor is restarted.

Earlier, we saw how the compiler can decide the type of a variable by the value it is assigned. Similarly, the compiler can interpret large data sets consisting of objects and arrays. Due to this, the compiler warns us if we try to do something suspicious with the JSON data we are handling. For example, if we are handling an array containing objects of a specific type, and we try to add an object which does not have all the fields the other objects have, or has type conflicts (for example, a number where there should be a string), the compiler can give us a warning.

Even though the compiler is pretty good at making sure we don't do anything unwanted, it is safer to define the types for the data ourselves.

Currently, we have a basic working TypeScript Express app, but there are barely any actual *typings* in the code. Since we know what type of data should be accepted for the *weather* and *visibility* fields, there is no reason for us not to include their types in the code.

Let's create a file for our types, *types.ts*, where we'll define all our types for this project.

First, let's type the *Weather* and *Visibility* values using a [union type](#) of the allowed strings:

```
export type Weather = 'sunny' | 'rainy' | 'cloudy' | 'windy' | 'stormy';

export type Visibility = 'great' | 'good' | 'ok' | 'poor';
```

And, from there, we can continue by creating a *DiaryEntry* type, which will be an [interface](#):

```
export interface DiaryEntry {
  id: number;
  date: string;
  weather: Weather;
  visibility: Visibility;
  comment: string;
}
```

We can now try to type our imported JSON:

```
import diaryData from '../data/entries.json';

import { DiaryEntry } from '../types';
const diaries: DiaryEntry[] = diaryData;
const getEntries = (): DiaryEntry[] => { return diaries; };

const addDiary = () => {
  return null;
};

export default [
  getEntries,
  addDiary
];
```

But since the JSON already has its values declared, assigning a type for the data set results in an error:

```

src > services > TS diaryService.ts > ...
1 import diaryData from '../../../../../data/diaries.json';
2
3 import { DiaryEntry } from '../types';
4
5 const diaries: Array<DiaryEntry> = diaryData;
6 |
7 const
8   retu Type '{ "id": number; "date": string; "weather": string;
9 }; "visibility": string; "comment": string; }[]' is not assignable to
10 type 'DiaryEntry[]'.
11 const
12   retu Type '{ "id": number; "date": string; "weather": string;
13 }; "visibility": string; "comment": string; }' is not assignable to
14 type 'DiaryEntry'.
15 export
16   getE Peek Problem No quick fixes available ts(2322)
17   addEntry
18 };

```

The end of the error message reveals the problem: the *weather* fields are incompatible. In *DiaryEntry*, we specified that its type is *Weather*, but the TypeScript compiler had inferred its type to be *string*.

We can fix the problem by doing a [type assertion](#). As we already mentioned type assertions should be done only if we are certain we know what we are doing!

If we assert the type of the variable *diaryData* to be *DiaryEntry* with the keyword *as*, everything should work:

```

import diaryData from '../../../../../data/entries.json'

import { Weather, Visibility, DiaryEntry } from '../types'

const diaries: DiaryEntry[] = diaryData as DiaryEntry[];
const getEntries = (): DiaryEntry[] => {
  return diaries;
}

const addDiary = () => {
  return null;
}

export default {
  getEntries,
  addDiary
};

```

We should never use type assertion unless there is no other way to proceed, as there is always the danger we assert an unfit type to an object and cause a nasty runtime error. While the compiler trusts you to know what you are doing when using *as*, by doing this, we are not using the full power of TypeScript but relying on the coder to secure the code.

In our case, we could change how we export our data so we can type it within the data file. Since we

cannot use typings in a JSON file, we should convert the JSON file to a ts file *diaries.ts* which exports the typed data like so:

```
import { DiaryEntry } from "../src/types";
const diaryEntries: DiaryEntry[] = [
  {
    "id": 1,
    "date": "2017-01-01",
    "weather": "rainy",
    "visibility": "poor",
    "comment": "Pretty scary flight, I'm glad I'm alive"
  },
  // ...
];

export default diaryEntries;
```

Now, when we import the array, the compiler interprets it correctly and the *weather* and *visibility* fields are understood right:

```
import diaries from '../data/entries';
import { DiaryEntry } from '../types';

const getEntries = (): DiaryEntry[] => {
  return diaries;
}

const addDiary = () => {
  return null;
}

export default [
  getEntries,
  addDiary
];
```

Note that, if we want to be able to save entries without a certain field, e.g. *comment*, we could set the type of the field as **optional** by adding `?>` to the type declaration:

```
export interface DiaryEntry {
  id: number;
  date: string;
  weather: Weather;
  visibility: Visibility;
  comment?: string;
}
```

Node and JSON modules

It is important to take note of a problem that may arise when using the tsconfig `resolveJsonModule` option:

```
[{"compilerOptions": {  
    // ...  
    "resolveJsonModule": true  
}]
```

According to the node documentation for [file modules](#), node will try to resolve modules in order of extensions:

```
["js", "json", "node"]
```

In addition to that, by default, *ts-node* and *ts-node-dev* extend the list of possible node module extensions to:

```
["js", "json", "node", "ts", "tsx"]
```

NB: The validity of *.js*, *.json* and *.node* files as modules in TypeScript depend on environment configuration, including *tsconfig* options such as *allowJs* and *resolveJsonModule*.

Consider a flat folder structure containing files:

```
├── myModule.json  
└── myModule.ts
```

In TypeScript, with the *resolveJsonModule* option set to true, the file *myModule.json* becomes a valid node module. Now, imagine a scenario where we wish to take the file *myModule.ts* into use:

```
import myModule from "./myModule";
```

Looking closely at the order of node module extensions:

```
["js", "json", "node", "ts", "tsx"]
```

We notice that the *.json* file extension takes precedence over *.ts* and so *myModule.json* will be imported and not *myModule.ts*.

To avoid time-eating bugs, it is recommended that within a flat directory, each file with a valid node module extension has a unique filename.

Utility Types

Sometimes, we might want to use a specific modification of a type. For example, consider a page for listing some data, some of which is sensitive and some of which is non-sensitive. We might want to be sure that no sensitive data is used or displayed. We could *pick* the fields of a type we allow to be used to enforce this. We can do that by using the utility type [Pick](#).

In our project, we should consider that Ilari might want to create a listing of all his diary entries *excluding* the comment field since, during a very scary flight, he might end up writing something he wouldn't necessarily want to show to anyone else.

The [Pick](#) utility type allows us to choose which fields of an existing type we want to use. Pick can be used to either construct a completely new type or to inform a function of what it should return on runtime. Utility types are a special kind of type, but they can be used just like regular types.

In our case, to create a "censored" version of the *DiaryEntry* for public displays, we can use *Pick* in the function declaration:

```
const getNonSensitiveEntries =  
  () : Pick<DiaryEntry, 'id' | 'date' | 'weather' | 'visibility'>[] => [  
    // ...  
]
```

and the compiler would expect the function to return an array of values of the modified *DiaryEntry* type, which includes only the four selected fields.

In this case, we want to exclude only one field, so it would be even better to use the [Omit](#) utility type, which we can use to declare which fields to exclude:

```
const getNonSensitiveEntries = () : Omit<DiaryEntry, 'comment'>[] => [  
  // ...  
]
```

To improve the readability, we should most definitively define a [type alias](#) *NonSensitiveDiaryEntry* in the file *types.ts*:

```
export type NonSensitiveDiaryEntry = Omit<DiaryEntry, 'comment'>;
```

The code becomes now much more clear and more descriptive:

```
import diaries from '../data/entries';  
import { NonSensitiveDiaryEntry, DiaryEntry } from '../types';  
const getEntries = () : DiaryEntry[] => [  
  return diaries;  
];  
  
const getNonSensitiveEntries = () : NonSensitiveDiaryEntry[] => [  
  return diaries;  
];  
  
const addDiary = () => [  
  return null;  
];  
  
export default [  
  getEntries,  
  addDiary,
```

```
getNonSensitiveEntries]:
```

One thing in our application is a cause for concern. In `getNonSensitiveEntries`, we are returning the complete diary entries, and *no error is given* despite typing!

This happens because [TypeScript only checks](#) whether we have all of the required fields or not, but excess fields are not prohibited. In our case, this means that it is *not prohibited* to return an object of type `DiaryEntry[]`, but if we were to try to access the `comment` field, it would not be possible because we would be accessing a field that TypeScript is unaware of even though it exists.

Unfortunately, this can lead to unwanted behavior if you are not aware of what you are doing; the situation is valid as far as TypeScript is concerned, but you are most likely allowing a use that is not wanted. If we were now to return all of the diary entries from the `getNonSensitiveEntries` function to the frontend, we would be *leaking the unwanted fields to the requesting browser* - even though our types seem to imply otherwise!

Because TypeScript doesn't modify the actual data but only its type, we need to exclude the fields ourselves:

```
import diaries from '../data/entries.ts'

import { NonSensitiveDiaryEntry, DiaryEntry } from '../types'

const getEntries = () : DiaryEntry[] => {
  return diaries
}

const getNonSensitiveEntries = (): NonSensitiveDiaryEntry[] => {
  return diaries.map(([id, date, weather, visibility]) => ({ id, date, weather, visibility }));
}

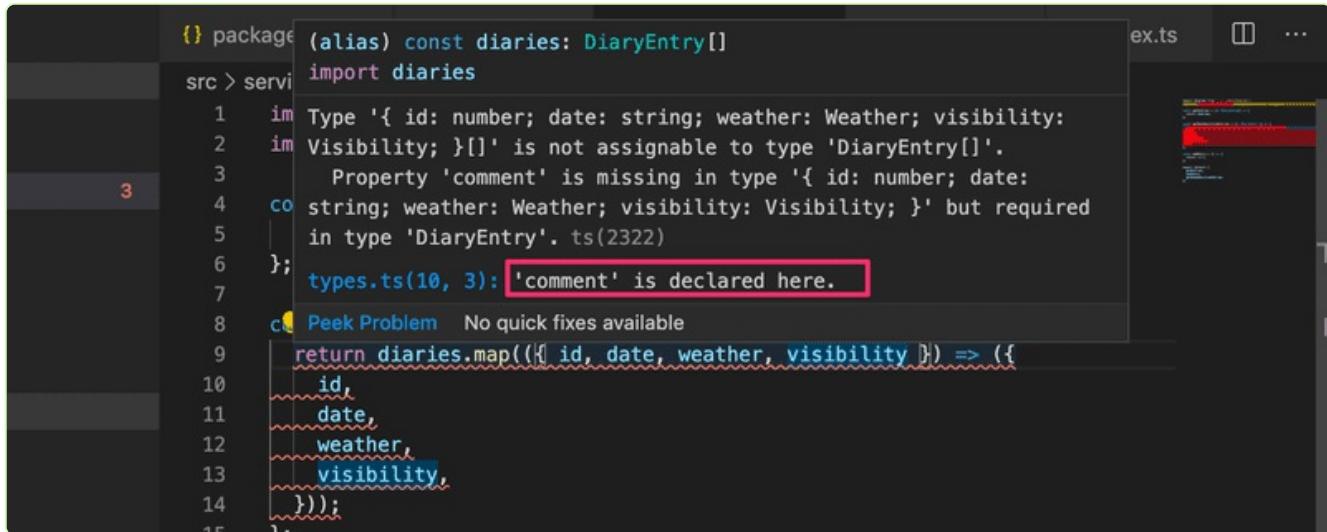
const addDiary = () => {
  return null;
}

export default [
  getEntries,
  getNonSensitiveEntries,
  addDiary
]
```

If we now try to return this data with the basic `DiaryEntry` type, i.e. if we type the function as follows:

```
const getNonSensitiveEntries = (): DiaryEntry[] => [
```

we would get the following error:



```
{ package (alias) const diaries: DiaryEntry[]
src > service
  1 import diaries
  2
  3     im Type '{ id: number; date: string; weather: Weather; visibility: Visibility; }[]' is not assignable to type 'DiaryEntry[]'.
  4         Property 'comment' is missing in type '{ id: number; date:
  5             co string; weather: Weather; visibility: Visibility; }' but required
  6             in type 'DiaryEntry'. ts(2322)
  7     );
  8     types.ts(10, 3): 'comment' is declared here.
  9     c Peek Problem No quick fixes available
10     return diaries.map(({ id, date, weather, visibility }) => ({
11         id,
12         date,
13         weather,
14         visibility,
15     }));
1.
```

Again, the last line of the error message is the most helpful one. Let's undo this undesired modification.

Note that if you make the comment field optional (using the ? operator), everything will work fine.

Utility types include many handy tools, and it is undoubtedly worth it to take some time to study [the documentation](#).

Finally, we can complete the route which returns all diary entries:

```
import express from 'express';
import diaryService from '../services/diaryService';
const router = express.Router();

router.get('/', (_req, res) => {
    res.send(diaryService.getNonSensitiveEntries());
}

router.post('/', (_req, res) => {
    res.send('Saving a diary!');
})

export default router;
```

The response is what we expect it to be:



A screenshot of a web browser window showing a JSON response at the URL `localhost:3000/api/diaries`. The JSON data consists of an array of three diary entries, each represented as an object with properties: `id`, `date`, `weather`, and `visibility`.

```
[  
  - {  
      id: 1,  
      date: "2017-01-01",  
      weather: "rainy",  
      visibility: "poor"  
    },  
  - {  
      id: 2,  
      date: "2017-04-01",  
      weather: "sunny",  
      visibility: "good"  
    },  
  - {  
      id: 3,  
      date: "2017-04-15",  
      weather: "windy",  
      visibility: "good"  
    },  
  - {  
  }]
```

Exercises 9.10-9.11

Similarly to Ilari's flight service, we do not use a real database in our app but instead use hardcoded data that is in the files `diagnoses.ts` and `patients.ts`. Get the files and store those in a directory called `data` in your project. All data modification can be done in runtime memory, so during this part, it is *not necessary to write to a file*.

9.10: Patient or backend, step3

Create a type `Diagnosis` and use it to create endpoint `/api/diagnoses` for fetching all diagnoses with HTTP GET.

Structure your code properly by using meaningfully-named directories and files.

Note that `diagnoses` may or may not contain the field `latin`. You might want to use [optional properties](#) in the type definition.

9.11: Patient or backend, step4

Create data type `Patient` and set up the GET endpoint `/api/patients` which returns all the patients to the frontend, excluding field `ssn`. Use a [utility type](#) to make sure you are selecting and returning only the wanted fields.

In this exercise, you may assume that field `gender` has type `string`.

Try the endpoint with your browser and ensure that `ssn` is not included in the response:

```
localhost:3001/api/patients
```

```
[  
  - {  
      id: "d2773336-f723-11e9-8f0b-362b9e155667",  
      name: "John McClane",  
      dateOfBirth: "1986-07-09",  
      gender: "male",  
      occupation: "New york city cop"  
    },  
  - {  
      id: "d2773598-f723-11e9-8f0b-362b9e155667",  
      name: "Martin Riggs",  
      dateOfBirth: "1986-07-09",  
      gender: "male",  
      occupation: "Cop"  
    },  
  - {  
      id: "d2773599-f723-11e9-8f0b-362b9e155667",  
      name: "Hans Gruber",  
      dateOfBirth: "1986-07-09",  
      gender: "male",  
      occupation: "Technician"  
    },  
  - {  
      id: "d2773600-f723-11e9-8f0b-362b9e155667",  
      name: "Dana Scully",  
      dateOfBirth: "1986-07-09",  
      gender: "female",  
      occupation: "Forensic Pathologist"  
    }  
]
```

After creating the endpoint, ensure that the *frontend* shows the list of patients:

Name	Gender	Occupation	Health Rating
John McClane	male	New york city cop	♥ ♥ ♥ ♥
Martin Riggs	male	Cop	♥ ♥ ♥ ♥
Hans Gruber	male	Technician	♥ ♥ ♥ ♥
Dana Scully	female	Forensic Pathologist	♥ ♥ ♥ ♥

Preventing an accidental undefined result

Let's extend the backend to support fetching one specific entry with an HTTP GET request to route `api/diaries/:id`.

The DiaryService needs to be extended with a `findById` function:

```
// ...  
  
const findById = (id: number): DiaryEntry => {  const entry = diaries.find(d => d  
.id === id);  return entry};  
export default [  
  getEntries,  
  getNonSensitiveEntries,  
  addDiary,  
  findById]
```

But once again, a new problem emerges:

```

16 } const entry: DiaryEntry | undefined
17 Type 'DiaryEntry | undefined' is not assignable to type
18 'DiaryEntry'.
19     Type 'undefined' is not assignable to type 'DiaryEntry'. ts(2322)
20 Peek Problem No quick fixes available
21 return entry;
22 }
23
24 const addDiaryEntry = () => {

```

The issue is that there is no guarantee that an entry with the specified id can be found. It is good that we are made aware of this potential problem already at compile phase. Without TypeScript, we would not be warned about this problem, and in the worst-case scenario, we could have ended up returning an *undefined* object instead of informing the user about the specified entry not being found.

First of all, in cases like this, we need to decide what the *return value* should be if an object is not found, and how the case should be handled. The *find* method of an array returns *undefined* if the object is not found, and this is fine. We can solve our problem by typing the return value as follows:

```

const findById = (id: number): DiaryEntry | undefined => [
  const entry = diaries
    .find(d => d.id === id);
  return entry;
]

```

The route handler is the following:

```

import express from 'express';
import diaryService from '../services/diaryService'

router.get('/:id', (req, res) => {
  const diary = diaryService.findById(Number(req.params.id));

  if (diary) {
    res.send(diary);
  } else {
    res.sendStatus(404);
  }
});

// ...

export default router;

```

Adding a new diary

Let's start building the HTTP POST endpoint for adding new flight diary entries. The new entries should have the same type as the existing data.

The code handling of the response looks as follows:

```

router.post('/ ', [req, res] => [
  const [ date, weather, visibility, comment ] = req.body,
  const addedEntry = diaryService.addDiary([
    date,
    weather,
    visibility,
    comment,
  ]),
  res.json(addedEntry),
])

```

The corresponding method in *diaryService* looks like this:

```

import {
  NonSensitiveDiaryEntry,
  DiaryEntry,
  Visibility, Weather} from '../types'

const addDiary = [
  date: string, weather: Weather, visibility: Visibility, comment: string
]: DiaryEntry => {

  const newDiaryEntry = [
    id: Math.max(...diaries.map(d => d.id)) + 1,
    date,
    weather,
    visibility,
    comment,
  ],
  diaries.push(newDiaryEntry),
  return newDiaryEntry,
}

```

As you can see, the *addDiary* function is becoming quite hard to read now that we have all the fields as separate parameters. It might be better to just send the data as an object to the function:

```

router.post('/ ', [req, res] => [
  const [ date, weather, visibility, comment ] = req.body,
  const addedEntry = diaryService.addDiary({
    date,
    weather,
    visibility,
    comment,
  }),
  res.json(addedEntry),
])

```

But wait, what is the type of this object? It is not exactly a *DiaryEntry*, since it is still missing the *id* field. It could be useful to create a new type, *NewDiaryEntry*, for an entry that hasn't been saved yet. Let's create that in *types.ts* using the existing *DiaryEntry* type and the [Omit](#) utility type:

```
export type NewDiaryEntry = Omit<DiaryEntry, 'id'>;
```

Now we can use the new type in our `DiaryService`, and destructure the new entry object when creating an entry to be saved:

```
import { NewDiaryEntry, NonSensitiveDiaryEntry, DiaryEntry } from '../types';
// ...

const addDiary = [ entry: NewDiaryEntry ]: DiaryEntry => {
  const newDiaryEntry =
    {
      id: Math.max(...diaries.map(d => d.id)) + 1,
      ...entry
    };

  diaries.push(newDiaryEntry);
  return newDiaryEntry;
};
```

Now the code looks much cleaner!

There is still a complaint from our code:

```
16 |   );
17 |   res. const comment: any
18 | });
19 | );
```

Unsafe assignment of an any value. eslint(@typescript-eslint/no-unsafe-assignment)

```
20 |outer.p Peek Problem (CF8) Quick Fix... (%)
21 | const { date, weather, visibility, comment } = req.body;
22 | const newDiaryEntry = diaryService.addDiary({
23 |   date,
24 |   weather,
25 |   visibility,
26 |   comment,
27 | });
28 |   res.json(newDiaryEntry);
29 | });
30 |
31 |
32 export default router;
```

The cause is the ESLint rule [@typescript-eslint/no-unsafe-assignment](#) that prevents us from assigning the fields of a request body to variables.

For the time being, let us just ignore the ESLint rule from the whole file by adding the following as the first line of the file:

```
/* eslint-disable @typescript-eslint/no-unsafe-assignment */
```

To parse the incoming data we must have the `json` middleware configured:

```
import express from 'express';
import diaryRouter from './routes/diaries';
const app = express();
app.use(express.json());
```

```

const PORT = 3000;

app.use('/api/diaries', diaryRouter);

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

```

Now the application is ready to receive HTTP POST requests for new diary entries of the correct type!

Proofing requests

There are plenty of things that can go wrong when we accept data from outside sources. Applications rarely work completely on their own, and we are forced to live with the fact that data from sources outside of our system cannot be fully trusted. When we receive data from an outside source, there is no way it can already be typed when we receive it. We need to make decisions on how to handle the uncertainty that comes with this.

The disabled ESLint rule was hinting to us that the following assignment is risky:

```

const newDiaryEntry = diaryService.addDiary([
  date,
  weather,
  visibility,
  comment,
])
;
```

We would like to have the assurance that the object in a POST request has the correct type. Let us now define a function `toNewDiaryEntry` that receives the request body as a parameter and returns a properly-typed `NewDiaryEntry` object. The function shall be defined in the file `utils.ts`.

The route definition uses the function as follows:

```

import toNewDiaryEntry from '../utils';
// ...

router.post('/', (req, res) => [
  try {
    const newDiaryEntry = toNewDiaryEntry(req.body);
    const addedEntry = diaryService.addDiary(newDiaryEntry);      res.json(addedEntry)
  } catch (error: unknown) {
    let errorMessage = 'Something went wrong.';
    if (error instanceof Error) {
      errorMessage += ` Error: ${error.message}`;
    }
    res.status(400).send(errorMessage);
  }
])
;
```

We can now also remove the first line that ignores the ESLint rule `no-unsafe-assignment`.

Since we are now writing secure code and trying to ensure that we are getting exactly the data we want from the requests, we should get started with parsing and validating each field we are expecting to receive.

The skeleton of the function `toNewDiaryEntry` looks like the following:

```
import { NewDiaryEntry } from './types';

const toNewDiaryEntry = (object): NewDiaryEntry => {
  const newEntry: NewDiaryEntry = {
    // ...
  };

  return newEntry;
};

export default toNewDiaryEntry;
```

The function should parse each field and make sure that the return value is exactly of type `NewDiaryEntry`. This means we should check each field separately.

Once again, we have a type issue: what is the type of the parameter `object`? Since the `object` is the body of a request, Express has typed it as `any`. Since the idea of this function is to map fields of unknown type to fields of the correct type and check whether they are defined as expected, this might be the rare case in which we *want to allow the `any` type*.

However, if we type the object as `any`, ESLint complains about that:

The screenshot shows a code editor with two files open: `types.ts` and `utils.ts`. The `utils.ts` file contains the following code:

```
src > TS utils.ts > [?] default
1 import { NewDiaryEntry }
2
3 const toNewDiaryEntry = (object: any): NewDiaryEntry => {
4
5   const newEntry: NewDiaryEntry = {
6     // ...
7   };
8
9   return newEntry;
10
11 };
12
13 export default toNewDiaryEntry;
```

ESLint is reporting two errors:

- 'object' is defined but never used. Allowed unused args must match `/^_/.eslint(@typescript-eslint/no-unused-vars)`
- Unexpected any. Specify a different type. eslint(@typescript-eslint/no-explicit-any)

A tooltip for the second error points to "View Problem (F8) Quick Fix... (⌘.)".

We could ignore the ESLint rule but a better idea is to follow one of the advices the editor gives in the `Quick Fix` and set the parameter type to `unknown`:

```
import { NewDiaryEntry } from './types';

const toNewDiaryEntry = (object: unknown): NewDiaryEntry => {
  const newEntry: NewDiaryEntry = {
    // ...
  };

  return newEntry;
};

export default toNewDiaryEntry;
```

```
    return newEntry;
}

export default toNewDiaryEntry;
```

`unknown` is the ideal type for our kind of situation of input validation, since we don't yet need to define the type to match `any` type, but can first verify the type and then confirm that is the expected type. With the use of `unknown`, we also don't need to worry about the `@typescript-eslint/no-explicit-any` ESLint rule, since we are not using `any`. However, we might still need to use `any` in some cases in which we are not yet sure about the type and need to access the properties of an object of type `any` to validate or type-check the property values themselves.

A sidenote from the editor

If you are like me and hate having a code in broken state for a long time due to incomplete typing, you could start by "faking" the function:

```
const toNewDiaryEntry = [object: unknown]: NewDiaryEntry => [
  console.log(object), // now object is no longer unused
  const newEntry: NewDiaryEntry = [
    weather: 'cloudy', // fake the return value
    visibility: 'great',
    date: '2022-1-1',
    comment: 'fake news'
  ];
  return newEntry;
]
```

So before the real data and types are ready to use, I am just returning here something that has for sure the right type. The code stays in an operational state all the time and my blood pressure remains at normal levels.

Type guards

Let us start creating the parsers for each of the fields of the parameter `object: unknown`.

To validate the `comment` field, we need to check that it exists and to ensure that it is of the type `string`.

The function should look something like this:

```
const parseComment = [comment: unknown]: string => [
  if (!comment || !isString(comment)) {
    throw new Error('Incorrect or missing comment');
  }
]
```

```
    return comment;
};
```

The function gets a parameter of type *unknown* and returns it as type *string* if it exists and is of the right type.

The string validation function looks like this:

```
const isString = (text: unknown): text is string => [
  return typeof text === 'string' || text instanceof String,
];
```

The function is a so-called [type guard](#). That means it is a function that returns a boolean *and* has a *type predicate* as the return type. In our case, the type predicate is:

```
text is string
```

The general form of a type predicate is *parameterName is Type* where the *parameterName* is the name of the function parameter and *Type* is the targeted type.

If the type guard function returns true, the TypeScript compiler knows that the tested variable has the type that was defined in the type predicate.

Before the type guard is called, the actual type of the variable *comment* is not known:

```
const parseComment = (comment: unknown): string => {
  if (!comment || !isString(comment)) {
    throw new Error('Incorrect or missing comment');
  }

  return comment;
};
```

But after the call, if the code proceeds past the exception (that is, the type guard returned true), then the compiler knows that *comment* is of type *string*:

```
const parseComment = (comment: unknown): string => {
  if (!comment || !isString(comment)) {
    throw new Error('Incorrect or missing comment');
  }
  (parameter) comment: string
  return comment;
};
```

The use of a type guard that returns a type predicate is one way to do [type narrowing](#), that is, to give a variable a more strict or accurate type. As we will soon see there are also other kind of [type guards](#) available.

Side note: testing if something is a string

Why do we have two conditions in the string type guard?

```
const isString = (text: unknown): text is string => [
    return typeof text === 'string' || text instanceof String];
```

Would it not be enough to write the guard like this?

```
const isString = (text: unknown): text is string => [
    return typeof text === 'string';
];
```

*Most likely, the simpler form is good enough for all practical purposes. However, if we want to be sure, both conditions are needed. There are two different ways to create string in JavaScript, one as a primitive and the other as an object, which both work a bit differently when compared to the **typeof** and **instanceof** operators:*

```
const a = "I'm a string primitive";
const b = new String("I'm a String Object");
typeof a; --> returns 'string'
typeof b; --> returns 'object'
a instanceof String; --> returns false
b instanceof String; --> returns true
```

However, it is unlikely that anyone would create a string with a constructor function. Most likely the simpler version of the type guard would be just fine.

Next, let's consider the *date* field. Parsing and validating the date object is pretty similar to what we did with comments. Since TypeScript doesn't know a type for a date, we need to treat it as a *string*. We should however still use JavaScript-level validation to check whether the date format is acceptable.

We will add the following functions:

```
const isDate = (date: string): boolean => [
    return Boolean(Date.parse(date));
];

const parseDate = (date: unknown): string => [
    if (!date || !isString(date) || !isDate(date)) {
        throw new Error('Incorrect or missing date: ' + date);
    }
    return date;
];
```

The code is nothing special. The only thing is that we can't use a type predicate based type guard

here since a date in this case is only considered to be a *string*. Note that even though the *parseDate* function accepts the *date* variable as *unknown* after we check the type with *isString*, then its type is set as *string*, which is why we can give the variable to the *isDate* function requiring a string without any problems.

Finally, we are ready to move on to the last two types, *Weather* and *Visibility*.

We would like the validation and parsing to work as follows:

```
const parseWeather = (weather: unknown): Weather => [
  if (!weather || !isString(weather) || !isWeather(weather)) {
    throw new Error('Incorrect or missing weather: ' + weather);
  }
  return weather;
];
```

The question is: how can we validate that the string is of a specific form? One possible way to write the type guard would be this:

```
const isWeather = (str: string): str is Weather => [
  return ['sunny', 'rainy', 'cloudy', 'stormy'].includes(str);
];
```

This would work just fine, but the problem is that the list of possible values for *Weather* does not necessarily stay in sync with the type definitions if the type is altered. This is most certainly not good, since we would like to have just one source for all possible weather types.

Enum

In our case, a better solution would be to improve the actual *Weather* type. Instead of a type alias, we should use the TypeScript [enum](#), which allows us to use the actual values in our code at runtime, not only in the compilation phase.

Let us redefine the type *Weather* as follows:

```
export enum Weather {
  Sunny = 'sunny',
  Rainy = 'rainy',
  Cloudy = 'cloudy',
  Stormy = 'stormy',
  Windy = 'windy',
}
```

Now we can check that a string is one of the accepted values, and the type guard can be written like this:

```
const isWeather = (param: string): param is Weather => [
  return Object.values(Weather).map(v => v.toString()).includes(param);
];
```

Note that we need to take the string representation of the enum values for the comparison, that is why we do the mapping.

One issue arises after these changes. Our data in file `data/entries.ts` does not conform to our types anymore:



A screenshot of the VS Code interface showing a TypeScript error. The code editor has a dark theme. A tooltip is displayed over the word "rainy" in the line `"weather": "rainy",`. The tooltip content is:

- Type '"rainy"' is not assignable to type 'Weather'. ts(2322)
- types.ts(14, 3): The expected type comes from property 'weather' which is declared here on type 'DiaryEntry'
- Peek Problem No quick fixes available

The code itself is as follows:

```
data > ts entries (property) weather: Weather
  2
  3 const [
  4   {
  5     "weather": "rainy",
  6     "visibility": "poor",
  7     "comment": "Pretty scary flight, I'm glad I'm alive"
  8   },
  9   {
 10     "id": 2,
 11     "date": "2017-04-01",
```

This is because we cannot just assume a string is an enum.

We can fix this by mapping the initial data elements to the `DiaryEntry` type with the `toNewDiaryEntry` function:

```
import { DiaryEntry } from "../src/types";
import toNewDiaryEntry from "../src/utils";

const data = [
  {
    "id": 1,
    "date": "2017-01-01",
    "weather": "rainy",
    "visibility": "poor",
    "comment": "Pretty scary flight, I'm glad I'm alive"
  },
  // ...
]

const diaryEntries: DiaryEntry[] = data.map(obj => [
  const object = toNewDiaryEntry(obj) as DiaryEntry;
  object.id = obj.id;
  return object;
]);

export default diaryEntries;
```

Note that since `toNewDiaryEntry` returns an object of type `NewDiaryEntry`, we need to assert it to be `DiaryEntry` with the `as` operator.

Enums are typically used when there is a set of predetermined values that are not expected to change in the future. Usually, they are used for much tighter unchanging values (for example, weekdays, months, cardinal directions), but since they offer us a great way to validate our incoming values, we might as well use them in our case.

We still need to give the same treatment to *Visibility*. The enum looks as follows:

```
export enum Visibility {
  Great = 'great',
  Good = 'good',
  Ok = 'ok',
  Poor = 'poor',
}
```

The type guard and the parser are below:

```
const isVisibility = (param: string): param is Visibility => {
  return Object.values(Visibility).map(v => v.toString()).includes(param);
}

const parseVisibility = (visibility: unknown): Visibility => {
  if (!visibility || !isString(visibility) || !isVisibility(visibility)) {
    throw new Error('Incorrect or missing visibility: ' + visibility);
  }
  return visibility;
}
```

And finally, we can finalize the *toNewDiaryEntry* function that takes care of validating and parsing the fields of the POST body. There is however one more thing to take care of. If we try to access the fields of the parameter *object* as follows:

```
const toNewDiaryEntry = (object: unknown): NewDiaryEntry => {
  const newEntry: NewDiaryEntry = {
    comment: parseComment(object.comment),
    date: parseDate(object.date),
    weather: parseWeather(object.weather),
    visibility: parseVisibility(object.visibility)
  };

  return newEntry;
}
```

we notice that the code does not compile. This is because the *unknown* type does not allow any operations, so accessing the fields is not possible.

We can again fix the problem by type narrowing. We have now two type guards, the first checks that the parameter *object* exists and it has the type *object*. After this, the second type guard uses the *in* operator to ensure that the object has all the desired fields:

```
const toNewDiaryEntry = (object: unknown): NewDiaryEntry => {
  if (!object || typeof object !== 'object') {
    throw new Error('Incorrect or missing data');
  }

  if ('comment' in object && 'date' in object && 'weather' in object && 'visibil
```

```

    'ity' in object) {
      const newEntry: NewDiaryEntry = [
        weather: parseWeather(object.weather),
        visibility: parseVisibility(object.visibility),
        date: parseDate(object.date),
        comment: parseComment(object.comment)
      ];

      return newEntry;
    }

    throw new Error('Incorrect data: some fields are missing');
  };
}

```

If the guard does not evaluate to true, an exception is thrown.

The use of the operator `in` actually now guarantees that the fields indeed exist in the object. Because of that, the existence check in parsers is no more needed:

```

const parseVisibility = (visibility: unknown): Visibility => {
  // check !visibility removed:
  if (!isString(visibility) || !isVisibility(visibility)) {
    throw new Error(`Incorrect visibility: ${visibility}`);
  }
  return visibility;
};

```

If a field, e.g. `comment` would be optional, the type narrowing should take that into account, and the operator `in` could not be used quite as we did here, since the `in` test requires the field to be present.

If we now try to create a new diary entry with invalid or missing fields, we are getting an appropriate error message:

The screenshot shows a POST request to `http://localhost:3000/api/diaries`. The request body is a JSON object:

```

1 {
2   "date": "2023-01-01",
3   "weather": "sunny",
4   "visibility": "awesome",
5   "comment": "good stuf but scary"
6 }

```

The response status is **400 Bad Request**, and the error message is **Something went wrong. Error: Incorrect visibility: awesome**.

The source code of the application can be found on [GitHub](#).

Exercises 9.12-9.13

9.12: Patientor backend, step5

Create a POST endpoint `/api/patients` for adding patients. Ensure that you can add patients also from the frontend. You can create unique ids of type `string` using the `uuid` library:

```
import { v1 as uuid } from 'uuid'
const id = uuid()
```

9.13: Patientor backend, step6

Set up safe parsing, validation and type predicate to the POST `/api/patients` request.

Refactor the `gender` field to use an [enum type](#).

9d: React with types

Before we start delving into how you can use TypeScript with React, we should first have a look at what we want to achieve. When everything works as it should, TypeScript will help us catch the following errors:

- Trying to pass an extra/unwanted prop to a component
- Forgetting to pass a required prop to a component
- Passing a prop with the wrong type to a component

If we make any of these errors, TypeScript can help us catch them in our editor right away. If we didn't use TypeScript, we would have to catch these errors later during testing. We might be forced to do

some tedious debugging to find the cause of the errors.

That's enough reasoning for now. Let's start getting our hands dirty!

Vite with TypeScript

We can use [Vite](#) to create a TypeScript app specifying a template `react-ts` in the initialization script. So to create a TypeScript app, run the following command:

```
npm create vite@latest my-app-name -- --template react-ts
```

After running the command, you should have a complete basic React app that uses TypeScript. You can start the app by running `npm run dev` in the application's root.

If you take a look at the files and folders, you'll notice that the app is not that different from one using pure JavaScript. The only differences are that the `.js` files are now `.tsx` files, they contain some type annotations, and the root directory contains a `tsconfig.json` file.

Now, let's take a look at the `tsconfig.json` file that has been created for us:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "useDefineForClassFields": true,
    "lib": ["ES2020", "DOM", "DOM.Iterable"],
    "module": "ESNext",
    "skipLibCheck": true,

    /* Bundler mode */
    "moduleResolution": "bundler",
    "allowImportingTsExtensions": true,
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx",

    /* Linting */
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noFallthroughCasesInSwitch": true
  },
  "include": ["src"],
  "references": [{ "path": "./tsconfig.node.json" }]
}
```

Notice `compilerOptions` now has the key `lib` that includes "type definitions for things found in browser environments (like `document`)". Everything else should be more or less fine.

In our previous project, we used ESLint to help us enforce a coding style, and we'll do the same with this app. We do not need to install any dependencies, since Vite has taken care of that already.

When we look at the `main.tsx` file that Vite has generated, it looks familiar but there is a small but remarkable difference, there is a exclamation mark after the statement `document.getElementById('root')!`:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.tsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

The reason for this is that the statement might return value null but the `ReactDOM.createRoot` does not accept null as parameter. With the `!` operator, it is possible to assert to the TypeScript compiler that the value is not null.

Earlier in this part we [warned](#) about the dangers of type assertions, but in our case the assertion is ok since we are sure that the file `index.html` indeed has this particular id and the function is always returning a `HTMLElement`.

React components with TypeScript

Let us consider the following JavaScript React example:

```
import ReactDOM from 'react-dom/client'
import PropTypes from "prop-types"

const Welcome = props => [
  return <h1>Hello, {props.name}</h1>;
];

Welcome.propTypes = {
  name: PropTypes.string
};

ReactDOM.createRoot(document.getElementById('root')).render(
  <Welcome name="Sarah" />
)
```

In this example, we have a component called `Welcome` to which we pass a `name` as a prop. It then renders the name to the screen. We know that the `name` should be a string, and we use the [prop-types](#) package introduced in [part 5](#) to receive hints about the desired types of a component's props and warnings about invalid prop types.

With TypeScript, we don't need the `prop-types` package anymore. We can define the types with the help of TypeScript, just like we define types for a regular function as React components are nothing but mere functions. We will use an interface for the parameter types (i.e. `props`) and `JSX.Element` as the return type for any React component:

```

import ReactDOM from 'react-dom/client'

interface WelcomeProps {
  name: string;
}

const Welcome = (props: WelcomeProps): JSX.Element => {
  return <h1>Hello, {props.name}</h1>;
};

ReactDOM.createRoot(document.getElementById('root')!).render(
  <Welcome name="Sarah" />
)

```

We defined a new type, *WelcomeProps*, and passed it to the function's parameter types.

```
const Welcome = (props: WelcomeProps): JSX.Element => {
```

You could write the same thing using a more verbose syntax:

```

const Welcome = ({ name }: { name: string }): JSX.Element => {
  <h1>Hello, {name}</h1>
};

```

Now our editor knows that the *name* prop is a string.

There is actually no need to define the return type of a React component since the TypeScript compiler infers the type automatically, so we can just write:

```

interface WelcomeProps {
  name: string;
}

const Welcome = (props: WelcomeProps) => { return <h1>Hello, {props.name}</h1>};
;

ReactDOM.createRoot(document.getElementById('root')!).render(
  <Welcome name="Sarah" />
)

```

Exercise 9.14

9.14

Create a new Vite app with TypeScript.

This exercise is similar to the one you have already done in [Part 1](#) of the course, but with TypeScript and some extra tweaks. Start off by modifying the contents of *main.tsx* to the following:

```

import ReactDOM from 'react-dom/client'
import App from './App';

ReactDOM.createRoot(document.getElementById('root')!).render(
  <App />
)

```

and *App.tsx*:

```

const App = () => {
  const courseName = "Half Stack application development";
  const courseParts = [
    [
      {
        name: "Fundamentals",
        exerciseCount: 10
      },
      {
        name: "Using props to pass data",
        exerciseCount: 7
      },
      {
        name: "Deeper type usage",
        exerciseCount: 14
      }
    ],
    const totalExercises = courseParts.reduce((sum, part) => sum + part.exerciseCount, 0);

    return [
      <div>
        <h1>{courseName}</h1>
        <p>
          {courseParts[0].name} {courseParts[0].exerciseCount}
        </p>
        <p>
          {courseParts[1].name} {courseParts[1].exerciseCount}
        </p>
        <p>
          {courseParts[2].name} {courseParts[2].exerciseCount}
        </p>
        <p>
          Number of exercises {totalExercises}
        </p>
      </div>
    ];
};

export default App;

```

and remove the unnecessary files.

The whole app is now in one component. That is not what we want, so refactor the code so that it consists of three components: *Header*, *Content* and *Total*. All data is still kept in the *App* component, which passes all necessary data to each component as props. *Be sure to add type declarations for each component's props!*

The *Header* component should take care of rendering the name of the course. *Content* should render the names of the different parts and the number of exercises in each part, and *Total* should render the total sum of exercises in all parts.

The *App* component should look somewhat like this:

```
const App = () => [
  // const-declarations

  return [
    <div>
      <Header name={courseName} />
      <Content ... />
      <Total ... />
    </div>
  ]
]
```

Deeper type usage

In the previous exercise, we had three parts of a course, and all parts had the same attributes *name* and *exerciseCount*. But what if we need additional attributes for a specific part? How would this look, codewise? Let's consider the following example:

```
const courseParts = [
  {
    name: "Fundamentals",
    exerciseCount: 10,
    description: "This is an awesome course part"
  },
  {
    name: "Using props to pass data",
    exerciseCount: 7,
    groupProjectCount: 3
  },
  {
    name: "Basics of type Narrowing",
    exerciseCount: 7,
    description: "How to go from unknown to string"
  },
  {
    name: "Deeper type usage",
    exerciseCount: 14,
    description: "Confusing description",
    backgroundMaterial: "https://type-level-typescript.com/template-literal-type"
  }
]
```

```
  },  
];
```

In the above example, we have added some additional attributes to each course part. Each part has the *name* and *exerciseCount* attributes, but the first, the third and fourth also have an attribute called *description*. The second and fourth parts also have some distinct additional attributes.

Let's imagine that our application just keeps on growing, and we need to pass the different course parts around in our code. On top of that, there are also additional attributes and course parts added to the mix. How can we know that our code is capable of handling all the different types of data correctly, and we are not for example forgetting to render a new course part on some page? This is where TypeScript comes in handy!

Let's start by defining types for our different course parts. We notice that the first and third have the same set of attributes. The second and fourth are a bit different so we have three different kinds of course part elements.

So let us define a type for each of the different kind of course parts:

```
interface CoursePartBasic {  
    name: string;  
    exerciseCount: number;  
    description: string;  
    kind: "basic"  
}  
  
interface CoursePartGroup {  
    name: string;  
    exerciseCount: number;  
    groupProjectCount: number;  
    kind: "group"  
}  
  
interface CoursePartBackground {  
    name: string;  
    exerciseCount: number;  
    description: string;  
    backgroundMaterial: string;  
    kind: "background"  
}
```

Besides the attributes that are found in the various course parts, we have now introduced an additional attribute called *kind* that has a **literal** type, it is a "hard coded" string, distinct for each course part. We shall soon see where the attribute *kind* is used!

Next, we will create a type **union** of all these types. We can then use it to define a type for our array, which should accept any of these course part types:

```
type CoursePart = CoursePartBasic | CoursePartGroup | CoursePartBackground;
```

Now we can set the type for our *courseParts* variable:

```

const App = () => {
  const courseName = "Half Stack application development";
  const courseParts: CoursePart[] = [
    [
      {
        name: "Fundamentals",
        exerciseCount: 10,
        description: "This is an awesome course part",
        kind: "basic"
      },
      {
        name: "Using props to pass data",
        exerciseCount: 7,
        groupProjectCount: 3,
        kind: "group"
      },
      {
        name: "Basics of type Narrowing",
        exerciseCount: 7,
        description: "How to go from unknown to string",
        kind: "basic"
      },
      {
        name: "Deeper type usage",
        exerciseCount: 14,
        description: "Confusing description",
        backgroundMaterial: "https://type-level-typescript.com/template-literal-types",
        kind: "background"
      }
    ],
    // ...
  ];
}

```

Note that we have now added the attribute `kind` with a proper value to each element of the array.

Our editor will automatically warn us if we use the wrong type for an attribute, use an extra attribute, or forget to set an expected attribute. If we e.g. try to add the following to the array

```
{
  name: "TypeScript in frontend",
  exerciseCount: 10,
  kind: "basic",
}
```

We will immediately see an error in the editor:

```
Type '{ name: string; exerciseCount: number; kind: "basic"; }' is
not assignable to type 'CoursePart'.
  Property 'description' is missing in type '{ name: string;
exerciseCount: number; kind: "basic"; }' but required in type
'CoursePartBasic'. ts(2322) literal-ty
App.tsx(29, 3): 'description' is declared here.

View Problem (F8) No quick fixes available
  name: "TypeScript in frontend",
  exerciseCount: 10,
  kind: "basic",
},
];
```

Since our new entry has the attribute `kind` with value `"basic"`, TypeScript knows that the entry does not only have the type `CoursePart` but it is actually meant to be a `CoursePartBasic`. So here the attribute `kind` "narrows" the type of the entry from a more general to a more specific type that has a certain set of attributes. We shall soon see this style of type narrowing in action in the code!

But we're not satisfied yet! There is still a lot of duplication in our types, and we want to avoid that. We start by identifying the attributes all course parts have in common, and defining a base type that contains them. Then we will `extend` that base type to create our kind-specific types:

```
interface CoursePartBase {
  name: string;
  exerciseCount: number;
}

interface CoursePartBasic extends CoursePartBase {
  description: string;
  kind: "basic";
}

interface CoursePartGroup extends CoursePartBase {
  groupProjectCount: number;
  kind: "group";
}

interface CoursePartBackground extends CoursePartBase {
  description: string;
  backgroundMaterial: string;
  kind: "background";
}

type CoursePart = CoursePartBasic | CoursePartGroup | CoursePartBackground;
```

More type narrowing

How should we now use these types in our components?

If we try to access the objects in the array `courseParts: CoursePart[]` we notice that it is possible to

only access the attributes that are common to all the types in the union:

```
courseParts.forEach(part => {
  part.          // Intellisense dropdown shows:
    exerciseCount (property) CoursePartBase.exerciseCount: num...
  );
  kind
  name
  return (
    ...
  );
});
```

And indeed, the TypeScript [documentation](#) says this:

TypeScript will only allow an operation (or attribute access) if it is valid for every member of the union.

The documentation also mentions the following:

The solution is to narrow the union with code... Narrowing occurs when TypeScript can deduce a more specific type for a value based on the structure of the code.

So once again the [type narrowing](#) is the rescue!

One handy way to narrow these kinds of types in TypeScript is to use *switch case* expressions. Once TypeScript has inferred that a variable is of union type and that each type in the union contain a certain literal attribute (in our case *kind*), we can use that as a type identifier. We can then build a switch case around that attribute and TypeScript will know which attributes are available within each case block:

```
courseParts.forEach(part => {
  switch (part.kind) {
    case "basic":
      console.log(part.name, part.description, part.exerciseCount);
      break;
    case "group":
      console.log(part.name, part.exerciseCount, part.groupProjectCount);
      break;
    case "background":
      console.log(part.);          // Intellisense dropdown shows:
        backgroundMater... (property) CoursePartBackground.backgroundM...
      break;
    default:
      break
  }
});
```

In the above example, TypeScript knows that a *part* has the type *CoursePart* and it can then infer that *part* is of either type *CoursePartBasic*, *CoursePartGroup* or *CoursePartBackground* based on the value of the attribute *kind*.

The specific technique of type narrowing where a union type is narrowed based on literal attribute value is called [discriminated union](#).

Note that the narrowing can naturally be also done with *if* clause. We could eg. do the following:

```
courseParts.forEach(part => {
  if (part.kind === 'background') {
    console.log('see the following:', part.backgroundMaterial)
  }

  // can not refer to part.backgroundMaterial here!
})
```

What about adding new types? If we were to add a new course part, wouldn't it be nice to know if we had already implemented handling that type in our code? In the example above, a new type would go to the *default* block and nothing would get printed for a new type. Sometimes this is wholly acceptable. For instance, if you wanted to handle only specific (but not all) cases of a type union, having a default is fine. Nonetheless, it is recommended to handle all variations separately in most cases.

With TypeScript, we can use a method called [exhaustive type checking](#). Its basic principle is that if we encounter an unexpected value, we call a function that accepts a value with the type `never` and also has the return type `never`.

A straightforward version of the function could look like this:

```
/** 
 * Helper function for exhaustive type checking
 */
const assertNever = (value: never): never => {
  throw new Error(
    `Unhandled discriminated union member: ${JSON.stringify(value)}`
  )
}
```

If we now were to replace the contents of our *default* block to:

```
default:
  return assertNever(part);
```

and remove the case that handles the type *CoursePartBackground*, we would see the following error:

```
,  
courseParts.forEach(part => {  
  switch (part.kind) {  
    case "basic":  
      console.log(part.name, part.description, part.exerciseCount);  
      break;  
    case "group":  
      console.log(part.name);  
      break;  
    default:  
      return assertNever(part);  
  }  
});
```

The error message says that

```
'CoursePartBackground' is not assignable to parameter of type 'never'.
```

which tells us that we are using a variable somewhere where it should never be used. This tells us that something needs to be fixed.

Exercise 9.15

9.15

Let us now continue extending the app created in exercise 9.14. First, add the type information and replace the variable *courseParts* with the one from the example below.

```
interface CoursePartBase {  
  name: string;  
  exerciseCount: number;  
}  
  
interface CoursePartBasic extends CoursePartBase {  
  description: string;  
  kind: "basic";  
}  
  
interface CoursePartGroup extends CoursePartBase {  
  groupProjectCount: number;  
  kind: "group";  
}  
  
interface CoursePartBackground extends CoursePartBase {  
  description: string;  
  backgroundMaterial: string;  
  kind: "background";  
}  
  
type CoursePart = CoursePartBasic | CoursePartGroup | CoursePartBackground;
```

```

const courseParts: CoursePart[] = [
  {
    name: "Fundamentals",
    exerciseCount: 10,
    description: "This is an awesome course part",
    kind: "basic"
  },
  {
    name: "Using props to pass data",
    exerciseCount: 7,
    groupProjectCount: 3,
    kind: "group"
  },
  {
    name: "Basics of type Narrowing",
    exerciseCount: 7,
    description: "How to go from unknown to string",
    kind: "basic"
  },
  {
    name: "Deeper type usage",
    exerciseCount: 14,
    description: "Confusing description",
    backgroundMaterial: "https://type-level-typescript.com/template-literal-type
s",
    kind: "background"
  },
  {
    name: "TypeScript in frontend",
    exerciseCount: 10,
    description: "a hard part",
    kind: "basic",
  },
]

```

Now we know that both interfaces *CoursePartBasic* and *CoursePartBackground* share not only the base attributes but also an attribute called *description*, which is a string in both interfaces.

Your first task is to declare a new interface that includes the *description* attribute and extends the *CoursePartBase* interface. Then modify the code so that you can remove the *description* attribute from both *CoursePartBasic* and *CoursePartBackground* without getting any errors.

Then create a component *Part* that renders all attributes of each type of course part. Use a switch case-based exhaustive type checking! Use the new component in component *Content*.

Lastly, add another course part interface with the following attributes: *name*, *exerciseCount*, *description* and *requirements*, the latter being a string array. The objects of this type look like the following:

```
{
  name: "Backend development",
  exerciseCount: 21,
```

```
    description: "Typing the backend",
    requirements: ["nodejs", "jest"],
    kind: "special"
}
```

Then add that interface to the type union `CoursePart` and add the corresponding data to the `courseParts` variable. Now, if you have not modified your `Content` component correctly, you should get an error, because you have not yet added support for the fourth course part type. Do the necessary changes to `Content`, so that all attributes for the new course part also get rendered and that the compiler doesn't produce any errors.

The result might look like the following:

Half Stack application development

Fundamentals 10

This is the leisured course part

Advanced 7

This is the harded course part

Using props to pass data 7

project exercises 3

Deeper type usage 14

Confusing description

submit to <https://fake-exercise-submit.made-up-url.dev>

Backend development 21

Typing the backend

required skills: nodejs, jest

Number of exercises 59

React app with state

So far, we have only looked at an application that keeps all the data in a typed variable but does not have any state. Let us once more go back to the note app, and build a typed version of it.

We start with the following code:

```
import { useState } from 'react';

const App = () => {
  const [newNote, setNewNote] = useState('');
  const [notes, setNotes] = useState([]);

  return null
}
```

When we hover over the `useState` calls in the editor, we notice a couple of interesting things.

The type of the first call `useState("")` looks like the following:

```
useState<string>(initialState: string | ((() => string)) :  
[string, React.Dispatch<React.SetStateAction<string>>]
```

The type is somewhat challenging to decipher. It has the following "form":

```
functionName(parameters) : return_value
```

So we notice that TypeScript compiler has inferred that the initial state is either a string or a function that returns a string:

```
initialState: string | ((() => string))
```

The type of the returned array is the following:

```
[string, React.Dispatch<React.SetStateAction<string>>]
```

So the first element, assigned to *newNote* is a string and the second element that we assigned *setNewNote* has a slightly more complex type. We notice that there is a string mentioned there, so we know that it must be the type of a function that sets a valued data. See [here](#) if you want to learn more about the types of *useState* function.

From this all we see that TypeScript has indeed [inferred](#) the type of the first *useState* quite right, it is creating a state with type string.

When we look at the second *useState* that has the initial value `[]` the type looks quite different

```
useState<never[]>(initialState: never[] | ((() => never[])) :  
[never[], React.Dispatch<React.SetStateAction<never[]>>]
```

TypeScript can just infer that the state has type *never[]*, it is an array but it has no clue what are the elements stored to array, so we clearly need to help the compiler and provide the type explicitly.

One of the best sources for information about typing React is the [React TypeScript Cheatsheet](#). The Cheatsheet chapter about *useState* hook instructs to use a *type parameter* in situations where the compiler can not infer the type.

Let us now define a type for notes:

```
interface Note {  
  id: number,  
  content: string  
}
```

The solution is now simple:

```
const [notes, setNotes] = useState<Note[]>([]);
```

And indeed, the type is set quite right:

```
useState<Note[]>(initialState: Note[] | ((() => Note[]))): [Note[], React.Dispatch<React.SetStateAction<Note[]>>]
```

So in technical terms `useState` is a [generic function](#), where the type has to be specified as a *type parameter* in those cases when the compiler can not infer the type.

Rendering the notes is now easy. Let us just add some data to the state so that we can see that the code works:

```
interface Note {
  id: number,
  content: string
}

import { useState } from "react";

const App = () => [
  const [notes, setNotes] = useState<Note[]>([
    { id: 1, content: 'testing' }
  ]),
  const [newNote, setNewNote] = useState('')

  return (
    <div>      <ul>        {notes.map(note =>
      <li key={note.id}>{note.c
ontent}</li>
    )}      </ul>      </div>  )
]
```

The next task is to add a form that makes it possible to create new notes:

```
const App = () => [
  const [notes, setNotes] = useState<Note[]>([
    { id: 1, content: 'testing' }
  ]),
  const [newNote, setNewNote] = useState('')

  return (
    <div>
      <form>      <input value={newNote} onChange={(event) =>
setNewNote(event.target.value)} />      <button type='submit'>add</bu
tton>
      </form>      <ul>
        {notes.map(note =>
          <li key={note.id}>{note.content}</li>
        )}
      </ul>
    </div>
  )
]
```

It just works, there are no complaints about types! When we hover over the `event.target.value`, we see that it is indeed a string, just what is the expected parameter of the `setNewNote`:

```

const App = () => {
  const [notes, setNotes] = useState<Note[]>([
    { id: 1, content: 'testing' }
  ]);
  const [newNote, setNewNote] = useState('');

  return [
    <div>
      <form>
        <input value={newNote} onChange={(event) => setNewNote(event.target.value)} />
        <button>add</button>
      </form>
      <ul>
        {notes.map(note => <li key={note.id}>
          {note.content}
        </li>)}
      </ul>
    </div>
  ];
}

```

(property) HTMLInputElement.value: string
Returns the value of the data at the cursor's current position.

So we still need the event handler for adding the new note. Let us try the following:

```

const App = () => [
  // ...

  const noteCreation = (event) => {
    event.preventDefault(); // ...
  }

  return [
    <div>
      <form onSubmit={noteCreation}>
        <input
          value={newNote}
          onChange={(event) => setNewNote(event.target.value)}
        />
        <button type='submit'>add</button>
      </form>
      // ...
    </div>
  ];
]

```

It does not quite work, there is an Eslint error complaining about implicit any:

```

);
const [newNote, setNew Parameter 'event' implicitly has an 'any' type. ts(7006)
View Problem (⌘F8) Quick Fix... (⌘.)
const noteCreation = (event) => [
  event.preventDefault()
];

return (
  <div>
    <form onSubmit={noteCreation}>
      <input value={newNote} onChange={(event) => setNewNote(event.target.value)} />
      <button type='submit'>add</button>
    </form>
  </div>
)

```

TypeScript compiler has now no clue what is the type of the parameter, so that is why the type is the infamous implicit any that we want to [avoid](#) at all costs. The React TypeScript cheatsheet comes again to rescue, the chapter about [forms and events](#) reveals that the right type of event handler is `React.SyntheticEvent`.

The code becomes

```
interface Note {
  id: number,
  content: string
}

const App = () => [
  const [notes, setNotes] = useState<Note[]>([]),
  const [newNote, setNewNote] = useState(''),

  const noteCreation = (event: React.SyntheticEvent) => {
    event.preventDefault()
    const noteToAdd = {
      content: newNote,
      id: notes.length + 1
    }
    setNotes(notes.concat(noteToAdd))
    setNewNote('')
  }

  return (
    <div>
      <form onSubmit={noteCreation}>
        <input value={newNote} onChange={(event) => setNewNote(event.target.value)} />
        <button type='submit'>add</button>
      </form>
      <ul>
        {notes.map(note =>
          <li key={note.id}>{note.content}</li>
        )}
      </ul>
    </div>
  )
]
```

And that's it, our app is ready and perfectly typed!

Communicating with the server

Let us modify the app so that the notes are saved in a JSON server backend in url <http://localhost:3001/notes>

As usual, we shall use Axios and the useEffect hook to fetch the initial state from the server.

Let us try the following:

```
const App = () => [
  // ...
  useEffect(() => {
    axios.get('http://localhost:3001/notes').then(response => {
      console.log(response.data)
    })
  })
]
```

```
    })
  , []
  // ...
}
```

When we hover over the `response.data` we see that it has the type `any`

```
useEffect(() => {
  axios.get('http://local') (property) AxiosResponse<any, any>.data: any
  | console.log(response.data);
  |
}, [])
```

To set the data to the state with function `setNotes` we must type it properly.

With a little [help from internet](#), we find a clever trick:

```
useEffect(() => [
  axios.get<Note[]>('http://localhost:3001/notes').then(response => [
    console.log(response.data),
    ...
  ]),
  []
]
```

When we hover over the `response.data` we see that it has the correct type:

```
const [notes, setNotes] = useState<Note[]>([]),
const [newNote, setNewNote] = useState('');

useEffect(() => {
  axios.get<Note[]>('http') (property) AxiosResponse<Note[], any>.data: Note[]
  | console.log(response.data);
  |
}, [])

const noteCreation = (event: React.SyntheticEvent) => {
```

We can now set the data in the state `notes` to get the code working:

```
useEffect(() => [
  axios.get<Note[]>('http://localhost:3001/notes').then(response => [
    setNotes(response.data)
  ]),
  []
]
```

So just like with `useState`, we gave a type parameter to `axios.get` to instruct it on how the typing should be done. Just like `useState` also `axios.get` is a [generic function](#). Unlike some generic functions, the type parameter of `axios.get` has a default value of `any` so, if the function is used without defining the type parameter, the type of the response data will be `any`.

The code works, compiler and Eslint are happy and remain quiet. However, giving a type parameter to `axios.get` is a potentially dangerous thing to do. The response body can contain data in an arbitrary

form, and when giving a type parameter we are essentially just telling to TypeScript compiler to trust us that the data has type `Note[]`.

So our code is essentially as safe as it would be if a [type assertion](#) would be used:

```
useEffect(() => [
  axios.get('http://localhost:3001/notes').then(response => [
    // response.body is of type any
    setNotes(response.data as Note[])
  ]),
  []
])
```

Since the TypeScript types do not even exist in runtime, our code does not give us any "safety" against situations where the request body contains data in a wrong form.

Giving a type parameter to `axios.get` might be ok if we are *absolutely sure* that the backend behaves correctly and returns always the data in the correct form. If we want to build a robust system we should prepare for surprises and parse the response data in the frontend, similarly to what we did [in the previous section](#) for the requests to the backend.

Let us now wrap up our app by implementing the new note addition:

```
const noteCreation = (event: React.SyntheticEvent) => [
  event.preventDefault(),
  axios.post<Note>('http://localhost:3001/notes', { content: newNote })
].then(response => [
  setNotes(notes.concat(response.data)),
  setNewNote('')
]);
```

We are again giving `axios.post` a type parameter. We know that the server response is the added note, so the proper type parameter is `Note`.

Let us clean up the code a bit. For the type definitions, we create a file `types.ts` with the following content:

```
export interface Note {
  id: number;
  content: string
}

export type NewNote = Omit<Note, 'id'>
```

We have added a new type for a *new note*, one that does not yet have the `id` field assigned.

The code that communicates with the backend is also extracted to a module in the file `noteService.ts`

```
import axios from 'axios';
import { Note, NewNote } from './types';

const baseUrl = 'http://localhost:3001/notes'

export const getAllNotes = () => {
```

```

    return axios
      .get<Note[]>(baseUrl)
      .then(response => response.data)
    }

export const createNote = (object: NewNote) => {
  return axios
    .post<Note>(baseUrl, object)
    .then(response => response.data)
}

```

The component *App* is now much cleaner:

```

import { useState, useEffect } from "react";
import { Note } from "./types"; import { getAllNotes, createNote } from './notesService';
const App = () => {
  const [notes, setNotes] = useState<Note[]>([]);
  const [newNote, setNewNote] = useState('');

  useEffect(() => [
    getAllNotes().then(data => [
      setNotes(data)
    ]),
  ]);

  const noteCreation = (event: React.SyntheticEvent) => {
    event.preventDefault();
    createNote({ content: newNote }).then(data => [
      setNotes(notes.concat([data]))
    ]);
    setNewNote('');
  };

  return [
    // ...
  ];
}

```

The app is now nicely typed and ready for further development!

The code of the typed notes can be found [here](#).

A note about defining object types

We have used [interfaces](#) to define object types, e.g. diary entries, in the previous section

```

interface DiaryEntry {
  id: number;
  date: string;
  weather: Weather;
  visibility: Visibility;
  comment?: string;
}

```

and in the course part of this section

```
interface CoursePartBase {  
  name: string;  
  exerciseCount: number;  
}
```

We actually could have had the same effect by using a [type alias](#)

```
type DiaryEntry = {  
  id: number;  
  date: string;  
  weather: Weather;  
  visibility: Visibility;  
  comment?: string;  
}
```

In most cases, you can use either *type* or *interface*, whichever syntax you prefer. However, there are a few things to keep in mind. For example, if you define multiple interfaces with the same name, they will result in a merged interface, whereas if you try to define multiple types with the same name, it will result in an error stating that a type with the same name is already declared.

TypeScript documentation [recommends using interfaces](#) in most cases.

Exercises 9.16-9.19

Let us now build a frontend for the Ilari's flight diaries that was developed in [the previous section](#). The source code of the backend can be found in [this GitHub repository](#).

Exercise 9.16

Create a TypeScript React app with similar configurations as the apps of this section. Fetch the diaries from the backend and render those to screen. Do all the required typing and ensure that there are no Eslint errors.

Remember to keep the network tab open. It might give you a valuable hint...

You can decide how the diary entries are rendered. If you wish, you may take inspiration from the figure below. Note that the backend API does not return the diary comments, you may modify it to return also those on a GET request.

Exercise 9.17

Make it possible to add new diary entries from the frontend. In this exercise you may skip all validations and assume that the user just enters the data in a correct form.

Exercise 9.18

Notify the user if the creation of a diary entry fails in the backend, show also the reason for the failure.

See eg. [this](#) to see how you can narrow the Axios error so that you can get hold of the error message.

Your solution may look like this:

The screenshot shows a web browser window with the URL `http://localhost:3000`. The page has a light gray background with a green border around the main content area.

Add new entry

Error: Incorrect visibility: best ever

Date: 2023-2-2
Visibility: best ever
Weather: sunny
Comment: nice flight but a shaky l

add

Diary entries

2017-01-01

visibility: poor
weather: rainy

2017-04-01

visibility: good
weather: sunny

Exercise 9.19

Addition of a diary entry is now very error prone since user can type anything to the input fields. The situation must be improved.

Modify the input form so that the date is set with a HTML `date` input element, and the weather and visibility are set with HTML `radio buttons`. We have already used radio buttons in [part 6](#), that material may or may not be useful...

Your app should all the time stay well typed and there should not be any Eslint errors and no Eslint rules should be ignored.

Your solution could look like this:

Add new entry

date

visibility great good ok poor

weather sunny rainy cloudy stormy windy

comment

Diary entries

2017-01-01

visibility: poor
weather: rainy

9e: Grande finale: Patientor

Working with an existing codebase

When diving into an existing codebase for the first time, it is good to get an overall view of the conventions and structure of the project. You can start your research by reading the *README.md* in the root of the repository. Usually, the README contains a brief description of the application and the requirements for using it, as well as how to start it for development. If the README is not available or someone has "saved time" and left it as a stub, you can take a peek at the *package.json*. It is always a good idea to start the application and click around to verify you have a functional development environment.

You can also browse the folder structure to get some insight into the application's functionality and/or the architecture used. These are not always clear, and the developers might have chosen a way to organize code that is not familiar to you. The [sample project](#) used in the rest of this part is organized, feature-wise. You can see what pages the application has, and some general components, e.g. modals and state. Keep in mind that the features may have different scopes. For example, modals are visible UI-level components whereas the state is comparable to business logic and keeps the data organized under the hood for the rest of the app to use.

TypeScript provides types for what kind of data structures, functions, components, and state to expect. You can try looking for *types.ts* or something similar to get started. VSCode is a big help and simply highlighting variables and parameters can provide quite a lot of insight. All this naturally depends on how types are used in the project.

If the project has unit, integration or end-to-end tests, reading those is most likely beneficial. Test cases are your most important tool when refactoring or adding new features to the application. You want to make sure not to break any existing features when hammering around the code. TypeScript can also give you guidance with argument and return types when changing the code.

Remember that reading code is a skill in itself, so don't worry if you don't understand the code on your first readthrough. The code may have a lot of corner cases, and pieces of logic may have been added here and there throughout its development cycle. It is hard to imagine what kind of problems the previous developer has wrestled with. Think of it all like [growth rings in trees](#). Understanding

everything requires digging deep into the code and business domain requirements. The more code you read, the better you will be at understanding it. You will most likely read far more code than you are going to produce throughout your life.

Patientor frontend

It's time to get our hands dirty finalizing the frontend for the backend we built in [exercises 9.8.-9.13](#). We will actually also need some new features to the backend for finishing the app.

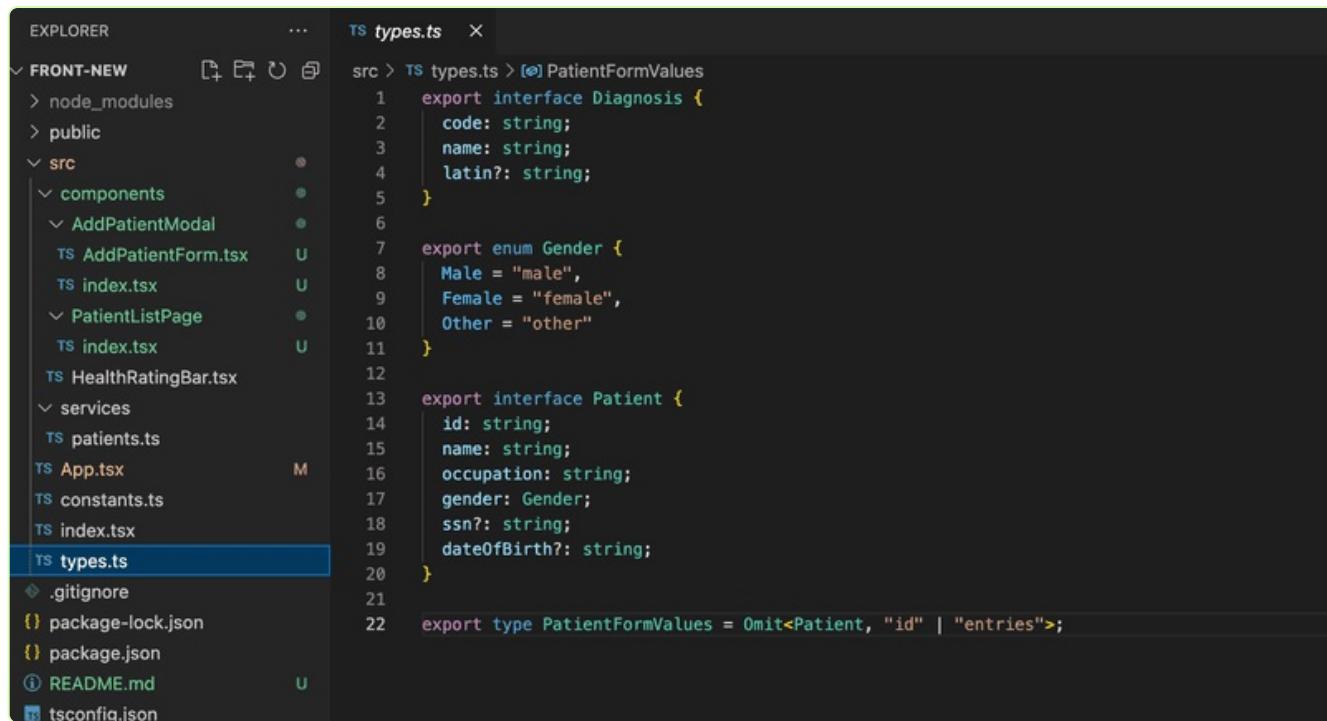
Before diving into the code, let us start both the frontend and the backend.

If all goes well, you should see a patient listing page. It fetches a list of patients from our backend, and renders it to the screen as a simple table. There is also a button for creating new patients on the backend. As we are using mock data instead of a database, the data will not persist - closing the backend will delete all the data we have added. UI design has not been a strong point of the creators, so let's disregard the UI for now.

After verifying that everything works, we can start studying the code. All the interesting stuff resides in the `src` folder. For your convenience, there is already a `types.ts` file for basic types used in the app, which you will have to extend or refactor in the exercises.

In principle, we could use the same types for both backend and frontend, but usually, the frontend has different data structures and use cases for the data, which causes the types to be different. For example, the frontend has a state and may want to keep data in objects or maps whereas the backend uses an array. The frontend might also not need all the fields of a data object saved in the backend, and it may need to add some new fields to use for rendering.

The folder structure looks as follows:



The screenshot shows a code editor with an Explorer sidebar on the left and a main editor area on the right. The Explorer sidebar shows a project structure with folders like FRONT-NEW, node_modules, public, and src. Inside src, there are components (AddPatientModal, PatientListPage), services (patients.ts), and other files (App.tsx, constants.ts, index.tsx). The file types.ts is selected in the Explorer and is open in the editor. The code in types.ts defines several interfaces and an enum:

```
EXPLORER ... TS types.ts X
FRONT-NEW ...
node_modules
public
src ...
components ...
AddPatientModal ...
  AddPatientForm.tsx U
  index.tsx U
PatientListPage ...
  index.tsx U
  HealthRatingBar.tsx
services ...
  patients.ts
App.tsx M
constants.ts
index.tsx
types.ts
.gitignore
package-lock.json
package.json
README.md
tsconfig.json

TS types.ts
src > TS types.ts > [e] PatientFormValues
  1  export interface Diagnosis {
  2    code: string;
  3    name: string;
  4    latin?: string;
  5  }
  6
  7  export enum Gender {
  8    Male = "male",
  9    Female = "female",
 10    Other = "other"
 11  }
 12
 13  export interface Patient {
 14    id: string;
 15    name: string;
 16    occupation: string;
 17    gender: Gender;
 18    ssn?: string;
 19    dateOfBirth?: string;
 20  }
 21
 22  export type PatientFormValues = Omit<Patient, "id" | "entries">;
```

Besides the component App and a directory for services, there are currently three main components: `AddPatientModal` and `PatientListPage` which are both defined in a directory, and a component `HealthRatingBar` defined in a file. If a component has some subcomponents not used elsewhere in the app, it might be a good idea to define the component and its subcomponents in a directory. For

example now the AddPatientModal is defined in the file `components/AddPatientModal/index.tsx` and its subcomponent `AddPatientForm` in its own file under the same directory.

There is nothing very surprising in the code. The state and communication with the backend are implemented with `useState` hook and Axios, similar to the notes app in the previous section. [Material UI](#) is used to style the app and the navigation structure is implemented with [React Router](#), both familiar to us from part 7 of the course.

From typing point of view, there are a couple of interesting things. Component `App` passes the function `setPatients` as a prop to the component `PatientListPage`:

```
const App = () => [
  const [patients, setPatients] = useState<Patient[]>([]);
  // ...

  return [
    <div className="App">
      <Router>
        <Container>
          <Routes>
            // ...
            <Route path="/" element={
              <PatientListPage
                patients={patients}
                setPatients={setPatients}
              />
            }
            </Routes>
          </Container>
        </Router>
    </div>
  ];
]
```

To keep the TypeScript compiler happy, the props should be typed as follows:

```
interface Props {
  patients : Patient[]
  setPatients: React.Dispatch<React.SetStateAction<Patient[]>>
}

const PatientListPage = ([ patients, setPatients ] : Props ) => [
  // ...
]
```

So the function `setPatients` has type `React.Dispatch<React.SetStateAction<Patient[]>>`. We can see the type in the editor when we hover over the function:

```

import patientService from "./services/patients";
import PatientListPage from "./PatientListPage";

const App = () => {
  const setPatients: React.Dispatch<React.SetStateAction<Patient[]>> = useState<Patient[]>([]);
  const [patients, setPatients] = useState<Patient[]>([]);

  useEffect(() => {
    void axios.get<void>(`${apiBaseUrl}/ping`);

    const fetchPatientList = async () => {
      const patients = await patientService.getAll();
      setPatients(patients);
    };
  }, []);
}

```

The [React TypeScript cheatsheet](#) has a pretty nice list of typical prop types, where we can seek for help if finding the proper typing for props is not obvious.

PatientListPage passes four props to the component *AddPatientModal*. Two of these props are functions. Let us have a look how these are typed:

```

const PatientListPage = ([ patients, setPatients ] : Props) => [
  // ...
  const [modalOpen, setModalOpen] = useState<boolean>(false);
  const [error, setError] = useState<string>("");
  // ...
  const closeModal = (): void => {
    setModalOpen(false);
    setError(undefined);
  };

  const submitNewPatient = async (values: PatientFormValues) => [
    // ...
  ];
  // ...

  return (
    <div className="App">
      // ...
      <AddPatientModal
        modalOpen={modalOpen}
        onSubmit={submitNewPatient}
        onClose={closeModal}
      />
    </div>
  );
]

```

Types look like the following:

```

interface Props {
  modalOpen: boolean;
  onClose: () => void;
  onSubmit: (values: PatientFormValues) => Promise<void>;
  error?: string;
}

```

```
const AddPatientModal = ([ modalOpen, onClose, onSubmit, error ]: Props) => [
  // ...
]
```

onClose is just a function that takes no parameters, and does not return anything, so the type is:

```
() => void
```

The type of *onSubmit* is a bit more interesting, it has one parameter that has the type *PatientFormValues*. The return value of the function is *Promise<void>*. So again the function type is written with the arrow syntax:

```
{values: PatientFormValues} => Promise<void>
```

The return value of a *async* function is a *promise* with the value that the function returns. Our function does not return anything so the proper return type is just *Promise<void>*.

Exercises 9.20-9.21

We will soon add a new type for our app, *Entry*, which represents a lightweight patient journal entry. It consists of a journal text, i.e. a *description*, a creation date, information regarding the specialist who created it and possible diagnosis codes. Diagnosis codes map to the ICD-10 codes returned from the */api/diagnoses* endpoint. Our naive implementation will be that a patient has an array of entries.

Before going into this, let us do some preparatory work.

9.20: Patientor, step1

Create an endpoint */api/patients/:id* to the backend that returns all of the patient information for one patient, including the array of patient entries that is still empty for all the patients. For the time being, expand the backend types as follows:

```
// eslint-disable-next-line @typescript-eslint/no-empty-interface
export interface Entry {
}

export interface Patient {
  id: string;
  name: string;
  ssn: string;
  occupation: string;
  gender: Gender;
  dateOfBirth: string;
  entries: Entry[]
}

export type NonSensitivePatient = Omit<Patient, 'ssn' | 'entries'>;
```

The response should look as follows:

```
← → ⌂ ⓘ localhost:3001/api/patients/d2773336-f723-11e9-8f0b-362b9e155667

{
  name: "John McClane",
  ssn: "090786-122X",
  occupation: "New york city cop",
  dateOfBirth: "1986-07-09",
  gender: "other",
  entries: [ ],
  id: "d2773336-f723-11e9-8f0b-362b9e155667"
}
```

9.21: Patientor, step2

Create a page for showing a patient's full information in the frontend.

The user should be able to access a patient's information by clicking the patient's name.

Fetch the data from the endpoint created in the previous exercise.

You may use [MaterialUI](#) for the new components but that is up to you since our main focus now is TypeScript.

You might want to have a look at [part 7](#) if you don't yet have a grasp on how the [React Router](#) works.

The result could look like this:

```
← → ⌂ ⓘ localhost:3000/patients/d2773822-f723-11e9-8f0b-362b9e155667
Reload this page, hold to see more options
```

Patientor

[HOME](#)

Dana Scully ♀

ssh: 050174-432N
occupation: Forensic Pathologist

The example uses [Material UI Icons](#) to represent genders.

Full entries

In [exercise 9.10](#) we implemented an endpoint for fetching information about various diagnoses, but we are still not using that endpoint at all. Since we now have a page for viewing a patient's information, it would be nice to expand our data a bit. Let's add an *Entry* field to our patient data so that a patient's data contains their medical entries, including possible diagnoses.

Let's ditch our old patient seed data from the backend and start using [this expanded format](#).

Let us now create a proper *Entry* type based on the data we have.

If we take a closer look at the data, we can see that the entries are quite different from one another. For example, let's take a look at the first two entries:

```
[{"id": "d811e46d-70b3-4d90-b090-4535c7cf8fb1",  
 "date": "2015-01-02",  
 "type": "Hospital",  
 "specialist": "MD House",  
 "diagnosisCodes": ["S62.5"],  
 "description":  
     "Healing time appr. 2 weeks. patient doesn't remember how he got the injury.",  
     "discharge": {  
         "date": "2015-01-16",  
         "criteria": "Thumb has healed.",  
     },  
     "sickLeave": {  
         "startDate": "2015-01-02",  
         "endDate": "2015-01-16",  
     },  
     "..."},  
 {"id": "fcd59fa6-c4b4-4fec-ac4d-df4fe1f85f62",  
 "date": "2019-08-05",  
 "type": "OccupationalHealthcare",  
 "specialist": "MD House",  
 "employerName": "HyPD",  
 "diagnosisCodes": ["Z57.1", "Z74.3", "M51.2"],  
 "description":  
     "Patient mistakenly found himself in a nuclear plant waste site without protection gear. Very minor radiation poisoning.",  
     "sickLeave": {  
         "startDate": "2019-08-05",  
         "endDate": "2019-08-28",  
     },  
     "..."}]
```

Immediately, we can see that while the first few fields are the same, the first entry has a *discharge* field and the second entry has *employerName* and *sickLeave* fields. All the entries seem to have some fields in common, but some fields are entry-specific.

When looking at the *type*, we can see that there are three kinds of entries: *OccupationalHealthcare*, *Hospital* and *HealthCheck*. This indicates we need three separate types. Since they all have some fields in common, we might just want to create a base entry interface that we can extend with the different fields in each type.

When looking at the data, it seems that the fields *id*, *description*, *date* and *specialist* are something that can be found in each entry. On top of that, it seems that *diagnosisCodes* is only found in one *OccupationalHealthcare* and one *Hospital* type entry. Since it is not always used, even in those types of entries, it is safe to assume that the field is optional. We could consider adding it to the *HealthCheck* type as well since it might just not be used in these specific entries.

So our *BaseEntry* from which each type could be extended would be the following:

```

interface BaseEntry {
  id: string;
  description: string;
  date: string;
  specialist: string;
  diagnosisCodes?: string[];
}

```

If we want to finetune it a bit further, since we already have a *Diagnosis* type defined in the backend, we might just want to refer to the code field of the *Diagnosis* type directly in case its type ever changes. We can do that like so:

```

interface BaseEntry {
  id: string;
  description: string;
  date: string;
  specialist: string;
  diagnosisCodes?: Diagnosis['code'][];
}

```

As was mentioned [earlier in this part](#), we could define an array with the syntax *Array<Type>* instead of defining it *Type[]*. In this particular case writing *Diagnosis['code'][]* starts to look a bit strange so we will decide to use the alternative syntax (that is also recommended by the ESLint rule [array-simple](#)):

```

interface BaseEntry {
  id: string;
  description: string;
  date: string;
  specialist: string;
  diagnosisCodes?: Array<Diagnosis['code']>;
}

```

Now that we have the *BaseEntry* defined, we can start creating the extended entry types we will actually be using. Let's start by creating the *HealthCheckEntry* type.

Entries of type *HealthCheck* contain the field *HealthCheckRating*, which is an integer from 0 to 3, zero meaning *Healthy* and three meaning *CriticalRisk*. This is a perfect case for an enum definition. With these specifications we could write a *HealthCheckEntry* type definition like so:

```

export enum HealthCheckRating {
  "Healthy" = 0,
  "LowRisk" = 1,
  "HighRisk" = 2,
  "CriticalRisk" = 3
}

interface HealthCheckEntry extends BaseEntry {
  type: "HealthCheck";
  healthCheckRating: HealthCheckRating;
}

```

Now we only need to create the *OccupationalHealthcareEntry* and *HospitalEntry* types so we can combine them in a union and export them as an Entry type like this:

```
export type Entry =
  | HospitalEntry
  | OccupationalHealthcareEntry
  | HealthCheckEntry
```

Omit with unions

An important point concerning unions is that, when you use them with *Omit* to exclude a property, it works in a possibly unexpected way. Suppose that we want to remove the *id* from each *Entry*. We could think of using

```
Omit<Entry, 'id'>
```

but [it wouldn't work as we might expect](#). In fact, the resulting type would only contain the common properties, but not the ones they don't share. A possible workaround is to define a special *Omit*-like function to deal with such situations:

```
// Define special omit for unions
type UnionOmit<T, K extends string | number | symbol> = T extends unknown ? Omit<
  T, K> : never
// Define Entry without the 'id' property
type EntryWithoutId = UnionOmit<Entry, 'id'>
```

Exercises 9.22-9.29

Now we are ready to put the finishing touches to the app!

9.22: Patientor, step 3

Define the types *OccupationalHealthcareEntry* and *HospitalEntry* so that those conform with the new example data. Ensure that your backend returns the entries properly when you go to an individual patient's route:

```

{
  name: "John McClane",
  ssn: "090786-122X",
  occupation: "New york city cop",
  dateOfBirth: "1986-07-09",
  gender: "male",
  - entries: [
    - {
      id: "d811e46d-70b3-4d90-b090-4535c7cf8fb1",
      date: "2015-01-02",
      type: "Hospital",
      specialist: "MD House",
      - diagnoseCodes: [
        "S62.5"
      ],
      description: "Healing time appr. 2 weeks. patient doesn't remember how he got the injury.",
      - discharge: {
        date: "2015-01-16",
        criteria: "Thumb has healed."
      }
    }
  ],
  id: "d2773336-f723-11e9-8f0b-362b9e155667"
}

```

Use types properly in the backend! For now, there is no need to do a proper validation for all the fields of the entries in the backend, it is enough e.g. to check that the field `type` has a correct value.

9.23: Patientor, step 4

Extend a patient's page in the frontend to list the `date`, `description` and `diagnoseCodes` of the patient's entries.

You can use the same type definition for an `Entry` in the frontend. For these exercises, it is enough to just copy/paste the definitions from the backend to the frontend.

Your solution could look like this:

→ C ⓘ localhost:3000/patients/d2773598-f723-11e9-8f0b-362b9e155667

Patientor

[Home](#)

Martin Riggs ♂

ssn: 300179-777A
occupation: Cop

entries

2019-08-05 Patient mistakenly found himself in a nuclear plant waste site without protection gear. Very minor radiation poisoning.

- Z57.1
- Z74.3
- M51.2

9.24: Patientor, step 5

Fetch and add diagnoses to the application state from the `/api/diagnoses` endpoint. Use the new diagnosis data to show the descriptions for patient's diagnosis codes:

The screenshot shows a web application titled "Patientor". The URL in the address bar is `localhost:3000/patients/d2773598-f723-11e9-8f0b-362b9e155667`. The page displays a patient profile for "Martin Riggs ♂". Basic details include SSN: 300179-777A and Occupation: Cop. A section titled "entries" contains a single entry: "2019-08-05 Patient mistakenly found himself in a nuclear plant waste site without protection gear. Very minor radiation poisoning." Below this entry is a list of three diagnoses: Z57.1 Occupational exposure to radiation, Z74.3 Need for continuous supervision, and M51.2 Other specified intervertebral disc displacement.

9.25: Patientor, step 6

Extend the entry listing on the patient's page to include the Entry's details, with a new component that shows the rest of the information of the patient's entries, distinguishing different types from each other.

You could use eg. [Icons](#) or some other [Material UI](#) component to get appropriate visuals for your listing.

You should use a *switch case*-based rendering and *exhaustive type checking* so that no cases can be forgotten.

Like this:

```
80
81  const EntryDetails: React.FC<{ entry: Entry }> = ({ entry }) => {
82    switch (entry.type) {
83      case "Hospital": var entry: HealthCheckEntry
84        return <HospitalEnt Argument of type 'HealthCheckEntry' is not assignable to parameter of type
85          case 'OccupationalHea 'never'. ts(2345)
86            return <Occupationa Peek Problem  No quick fixes available
87            default:
88              return assertNever(entry);
89    }
90  };
91
```

The resulting entries in the listing *could* look something like this:

Dana Scully ♀

ssh: 050174-432N
occupation: Forensic Pathologist

entries

2019-10-20 
Yearly control visit. Cholesterol levels back to normal.

diagnose by MD House

2019-09-10  **FBI**
Prescriptions renewed.
diagnose by MD House

2018-10-05 
Yearly control visit. Due to high cholesterol levels recommended to eat more vegetables.

diagnose by MD House

[ADD NEW ENTRY](#)

9.26: Patientor, step 7

We have established that patients can have different kinds of entries. We don't yet have any way of adding entries to patients in our app, so, at the moment, it is pretty useless as an electronic medical record.

Your next task is to add endpoint `/api/patients/:id/entries` to your backend, through which you can POST an entry for a patient.

Remember that we have different kinds of entries in our app, so our backend should support all those types and check that at least all required fields are given for each type.

In this exercise you quite likely need to remember [this trick](#).

You may assume that the diagnostic codes are sent in a correct form and use eg. the following kind of parser to extract those from the request body:

```
const parseDiagnosisCodes = [object: unknown]: Array<Diagnosis['code']> => {
  if (!object || typeof object !== 'object' || !('diagnosisCodes' in object)) {
    // we will just trust the data to be in correct form
    return [] as Array<Diagnosis['code']>;
  }

  return object.diagnosisCodes as Array<Diagnosis['code']>;
};
```

9.27: Patientor, step 8

Now that our backend supports adding entries, we want to add the corresponding functionality to the frontend. In this exercise, you should add a form for adding an entry to a patient. An intuitive place for accessing the form would be on a patient's page.

In this exercise, it is enough to **support one entry type**. All the fields in the form can be just plain text inputs, so it is up to user to enter valid values.

Upon a successful submit, the new entry should be added to the correct patient and the patient's entries on the patient page should be updated to contain the new entry.

Your form might look something like this:

Patientor

[HOME](#)

John McClane ⚡

ssn 090786-122X
occupation: New york city cop

New HealthCheck entry

Description
good

Date
2023-1-4

Specialist
Dr Alban

Healthcheck rating
1

Diagnosis codes
Z57.1, N30.0

[CANCEL](#) [ADD](#)

entries

2015-01-02 Healing time appr. 2 weeks. patient doesn't remember how he got the injury. 

diagnose by MD House

discharged 2015-01-16. Thumb has healed.

If user enters invalid values to the form and backend rejects the addition, show a proper error message to user

← → ⌂ ⓘ http://localhost:3000/patients/d2773336-f723-11e9-8f0b-362b9e155667

John McClane ⚡

ssn 090786-122X
occupation: New york city cop

! Value of healthCheckRating incorrect: 15

New HealthCheck entry

Description
problematic entry

Date
2023-1-4

Specialist
Dr House

Healthcheck rating
15

9.28: Patientor, step 9

Extend your solution so that it supports *all the entry types*

9.29: Patientor, step 10

Improve the entry creation forms so that it makes hard to enter incorrect dates, diagnosis codes and health rating.

Your improved form might look something like this:

The screenshot shows a patient form with the following fields:

- Diagnosis codes:** Z57.1, N30.0, L60.1
- Employee:** Nokia
- Sickleave:** start dd.mm.yyyy (with a placeholder value of dd.mm.yyyy)
- Calendar:** A date picker for February 2023. The date 11 is selected and highlighted in blue. The previous month (January) has a red background.
- Text area:** A large text area containing the text: "and himself in a nuclear plant waste site without protection gear. Very minor radiation poisoni".

Diagnosis codes are now set with Material UI [multiple select](#) and dates with [Input](#) elements with type [date](#).

Submitting exercises and getting the credits

Exercises of this part are submitted via [the submissions system](#) just like in the previous parts, but unlike previous parts, the submission goes to a different "course instance". Remember that you have to finish at least 24 exercises to pass this part!

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:

My submissions

part	exercises	hours	github	comment	solution
1	22	29	https://github.com/Kaltsoon/fs-cicd		show
total	22	29			

credits 1 based on exercises

Certificate  

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

Note that you need a registration to the corresponding course part for getting the credits registered, see [here](#) for more information.

You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the certificate's language.

10: React Native

10a: Introduction to React Native

Note: This course material was updated in Feb 2024. Some updates are not compatible anymore with older material. We recommend a fresh start with this new Part 10 material. However, if you're returning to this course after a break, and you want to continue the exercises in your older project, please use [Part 10 material before the update](#).

Traditionally, developing native iOS and Android applications has required the developer to use platform-specific programming languages and development environments. For iOS development, this means using Objective C or Swift and for Android development using JVM-based languages such as Java, Scala or Kotlin. Releasing an application for both these platforms technically requires developing two separate applications with different programming languages. This requires lots of development resources.

One of the popular approaches to unify the platform-specific development has been to utilize the browser as the rendering engine. [Cordova](#) is one of the most popular platforms for building cross-platform applications. It allows for developing multi-platform applications using standard web technologies - HTML5, CSS3, and JavaScript. However, Cordova applications are running within an embedded browser window in the user's device. That is why these applications can not achieve the performance nor the look-and-feel of native applications that utilize actual native user interface components.

[React Native](#) is a framework for developing native Android and iOS applications using JavaScript and React. It provides a set of cross-platform components that behind the scenes utilize the platform's native components. Using React Native allows us to bring all the familiar features of React such as JSX, components, props, state, and hooks into native application development. On top of that, we can utilize many familiar libraries in the React ecosystem such as [React Redux](#), [Apollo](#), [React Router](#) and many more.

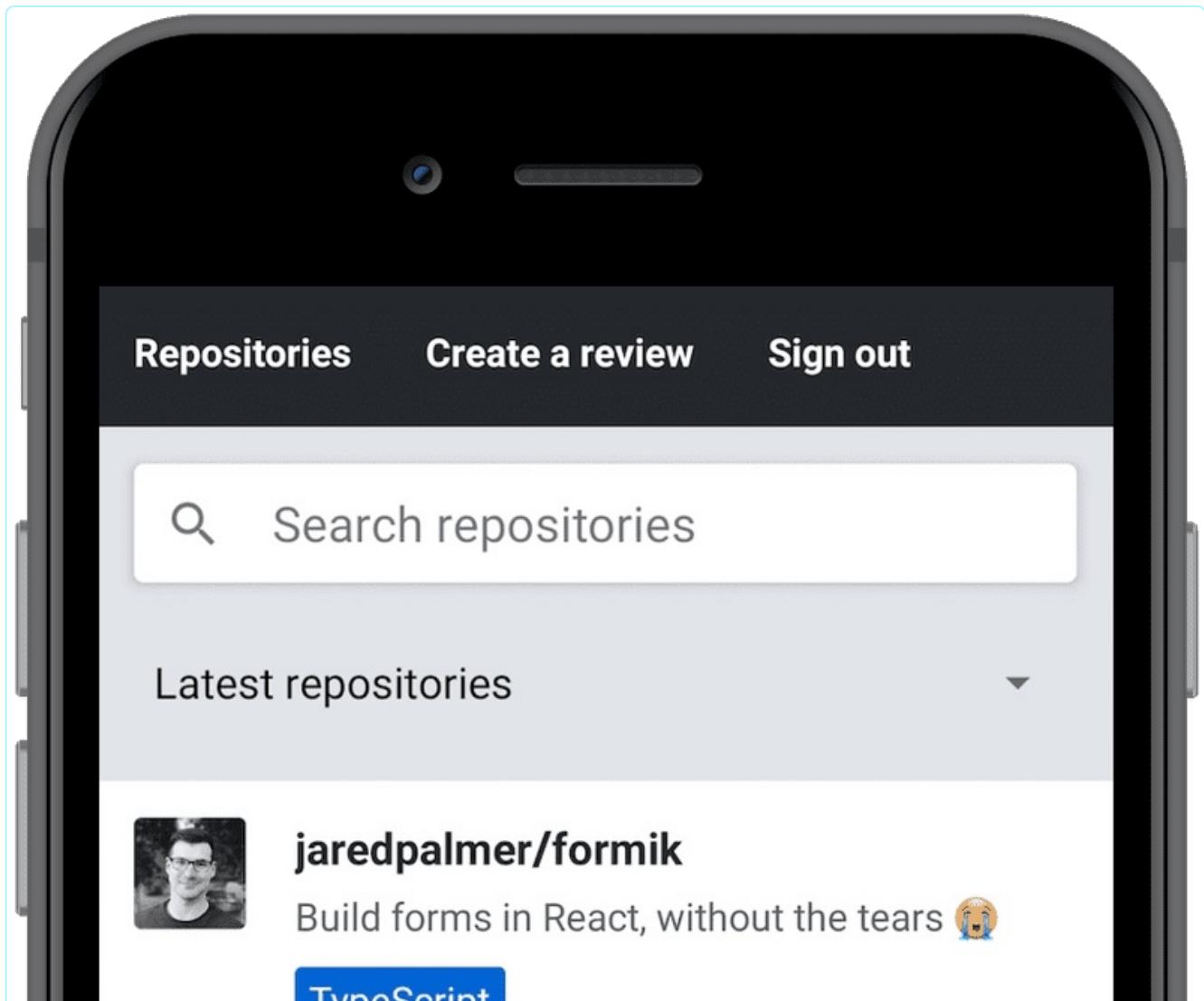
The speed of development and gentle learning curve for developers familiar with React is one of the most important benefits of React Native. Here's a motivational quote from Coinbase's article [Onboarding thousands of users with React Native](#) on the benefits of React Native:

If we were to reduce the benefits of React Native to a single word, it would be "velocity". On average, our team was able to onboard engineers in less time, share more code (which we expect will lead to future productivity boosts), and ultimately deliver features faster than if we had taken a purely native approach.

About this part

During this part, we will learn how to build an actual React Native application from the bottom up. We will learn concepts such as what are React Native's core components, how to create beautiful user interfaces, how to communicate with a server and how to test a React Native application.

We will be developing an application for rating [GitHub](#) repositories. Our application will have features such as, sorting and filtering reviewed repositories, registering a user, logging in and creating a review for a repository. The back end for the application will be provided for us so that we can solely focus on the React Native development. The final version of our application will look something like this:



typescript

22.1k

1.6k

6

87

Stars

Forks

Reviews

Rating



async-library/react-async

Flexible promise-based React data loader

JavaScript

1.8k

69

3

72

Stars

Forks

Reviews

Rating



rzwitserloot/lombok

Very spicy additions to the Java programming language.

Java

9.2k

1.8k

0

0



All the exercises in this part have to be submitted into *a single GitHub repository* which will eventually contain the entire source code of your application. There will be model solutions available for each section of this part which you can use to fill in incomplete submissions. This part is structured based on the idea that you develop your application as you progress in the material. So *do not* wait until the exercises to start the development. Instead, develop your application at the same pace as the material progresses.

This part will heavily rely on concepts covered in the previous parts. Before starting this part you will need basic knowledge of JavaScript, React and GraphQL. Deep knowledge of server-side development is not required and all the server-side code is provided for you. However, we will be making network requests from your React Native applications, for example, using GraphQL queries. The recommended

parts to complete before this part are [part 1](#), [part 2](#), [part 5](#), [part 7](#) and [part 8](#).

Submitting exercises and earning credits

Exercises are submitted via the [submissions system](#) just like in the previous parts. Note that, exercises in this part are submitted to *a different course instance* than in parts 0-9. Parts 1-4 in the submission system refer to sections a-d in this part. This means that you will be submitting exercises a single section at a time starting with this section, "Introduction to React Native", which is part 1 in the submission system.

During this part, you will earn credits based on the number of exercises you complete. Completing *at least 25 exercises* in this part will earn you *2 credits*. Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:

My submissions

part	exercises	hours	github	comment	solu
1	2	3	https://github.com/Kaltsoon/rate-repository-app		shov
2	8	12	https://github.com/Kaltsoon/rate-repository-app		shov
3	6	24	https://github.com/Kaltsoon/rate-repository-app		shov
4	11	39	https://github.com/Kaltsoon/rate-repository-app		shov
total	27	78			

credits 2 based on exercises

Certificate

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

Note that you need a registration to the corresponding course part for getting the credits registered, see [here](#) for more information.

You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the certificate's language. Note that you must have completed at least one credit worth of exercises before you can download the certificate.

Initializing the application

To get started with our application we need to set up our development environment. We have learned from previous parts that there are useful tools for setting up React applications quickly such as Create React App. Luckily React Native has these kinds of tools as well.

For the development of our application, we will be using [Expo](#). Expo is a platform that eases the setup, development, building, and deployment of React Native applications. Let's get started with Expo by initializing our project with *create-expo-app*:

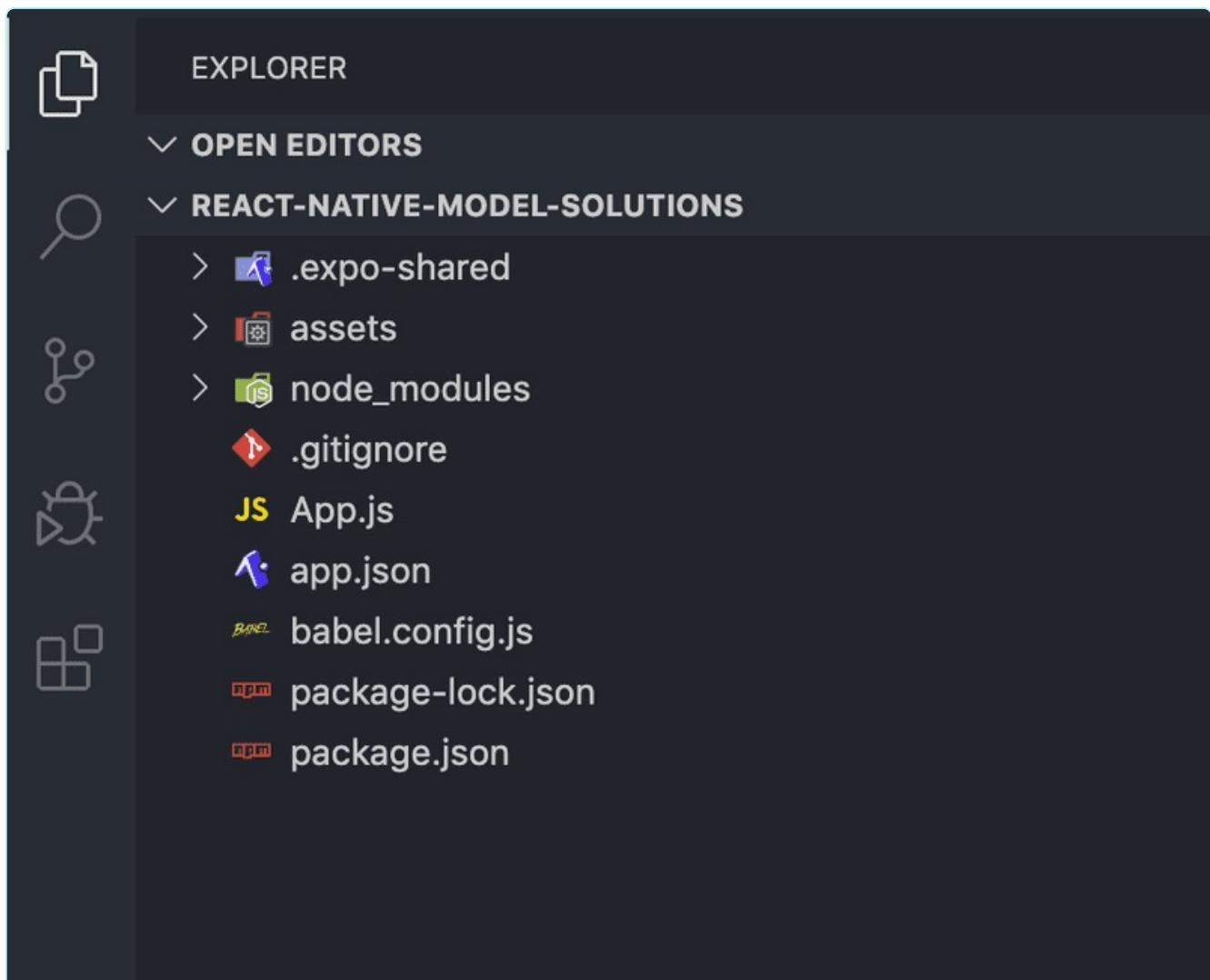
```
npx create-expo-app rate-repository-app --template expo-template-blank@sdk-50
```

Note, that the `@sdk-50` sets the project's *Expo SDK version to 50*, which supports *React Native version 0.73*. Using other Expo SDK versions might cause you trouble while following this material. Also, Expo has a [few limitations](#) when compared to plain React Native CLI. However, these limitations do not affect the application implemented in the material.

Next, let's navigate to the created `rate-repository-app` directory with the terminal and install a few dependencies we'll be needing soon:

```
npx expo install react-native-web@~0.19.6 react-dom@18.2.0 @expo/metro-runtime@~3.1.1
```

Now that our application has been initialized, open the created `rate-repository-app` directory with an editor such as [Visual Studio Code](#). The structure should be more or less the following:



We might spot some familiar files and directories such as `package.json` and `node_modules`. On top of those, the most relevant files are the `app.json` file which contains Expo-related configuration and `App.js` which is the root component of our application. *Do not rename or move the App.js file because by default Expo imports it to register the root component.*

Let's look at the `scripts` section of the `package.json` file which has the following scripts:

```
{}  
// ...
```

```
"scripts": {  
  "start": "expo start",  
  "android": "expo start --android",  
  "ios": "expo start --ios",  
  "web": "expo start --web"  
}  
// ...  
}
```

Let us now run the script `npm start`

```
Starting Metro Bundler  
  
> Metro waiting on exp://10.112.31.111:19000  
> Scan the QR code above with Expo Go (Android) or the Camera app (iOS)  
  
> Press a | open Android  
> Press i | open iOS simulator  
> Press w | open web  
  
> Press r | reload app  
> Press m | toggle menu  
  
> Press ? | show all commands  
  
Logs for your project will appear below. Press Ctrl+C to exit.
```

If the script fails with error the problem is most likely your Node version. In case of problems, switch to version 20.

The script starts the [Metro bundler](#) which is a JavaScript bundler for React Native. In addition to the Metro bundler, the Expo command-line interface should be open in the terminal window. The command-line interface has a useful set of commands for viewing the application logs and starting the application in an emulator or in Expo's mobile application. We will get to emulators and Expo's mobile application soon, but first, let's open our application.

Expo command-line interface suggests a few ways to open our application. Let's press the "w" key in

the terminal window to open the application in a browser. We should soon see the text defined in the `App.js` file in a browser window. Open the `App.js` file with an editor and make a small change to the text in the `Text` component. After saving the file you should be able to see that the changes you have made in the code are visible in the browser window.

Setting up the development environment

We have had the first glance of our application using the Expo's browser view. Although the browser view is quite usable, it is still a quite poor simulation of the native environment. Let's have a look at the alternatives we have regarding the development environment.

Android and iOS devices such as tablets and phones can be emulated in computers using specific *emulators*. This is very useful for developing native applications. macOS users can use both Android and iOS emulators with their computers. Users of other operating systems such as Linux or Windows have to settle for Android emulators. Next, depending on your operating system follow one of these instructions on setting up an emulator:

- [Set up the Android emulator with Android Studio](#) (any operating system)
- [Set up the iOS simulator with Xcode](#) (macOS operating system)

After you have set up the emulator and it is running, start the Expo development tools as we did before, by running `npm start`. Depending on the emulator you are running either press the corresponding key for the "open Android" or "open iOS simulator". After pressing the key, Expo should connect to the emulator and you should eventually see the application in your emulator. Be patient, this might take a while.

In addition to emulators, there is one extremely useful way to develop React Native applications with Expo, the Expo mobile app. With the Expo mobile app, you can preview your application using your actual mobile device, which provides a bit more concrete development experience compared to emulators. To get started, install the Expo mobile app by following the instructions in the [Expo's documentation](#). Note that the Expo mobile app can only open your application if your mobile device is connected to *the same local network* (e.g. connected to the same Wi-Fi network) as the computer you are using for development.

When the Expo mobile app has finished installing, open it up. Next, if the Expo development tools are not already running, start them by running `npm start`. You should be able to see a QR code at the beginning of the command output. Open the app by scanning the QR code, in Android with Expo app or in iOS with the Camera app. The Expo mobile app should start building the JavaScript bundle and after it is finished you should be able to see your application. Now, every time you want to reopen your application in the Expo mobile app, you should be able to access the application without scanning the QR code by pressing it in the *Recently opened* list in the *Projects* view.

Exercise 10.1

Exercise 10.1: initializing the application

Initialize your application with Expo command-line interface and set up the development environment either using an emulator or Expo's mobile app. It is recommended to try both and find out which development environment is the most suitable for you. The name of the application is not that relevant. You can, for example, go with `rate-repository-app`.

To submit this exercise and all future exercises you need to [create a new GitHub repository](#). The name of the repository can be for example the name of the application you initialized with `expo init`. If

you decide to create a private repository, add GitHub user [mluukkai](#) as a [repository collaborator](#). The collaborator status is only used for verifying your submissions.

Now that the repository is created, run `git init` within your application's root directory to make sure that the directory is initialized as a Git repository. Next, to add the created repository as the remote run `git remote add origin git@github.com:`

`<YOURGITHUBUSERNAME>/<NAMEOYOUR_REPOSITORY>.git` (remember to replace the placeholder values in the command). Finally, just commit and push your changes into the repository and you are all done.

ESLint

Now that we are somewhat familiar with the development environment let's enhance our development experience even further by configuring a linter. We will be using [ESLint](#) which is already familiar to us from the previous parts. Let's get started by installing the dependencies:

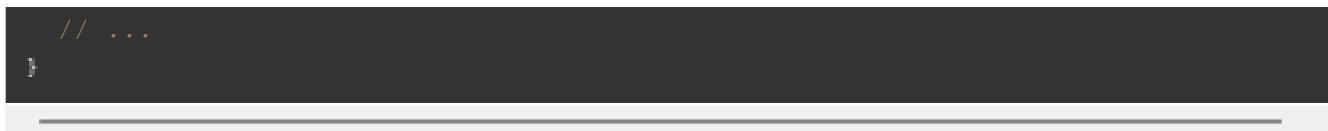
```
npm install --save-dev eslint @babel/eslint-parser eslint-plugin-react eslint-plugin-react-native
```

Next, let's add the ESLint configuration into a `.eslintrc.json` file in the `rate-repository-app` directory with the following content:

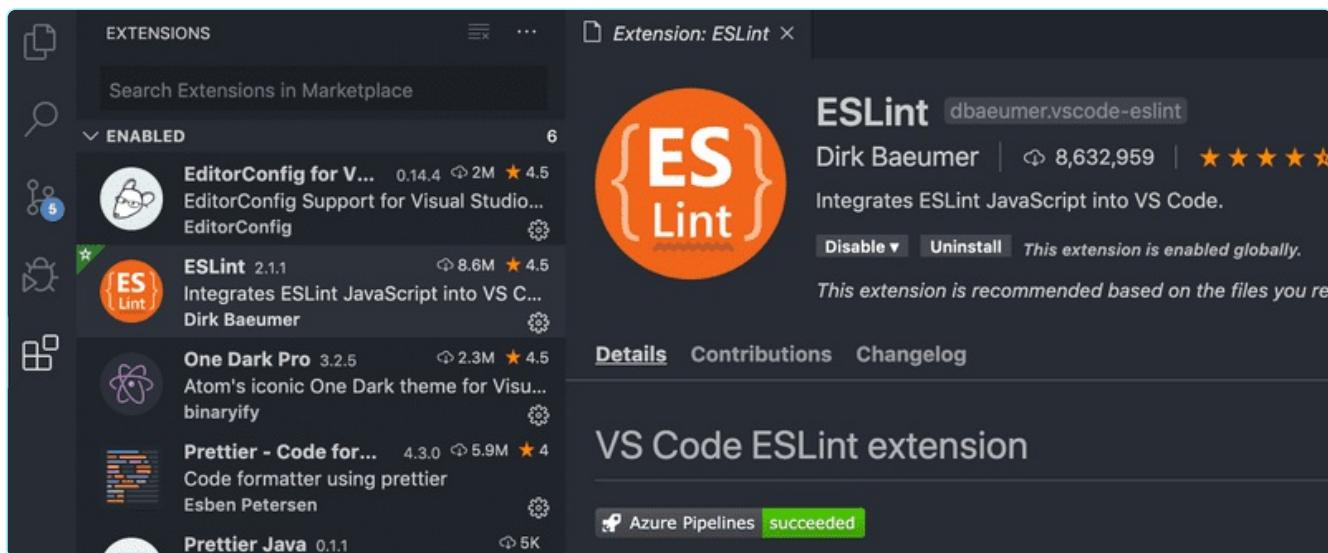
```
{  
  "plugins": ["react", "react-native"],  
  "settings": {  
    "react": {  
      "version": "detect"  
    }  
  },  
  "extends": ["eslint:recommended", "plugin:react/recommended"],  
  "parser": "@babel/eslint-parser",  
  "env": {  
    "react-native/react-native": true  
  },  
  "rules": [  
    "react/prop-types": "off",  
    "react/react-in-jsx-scope": "off"  
  ]  
}
```

And finally, let's add a `/lint` script to the `package.json` file to check the linting rules in specific files:

```
{  
  // ...  
  "scripts": {  
    "start": "expo start",  
    "android": "expo start --android",  
    "ios": "expo start --ios",  
    "web": "expo start --web",  
    "lint": "eslint ./src/**/*.{js,jsx} App.js --no-error-on-unmatched-pattern"  
  },  
}
```



Now we can check that the linting rules are obeyed in JavaScript files in the `src` directory and in the `App.js` file by running `npm run lint`. We will be adding our future code to the `src` directory but because we haven't added any files there yet, we need the `no-error-on-unmatched-pattern` flag. Also if possible integrate ESLint with your editor. If you are using Visual Studio Code you can do that by, going to the extensions section and checking that the ESLint extension is installed and enabled:



The provided ESLint configuration contains only the basis for the configuration. Feel free to improve the configuration and add new plugins if you feel like it.

Exercise 10.2

Exercise 10.2: setting up the ESLint

Set up ESLint in your project so that you can perform linter checks by running `npm run lint`. To get most of linting it is also recommended to integrate ESLint with your editor.

This was the last exercise in this section. It's time to push your code to GitHub and mark all of your finished exercises to the [exercise submission system](#). Note that exercises in this section should be submitted to part 1 in the exercise submission system.

Debugging

When our application doesn't work as intended, we should immediately start *debugging* it. In practice, this means that we'll need to reproduce the erroneous behavior and monitor the code execution to find out which part of the code behaves incorrectly. During the course, we have already done a bunch of debugging by logging messages, inspecting network traffic, and using specific development tools, such as *React Development Tools*. In general, debugging isn't that different in React Native, we'll just need the right tools for the job.

The good old `console.log` messages appear in the Expo development tools command line:

```
> Press w | open web
> Press r | reload app
> Press m | toggle menu
> Press ? | show all commands

Logs for your project will appear below. Press Ctrl+C to exit.
> Opening on iOS...
> Opening exp://192.168.1.33:19000 on iPhone SE (3rd generation)
> Opening the iOS simulator, this might take a moment.
> Press ? | show all commands
iOS Bundling complete 1217ms
LOG good old console log works!
LOG number of items fetched from server 0
LOG good old console log works!
LOG number of items fetched from server 10
```

That might actually be enough in most cases, but sometimes we need more. React Native provides an in-app developer menu which offers several debugging options. Read more about [debugging react native applications](#).

To inspect the React element tree, props, and state you can install React DevTools.

```
npx react-devtools
```

Read here about [React DevTools](#). For more useful React Native application debugging tools, also head out to the Expo's [debugging documentation](#).

10b: React Native basics

Note: This course material was updated in Feb 2024. Some updates are not compatible anymore with older material. We recommend a fresh start with this new Part 10 material. However, if you're returning to this course after a break, and you want to continue the exercises in your older project, please use [Part 10 material before the upgrade](#).

Now that we have set up our development environment we can get into React Native basics and get started with the development of our application. In this section, we will learn how to build user interfaces with React Native's core components, how to add style properties to these core components, how to transition between views, and how to manage the form's state efficiently.

Core components

In the previous parts, we have learned that we can use React to define components as functions, which receive props as an argument and returns a tree of React elements. This tree is usually represented with JSX syntax. In the browser environment, we have used the [ReactDOM](#) library to turn these components into a DOM tree that can be rendered by a browser. Here is a concrete example of a very simple component:

```
const HelloWorld = props => {
```

```
    return <div>Hello world!</div>;
};
```

The `HelloWorld` component returns a single `div` element which is created using the JSX syntax. We might remember that this JSX syntax is compiled into `React.createElement` method calls, such as this:

```
React.createElement('div', null, 'Hello world!');
```

This line of code creates a `div` element without any props and with a single child element which is a string `"Hello world"`. When we render this component into a root DOM element using the `ReactDOM.render` method the `div` element will be rendered as the corresponding DOM element.

As we can see, React is not bound to a certain environment, such as the browser environment. Instead, there are libraries such as `ReactDOM` that can render a *set of predefined components*, such as DOM elements, in a specific environment. In React Native these predefined components are called *core components*.

Core components are a set of components provided by React Native, which behind the scenes utilize the platform's native components. Let's implement the previous example using React Native:

```
import { Text } from 'react-native';
const HelloWorld = props => [
  return <Text>Hello world!</Text>,
];
```

So we import the `Text` component from React Native and replace the `div` element with a `Text` element. Many familiar DOM elements have their React Native "counterparts". Here are some examples picked from React Native's [Core Components documentation](#):

- `Text` component is *the only* React Native component that can have textual children. It is similar to for example the `` and the `<h1>` elements.
- `View` component is the basic user interface building block similar to the `<div>` element.
- `TextInput` component is a text field component similar to the `<input>` element.
- `Pressable` component is for capturing different press events. It is similar to for example the `<button>` element.

There are a few notable differences between core components and DOM elements. The first difference is that the `Text` component is *the only* React Native component that can have textual children. This means that you can't, for example, replace the `Text` component with the `View` component in the previous example.

The second notable difference is related to the event handlers. While working with the DOM elements we are used to adding event handlers such as `onClick` to basically any element such as `<div>` and `<button>`. In React Native we have to carefully read the [API documentation](#) to know what event handlers (as well as other props) a component accepts. For example, the `Pressable` component provides props for listening to different kinds of press events. We can for example use the component's `onPress` prop for listening to press events:

```
import { Text, Pressable, Alert } from 'react-native';

const PressableText = props => [
  return [
    <Text>Press me!</Text>
  ],
];
```

```

<Pressable
  onPress={() => Alert.alert('You pressed the text!')}
>
  <Text>You can press me</Text>
</Pressable>
);
}

```

Now that we have a basic understanding of the core components, let's start to give our project some structure. Create a `src` directory in the root directory of your project and in the `src` directory create a `components` directory. In the `components` directory create a file `Main.jsx` with the following content:

```

import Constants from 'expo-constants';
import { Text, StyleSheet, View } from 'react-native';

const styles = StyleSheet.create({
  container: {
    marginTop: Constants.statusBarHeight,
    flexGrow: 1,
    flexShrink: 1,
  },
});

const Main = () => {
  return [
    <View style={styles.container}>
      <Text>Rate Repository Application</Text>
    </View>
  ];
};

export default Main;

```

Next, let's use the `Main` component in the `App` component in the `App.js` file which is located in our project's root directory. Replace the current content of the file with this:

```

import Main from './src/components/Main';

const App = () => {
  return <Main />;
};

export default App;

```

Manually reloading the application

As we have seen, Expo will automatically reload the application when we make changes to the code. However, there might be times when automatic reload isn't working and the application has to be reloaded manually. This can be achieved through the in-app developer menu.

You can access the developer menu by shaking your device or by selecting "Shake Gesture" inside the

Hardware menu in the iOS Simulator. You can also use the $\mathcal{H}D$ keyboard shortcut when your app is running in the iOS Simulator, or $\mathcal{H}M$ when running in an Android emulator on Mac OS and *Ctrl+M* on Windows and Linux.

Once the developer menu is open, simply press "Reload" to reload the application. After the application has been reloaded, automatic reloads should work without the need for a manual reload.

Exercise 10.3

Exercise 10.3: the reviewed repositories list

In this exercise, we will implement the first version of the reviewed repositories list. The list should contain the repository's full name, description, language, number of forks, number of stars, rating average and number of reviews. Luckily React Native provides a handy component for displaying a list of data, which is the [FlatList](#) component.

Implement components *RepositoryList* and *RepositoryItem* in the *components* directory's files *RepositoryList.jsx* and *RepositoryItem.jsx*. The *RepositoryList* component should render the *FlatList* component and *RepositoryItem* a single item on the list (hint: use the *FlatList* component's [renderItem](#) prop). Use this as the basis for the *RepositoryList.jsx* file:

```
import { FlatList, View, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  separator: {
    height: 10,
  },
});

const repositories = [
  {
    id: 'jaredpalmer.formik',
    fullName: 'jaredpalmer/formik',
    description: 'Build forms in React, without the tears',
    language: 'TypeScript',
    forksCount: 1589,
    stargazersCount: 21553,
    ratingAverage: 88,
    reviewCount: 4,
    ownerAvatarUrl: 'https://avatars2.githubusercontent.com/u/4060187?v=4',
  },
  {
    id: 'rails.rails',
    fullName: 'rails/rails',
    description: 'Ruby on Rails',
    language: 'Ruby',
    forksCount: 18349,
    stargazersCount: 45377,
    ratingAverage: 100,
    reviewCount: 2,
    ownerAvatarUrl: 'https://avatars1.githubusercontent.com/u/4223?v=4',
  },
]
```

```

[{"id": "django.django", "fullName": "django/django", "description": "The Web framework for perfectionists with deadlines.", "language": "Python", "forksCount": 21015, "stargazersCount": 48496, "ratingAverage": 73, "reviewCount": 5, "ownerAvatarUrl": "https://avatars2.githubusercontent.com/u/27804?v=4"}, {"id": "reduxjs.redux", "fullName": "reduxjs/redux", "description": "Predictable state container for JavaScript apps", "language": "TypeScript", "forksCount": 13902, "stargazersCount": 52869, "ratingAverage": 0, "reviewCount": 0, "ownerAvatarUrl": "https://avatars3.githubusercontent.com/u/13142323?v=4"}];

```

```

const ItemSeparator = () => <View style={styles.separator} />

const RepositoryList = () => {
  return [
    <FlatList
      data={repositories}
      ItemSeparatorComponent={ItemSeparator}
      // other props
    />
  ];
}

export default RepositoryList;

```

Do not alter the contents of the `repositories` variable, it should contain everything you need to complete this exercise. Render the `RepositoryList` component in the `Main` component which we previously added to the `Main.jsx` file. The reviewed repository list should roughly look something like this:

15.19  

 VoIP 4G+ LTE2  

Full name: jaredpalmer/formik

Description: Build forms in React, without the tears

Language: TypeScript

Stars: 21553

Forks: 1589

Reviews: 4

Rating: 88

Full name: rails/rails

Description: Ruby on Rails

Language: Ruby

Stars: 45377

Forks: 18349

Reviews: 2

Rating: 100

Full name: django/django

Description: The Web framework for perfectionists with deadlines.

Language: Python

Stars: 48496

Forks: 21015

Reviews: 5

Rating: 73

Full name: reduxjs/redux

Description: Predictable state container for JavaScript apps

Language: TypeScript

Stars: 52869

Forks: 13902

Reviews: 0

Rating: 0

Style

Now that we have a basic understanding of how core components work and we can use them to build

a simple user interface it is time to add some style. In [part 2](#) we learned that in the browser environment we can define React component's style properties using CSS. We had the option to either define these styles inline using the `style` prop or in a CSS file with a suitable selector.

There are many similarities in the way style properties are attached to React Native's core components and the way they are attached to DOM elements. In React Native most of the core components accept a prop called `style`. The `style` prop accepts an object with style properties and their values. These style properties are in most cases the same as in CSS, however, property names are in *camelCase*. This means that CSS properties such as `padding-top` and `font-size` are written as `paddingTop` and `fontSize`. Here is a simple example of how to use the `style` prop:

```
import { Text, View } from 'react-native';

const BigBlueText = () => {
  return [
    <View style={{ padding: 20 }}>
      <Text style={{ color: 'blue', fontSize: 24, fontWeight: '700' }}>
        Big blue text
      </Text>
    </View>
  ];
}
```

On top of the property names, you might have noticed another difference in the example. In CSS numerical property values commonly have a unit such as `px`, `%`, `em` or `rem`. In React Native all dimension-related property values such as `width`, `height`, `padding`, and `margin` as well as font sizes are *unitless*. These unitless numeric values represent *density-independent pixels*. In case you are wondering what are the available style properties for certain core components, check the [React Native Styling Cheat Sheet](#).

In general, defining styles directly in the `style` prop is not considered such a great idea, because it makes components bloated and unclear. Instead, we should define styles outside the component's render function using the `StyleSheet.create` method. The `StyleSheet.create` method accepts a single argument which is an object consisting of named style objects and it creates a `StyleSheet` style reference from the given object. Here is an example of how to refactor the previous example using the `StyleSheet.create` method:

```
import { Text, View, StyleSheet } from 'react-native';
const styles = StyleSheet.create({ container: { padding: 20 }, text: { color: 'blue', fontSize: 24, fontWeight: '700' } });
const BigBlueText = () => {
  return [
    <View style={styles.container}> <Text style={styles.text}> Big b
    lue text
    </Text>
  </View>
];
}
```

We create two named style objects, `styles.container` and `styles.text`. Inside the component, we can access specific style objects the same way we would access any key in a plain object.

In addition to an object, the `style` prop also accepts an array of objects. In the case of an array, the objects are merged from left to right so that latter-style properties take precedence. This works recursively, so we can have for example an array containing an array of styles and so forth. If an array contains values that evaluate to false, such as `null` or `undefined`, these values are ignored. This makes it easy to define *conditional styles* for example, based on the value of a prop. Here is an example of conditional styles:

```
import { Text, View, StyleSheet } from 'react-native'

const styles = StyleSheet.create({
  text: {
    color: 'grey',
    fontSize: 14,
  },
  blueText: {
    color: 'blue',
  },
  bigText: {
    fontSize: 24,
    fontWeight: '700',
  },
})

const FancyText = ({ isBlue, isBig, children }) => {
  const textStyles = [
    styles.text,
    isBlue && styles.blueText,
    isBig && styles.bigText,
  ];

  return <Text style={textStyles}>{children}</Text>;
}

const Main = () => {
  return (
    <>
      <FancyText>Simple text</FancyText>
      <FancyText isBlue>Blue text</FancyText>
      <FancyText isBig>Big text</FancyText>
      <FancyText isBig isBlue>
        Big blue text
      </FancyText>
    </>
  );
}
```

In the example, we use the `&&` operator with the expression `condition && exprIfTrue`. This expression yields `exprIfTrue` if the `condition` evaluates to true, otherwise it will yield `condition`, which in that case is a value that evaluates to false. This is an extremely widely used and handy shorthand. Another option would be to use the [conditional operator](#) like this:

```
condition ? exprIfTrue : exprIfFalse
```

Consistent user interface with theming

Let's stick with the concept of styling but with a bit wider perspective. Most of us have used a multitude of different applications and might agree that one trait that makes a good user interface is *consistency*. This means that the appearance of user interface components such as their font size, font family and color follows a consistent pattern. To achieve this we have to somehow *parametrize* the values of different style properties. This method is commonly known as *theming*.

Users of popular user interface libraries such as [Bootstrap](#) and [Material UI](#) might already be quite familiar with theming. Even though the theming implementations differ, the main idea is always to use variables such as `colors.primary` instead of "magic numbers" such as `#0366d6` when defining styles. This leads to increased consistency and flexibility.

Let's see how theming could work in practice in our application. We will be using a lot of text with different variations, such as different font sizes and colors. Because React Native does not support global styles, we should create our own `Text` component to keep the textual content consistent. Let's get started by adding the following theme configuration object in a `theme.js` file in the `src` directory:

```
const theme = {
  colors: {
    textPrimary: '#24292e',
    textSecondary: '#586069',
    primary: '#0366d6',
  },
  fontSizes: {
    body: 14,
    subheading: 16,
  },
  fonts: {
    main: 'System',
  },
  fontWeights: {
    normal: '400',
    bold: '700',
  },
};

export default theme;
```

Next, we should create the actual `Text` component which uses this theme configuration. Create a `Text.jsx` file in the `components` directory where we already have our other components. Add the following content to the `Text.jsx` file:

```
import { Text as NativeText, StyleSheet } from 'react-native';

import theme from '../theme';

const styles = StyleSheet.create({
  text: {
```

```

        color: theme.colors.textPrimary,
        fontSize: theme.fontSizes.body,
        fontFamily: theme.fonts.main,
        fontWeight: theme.fontWeights.normal,
    },
    colorTextSecondary: [
        color: theme.colors.textSecondary,
    ],
    colorPrimary: [
        color: theme.colors.primary,
    ],
    fontSizeSubheading: [
        fontSize: theme.fontSizes.subheading,
    ],
    fontWeightBold: [
        fontWeight: theme.fontWeights.bold,
    ],
},
),
}

const Text = ({ color, fontSize, fontWeight, style, ...props }) => {
    const textStyle = [
        styles.text,
        color === 'textSecondary' && styles.colorTextSecondary,
        color === 'primary' && styles.colorPrimary,
        fontSize === 'subheading' && styles.fontSizeSubheading,
        fontWeight === 'bold' && styles.fontWeightBold,
        style,
    ];
    return <NativeText style={textStyle} {...props} />;
};

export default Text;

```

Now we have implemented our text component. This text component has consistent color, font size and font weight variants that we can use anywhere in our application. We can get different text variations using different props like this:

```

import Text from './Text';

const Main = () => {
    return [
        <>
            <Text>Simple text</Text>
            <Text style={{ paddingBottom: 10 }}>Text with custom style</Text>
            <Text fontWeight="bold" fontSize="subheading">
                Bold subheading
            </Text>
            <Text color="textSecondary">Text with secondary color</Text>
        </>
    ];
};

```

```
export default Main;
```

Feel free to extend or modify this component if you feel like it. It might also be a good idea to create reusable text components such as *Subheading* which use the *Text* component. Also, keep on extending and modifying the theme configuration as your application progresses.

Using flexbox for layout

The last concept we will cover related to styling is implementing layouts with **flexbox**. Those who are more familiar with CSS know that flexbox is not related only to React Native, it has many use cases in web development as well. Those who know how flexbox works in web development won't probably learn that much from this section. Nevertheless, let's learn or revise the basics of flexbox.

Flexbox is a layout entity consisting of two separate components: a *flex container* and inside it a set of *flex items*. A Flex container has a set of properties that control the flow of its items. To make a component a flex container it must have the style property *display* set as *flex* which is the default value for the *display* property. Here is an example of a flex container:

```
import { View, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  flexContainer: [
    flexDirection: 'row',
  ],
});

const FlexboxExample = () => {
  return <View style={styles.flexContainer}>/* ... */</View>;
};
```

Perhaps the most important properties of a flex container are the following:

- **flexDirection** property controls the direction in which the flex items are laid out within the container. Possible values for this property are *row*, *row-reverse*, *column* (default value) and *column-reverse*. Flex direction *row* will lay out the flex items from left to right, whereas *column* from top to bottom. **-reverse* directions will just reverse the order of the flex items.
- **justifyContent** property controls the alignment of flex items along the main axis (defined by the *flexDirection* property). Possible values for this property are *flex-start* (default value), *flex-end*, *center*, *space-between*, *space-around* and *space-evenly*.
- **alignItems** property does the same as *justifyContent* but for the opposite axis. Possible values for this property are *flex-start*, *flex-end*, *center*, *baseline* and *stretch* (default value).

Let's move on to flex items. As mentioned, a flex container can contain one or many flex items. Flex items have properties that control how they behave in respect of other flex items in the same flex container. To make a component a flex item all you have to do is to set it as an immediate child of a flex container:

```
import { View, Text, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
```

```

flexContainer: [
  display: 'flex',
],
flexItemA: {
  flexGrow: 0,
  backgroundColor: 'green',
},
flexItemB: {
  flexGrow: 1,
  backgroundColor: 'blue',
},
),
),
};

const FlexboxExample = () => {
return (
<View style={styles.flexContainer}>
  <View style={styles.flexItemA}>
    <Text>Flex item A</Text>
  </View>
  <View style={styles.flexItemB}>
    <Text>Flex item B</Text>
  </View>
</View>
)
}

```

One of the most commonly used properties of flex items is the `flexGrow` property. It accepts a unitless value which defines the ability for a flex item to grow if necessary. If all flex items have a `flexGrow` of `1`, they will share all the available space evenly. If a flex item has a `flexGrow` of `0`, it will only use the space its content requires and leave the rest of the space for other flex items.

Here you can find how to simplify layouts with Flexbox gap: [Flexbox gap](#).

Next, read the article [A Complete Guide to Flexbox](#) which has comprehensive visual examples of flexbox. It is also a good idea to play around with the flexbox properties in the [Flexbox Playground](#) to see how different flexbox properties affect the layout. Remember that in React Native the property names are the same as the ones in CSS except for the *camelCase* naming. However, the *property values* such as `flex-start` and `space-between` are exactly the same.

NB: React Native and CSS has some differences regarding the flexbox. The most important difference is that in React Native the default value for the `flexDirection` property is `column`. It is also worth noting that the `flex` shorthand doesn't accept multiple values in React Native. More on React Native's flexbox implementation can be read in the [documentation](#).

Exercises 10.4-10.5

Exercise 10.4: the app bar

We will soon need to navigate between different views in our application. That is why we need an [app bar](#) to display tabs for switching between different views. Create a file `AppBar.jsx` in the `components` folder with the following content:

```

import { View, StyleSheet } from 'react-native';
import Constants from 'expo-constants';

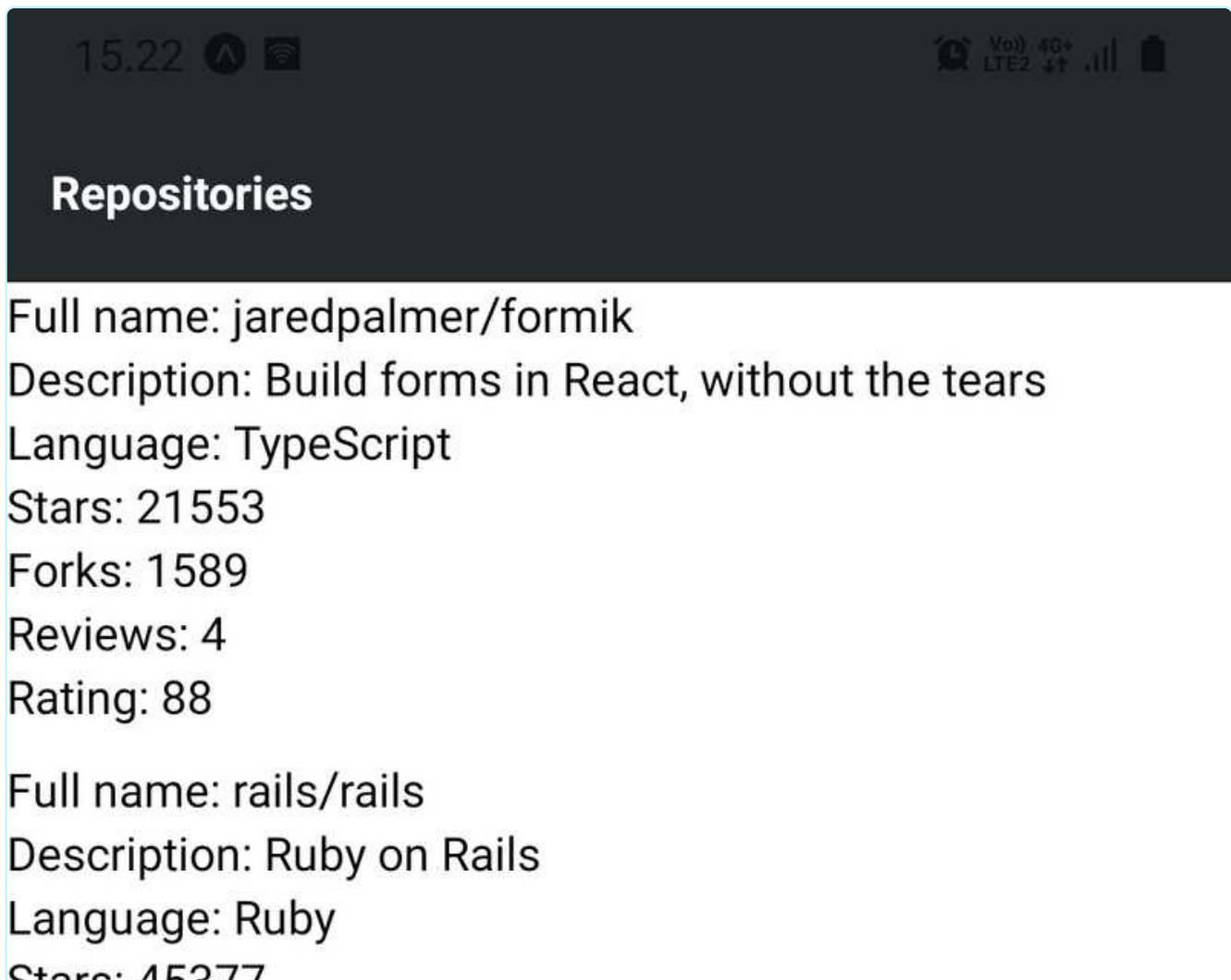
const styles = StyleSheet.create({
  container: {
    paddingTop: Constants.statusBarHeight,
    // ...
  },
  // ...
});

const AppBar = () => {
  return <View style={styles.container}>{/* ... */}</View>;
};

export default AppBar;

```

Now that the *AppBar* component will prevent the status bar from overlapping the content, you can remove the *marginTop* style we added for the *Main* component earlier in the *Main.jsx* file. The *AppBar* component should currently contain a tab with the text "*Repositories*". Make the tab pressable by using the *Pressable* component but you don't have to handle the *onPress* event in any way. Add the *AppBar* component to the *Main* component so that it is the uppermost component on the screen. The *AppBar* component should look something like this:



The background color of the app bar in the image is #24292e but you can use any other color as well. It might be a good idea to add the app bar's background color into the theme configuration so that it is easy to change it if needed. Another good idea might be to separate the app bar's tab into a component like *AppBarTab* so that it is easy to add new tabs in the future.

Exercise 10.5: polished reviewed repositories list

The current version of the reviewed repositories list looks quite grim. Modify the *RepositoryItem* component so that it also displays the repository author's avatar image. You can implement this by using the *Image* component. Counts, such as the number of stars and forks, larger than or equal to 1000 should be displayed in thousands with the precision of one decimal and with a "k" suffix. This means that for example fork count of 8439 should be displayed as "8.4k". Also, polish the overall look of the component so that the reviewed repositories list looks something like this:

The screenshot shows a mobile application interface for reviewing repositories. At the top, there is a dark header bar with the text "Repositories". Below this, the first repository entry is displayed:

jaredpalmer/formik
Build forms in React, without the tears
TypeScript

21.6k Stars 1.6k Forks 4 Reviews 88 Rating

The second repository entry is partially visible below it:

rails/rails
Ruby on Rails
Ruby

45.4k Stars 18.3k Forks 2 Reviews 100 Rating

The interface uses a clean, modern design with blue and white colors. Repository names and descriptions are bolded. Metrics like stars, forks, reviews, and rating are clearly labeled and aligned. The GitHub logo is present in the bottom right corner of the screenshot.



django/django

The Web framework for perfectionists with deadlines.

Python

48.5k

21k

5

73

Stars

Forks

Reviews

Rating



reduxjs/redux

In the image, the *Main* component's background color is set to `#e1e4e8` whereas *RepositoryItem* component's background color is set to `white`. The language tag's background color is `#0366d6` which is the value of the `colors.primary` variable in the theme configuration. Remember to exploit the *Text* component we implemented earlier. Also when needed, split the *RepositoryItem* component into smaller components.

Routing

When we start to expand our application we will need a way to transition between different views such as the repositories view and the sign-in view. In [part 7](#) we got familiar with [React router](#) library and learned how to use it to implement routing in a web application.

Routing in a React Native application is a bit different from routing in a web application. The main difference is that we can't reference pages with URLs, which we type into the browser's address bar, and can't navigate back and forth through the user's history using the browser's [history API](#). However, this is just a matter of the router interface we are using.

With React Native we can use the entire React router's core, including the hooks and components. The only difference to the browser environment is that we must replace the *BrowserRouter* with React Native compatible [NativeRouter](#), provided by the [react-router-native](#) library. Let's get started by installing the *react-router-native* library:

```
npm install react-router-native
```

Next, open the *App.js* file and add the *NativeRouter* component to the *App* component:

```
import { StatusBar } from 'expo-status-bar';
import { NativeRouter } from 'react-router-native';
```

```

import Main from './src/components/Main';

const App = () => {
  return [
    <>          <NativeRouter>          <Main />          </NativeRouter>          <StatusBar
style="auto" />      </>  </>
  ];
}

export default App;

```

Once the router is in place, let's add our first route to the *Main* component in the *Main.jsx* file:

```

import { StyleSheet, View } from 'react-native';
import { Router, Routes, Navigate } from 'react-router-native';
import RepositoryList from './RepositoryList';
import AppBar from './AppBar';
import theme from '../theme';

const styles = StyleSheet.create({
  container: {
    backgroundColor: theme.colors.mainBackground,
    flexGrow: 1,
    flexShrink: 1,
  },
});

const Main = () => {
  return [
    <View style={styles.container}>
      <AppBar />
      <Routes>          <Route path="/" element={<RepositoryList />} />          <Ro
ute path="*" element={<Navigate to="/" replace />} />          </Routes>      </View>
    </>
  ];
}

export default Main;

```

That's it! The last *Route* inside the *Routes* is for catching paths that don't match any previously defined path. In this case, we want to navigate to the home view.

Exercises 10.6-10.7

Exercise 10.6: the sign-in view

We will soon implement a form, that a user can use to *sign in* to our application. Before that, we must implement a view that can be accessed from the app bar. Create a file *SignIn.jsx* in the *components* directory with the following content:

```

import Text from './Text';

const SignIn = () => [

```

```
    return <Text>The sign-in view</Text>;
}

export default SignIn;
```

Set up a route for this *SignIn* component in the *Main* component. Also, add a tab with the text "Sign in" to the app bar next to the "Repositories" tab. Users should be able to navigate between the two views by pressing the tabs (hint: you can use the React router's [Link](#) component).

Exercise 10.7: scrollable app bar

As we are adding more tabs to our app bar, it is a good idea to allow horizontal scrolling once the tabs won't fit the screen. The [ScrollView](#) component is just the right component for the job.

Wrap the tabs in the *AppBar* component's tabs with a *ScrollView* component:

```
const AppBar = () => {
  return [
    <View style={styles.container}>
      <ScrollView horizontal>[* ... *]</ScrollView>      </View>
    ],
];
```

Setting the [horizontal](#) prop *true* will cause the *ScrollView* component to scroll horizontally once the content won't fit the screen. Note that, you will need to add suitable style properties to the *ScrollView* component so that the tabs will be laid in a *row* inside the flex container. You can make sure that the app bar can be scrolled horizontally by adding tabs until the last tab won't fit the screen. Just remember to remove the extra tabs once the app bar is working as intended.

Form state management

Now that we have a placeholder for the sign-in view the next step would be to implement the sign-in form. Before we get to that let's talk about forms from a wider perspective.

Implementation of forms relies heavily on state management. Using React's [useState](#) hook for state management might get the job done for smaller forms. However, it will quickly make state management for more complex forms quite tedious. Luckily there are many good libraries in the React ecosystem that ease the state management of forms. One of these libraries is [Formik](#).

The main concepts of Formik are the *context* and the *field*. However, the easiest way to do a simple form submit is by using [useFormik\(\)](#). It is a custom React hook that will return all Formik state and helpers directly.

There are some restrictions concerning the use of [useFormik\(\)](#). Read this to become familiar with [useFormik\(\)](#)

Let's see how this works by creating a form for calculating the [body mass index](#):

```
import { Text, TextInput, Pressable, View } from 'react-native';
import { useFormik } from 'formik';
```

```

const initialValues = {
  mass: '',
  height: '',
};

const getBodyMassIndex = (mass, height) => {
  return Math.round(mass / Math.pow(height, 2));
};

const BodyMassIndexForm = ({ onSubmit }) => {
  const formik = useFormik({
    initialValues,
    onSubmit,
  });

  return (
    <View>
      <TextInput
        placeholder="Weight (kg)"
        value={formik.values.mass}
        onChangeText={formik.handleChange('mass')}
      />
      <TextInput
        placeholder="Height (m)"
        value={formik.values.height}
        onChangeText={formik.handleChange('height')}
      />
      <Pressable onPress={formik.handleSubmit}>
        <Text>Calculate</Text>
      </Pressable>
    </View>
  );
};

const BodyMassIndexCalculator = () => {
  const onSubmit = values => {
    const mass = parseFloat(values.mass);
    const height = parseFloat(values.height);

    if (!isNaN(mass) && !isNaN(height) && height !== 0) {
      console.log(`Your body mass index is: ${getBodyMassIndex(mass, height)}`);
    }
  };

  return <BodyMassIndexForm onSubmit={onSubmit} />;
};

export default BodyMassIndexCalculator;

```

This example is not part of our application, so you don't need to add this code to the application. You can however try it out for example in [Expo Snack](#). Expo Snack is an online editor for React Native, similar to [JSFiddle](#) and [CodePen](#). It is a useful platform for quickly trying out code. You can share Expo

Snacks with others using a link or embedding them as a *Snack Player* on a website. You might have bumped into Snack Players for example in this material and React Native documentation.

Exercise 10.8

Exercise 10.8: the sign-in form

Implement a sign-in form to the *SignIn* component we added earlier in the *SignIn.jsx* file. The sign-in form should include two text fields, one for the username and one for the password. There should also be a button for submitting the form. You don't need to implement an *onSubmit* callback function, it is enough that the form values are logged using *console.log* when the form is submitted:

```
const onSubmit = [values] => [
  console.log(values),
];
```

The first step is to install Formik:

```
npm install formik
```

You can use the `secureTextEntry` prop in the *TextInput* component to obscure the password input.

The sign-in form should look something like this:

[Repositories](#)[Sign in](#)

Username

Password

[Sign in](#)

Form validation

Formik offers two approaches to form validation: a validation function or a validation schema. A validation function is a function provided for the *Formik* component as the value of the `validate` prop. It receives the form's values as an argument and returns an object containing possible field-specific error messages.

The second approach is the validation schema which is provided for the *Formik* component as the value of the `validationSchema` prop. This validation schema can be created with a validation library called [Yup](#). Let's get started by installing Yup:

```
npm install yup
```

Next, as an example, let's create a validation schema for the body mass index form we implemented earlier. We want to validate that both *mass* and *height* fields are present and they are numeric. Also, the value of *mass* should be greater or equal to 1 and the value of *height* should be greater or equal to 0.5. Here is how we define the schema:

```

import * as yup from 'yup';
// ...

const validationSchema = yup.object().shape({
  mass: yup.number().min(1, 'Weight must be greater or equal to 1')
    .required('Weight is required'),
  height: yup.number().min(0.5, 'Height must be greater or equal to 0.5')
    .required('Height is required'),
});
const BodyMassIndexForm = ({ onSubmit }) => {
  const formik = useFormik({
    initialValues,
    validationSchema,
    onSubmit,
  });
  return (
    <View>
      <TextInput
        placeholder="Weight (kg)"
        value={formik.values.mass}
        onChangeText={formik.handleChange('mass')}
      />
      {formik.touched.mass && formik.errors.mass && (
        <Text style={{ color: 'red' }}>{formik.errors.mass}</Text>
      )}
      <TextInput
        placeholder="Height (m)"
        value={formik.values.height}
        onChangeText={formik.handleChange('height')}
      />
      {formik.touched.height && formik.errors.height && (
        <Text style={{ color: 'red' }}>{formik.errors.height}</Text>
      )}
      <Pressable onPress={formik.handleSubmit}>
        <Text>Calculate</Text>
      </Pressable>
    </View>
  );
};

const BodyMassIndexCalculator = () => {
  // ...

```

Be aware that you need to include these Text components within the View returned by the form to display the validation errors:

```
{formik.touched.mass && formik.errors.mass && (
  <Text style={{ color: 'red' }}>{formik.errors.mass}</Text>
)}
```

```
{formik.touched.height && formik.errors.height && (
  <Text style={{ color: 'red' }}>{formik.errors.height}</Text>
```

```
) }
```

The validation is performed by default every time a field's value changes and when the `handleSubmit` function is called. If the validation fails, the function provided for the `onSubmit` prop of the `Formik` component is not called.

Exercise 10.9

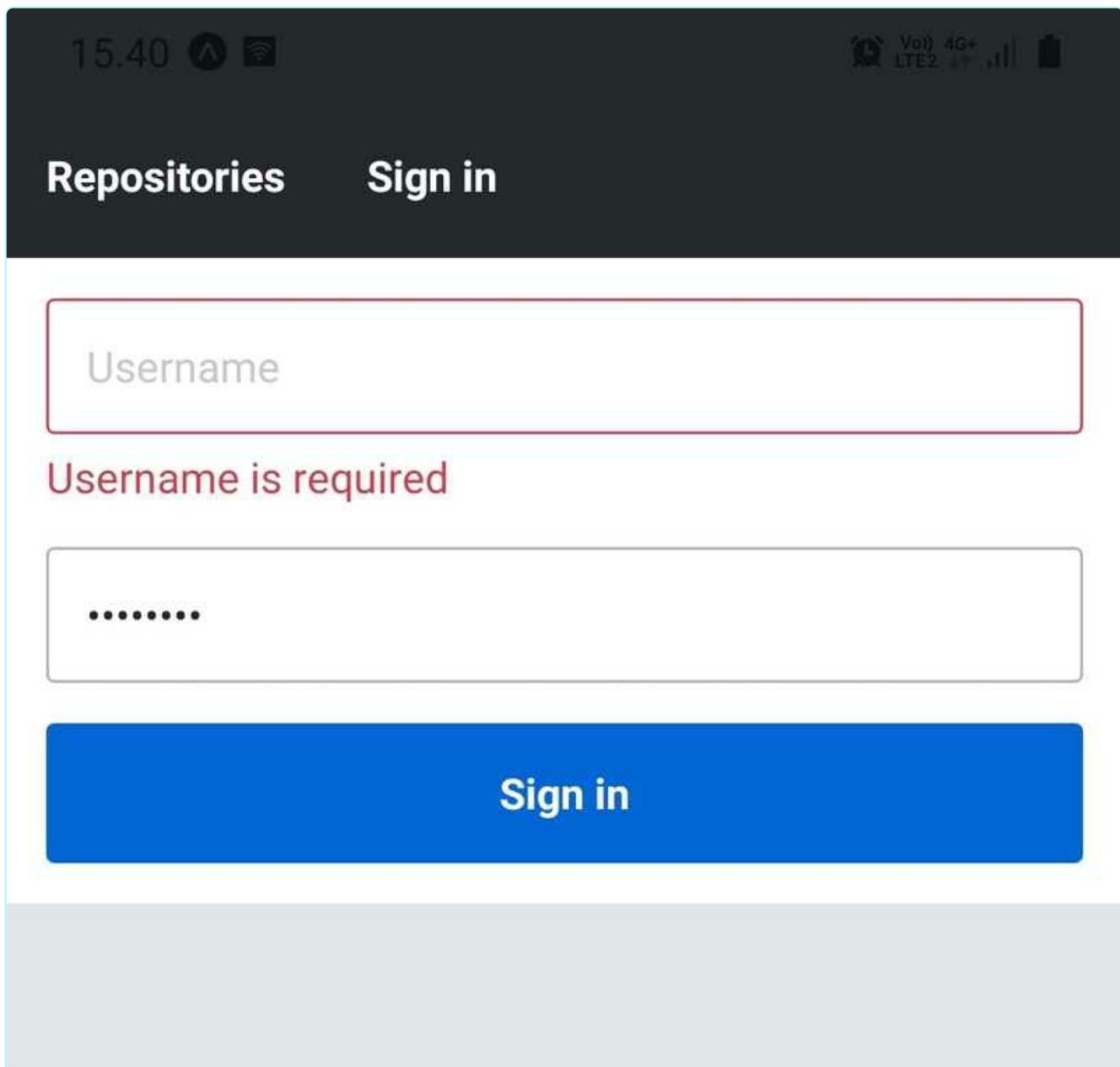
Exercise 10.9: validating the sign-in form

Validate the sign-in form so that both username and password fields are required. Note that the `onSubmit` callback implemented in the previous exercise, *should not be called* if the form validation fails.

The current implementation of the `TextInput` component should display an error message if a touched field has an error. Emphasize this error message by giving it a red color.

On top of the red error message, give an invalid field a visual indication of an error by giving it a red border color. Remember that if a field has an error, the `TextInput` component sets the `TextInput` component's `error` prop as `true`. You can use the value of the `error` prop to attach conditional styles to the `TextInput` component.

Here's what the sign-in form should roughly look like with an invalid field:



The red color used in this implementation is `#d73a4a`.

Platform-specific code

A big benefit of React Native is that we don't need to worry about whether the application is run on an Android or iOS device. However, there might be cases where we need to execute *platform-specific code*. Such cases could be for example using a different implementation of a component on a different platform.

We can access the user's platform through the *Platform.OS* constant:

```
import { React } from 'react';
import { Platform, Text, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  text: {
    color: Platform.OS === 'android' ? 'green' : 'blue',
  },
});
```

```

});;

const WhatIsMyPlatform = () => [
  return <Text style={styles.text}>Your platform is: {Platform.OS}</Text>;
];

```

Possible values for the `Platform.OS` constants are `android` and `ios`. Another useful way to define platform-specific code branches is to use the `Platform.select` method. Given an object where keys are one of `ios`, `android`, `native` and `default`, the `Platform.select` method returns the most fitting value for the platform the user is currently running on. We can rewrite the `styles` variable in the previous example using the `Platform.select` method like this:

```

const styles = StyleSheet.create({
  text: {
    color: Platform.select({
      android: 'green',
      ios: 'blue',
      default: 'black'
    })
  }
});

```

We can even use the `Platform.select` method to require a platform-specific component:

```

const MyComponent = Platform.select({
  ios: () => require('./MyIOSComponent'),
  android: () => require('./MyAndroidComponent'),
});

<MyComponent />;

```

However, a more sophisticated method for implementing and importing platform-specific components (or any other piece of code) is to use the `.ios.jsx` and `.android.jsx` file extensions. Note that the `.jsx` extension can as well be any extension recognized by the bundler, such as `.js`. We can for example have files `Button.ios.jsx` and `Button.android.jsx` which we can import like this:

```

import Button from './Button';

const PlatformSpecificButton = () => [
  return <Button />;
];

```

Now, the Android bundle of the application will have the component defined in the `Button.android.jsx` whereas the iOS bundle the one defined in the `Button.ios.jsx` file.

Exercise 10.10

Exercise 10.10: a platform-specific font

Currently, the font family of our application is set to `System` in the theme configuration located in the

theme.js file. Instead of the *System* font, use a platform-specific [Sans-serif](#) font. On the Android platform, use the *Roboto* font and on the iOS platform, use the *Arial* font. The default font can be *System*.

This was the last exercise in this section. It's time to push your code to GitHub and mark all of your finished exercises to the [exercise submission system](#). Note that exercises in this section should be submitted to the section named part 2 in the exercise submission system.

10c: Communicating with server

Note: This course material was updated in Feb 2024. Some updates are not compatible anymore with older material. We recommend a fresh start with this new Part 10 material. However, if you're returning to this course after a break, and you want to continue the exercises in your older project, please use [Part 10 material before the upgrade](#).

So far we have implemented features to our application without any actual server communication. For example, the reviewed repositories list we have implemented uses mock data and the sign in form doesn't send the user's credentials to any authentication endpoint. In this section, we will learn how to communicate with a server using HTTP requests, how to use Apollo Client in a React Native application, and how to store data in the user's device.

Soon we will learn how to communicate with a server in our application. Before we get to that, we need a server to communicate with. For this purpose, we have a completed server implementation in the [rate-repository-api](#) repository. The rate-repository-api server fulfills all our application's API needs during this part. It uses [SQLite](#) database which doesn't need any setup and provides an Apollo GraphQL API along with a few REST API endpoints.

Before heading further into the material, set up the rate-repository-api server by following the setup instructions in the repository's [README](#). Note that if you are using an emulator for development it is recommended to run the server and the emulator *on the same computer*. This eases network requests considerably.

HTTP requests

React Native provides [Fetch API](#) for making HTTP requests in our applications. React Native also supports the good old [XMLHttpRequest API](#) which makes it possible to use third-party libraries such as [Axios](#). These APIs are the same as the ones in the browser environment and they are globally available without the need for an import.

People who have used both Fetch API and XMLHttpRequest API most likely agree that the Fetch API is easier to use and more modern. However, this doesn't mean that XMLHttpRequest API doesn't have its uses. For the sake of simplicity, we will be only using the Fetch API in our examples.

Sending HTTP requests using the Fetch API can be done using the *fetch* function. The first argument of the function is the URL of the resource:

```
fetch('https://my-api.com/get-end-point');
```

The default request method is *GET*. The second argument of the *fetch* function is an options object, which you can use to for example to specify a different request method, request headers, or request body:

```

fetch('https://my-api.com/post-end-point', {
  method: 'POST',
  headers: [
    Accept: 'application/json',
    'Content-Type': 'application/json',
  ],
  body: JSON.stringify({
    firstParam: 'firstValue',
    secondParam: 'secondValue',
  })
});

```

Note that these URLs are made up and won't (most likely) send a response to your requests. In comparison to Axios, the Fetch API operates on a bit lower level. For example, there isn't any request or response body serialization and parsing. This means that you have to for example set the *Content-Type* header by yourself and use *JSON.stringify* method to serialize the request body.

The *fetch* function returns a promise which resolves a [Response](#) object. Note that error status codes such as 400 and 500 *are not rejected* like for example in Axios. In case of a JSON formatted response we can parse the response body using the *Response.json* method:

```

const fetchMovies = async () => {
  const response = await fetch('https://reactnative.dev/movies.json');
  const json = await response.json();

  return json;
};

```

For a more detailed introduction to the Fetch API, read the [Using Fetch](#) article in the MDN web docs.

Next, let's try the Fetch API in practice. The rate-repository-api server provides an endpoint for returning a paginated list of reviewed repositories. Once the server is running, you should be able to access the endpoint at <http://localhost:5000/api/repositories> (unless you have changed the port). The data is paginated in a common [cursor based pagination format](#). The actual repository data is behind the *node* key in the *edges* array.

Unfortunately, if we're using external device, we can't access the server directly in our application by using the <http://localhost:5000/api/repositories> URL. To make a request to this endpoint in our application we need to access the server using its IP address in its local network. To find out what it is, open the Expo development tools by running *npm start*. In the console you should be able to see an URL starting with *exp://* below the QR code, after the "Metro waiting on" text:



```
> Metro waiting on exp://192.168.1.33:19000
> Scan the QR code above with Expo Go (Android) or the Camera app (iOS)

> Press a | open Android
> Press i | open iOS simulator
> Press w | open web
```

Copy the IP address between the `exp://` and `:`, which is in this example `192.168.1.33`. Construct an URL in format `http://<IP_ADDRESS>:5000/api/repositories` and open it in the browser. You should see the same response as you did with the `localhost` URL.

Now that we know the end point's URL let's use the actual server-provided data in our reviewed repositories list. We are currently using mock data stored in the `repositories` variable. Remove the `repositories` variable and replace the usage of the mock data with this piece of code in the `RepositoryList.jsx` file in the `components` directory:

```
import { useState, useEffect } from 'react';
// ...

const RepositoryList = () => {
  const [repositories, setRepositories] = useState([]);

  const fetchRepositories = async () => {
    // Replace the IP address part with your own IP address!
    const response = await fetch('http://192.168.1.33:5000/api/repositories');
    const json = await response.json();

    console.log(json);

    setRepositories(json);
  };

  useEffect(() => {
    fetchRepositories();
  }, []);

  // Get the nodes from the edges array
  const repositoryNodes = repositories
    ? repositories.edges.map(edge => edge.node)
    : [];

  return (
    <FlatList
      data={repositoryNodes}
```

```

        // Other props
      />
    );
}

export default RepositoryList;

```

We are using the React's `useState` hook to maintain the repository list state and the `useEffect` hook to call the `fetchRepositories` function when the `RepositoryList` component is mounted. We extract the actual repositories into the `repositoryNodes` variable and replace the previously used `repositories` variable in the `FlatList` component's `data` prop with it. Now you should be able to see actual server-provided data in the reviewed repositories list.

It is usually a good idea to log the server's response to be able to inspect it as we did in the `fetchRepositories` function. You should be able to see this log message in the Expo development tools if you navigate to your device's logs as we learned in the [Debugging](#) section. If you are using the Expo's mobile app for development and the network request is failing, make sure that the computer you are using to run the server and your phone are *connected to the same Wi-Fi network*. If that's not possible either use an emulator in the same computer as the server is running in or set up a tunnel to the localhost, for example, using [Ngrok](#).

The current data fetching code in the `RepositoryList` component could do with some refactoring. For instance, the component is aware of the network request's details such as the end point's URL. In addition, the data fetching code has lots of reuse potential. Let's refactor the component's code by extract the data fetching code into its own hook. Create a directory `hooks` in the `src` directory and in that `hooks` directory create a file `useRepositories.js` with the following content:

```

import { useState, useEffect } from 'react';

const useRepositories = () => {
  const [repositories, setRepositories] = useState([]);
  const [loading, setLoading] = useState(false);

  const fetchRepositories = async () => {
    setLoading(true);

    // Replace the IP address part with your own IP address!
    const response = await fetch('http://192.168.1.33:5000/api/repositories');
    const json = await response.json();

    setLoading(false);
    setRepositories(json);
  };

  useEffect(() => {
    fetchRepositories();
  }, []);

  return { repositories, loading, refetch: fetchRepositories };
};

export default useRepositories;

```

Now that we have a clean abstraction for fetching the reviewed repositories, let's use the `useRepositories` hook in the `RepositoryList` component:

```
// ...
import useRepositories from '../hooks/useRepositories';
const RepositoryList = () => {
  const { repositories } = useRepositories();
  const repositoryNodes = repositories
    ? repositories.edges.map(edge => edge.node)
    : [];

  return (
    <FlatList
      data={repositoryNodes}
      // Other props
    />
  );
}

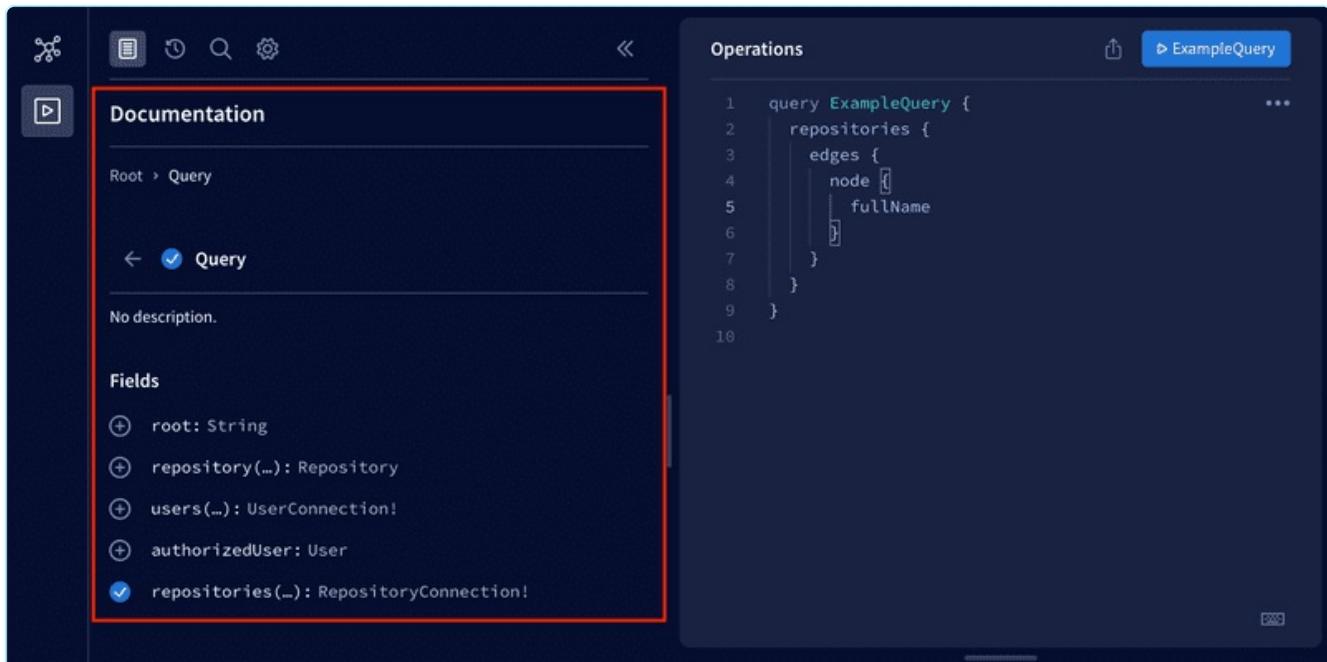
export default RepositoryList;
```

That's it, now the `RepositoryList` component is no longer aware of the way the repositories are acquired. Maybe in the future, we will acquire them through a GraphQL API instead of a REST API. We will see what happens.

GraphQL and Apollo client

In [part 8](#) we learned about GraphQL and how to send GraphQL queries to an Apollo Server using the [Apollo Client](#) in React applications. The good news is that we can use the Apollo Client in a React Native application exactly as we would with a React web application.

As mentioned earlier, the `rate-repository-api` server provides a GraphQL API which is implemented with Apollo Server. Once the server is running, you can access the [Apollo Sandbox](#) at <http://localhost:4000>. Apollo Sandbox is a tool for making GraphQL queries and inspecting the GraphQL APIs schema and documentation. If you need to send a query in your application *always* test it with the Apollo Sandbox first before implementing it in the code. It is much easier to debug possible problems in the query in the Apollo Sandbox than in the application. If you are uncertain what the available queries are or how to use them, you can see the documentation next to the operations editor:



In our React Native application, we will be using the same [@apollo/client](#) library as in part 8. Let's get started by installing the library along with the [graphql](#) library which is required as a peer dependency:

```
npm install @apollo/client graphql
```

Before we can start using Apollo Client, we will need to slightly configure the Metro bundler so that it handles the `.cjs` file extensions used by the Apollo Client. First, let's install the [@expo/metro-config](#) package which has the default Metro configuration:

```
npm install @expo/metro-config@0.17.4
```

Then, we can add the following configuration to a `metro.config.js` in the root directory of our project:

```
const { getDefaultConfig } = require('@expo/metro-config');

const defaultConfig = getDefaultConfig(__dirname);

defaultConfig.resolver.sourceExts.push('cjs');

module.exports = defaultConfig;
```

Restart the Expo development tools so that changes in the configuration are applied.

Now that the Metro configuration is in order, let's create a utility function for creating the Apollo Client with the required configuration. Create a `utils` directory in the `src` directory and in that `utils` directory create a file `apolloClient.js`. In that file configure the Apollo Client to connect to the Apollo Server:

```
import { ApolloClient, InMemoryCache } from '@apollo/client';
```

```

const createApolloClient = () => {
  return new ApolloClient({
    uri: 'http://192.168.1.100:4000/graphql',
    cache: new InMemoryCache(),
  });
};

export default createApolloClient;

```

The URL used to connect to the Apollo Server is otherwise the same as the one you used with the Fetch API except the port is *4000* and the path is */graphql*. Lastly, we need to provide the Apollo Client using the [ApolloProvider](#) context. We will add it to the *App* component in the *App.js* file:

```

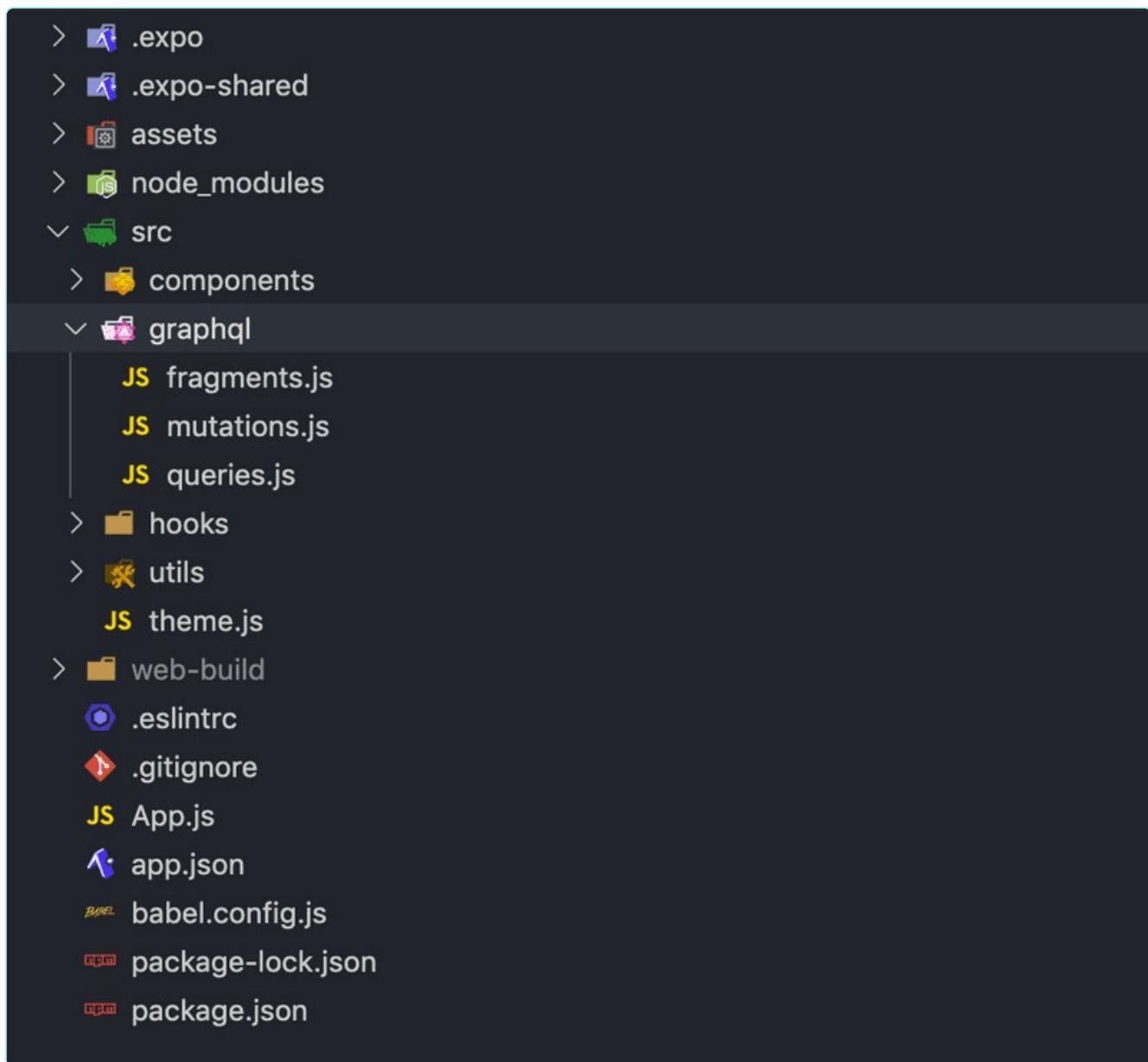
import { NativeRouter } from 'react-router-native';
import { ApolloProvider } from '@apollo/client';
import Main from './src/components/Main';
import createApolloClient from './src/utils/apolloClient';
const apolloClient = createApolloClient();
const App = () => {
  return (
    <NativeRouter>
      <ApolloProvider client={apolloClient}>          <Main />
      </ApolloProvider>    </NativeRouter>
    )
};

export default App;

```

Organizing GraphQL related code

It is up to you how to organize the GraphQL related code in your application. However, for the sake of a reference structure, let's have a look at one quite simple and efficient way to organize the GraphQL related code. In this structure, we define queries, mutations, fragments, and possibly other entities in their own files. These files are located in the same directory. Here is an example of the structure you can use to get started:



You can import the `gql` template literal tag used to define GraphQL queries from `@apollo/client` library. If we follow the structure suggested above, we could have a `queries.js` file in the `graphql` directory for our application's GraphQL queries. Each of the queries can be stored in a variable and exported like this:

```
import { gql } from '@apollo/client'

export const GET_REPOSITORIES = gql`  
query {  
  repositories {  
    /* ... */  
  }  
}  
  
// other queries...
```

We can import these variables and use them with the `useQuery` hook like this:

```

import { useQuery } from '@apollo/client';

import { GET_REPOSITORIES } from '../graphql/queries';

const Component = () => {
  const [ data, error, loading ] = useQuery(GET_REPOSITORIES);
  // ...
};


```

The same goes for organizing mutations. The only difference is that we define them in a different file, *mutations.js*. It is recommended to use **fragments** in queries to avoid retyping the same fields over and over again.

Evolving the structure

Once our application grows larger there might be times when certain files grow too large to manage. For example, we have component *A* which renders the components *B* and *C*. All these components are defined in a file *A.jsx* in a *components* directory. We would like to extract components *B* and *C* into their own files *B.jsx* and *C.jsx* without major refactors. We have two options:

- Create files *B.jsx* and *C.jsx* in the *components* directory. This results in the following structure:

```

components/
  A.jsx
  B.jsx
  C.jsx
  ...

```

- Create a directory *A* in the *components* directory and create files *B.jsx* and *C.jsx* there. To avoid breaking components that import the *A.jsx* file, move the *A.jsx* file to the *A* directory and rename it to *index.jsx*. This results in the following structure:

```

components/
  A/
    B.jsx
    C.jsx
    index.jsx
  ...

```

The first option is fairly decent, however, if components *B* and *C* are not reusable outside the component *A*, it is useless to bloat the *components* directory by adding them as separate files. The second option is quite modular and doesn't break any imports because importing a path such as *./A* will match both *A.jsx* and *A/index.jsx*.

Exercise 10.11

Exercise 10.11: fetching repositories with Apollo Client

We want to replace the Fetch API implementation in the *useRepositories* hook with a GraphQL query. Open the Apollo Sandbox at <http://localhost:4000> and take a look at the documentation next to the

operations editor. Look up the *repositories* query. The query has some arguments, however, all of these are optional so you don't need to specify them. In the Apollo Sandbox form a query for fetching the repositories with the fields you are currently displaying in the application. The result will be paginated and it contains the up to first 30 results by default. For now, you can ignore the pagination entirely.

Once the query is working in the Apollo Sandbox, use it to replace the Fetch API implementation in the *useRepositories* hook. This can be achieved using the *useQuery* hook. The *gql* template literal tag can be imported from the *@apollo/client* library as instructed earlier. Consider using the structure recommended earlier for the GraphQL related code. To avoid future caching issues, use the *cache-and-network fetch policy* in the query. It can be used with the *useQuery* hook like this:

```
useQuery[MY_QUERY, {
  fetchPolicy: 'cache-and-network',
  // Other options
}];
```

The changes in the *useRepositories* hook should not affect the *RepositoryList* component in any way.

Environment variables

Every application will most likely run in more than one environment. Two obvious candidates for these environments are the development environment and the production environment. Out of these two, the development environment is the one we are running the application right now. Different environments usually have different dependencies, for example, the server we are developing locally might use a local database whereas the server that is deployed to the production environment uses the production database. To make the code environment independent we need to parametrize these dependencies. At the moment we are using one very environment dependant hardcoded value in our application: the URL of the server.

We have previously learned that we can provide running programs with environment variables. These variables can be defined in the command line or using environment configuration files such as *.env* files and third-party libraries such as *Dotenv*. Unfortunately, React Native doesn't have direct support for environment variables. However, we can access the Expo configuration defined in the *app.json* file at runtime from our JavaScript code. This configuration can be used to define and access environment dependant variables.

The configuration can be accessed by importing the *Constants* constant from the *expo-constants* module as we have done a few times before. Once imported, the *Constants.expoConfig* property will contain the configuration. Let's try this by logging *Constants.expoConfig* in the *App* component:

```
import { NativeRouter } from 'react-router-native';
import { ApolloProvider } from '@apollo/client';
import Constants from 'expo-constants';
import Main from './src/components/Main';
import createApolloClient from './src/utils/apolloClient';

const apolloClient = createApolloClient();

const App = () => [
  console.log(Constants.expoConfig),
  return [
    <ApolloProvider client={apolloClient}>
      <Main />
    </ApolloProvider>
  ],
];
```

```
<NativeRouter>
  <ApolloProvider client={apolloClient}>
    <Main />
  </ApolloProvider>
</NativeRouter>
}
}

export default App;
```

You should now see the configuration in the logs.

The next step is to use the configuration to define environment dependant variables in our application. Let's get started by renaming the `app.json` file to `app.config.js`. Once the file is renamed, we can use JavaScript inside the configuration file. Change the file contents so that the previous object:

```
{
  "expo": [
    "name": "rate-repository-app",
    // rest of the configuration...
  ],
}
```

Is turned into an export, which contains the contents of the `expo` property:

```
export default {
  name: 'rate-repository-app',
  // rest of the configuration...
};
```

Expo has reserved an `extra` property in the configuration for any application-specific configuration. To see how this works, let's add an `env` variable into our application's configuration. Note, that the older versions used (now deprecated) manifest instead of `expoConfig`.

```
export default {
  name: 'rate-repository-app',
  // rest of the configuration...
  extra: { env: 'development' },
};
```

If you make changes in configuration, the restart may not be enough. You may need to start the application with cache cleared by command:

```
npx expo start --clear
```

Now, restart Expo development tools to apply the changes and you should see that the value of `Constants.expoConfig` property has changed and now includes the `extra` property containing our application-specific configuration. Now the value of the `env` variable is accessible through the `Constants.expoConfig.extra.env` property.

Because using hard coded configuration is a bit silly, let's use an environment variable instead:

```
export default {
  name: 'rate-repository-app',
  // rest of the configuration...
  extra: [
    env: process.env.ENV,
  ],
};
```

As we have learned, we can set the value of an environment variable through the command line by defining the variable's name and value before the actual command. As an example, start Expo development tools and set the environment variable *ENV* as *test* like this:

```
ENV=test npm start
```

If you take a look at the logs, you should see that the *Constants.expoConfig.extra.env* property has changed.

We can also load environment variables from an *.env* file as we have learned in the previous parts. First, we need to install the [Dotenv](#) library:

```
npm install dotenv
```

Next, add a *.env* file in the root directory of our project with the following content:

```
ENV=development
```

Finally, import the library in the *app.config.js* file:

```
import 'dotenv/config';
export default {
  name: 'rate-repository-app',
  // rest of the configuration...
  extra: [
    env: process.env.ENV,
  ],
};
```

You need to restart Expo development tools to apply the changes you have made to the *.env* file.

Note that it is *never* a good idea to put sensitive data into the application's configuration. The reason for this is that once a user has downloaded your application, they can, at least in theory, reverse engineer your application and figure out the sensitive data you have stored into the code.

Exercise 10.12

Exercise 10.12: environment variables

Instead of the hardcoded Apollo Server's URL, use an environment variable defined in the *.env* file when initializing the Apollo Client. You can name the environment variable for example *APOLLO_URI*.

Do not try to access environment variables like `process.env.APOLLO_URI` outside the `app.config.js` file. Instead use the `Constants.expoConfig.extra` object like in the previous example. In addition, do not import the `dotenv` library outside the `app.config.js` file or you will most likely face errors.

Storing data in the user's device

There are times when we need to store some persisted pieces of data in the user's device. One such common scenario is storing the user's authentication token so that we can retrieve it even if the user closes and reopens our application. In web development, we have used the browser's `localStorage` object to achieve such functionality. React Native provides similar persistent storage, the [AsyncStorage](#).

We can use the `npx expo install` command to install the version of the `@react-native-async-storage/async-storage` package that is suitable for our Expo SDK version:

```
npx expo install @react-native-async-storage/async-storage
```

The API of the `AsyncStorage` is in many ways same as the `localStorage` API. They are both key-value storages with similar methods. The biggest difference between the two is that, as the name implies, the operations of `AsyncStorage` are *asynchronous*.

Because `AsyncStorage` operates with string keys in a global namespace it is a good idea to create a simple abstraction for its operations. This abstraction can be implemented for example using a [class](#). As an example, we could implement a shopping cart storage for storing the products user wants to buy:

```
import AsyncStorage from '@react-native-async-storage/async-storage';

class ShoppingCartStorage {
  constructor(namespace = 'shoppingCart') {
    this.namespace = namespace;
  }

  async getProducts() {
    const rawProducts = await AsyncStorage.getItem(`:${this.namespace}:products`);
    return rawProducts ? JSON.parse(rawProducts) : [];
  }

  async addProduct(productId) {
    const currentProducts = await this.getProducts();
    const newProducts = [...currentProducts, productId];

    await AsyncStorage.setItem(`:${this.namespace}:products`, JSON.stringify(newProducts));
  }

  async clearProducts() {
    await AsyncStorage.removeItem(`:${this.namespace}:products`);
  }
}
```

```

    await AsyncStorage.removeItem(`#${this.namespace}:products`);

}

const doShopping = async () => {
  const shoppingCartA = new ShoppingCartStorage('shoppingCartA');
  const shoppingCartB = new ShoppingCartStorage('shoppingCartB');

  await shoppingCartA.addProduct('chips');
  await shoppingCartA.addProduct('soda');

  await shoppingCartB.addProduct('milk');

  const productsA = await shoppingCartA.getProducts();
  const productsB = await shoppingCartB.getProducts();

  console.log(productsA, productsB);

  await shoppingCartA.clearProducts();
  await shoppingCartB.clearProducts();
}

doShopping();

```

Because `AsyncStorage` keys are global, it is usually a good idea to add a *namespace* for the keys. In this context, the namespace is just a prefix we provide for the storage abstraction's keys. Using the namespace prevents the storage's keys from colliding with other `AsyncStorage` keys. In this example, the namespace is defined as the constructor's argument and we are using the `namespace:key` format for the keys.

We can add an item to the storage using the `AsyncStorage.setItem` method. The first argument of the method is the item's key and the second argument its value. The value *must be a string*, so we need to serialize non-string values as we did with the `JSON.stringify` method. The `AsyncStorage.getItem` method can be used to get an item from the storage. The argument of the method is the item's key, of which value will be resolved. The `AsyncStorage.removeItem` method can be used to remove the item with the provided key from the storage.

NB: `SecureStore` is similar persisted storage as the `AsyncStorage` but it encrypts the stored data. This makes it more suitable for storing more sensitive data such as the user's credit card number.

Exercises 10.13. - 10.14

Exercise 10.13: the sign in form mutation

The current implementation of the sign in form doesn't do much with the submitted user's credentials. Let's do something about that in this exercise. First, read the rate-repository-api server's [authentication documentation](#) and test the provided queries and mutations in the Apollo Sandbox. If the database doesn't have any users, you can populate the database with some seed data. Instructions for this can be found in the [getting started](#) section of the README.

Once you have figured out how the authentication works, create a file `useSignIn.js` in the `hooks` directory. In that file implement a `useSignIn` hook that sends the `authenticate` mutation using the

`useMutation` hook. Note that the `authenticate` mutation has a *single* argument called `credentials`, which is of type `AuthenticateInput`. This `input type` contains `username` and `password` fields.

The return value of the hook should be a tuple `[signIn, result]` where `result` is the mutations result as it is returned by the `useMutation` hook and `signIn` a function that runs the mutation with a `{ username, password }` object argument. Hint: don't pass the mutation function to the return value directly. Instead, return a function that calls the mutation function like this:

```
const useSignIn = () => {
  const [mutate, result] = useMutation(/* mutation arguments */);

  const signIn = async ({ username, password }) => {
    // call the mutate function here with the right arguments
  };

  return [signIn, result];
};
```

Once the hook is implemented, use it in the `SignIn` component's `onSubmit` callback for example like this:

```
const SignIn = () => {
  const [signIn] = useSignIn();

  const onSubmit = async (values) => {
    const { username, password } = values;

    try {
      const { data } = await signIn({ username, password });
      console.log(data);
    } catch (e) {
      console.log(e);
    }
  };

  // ...
};
```

This exercise is completed once you can log the user's `authenticate` mutations result after the sign in form has been submitted. The mutation result should contain the user's access token.

Exercise 10.14: storing the access token step1

Now that we can obtain the access token we need to store it. Create a file `authStorage.js` in the `utils` directory with the following content:

```
import AsyncStorage from '@react-native-async-storage/async-storage';

class AuthStorage {
  constructor(namespace = 'auth') {
    this.namespace = namespace;
  }

  // ...
}
```

```

    }

    getAccessToken() {
        // Get the access token for the storage
    }

    setAccessToken(accessToken) {
        // Add the access token to the storage
    }

    removeAccessToken() {
        // Remove the access token from the storage
    }
}

export default AuthStorage;

```

Next, implement the methods `AuthStorage.getAccessToken`, `AuthStorage.setAccessToken` and `AuthStorage.removeAccessToken`. Use the `namespace` variable to give your keys a namespace like we did in the previous example.

Enhancing Apollo Client's requests

Now that we have implemented storage for storing the user's access token, it is time to start using it. Initialize the storage in the `App` component:

```

import { NativeRouter } from 'react-router-native';
import { ApolloProvider } from '@apollo/client';

import Main from './src/components/Main';
import createApolloClient from './src/utils/apolloClient';
import AuthStorage from './src/utils/authStorage';
const authStorage = new AuthStorage(); const apolloClient = createApolloClient(authStorage);
const App = () => [
    return [
        <NativeRouter>
            <ApolloProvider client={apolloClient}>
                <Main />
            </ApolloProvider>
        </NativeRouter>
    ]
];

export default App;

```

We also provided the storage instance for the `createApolloClient` function as an argument. This is because next, we will send the access token to Apollo Server in each request. The Apollo Server will expect that the access token is present in the `Authorization` header in the format `Bearer <ACCESS_TOKEN>`. We can enhance the Apollo Client's request by using the `setContext` function. Let's send the access token to the Apollo Server by modifying the `createApolloClient` function in the

apolloClient.js file:

```
import { ApolloClient, InMemoryCache, createHttpLink } from '@apollo/client';
import Constants from 'expo-constants';
import { setContext } from '@apollo/client/link/context';
// You might need to change this depending on how you have configured the Apollo
// Server's URI
const [ apolloUri ] = Constants.expoConfig.extra;

const httpLink = createHttpLink({
  uri: apolloUri,
});

const createApolloClient = (authStorage) => {
  const authLink = setContext(async (_r, { headers }) => {
    try {
      const accessToken = await authStorage.getAccessToken();
      return {
        headers: {
          ...headers,
          authorization: accessToken ? `Bearer ${accessToken}` : '',
        },
      };
    } catch (e) {
      console.log(e);
      return {
        headers,
      };
    }
  });
  return new ApolloClient({
    link: authLink.concat(httpLink),
    cache: new InMemoryCache(),
  });
}
export default createApolloClient;
```

Using React Context for dependency injection

The last piece of the sign-in puzzle is to integrate the storage to the *useSignIn* hook. To achieve this the hook must be able to access token storage instance we have initialized in the *App* component. React [Context](#) is just the tool we need for the job. Create a directory *contexts* in the *src* directory. In that directory create a file *AuthStorageContext.js* with the following content:

```
import { createContext } from 'react';

const AuthStorageContext = createContext();

export default AuthStorageContext;
```

Now we can use the *AuthStorageContext.Provider* to provide the storage instance to the descendants of the context. Let's add it to the *App* component:

```
import { NativeRouter } from 'react-router-native';
import { ApolloProvider } from '@apollo/client';

import Main from './src/components/Main';
import createApolloClient from './src/utils/apolloClient';
import AuthStorage from './src/utils/authStorage';
import AuthStorageContext from './src/context/AuthStorageContext';
const authStorage = new AuthStorage();
const apolloClient = createApolloClient(authStorage);

const App = () => [
```

```

    return [
      <NativeRouter>
        <ApolloProvider client={apolloClient}>
          <AuthStorageContext.Provider value={authStorage}>           <Main />
            </AuthStorageContext.Provider>      </ApolloProvider>
        </NativeRouter>
    ];
}

export default App;

```

Accessing the storage instance in the `useSignIn` hook is now possible using the React's `useContext` hook like this:

```

// ...
import { useContext } from 'react';
import AuthStorageContext from '../contexts/AuthStorageContext';
const useSignIn = () => {
  const authStorage = useContext(AuthStorageContext); // ...
};

```

Note that accessing a context's value using the `useContext` hook only works if the `useContext` hook is used in a component that is a *descendant* of the `Context.Provider` component.

Accessing the `AuthStorage` instance with `useContext(AuthStorageContext)` is quite verbose and reveals the details of the implementation. Let's improve this by implementing a `useAuthStorage` hook in a `useAuthStorage.js` file in the `hooks` directory:

```

import { useContext } from 'react';
import AuthStorageContext from '../contexts/AuthStorageContext';

const useAuthStorage = () => {
  return useContext(AuthStorageContext);
};

export default useAuthStorage;

```

The hook's implementation is quite simple but it improves the readability and maintainability of the hooks and components using it. We can use the hook to refactor the `useSignIn` hook like this:

```

// ...
import useAuthStorage from '../hooks/useAuthStorage';
const useSignIn = () => {
  const authStorage = useAuthStorage(); // ...
};

```

The ability to provide data to component's descendants opens tons of use cases for React Context, as we already saw in the [last chapter](#) of part 6.

To learn more about these use cases, read Kent C. Dodds' enlightening article [How to use React](#)

[Context effectively](#) to find out how to combine the `useReducer` hook with the context to implement state management. You might find a way to use this knowledge in the upcoming exercises.

Exercises 10.15. - 10.16

Exercise 10.15: storing the access token step2

Improve the `useSignIn` hook so that it stores the user's access token retrieved from the `authenticate` mutation. The return value of the hook should not change. The only change you should make to the `SignIn` component is that you should redirect the user to the reviewed repositories list view after a successful sign in. You can achieve this by using the `useNavigate` hook.

After the `authenticate` mutation has been executed and you have stored the user's access token to the storage, you should reset the Apollo Client's store. This will clear the Apollo Client's cache and re-execute all active queries. You can do this by using the Apollo Client's `resetStore` method. You can access the Apollo Client in the `useSignIn` hook using the `useApolloClient` hook. Note that the order of the execution is crucial and should be the following:

```
const [ data ] = await mutate/* options */;
await authStorage.setAccessToken/* access token from the data */;
apolloClient.resetStore();
```

Exercise 10.16: sign out

The final step in completing the sign in feature is to implement a sign out feature. The `me` query can be used to check the authenticated user's information. If the query's result is `null`, that means that the user is not authenticated. Open the Apollo Sandbox and run the following query:

```
{
  me {
    id
    username
  }
}
```

You will probably end up with the `null` result. This is because the Apollo Sandbox is not authenticated, meaning that it doesn't send a valid access token with the request. Revise the [authentication documentation](#) and retrieve an access token using the `authenticate` mutation. Use this access token in the `Authorization` header as instructed in the documentation. Now, run the `me` query again and you should be able to see the authenticated user's information.

Open the `AppBar` component in the `AppBar.jsx` file where you currently have the tabs "Repositories" and "Sign in". Change the tabs so that if the user is signed in the tab "Sign out" is displayed, otherwise show the "Sign in" tab. You can achieve this by using the `me` query with the `useQuery` hook.

Pressing the "Sign out" tab should remove the user's access token from the storage and reset the Apollo Client's store with the `resetStore` method. Calling the `resetStore` method should automatically re-execute all active queries which means that the `me` query should be re-executed. Note that the order of execution is crucial: access token must be removed from the storage *before* the Apollo Client's store is reset.

This was the last exercise in this section. It's time to push your code to GitHub and mark all of your finished exercises to the [exercise submission system](#). Note that exercises in this section should be submitted to the part 3 in the exercise submission system.

10d: Testing and extending our application

Now that we have established a good foundation for our project, it is time to start expanding it. In this section you can put to use all the React Native knowledge you have gained so far. Along with expanding our application we will cover some new areas, such as testing, and additional resources.

Testing React Native applications

To start testing code of any kind, the first thing we need is a testing framework, which we can use to run a set of test cases and inspect their results. For testing a JavaScript application, [Jest](#) is a popular candidate for such testing framework. For testing an Expo based React Native application with Jest, Expo provides a set of Jest configuration in a form of [jest-expo](#) preset. In order to use ESLint in the Jest's test files, we also need the [eslint-plugin-jest](#) plugin for ESLint. Let's get started by installing the packages:

```
npm install --save-dev jest jest-expo eslint-plugin-jest
```

To use the [jest-expo](#) preset in Jest, we need to add the following Jest configuration to the `package.json` file along with the `test` script:

```
{
  // ...
  "scripts": {
    // other scripts...
    "test": "jest",
    "jest": [
      "preset": "jest-expo",
      "transform": {
        "^.+\\.js$": "babel-jest",
        "^.+\\.jsx?$": "babel-jest"
      },
      "transformIgnorePatterns": [
        "node_modules/(?!((jest-)?react-native|@react-native(-community)?)|expo(nent)?|@expo(nent)?/.*/@expo-google-fonts/.*/|react-navigation|@react-navigation/.*/|@unimodules/.*/|unimodules|sentry-expo|native-base|react-native-svg|react-router-native)" ]
    ],
    // ...
  }
}
```

The `transform` option tells Jest to transform `.js` and `.jsx` files with the [Babel](#) compiler. The `transformIgnorePatterns` option is for ignoring certain directories in the `node_modules` directory while transforming files. This Jest configuration is almost identical to the one proposed in the [Expo's documentation](#).

To use the [eslint-plugin-jest](#) plugin in ESLint, we need to include it in the `plugins` and `extensions` array in the `.eslintrc` file:

```
{
  "plugins": ["react", "react-native"],
  "settings": {
    "react": {
      "version": "detect"
    }
  }
}
```

```
  },
  "extends": ["eslint:recommended", "plugin:react/recommended", "plugin:jest/recommended"], "parser": "@babel/eslint-parser",
  "env": {
    "react-native/react-native": true
  },
  "rules": [
    "react/prop-types": "off",
    "react/react-in-jsx-scope": "off"
  ]
}
```

To see that the setup is working, create a directory `__tests__` in the `src` directory and in the created directory create a file `example.test.js`. In that file, add this simple test:

```
describe('Example', () => {
  it('works', () => {
    expect(1).toBe(1);
  });
});
```

Now, let's run our example test by running `npm test`. The command's output should indicate that the test located in the `src/__tests__/example.test.js` file is passed.

Organizing tests

Organizing test files in a single `__tests__` directory is one approach in organizing the tests. When choosing this approach, it is recommended to put the test files in their corresponding subdirectories just like the code itself. This means that for example tests related to components are in the `components` directory, tests related to utilities are in the `utils` directory, and so on. This will result in the following structure:

```
src/
  __tests__/
    components/
      AppBar.js
      RepositoryList.js
      ...
    utils/
      authStorage.js
      ...
      ...
```

Another approach is to organize the tests near the implementation. This means that for example, the test file containing tests for the `AppBar` component is in the same directory as the component's code. This will result in the following structure:

```
src/
  components/
    AppBar/
```

```
AppBar.test.jsx  
index.jsx  
...  
...
```

In this example, the component's code is in the `index.jsx` file and the test in the `AppBar.test.jsx` file. Note that in order for Jest to find your test files you either have to put them into a `__tests__` directory, use the `.test` or `.spec` suffix, or [manually configure](#) the global patterns.

Testing components

Now that we have managed to set up Jest and run a very simple test, it is time to find out how to test components. As we know, testing components requires a way to serialize a component's render output and simulate firing different kind of events, such as pressing a button. For these purposes, there is the [Testing Library](#) family, which provides libraries for testing user interface components in different platforms. All of these libraries share similar API for testing user interface components in a user-centric way.

In [part 5](#) we got familiar with one of these libraries, the [React Testing Library](#). Unfortunately, this library is only suitable for testing React web applications. Luckily, there exists a React Native counterpart for this library, which is the [React Native Testing Library](#). This is the library we will be using while testing our React Native application's components. The good news is, that these libraries share a very similar API, so there aren't too many new concepts to learn. In addition to the React Native Testing Library, we need a set of React Native specific Jest matchers such as `toHaveTextContent` and `toHaveProp`. These matchers are provided by the [jest-native](#) library. Before getting into the details, let's install these packages:

```
npm install --save-dev --legacy-peer-deps react-test-renderer@18.2.0 @testing-library/react-native @testing-library/jest-native
```

NB: If you face peer dependency issues, make sure that the `react-test-renderer` version matches the project's React version in the `npm install` command above. You can check the React version by running `npm list react --depth=0`.

If the installation fails due to peer dependency issues, try again using the `--legacy-peer-deps` flag with the `npm install` command.

To be able to use these matchers we need to extend the Jest's `expect` object. This can be done by using a global setup file. Create a file `setupTests.js` in the root directory of your project, that is, the same directory where the `package.json` file is located. In that file add the following line:

```
import '@testing-library/jest-native/extend-expect';
```

Next, configure this file as a setup file in the Jest's configuration in the `package.json` file (note that the `<rootDir>` in the path is intentional and there is no need to replace it):

```
{  
  // ...  
  "jest": [  
    "preset": "jest-expo",  
    "transform": [  
      // ...  
    ]  
  ]  
}
```

```

    "^.+\\.jsx?$$": "babel-jest"
},
"transformIgnorePatterns": [
  "node_modules/(?!jest-)?react-native|react-clone-referenced-element|@react-native-community|expo(nent)?|@expo(nent)?:.*|react-navigation|@react-navigation/.*|@unimodules/.*|unimodules|sentry-expo|native-base|@sentry/.*|react-router-native)"
],
"setupFilesAfterEnv": ["<rootDir>/setupTests.js"]
// ...
}

```

The main concepts of the React Native Testing Library are the [queries](#) and [firing events](#). Queries are used to extract a set of nodes from the component that is rendered using the [render](#) function. Queries are useful in tests where we expect for example some text, such as the name of a repository, to be present in the rendered component. Here's an example how to use the [ByText](#) query to check if the component's *Text* element has the correct textual content:

```

import { Text, View } from 'react-native';
import { render, screen } from '@testing-library/react-native';

const Greeting = ({ name }) => {
  return (
    <View>
      <Text>Hello {name}!</Text>
    </View>
  );
}

describe('Greeting', () => {
  it('renders a greeting message based on the name prop', () => {
    render(<Greeting name="Kalle" />);

    screen.debug();

    expect(screen.getByText('Hello Kalle!')).toBeDefined();
  });
});

```

Tests use the object [screen](#) to do the queries to the rendered component.

We acquire the *Text* node containing certain text by using the [getByText](#) function. The Jest matcher [toBeDefined](#) is used to ensure that the query has found the element.

React Native Testing Library's documentation has some good hints on [how to query different kinds of elements](#). Another guide worth reading is Kent C. Dodds article [Making your UI tests resilient to change](#).

The object [screen](#) also has a helper method [debug](#) that prints the rendered React tree in a user-friendly format. Use it if you are unsure what the React tree rendered by the [render](#) function looks like.

For all available queries, check the React Native Testing Library's [documentation](#). The full list of available React Native specific matchers can be found in the [documentation](#) of the jest-native library. Jest's [documentation](#) contains every universal Jest matcher.

The second very important React Native Testing Library concept is firing events. We can fire an event in a provided node by using the `fireEvent` object's methods. This is useful for example typing text into a text field or pressing a button. Here is an example of how to test submitting a simple form:

```
import { useState } from 'react';
import { Text, TextInput, Pressable, View } from 'react-native';
import { render, fireEvent, screen } from '@testing-library/react-native';

const Form = ({ onSubmit }) => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const handleSubmit = () => {
    onSubmit({ username, password });
  };

  return (
    <View>
      <View>
        <TextInput
          value={username}
          onChangeText={(text) => setUsername(text)}
          placeholder="Username"
        />
      </View>
      <View>
        <TextInput
          value={password}
          onChangeText={(text) => setPassword(text)}
          placeholder="Password"
        />
      </View>
      <View>
        <Pressable onPress={handleSubmit}>
          <Text>Submit</Text>
        </Pressable>
      </View>
    </View>
  );
}

describe('Form', () => {
  it('calls function provided by onSubmit prop after pressing the submit button', () => {
    const onSubmit = jest.fn();
    render(<Form onSubmit={onSubmit} />);

    fireEvent.changeText(screen.getByPlaceholderText('Username'), 'kalle');
  });
});
```

```

fireEvent.changeText(screen.getByPlaceholderText('Password'), 'password');
fireEvent.press(screen.getByText('Submit'));

expect(onSubmit).toHaveBeenCalledWith(1);

// onSubmit.mock.calls[0][0] contains the first argument of the first call
expect(onSubmit.mock.calls[0][0]).toEqual({
  username: 'kalle',
  password: 'password',
});
});
});
}

```

In this test, we want to test that after filling the form's fields using the `fireEvent.changeText` method and pressing the submit button using the `fireEvent.press` method, the `onSubmit` callback function is called correctly. To inspect whether the `onSubmit` function is called and with which arguments, we can use a [mock function](#). Mock functions are functions with preprogrammed behavior such as a specific return value. In addition, we can create expectations for the mock functions such as "expect the mock function to have been called once". The full list of available expectations can be found in the Jest's [expect documentation](#).

Before heading further into the world of testing React Native applications, play around with these examples by adding a test file in the `__tests__` directory we created earlier.

Handling dependencies in tests

Components in the previous examples are quite easy to test because they are more or less *pure*. Pure components don't depend on *side effects* such as network requests or using some native API such as the `AsyncStorage`. The `Form` component is much less pure than the `Greeting` component because its state changes can be counted as a side effect. Nevertheless, testing it isn't too difficult.

Next, let's have a look at a strategy for testing components with side effects. Let's pick the `RepositoryList` component from our application as an example. At the moment the component has one side effect, which is a GraphQL query for fetching the reviewed repositories. The current implementation of the `RepositoryList` component looks something like this:

```

const RepositoryList = () => {
  const { repositories } = useRepositories();

  const repositoryNodes = repositories
    ? repositories.edges.map((edge) => edge.node)
    : [];

  return [
    <FlatList
      data={repositoryNodes}
      // ...
    />
  ];
}

```

```
export default RepositoryList;
```

The only side effect is the use of the `useRepositories` hook, which sends a GraphQL query. There are a few ways to test this component. One way is to mock the Apollo Client's responses as instructed in the Apollo Client's [documentation](#). A more simple way is to assume that the `useRepositories` hook works as intended (preferably through testing it) and extract the components "pure" code into another component, such as the `RepositoryListContainer` component:

```
export const RepositoryListContainer = ({ repositories }) => {
  const repositoryNodes = repositories
    ? repositories.edges.map(({ edge }) => edge.node)
    : [];

  return (
    <FlatList
      data={repositoryNodes}
      // ...
    />
  );
};

const RepositoryList = () => {
  const { repositories } = useRepositories();

  return <RepositoryListContainer repositories={repositories} />;
};

export default RepositoryList;
```

Now, the `RepositoryList` component contains only the side effects and its implementation is quite simple. We can test the `RepositoryListContainer` component by providing it with paginated repository data through the `repositories` prop and checking that the rendered content has the correct information.

Exercises 10.17. - 10.18

Exercise 10.17: testing the reviewed repositories list

Implement a test that ensures that the `RepositoryListContainer` component renders repository's name, description, language, forks count, stargazers count, rating average, and review count correctly. One approach in implementing this test is to add a `testID` prop for the element wrapping a single repository's information:

```
const RepositoryItem = /* ... */ => [
  // ...

  return (
    <View testID="repositoryItem" /* ... */>
      /* ... */
    </View>
  )
];
```

Once the `testID` prop is added, you can use the `getAllByTestId` query to get those elements:

```
const repositoryItems = screen.getAllByTestId('repositoryItem');
const [firstRepositoryItem, secondRepositoryItem] = repositoryItems;

// expect something from the first and the second repository item
```

Having those elements you can use the `toHaveTextContent` matcher to check whether an element has certain textual content. You might also find the [Querying Within Elements](#) guide useful. If you are unsure what is being rendered, use the `debug` function to see the serialized rendering result.

Use this as a base for your test:

```
describe('RepositoryList', () => {
  describe('RepositoryListContainer', () => {
    it('renders repository information correctly', () => {
      const repositories = [
        {
          totalCount: 8,
          pageInfo: {
            hasNextPage: true,
            endCursor: 'WyJhc3luYy1saWJyYXJ5LnJ1YWN0LWFzeW5jIiwxNTg4NjU2NzUwMDc2XQ==',
            startCursor: 'WyJqYXJ1ZHBhbG1lci5mb3JtaWsiLDE1ODg2NjAzNTAwNzZd',
          },
          edges: [
            {
              node: {
                id: 'jaredpalmer.formik',
                fullName: 'jaredpalmer/formik',
                description: 'Build forms in React, without the tears',
                language: 'TypeScript',
                forksCount: 1619,
                stargazersCount: 21856,
                ratingAverage: 88,
                reviewCount: 3,
                ownerAvatarUrl:
                  'https://avatars2.githubusercontent.com/u/4060187?v=4',
              },
              cursor: 'WyJqYXJ1ZHBhbG1lci5mb3JtaWsiLDE1ODg2NjAzNTAwNzZd',
            },
            {
              node: {
                id: 'async-library.react-async',
                fullName: 'async-library/react-async',
                description: 'Flexible promise-based React data loader',
                language: 'JavaScript',
                forksCount: 69,
                stargazersCount: 1760,
                ratingAverage: 72,
              }
            }
          ]
        }
      ]
    })
  })
})
```

```

        reviewCount: 3,
        ownerAvatarUrl:
          'https://avatars1.githubusercontent.com/u/54310907?v=4',
      },
      cursor:
        'WyJhc3luYy1saWJyYXJ5LnJlYWN0LWFzeW5jIiwxNTg4NjU2NzUwMDc2XQ==',
    ],
  },
};

// Add your test code here
});;
});;
})();

```

You can put the test file where you please. However, it is recommended to follow one of the ways of organizing test files introduced earlier. Use the *repositories* variable as the repository data for the test. There should be no need to alter the variable's value. Note that the repository data contains two repositories, which means that you need to check that both repositories' information is present.

Exercise 10.18: testing the sign in form

Implement a test that ensures that filling the sign in form's username and password fields and pressing the submit button *will call* the *onSubmit* handler with *correct arguments*. The *first argument* of the handler should be an object representing the form's values. You can ignore the other arguments of the function. Remember that the *fireEvent* methods can be used for triggering events and a *mock function* for checking whether the *onSubmit* handler is called or not.

You don't have to test any Apollo Client or AsyncStorage related code which is in the *useSignIn* hook. As in the previous exercise, extract the pure code into its own component and test it in the test.

Note that Formik's form submissions are *asynchronous* so expecting the *onSubmit* function to be called immediately after pressing the submit button won't work. You can get around this issue by making the test function an *async* function using the *async* keyword and using the React Native Testing Library's *waitFor* helper function. The *waitFor* function can be used to wait for expectations to pass. If the expectations don't pass within a certain period, the function will throw an error. Here is a rough example of how to use it:

```

import { render, screen, fireEvent, waitFor } from '@testing-library/react-native';
// ...

describe('SignIn', () => {
  describe('SignInContainer', () => {
    it('calls onSubmit function with correct arguments when a valid form is submitted', async () => {
      // render the SignInContainer component, fill the text inputs and press the submit button

      await waitFor(() => {
        // expect the onSubmit function to have been called once and with a correct first argument
      })
    })
  })
})

```

```
    }) ;
  }) ;
}) ;
}) ;
```

Extending our application

It is time to put everything we have learned so far to good use and start extending our application. Our application still lacks a few important features such as reviewing a repository and registering a user. The upcoming exercises will focus on these essential features.

Exercises 10.19. - 10.26

Exercise 10.19: the single repository view

Implement a view for a single repository, which contains the same repository information as in the reviewed repositories list but also a button for opening the repository in GitHub. It would be a good idea to reuse the *RepositoryItem* component used in the *RepositoryList* component and display the GitHub repository button for example based on a boolean prop.

The repository's URL is in the *url* field of the *Repository* type in the GraphQL schema. You can fetch a single repository from the Apollo server with the *repository* query. The query has a single argument, which is the id of the repository. Here's a simple example of the *repository* query:

```
{
  repository(id: "jaredpalmer.formik") {
    id
    fullName
    url
  }
}
```

As always, test your queries in the Apollo Sandbox first before using them in your application. If you are unsure about the GraphQL schema or what are the available queries, take a look at the documentation next to the operations editor. If you have trouble using the id as a variable in the query, take a moment to study the Apollo Client's [documentation](#) on queries.

To learn how to open a URL in a browser, read the Expo's [Linking API documentation](#). You will need this feature while implementing the button for opening the repository in GitHub. Hint: [Linking.openURL](#) method will come in handy.

The view should have its own route. It would be a good idea to define the repository's id in the route's path as a path parameter, which you can access by using the [useParams](#) hook. The user should be able to access the view by pressing a repository in the reviewed repositories list. You can achieve this by for example wrapping the *RepositoryItem* with a [Pressable](#) component in the *RepositoryList* component and using *navigate* function to change the route in an *onPress* event handler. You can access the *navigate* function with the [useNavigate](#) hook.

The final version of the single repository view should look something like this:

[Repositories](#)[Sign in](#)**jaredpalmer/formik**

Build forms in React, without the tears 😭

[TypeScript](#)**21.9k**

Stars

1.6k

Forks

3

Reviews

88

Rating

[Open in GitHub](#)

Note if the peer dependency issues prevent installing the library, try the `--legacy-peer-deps` option:

```
npm install expo-linking --legacy-peer-deps
```

Exercise 10.20: repository's review list

Now that we have a view for a single repository, let's display repository's reviews there. Repository's reviews are in the `reviews` field of the `Repository` type in the GraphQL schema. `reviews` is a similar paginated list as in the `repositories` query. Here's an example of getting reviews of a repository:

```
{  
  repository(id: "jaredpalmer.formik") {  
    id  
    fullName  
    reviews {
```

```
edges [
    node [
        id
        text
        rating
        createdAt
        user [
            id
            username
        ]
    ]
]
}
}
```

Review's *text* field contains the textual review, *rating* field a numeric rating between 0 and 100, and *createdAt* the date when the review was created. Review's *user* field contains the reviewer's information, which is of type *User*.

We want to display reviews as a scrollable list, which makes `FlatList` a suitable component for the job. To display the previous exercise's repository's information at the top of the list, you can use the `FlatList` component's `ListHeaderComponent` prop. You can use the `ItemSeparatorComponent` to add some space between the items like in the `RepositoryList` component. Here's an example of the structure:

The final version of the repository's reviews list should look something like this:

[Repositories](#)[Sign in](#)**jaredpalmer/formik**

Build forms in React, without the tears 😭

TypeScript

21.9k

Stars

1.6k

Forks

3

Reviews

88

Rating

[Open in GitHub](#)**95****kalle**

05.05.2020

Lorem ipsum dolor sit amet, per brute
apeirian ei. Malis facilisis vel ex, ex vivendo
signiferumque nam, nam ad natum electram
constituto. Causae latine at sea, ex nec
ullum ceteros, est ut dicant splendide. Omnis
electram ullamcorper est ut.

70**matti**

05.05.2020

lorem ipsum dolor sit amet, per brute

Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.



elina

05.05.2020

Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo

The date under the reviewer's username is the creation date of the review, which is in the `createdAt` field of the `Review` type. The date format should be user-friendly such as `date.month.year`. You can for example install the [date-fns](#) library and use the `format` function for formatting the creation date.

The round shape of the rating's container can be achieved with the `borderRadius` style property. You can make it round by fixing the container's `width` and `height` style property and setting the border-radius as `width / 2`.

Exercise 10.21: the review form

Implement a form for creating a review using Formik. The form should have four fields: repository owner's GitHub username (for example "jaredpalmer"), repository's name (for example "formik"), a numeric rating, and a textual review. Validate the fields using Yup schema so that it contains the following validations:

- Repository owner's username is a required string
- Repository's name is a required string
- Rating is a required number between 0 and 100
- Review is a optional string

Explore Yup's [documentation](#) to find suitable validators. Use sensible error messages with the validators. The validation message can be defined as the validator method's `message` argument. You can make the review field expand to multiple lines by using `TextInput` component's `multiline` prop.

You can create a review using the `createReview` mutation. Check this mutation's arguments in the Apollo Sandbox. You can use the [useMutation](#) hook to send a mutation to the Apollo Server.

After a successful `createReview` mutation, redirect the user to the repository's view you implemented in the previous exercise. This can be done with the `navigate` function after you have obtained it using the [useNavigate](#) hook. The created review has a `repositoryId` field which you can use to construct the route's path.

To prevent getting cached data with the `repository` query in the single repository view, use the `cache-`

and-network [fetch policy](#) in the query. It can be used with the `useQuery` hook like this:

```
useQuery(GET_REPOSITORY, [
  fetchPolicy: 'cache-and-network',
  // Other options
]);
```

Note that only *an existing public GitHub repository* can be reviewed and a user can review the same repository *only once*. You don't have to handle these error cases, but the error payload includes specific codes and messages for these errors. You can try out your implementation by reviewing one of your own public repositories or any other public repository.

The review form should be accessible through the app bar. Create a tab to the app bar with a label "Create a review". This tab should only be visible to users who have signed in. You will also need to define a route for the review form.

The final version of the review form should look something like this:

[Repositories](#)[Create a review](#)[Sign out](#)

Repository owner name

Repository owner name is required

Repository name

Repository name is required

Rating between 0 and 100

Rating is required

Review

[Create a review](#)

This screenshot has been taken after invalid form submission to present what the form should look like in an invalid state.

Exercise 10.22: the sign up form

Implement a sign up form for registering a user using Formik. The form should have three fields: username, password, and password confirmation. Validate the form using Yup schema so that it contains the following validations:

- Username is a required string with a length between 5 and 30
- Password is a required string with a length between 5 and 50
- Password confirmation matches the password

The password confirmation field's validation can be a bit tricky, but it can be done for example by using the [oneOf](#) and [ref](#) methods like suggested in [this issue](#).

You can create a new user by using the *createUser* mutation. Find out how this mutation works by exploring the documentation in the Apollo Sandbox. After a successful *createUser* mutation, sign the created user in by using the *useSignIn* hook as we did in the sign in the form. After the user has been signed in, redirect the user to the reviewed repositories list view.

The user should be able to access the sign-up form through the app bar by pressing a "Sign up" tab. This tab should only be visible to users that aren't signed in.

The final version of the sign up form should look something like this:

[Repositories](#)[Sign in](#)[Sign up](#)

Username

Username is required

Password

Password is required

Password confirmation

Password confirmation is required

[Sign up](#)

This screenshot has been taken after invalid form submission to present what the form should look like in an invalid state.

Exercise 10.23: sorting the reviewed repositories list

At the moment repositories in the reviewed repositories list are ordered by the date of repository's first review. Implement a feature that allows users to select the principle, which is used to order the repositories. The available ordering principles should be:

- Latest repositories. The repository with the latest first review is on the top of the list. This is the current behavior and should be the default principle.

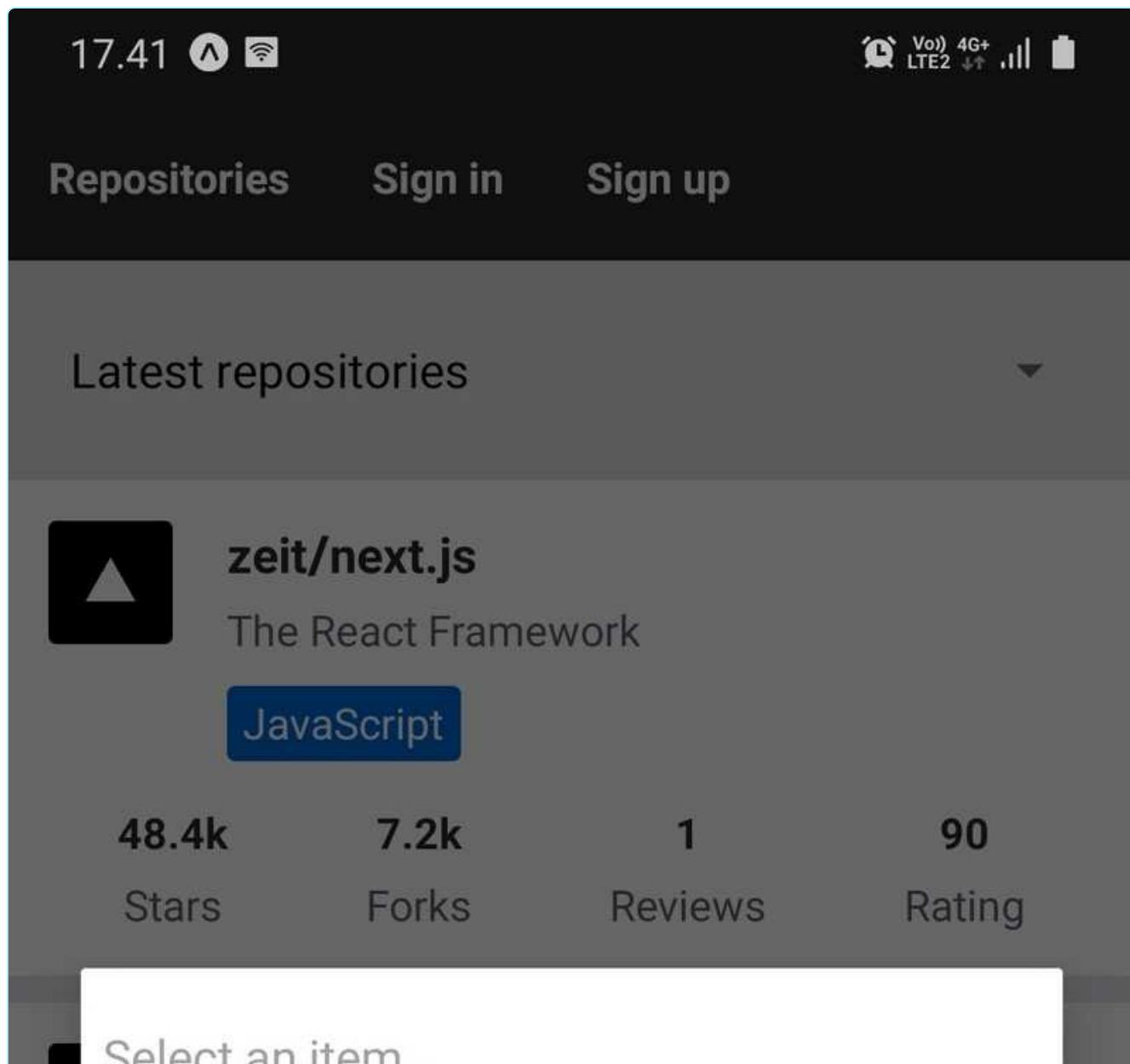
- Highest rated repositories. The repository with the *highest* average rating is on the top of the list.
- Lowest rated repositories. The repository with the *lowest* average rating is on the top of the list.

The *repositories* query used to fetch the reviewed repositories has an argument called *orderBy*, which you can use to define the ordering principle. The argument has two allowed values: `CREATED_AT` (order by the date of repository's first review) and `RATING_AVERAGE`, (order by the repository's average rating). The query also has an argument called *orderDirection* which can be used to change the order direction. The argument has two allowed values: `ASC` (ascending, smallest value first) and `DESC` (descending, biggest value first).

The selected ordering principle state can be maintained for example using the React's `useState` hook. The variables used in the *repositories* query can be given to the `useRepositories` hook as an argument.

You can use for example [@react-native-picker/picker](#) library, or [React Native Paper](#) library's `Menu` component to implement the ordering principle's selection. You can use the `FlatList` component's `ListHeaderComponent` prop to provide the list with a header containing the selection component.

The final version of the feature, depending on the selection component in use, should look something like this:



Latest repositories

Highest rated repositories

Lowest rated repositories



jaredpalmer/formik

Build forms in React, without the tears 😭

TypeScript

22.1k

1.6k

3

88

Stars

Forks

Reviews

Rating



async-library/react-async

Flexible promise-based React data loader

Exercise 10.24: filtering the reviewed repositories list

The Apollo Server allows filtering repositories using the repository's name or the owner's username. This can be done using the `searchKeyword` argument in the `repositories` query. Here's an example of how to use the argument in a query:

```
{  
  repositories(searchKeyword: "ze") {  
    edges {  
      node {  
        id  
        fullName  
      }  
    }  
  }  
}
```

Implement a feature for filtering the reviewed repositories list based on a keyword. Users should be able to type in a keyword into a text input and the list should be filtered as the user types. You can use a simple `TextInput` component or something a bit fancier such as React Native Paper's `Searchbar` component as the text input. Put the text input component in the `FlatList` component's header.

To avoid a multitude of unnecessary requests while the user types the keyword fast, only pick the latest input after a short delay. This technique is often referred to as `debouncing`. `use-debounce` library is a handy hook for debouncing a state variable. Use it with a sensible delay time, such as 500 milliseconds. Store the text input's value by using the `useState` hook and then pass the debounced value to the query as the value of the `searchKeyword` argument.

You probably face an issue that the text input component loses focus after each keystroke. This is because the content provided by the `ListHeaderComponent` prop is constantly unmounted. This can be fixed by turning the component rendering the `FlatList` component into a class component and defining the header's render function as a class property like this:

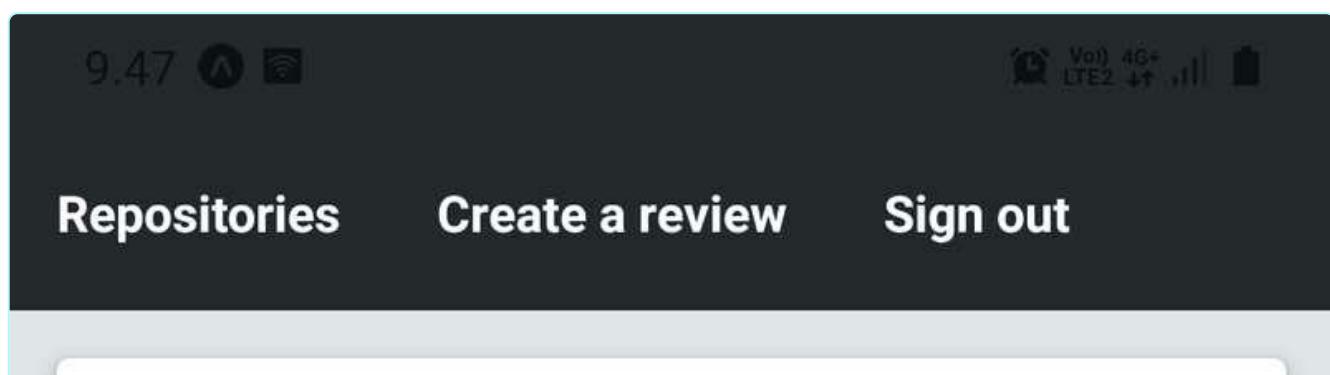
```
export class RepositoryListContainer extends React.Component {
  renderHeader = () => [
    // this.props contains the component's props
    const props = this.props;

    // ...

    return [
      <RepositoryListHeader
        // ...
      />
    ];
  };

  render() {
    return [
      <FlatList
        // ...
        ListHeaderComponent={this.renderHeader}
      />
    ];
  }
}
```

The final version of the filtering feature should look something like this:

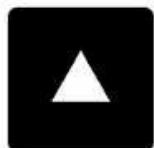




ze



Latest repositories

**zeit/next.js**

The React Framework

JavaScript

48.4k

Stars

7.2k

Forks

0

Reviews

0

Rating

**zeit/swr**

React Hooks library for remote data fetching

TypeScript

9.4k

Stars

287

Forks

0

Reviews

0

Rating

Exercise 10.25: the user's reviews view

Implement a feature which allows user to see their reviews. Once signed in, the user should be able to access this view by pressing a "My reviews" tab in the app bar. Here is what the review list view should roughly look like:

[Repositories](#)[Create a review](#)[My reviews](#)[Sign](#)

99

rails/rails

17.08.2020

Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.

95

jaredpalmer/formik

17.08.2020

Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.

78

django/django

17.08.2020

Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram

Remember that you can fetch the authenticated user from the Apollo Server with the `me` query. This query returns a `User` type, which has a field `reviews`. If you have already implemented a reusable `me` query in your code, you can customize this query to fetch the `reviews` field conditionally. This can be done using GraphQL's `include` directive.

Let's say that the current query is implemented roughly in the following manner:

```
const GET_CURRENT_USER = gql`  
query {  
  me {  
    # user fields...  
  }  
}
```

You can provide the query with an `includeReviews` argument and use that with the `include` directive:

```
const GET_CURRENT_USER = gql`  
query getCurrentUser($includeReviews: Boolean = false) {  
  me {  
    # user fields...  
    reviews @include(if: $includeReviews) {  
      edges {  
        node {  
          # review fields...  
        }  
      }  
    }  
  }  
}
```

The `includeReviews` argument has a default value of `false`, because we don't want to cause additional server overhead unless we explicitly want to fetch authenticated user's reviews. The principle of the `include` directive is quite simple: if the value of the `if` argument is `true`, include the field, otherwise omit it.

Exercise 10.26: review actions

Now that user can see their reviews, let's add some actions to the reviews. Under each review on the review list, there should be two buttons. One button is for viewing the review's repository. Pressing this button should take the user to the single repository review implemented in the previous exercise. The other button is for deleting the review. Pressing this button should delete the review. Here is what the actions should roughly look like:

12.00

Voice 4G+ LTE2 86%

Repositories

Create a review

My reviews

Sign



99

rails/rails

17.08.2020

Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.

[View repository](#)[Delete review](#)

95

jaredpalmer/formik

17.08.2020

Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.

[View repository](#)[Delete review](#)

78

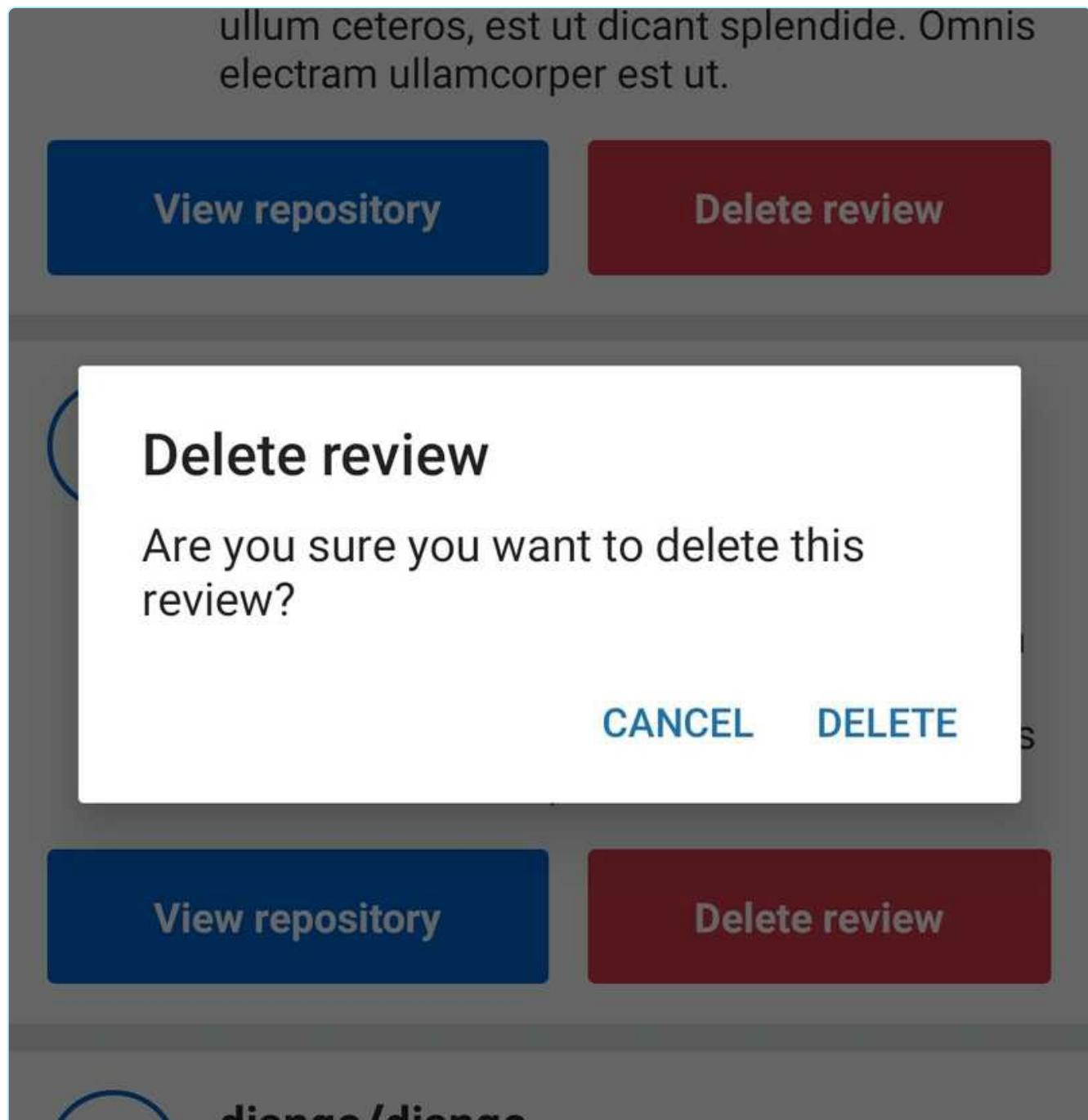
django/django

17.08.2020

**“Lorem ipsum dolor sit amet, per brute
apeirian ei. Malis facilisis vel ex, ex vivendo**

Pressing the delete button should be followed by a confirmation alert. If the user confirms the deletion, the review is deleted. Otherwise, the deletion is discarded. You can implement the confirmation using the [Alert](#) module. Note that calling the `Alert.alert` method won't open any window in Expo web preview. Use either Expo mobile app or an emulator to see the what the alert window looks like.

Here is the confirmation alert that should pop out once the user presses the delete button:



You can delete a review using the `deleteReview` mutation. This mutation has a single argument, which is the id of the review to be deleted. After the mutation has been performed, the easiest way to update the review list's query is to call the `refetch` function.

Cursor-based pagination

When an API returns an ordered list of items from some collection, it usually returns a subset of the whole set of items to reduce the required bandwidth and to decrease the memory usage of the client applications. The desired subset of items can be parameterized so that the client can request for example the first twenty items on the list after some index. This technique is commonly referred to as *pagination*. When items can be requested after a certain item defined by a *cursor*, we are talking about *cursor-based pagination*.

So cursor is just a serialized presentation of an item in an ordered list. Let's have a look at the paginated repositories returned by the *repositories* query using the following query:

```
{
  repositories(first: 2) {
    totalCount
    edges {
      node {
        id
        fullName
        createdAt
      }
      cursor
    }
    pageInfo {
      endCursor
      startCursor
      hasNextPage
    }
  }
}
```

The *first* argument tells the API to return only the first two repositories. Here's an example of a result of the query:

```
{
  "data": [
    "repositories": {
      "totalCount": 10,
      "edges": [
        {
          "node": {
            "id": "zeit.next.js",
            "fullName": "zeit/next.js",
            "createdAt": "2020-05-15T11:59:57.557Z"
          },
          "cursor": "WyJ6ZWl0Lm5leHQuanMiLDE1ODk1NDM5OTc1NTdd"
        },
        {
          "node": {
            "id": "zeit.swr",
            "fullName": "zeit/swr",
            "createdAt": "2020-05-15T11:59:57.557Z"
          }
        }
      ]
    }
  ]
}
```

```

    "createdAt": "2020-05-15T11:58:53.867Z"
  },
  "cursor": "WyJ6ZWl0LnN3ciIsMTU4OTU0MzkzMzg2N10="
],
{
  "pageInfo": {
    "endCursor": "WyJ6ZWl0LnN3ciIsMTU4OTU0MzkzMzg2N10=",
    "startCursor": "WyJ6ZWl0Lm5leHQuanMiLDE1ODk1NDM5OTc1NTdd",
    "hasNextPage": true
  }
}
]
}

```

The format of the result object and the arguments are based on the [Relay's GraphQL Cursor Connections Specification](#), which has become a quite common pagination specification and has been widely adopted for example in the [GitHub's GraphQL API](#). In the result object, we have the `edges` array containing items with `node` and `cursor` attributes. As we know, the `node` contains the repository itself. The `cursor` on the other hand is a Base64 encoded representation of the node. In this case, it contains the repository's id and date of repository's creation as a timestamp. This is the information we need to point to the item when they are ordered by the creation time of the repository. The `pageInfo` contains information such as the cursor of the first and the last item in the array.

Let's say that we want to get the next set of items *after* the last item of the current set, which is the "zeit/swr" repository. We can set the *after* argument of the query as the value of the `endCursor` like this:

```

{
  repositories(first: 2, after: "WyJ6ZWl0LnN3ciIsMTU4OTU0MzkzMzg2N10=") {
    totalCount
    edges {
      node {
        id
        fullName
        createdAt
      }
      cursor
    }
    pageInfo {
      endCursor
      startCursor
      hasNextPage
    }
  }
}

```

Now that we have the next two items and we can keep on doing this until the `hasNextPage` has the value `false`, meaning that we have reached the end of the list. To dig deeper into cursor-based pagination, read Shopify's article [Pagination with Relative Cursors](#). It provides great details on the implementation itself and the benefits over the traditional index-based pagination.

Infinite scrolling

Vertically scrollable lists in mobile and desktop applications are commonly implemented using a technique called *infinite scrolling*. The principle of infinite scrolling is quite simple:

- Fetch the initial set of items
- When the user reaches the last item, fetch the next set of items after the last item

The second step is repeated until the user gets tired of scrolling or some scrolling limit is exceeded. The name "infinite scrolling" refers to the way the list seems to be infinite - the user can just keep on scrolling and new items keep on appearing on the list.

Let's have a look at how this works in practice using the Apollo Client's `useQuery` hook. Apollo Client has a great [documentation](#) on implementing the cursor-based pagination. Let's implement infinite scrolling for the reviewed repositories list as an example.

First, we need to know when the user has reached the end of the list. Luckily, the `FlatList` component has a prop `onEndReached`, which will call the provided function once the user has scrolled to the last item on the list. You can change how early the `onEndReach` callback is called using the `onEndReachedThreshold` prop. Alter the `RepositoryList` component's `FlatList` component so that it calls a function once the end of the list is reached:

```
export const RepositoryListContainer = ({  
  repositories,  
  onEndReach,  
  /* ... */  
}) => {  
  const repositoryNodes = repositories  
    ? repositories.edges.map(({edge}) => edge.node)  
    : [];  
  
  return (  
    <FlatList  
      data={repositoryNodes}  
      // ...  
      onEndReached={onEndReach}  
      onEndReachedThreshold={0.5}  
    />  
  );  
};  
  
const RepositoryList = () => {  
  // ...  
  
  const { repositories } = useRepositories(/* ... */);  
  
  const onEndReach = () => {  
    console.log('You have reached the end of the list');  
  };  
  
  return (  
    <RepositoryListContainer  
      repositories={repositories}  
    />  
  );  
};
```

```

        onEndReach={onEndReach}
        // ...
      />
    );
}

export default RepositoryList;

```

Try scrolling to the end of the reviewed repositories list and you should see the message in the logs.

Next, we need to fetch more repositories once the end of the list is reached. This can be achieved using the `fetchMore` function provided by the `useQuery` hook. To describe to Apollo Client how to merge the existing repositories in the cache with the next set of repositories, we can use a `field policy`. In general, field policies can be used to customize the cache behavior during read and write operations with `read` and `merge` functions.

Let's add a field policy for the `repositories` query in the `apolloClient.js` file:

```

import { ApolloClient, InMemoryCache, createHttpLink } from '@apollo/client';
import { setContext } from '@apollo/client/link/context';
import Constants from 'expo-constants';
import { relayStylePagination } from '@apollo/client/utilities';
const { apolloUri } = Constants.manifest.extra;

const httpLink = createHttpLink({
  uri: apolloUri,
});

const cache = new InMemoryCache({ typePolicies: { Query: { fields: {
  repositories: relayStylePagination(),
} }, } });
const createApolloClient = (authStorage) => {
  const authLink = setContext(async ({ headers }) => {
    try {
      const accessToken = await authStorage.getAccessToken();

      return {
        headers: {
          ...headers,
          authorization: accessToken ? `Bearer ${accessToken}` : '',
        },
      };
    } catch (e) {
      console.log(e);
    }
  });

  return new ApolloClient({
    link: authLink.concat(httpLink),
  });
}

```

```

    cache,  () =>
};

export default createApolloClient;

```

As mentioned earlier, the format of the pagination's result object and the arguments are based on the Relay's pagination specification. Luckily, Apollo Client provides a predefined field policy, *relayStylePagination*, which can be used in this case.

Next, let's alter the *useRepositories* hook so that it returns a decorated *fetchMore* function, which calls the actual *fetchMore* function with appropriate arguments so that we can fetch the next set of repositories:

```

const useRepositories = (variables) => {
  const { data, loading, fetchMore, ...result } = useQuery(GET_REPOSITORIES, [
    variables,
    // ...
  ]);

  const handleFetchMore = () => {
    const canFetchMore = !loading && data?.repositories.pageInfo.hasNextPage;

    if (!canFetchMore) {
      return;
    }

    fetchMore({
      variables: {
        after: data.repositories.pageInfo.endCursor,
        ...variables,
      },
    });
  };
}

return {
  repositories: data?.repositories,
  fetchMore: handleFetchMore,
  loading,
  ...result,
};
}

```

Make sure you have the *pageInfo* and the *cursor* fields in your *repositories* query as described in the pagination examples. You will also need to include the *after* and *first* arguments for the query.

The *handleFetchMore* function will call the Apollo Client's *fetchMore* function if there are more items to fetch, which is determined by the *hasNextPage* property. We also want to prevent fetching more items if fetching is already in process. In this case, *loading* will be *true*. In the *fetchMore* function we are providing the query with an *after* variable, which receives the latest *endCursor* value.

The final step is to call the *fetchMore* function in the *onEndReach* handler:

```

const RepositoryList = () => {
  // ...

  const { repositories, fetchMore } = useRepositories({
    first: 8,
    // ...
  });

  const onEndReach = () => {
    fetchMore();
  };

  return (
    <RepositoryListContainer
      repositories={repositories}
      onEndReach={onEndReach}
      // ...
    />
  );
}

export default RepositoryList;

```

Use a relatively small *first* argument value such as 3 while trying out the infinite scrolling. This way you don't need to review too many repositories. You might face an issue that the *onEndReach* handler is called immediately after the view is loaded. This is most likely because the list contains so few repositories that the end of the list is reached immediately. You can get around this issue by increasing the value of *first* argument. Once you are confident that the infinite scrolling is working, feel free to use a larger value for the *first* argument.

Exercise 10.27

Exercise 10.27: infinite scrolling for the repository's reviews list

Implement infinite scrolling for the repository's reviews list. The *Repository* type's *reviews* field has the *first* and *after* arguments similar to the *repositories* query. *ReviewConnection* type also has the *pageInfo* field just like the *RepositoryConnection* type.

Here's an example query:

```
{
  repository(id: "jaredpalmer.formik") {
    id
    fullName
    reviews(first: 2, after: "WyIxYjEwZTRkOC01N2VlLTRkMDAtODg4Ni1lNGEwNDlkN2ZmOG
YuamFyZWRwYWxtZXIuZm9ybWlrIiwxNTg4NjU2NzUwMDgwXQ==") {
      totalCount
      edges {
        node {
          id
          text
        }
      }
    }
  }
}
```

```
        rating
        createdAt
        repositoryId
      user {
        id
        username
      }
    }
  cursor
}
pageInfo {
  endCursor
  startCursor
  hasNextPage
}
}
}
```

The cache's field policy can be similar as with the *repositories* query:

```
const cache = new InMemoryCache({
  typePolicies: [
    Query: [
      fields: [
        repositories: relayStylePagination(),
      ],
    ],
    Repository: [
      fields: [
        reviews: relayStylePagination(),
      ],
    ],
  ],
})
```

As with the reviewed repositories list, use a relatively small *first* argument value while you are trying out the infinite scrolling. You might need to create a few new users and use them to create a few new reviews to make the reviews list long enough to scroll. Set the value of the *first* argument high enough so that the *onEndReach* handler isn't called immediately after the view is loaded, but low enough so that you can see that more reviews are fetched once you reach the end of the list. Once everything is working as intended you can use a larger value for the *first* argument.

This was the last exercise in this section. It's time to push your code to GitHub and mark all of your finished exercises to the [exercise submission system](#). Note that exercises in this section should be submitted to the part 4 in the exercise submission system.

Additional resources

As we are getting closer to the end of this part, let's take a moment to look at some additional React Native related resources. [Awesome React Native](#) is an extremely encompassing curated list of React Native resources such as libraries, tutorials, and articles. Because the list is exhaustively long, let's have a closer look at few of its highlights

React Native Paper

Paper is a collection of customizable and production-ready components for React Native, following Google's Material Design guidelines.

[React Native Paper](#) is for React Native what [Material-UI](#) is for React web applications. It offers a wide range of high-quality UI components, support for [custom themes](#) and a fairly simple [setup](#) for Expo based React Native applications.

Styled-components

Utilising tagged template literals (a recent addition to JavaScript) and the power of CSS, styled-components allows you to write actual CSS code to style your components. It also removes the mapping between components and styles – using components as a low-level styling construct could not be easier!

[Styled-components](#) is a library for styling React components using [CSS-in-JS](#) technique. In React Native we are already used to defining component's styles as a JavaScript object, so CSS-in-JS is not so uncharted territory. However, the approach of styled-components is quite different from using the `StyleSheet.create` method and the `style` prop.

In styled-components components' styles are defined with the component using a feature called [tagged template literal](#) or a plain JavaScript object. Styled-components makes it possible to define new style properties for component based on its props *at runtime*. This brings many possibilities, such as seamlessly switching between a light and a dark theme. It also has a full [theming support](#). Here is an example of creating a `Text` component with style variations based on props:

```
import styled from 'styled-components/native';
import { css } from 'styled-components';

const FancyText = styled.Text`  

  color: grey;  

  font-size: 14px;  

  ${({ isBlue }) =>  

    isBlue &&  

    css`  

      color: blue;  

`}  

  ${({ isBig }) =>  

    isBig &&  

    css`  

      font-size: 24px;  

      font-weight: 700;  

`}  

`
```

```
const Main = () => {
  return (
    <>
      <FancyText>Simple text</FancyText>
      <FancyText isBlue>Blue text</FancyText>
      <FancyText isBig>Big text</FancyText>
      <FancyText isBig isBlue>
        Big blue text
      </FancyText>
    </>
  );
};
```

Because styled-components processes the style definitions, it is possible to use CSS-like snake case syntax with the property names and units in property values. However, units don't have any effect because property values are internally unitless. For more information on styled-components, head out to the [documentation](#).

React-spring

react-spring is a spring-physics based animation library that should cover most of your UI related animation needs. It gives you tools flexible enough to confidently cast your ideas into moving interfaces.

[React-spring](#) is a library that provides a clean [API](#) for animating React Native components.

React Navigation

Routing and navigation for your React Native apps

[React Navigation](#) is a routing library for React Native. It shares some similarities with the React Router library we have been using during this and earlier parts. However, unlike React Router, React Navigation offers more native features such as native gestures and animations to transition between views.

Closing words

That's it, our application is ready. Good job! We have learned many new concepts during our journey such as setting up our React Native application using Expo, using React Native's core components and adding style to them, communicating with the server, and testing React Native applications. The final piece of the puzzle would be to deploy the application to the Apple App Store and Google Play Store.

Deploying the application is entirely *optional* and it isn't quite trivial, because you also need to fork and deploy the [rate-repository-api](#). For the React Native application itself, you first need to create either iOS or Android builds by following Expo's [documentation](#). Then you can upload these builds to

either Apple App Store or Google Play Store. Expo has [documentation](#) for this as well.

11: CI/CD

11a: Introduction to CI/CD

During this part, you will build a robust *deployment pipeline* to a ready made [example project](#) starting in [exercise 11.2](#). You will [fork](#) the example project and that will create you a personal copy of the repository. In the [last two](#) exercises, you will build another deployment pipeline for some of *your own* previously created apps!

There are 21 exercises in this part, and you need to complete *each* exercise for completing the course. Exercises are submitted via [the submissions system](#) just like in the previous parts, but unlike parts 0 to 7, the submission goes to a different "course instance".

This part will rely on many concepts covered in the previous parts of the course. It is recommended that you finish at least parts 0 to 5 before starting this part.

Unlike the other parts of this course, you do not write many lines of code in this part, it is much more about configuration. Debugging code might be hard but debugging configurations is way harder, so in this part, you need lots of patience and discipline!

Getting software to production

Writing software is all well and good but nothing exists in a vacuum. Eventually, we'll need to deploy the software to production, i.e. give it to the real users. After that we need to maintain it, release new versions, and work with other people to expand that software.

We've already used GitHub to store our source code, but what happens when we work within a team with more developers?

Many problems may arise when several developers are involved. The software might work just fine on *my computer*, but maybe some of the other developers are using a different operating system or different library versions. It is not uncommon that a code works just fine in one developer's machine but another developer can not even get it started. This is often called the "works on my machine" problem.

There are also more involved problems. If two developers are both working on changes and they haven't decided on a way to deploy to production, whose changes get deployed? How would it be possible to prevent one developer's changes from overwriting another's?

In this part, we'll cover ways to work together and build and deploy software in a strictly defined way so that it's clear *exactly* what will happen under any given circumstance.

Some useful terms

In this part we'll be using some terms you may not be familiar with or you may not have a good understanding of. We'll discuss some of these terms here. Even if you are familiar with the terms, give this section a read so when we use the terms in this part, we're on the same page.

Branches

Git allows multiple copies, streams, or versions of the code to co-exist without overwriting each other. When you first create a repository, you will be looking at the main branch (usually in Git, we call this *main* or *master*, but that does vary in older projects). This is fine if there's only one developer for a project and that developer only works on one feature at a time.

Branches are useful when this environment becomes more complex. In this context, each developer can have one or more branches. Each branch is effectively a copy of the main branch with some changes that make it diverge from it. Once the feature or change in the branch is ready it can be *merged* back into the main branch, effectively making that feature or change part of the main software. In this way, each developer can work on their own set of changes and not affect any other developer until the changes are ready.

But once one developer has merged their changes into the main branch, what happens to the other developers' branches? They are now diverging from an older copy of the main branch. How will the developer on the later branch know if their changes are compatible with the current state of the main branch? That is one of the fundamental questions we will be trying to answer in this part.

You can read more about branches e.g. from [here](#).

Pull request

In GitHub merging a branch back to the main branch of software is quite often happening using a mechanism called [pull request](#), where the developer who has done some changes is requesting the changes to be merged to the main branch. Once the pull request, or PR as it's often called, is made or *opened*, another developer checks that all is ok and *merges* the PR.

If you have proposed changes to the material of this course, you have already made a pull request!

Build

The term "build" has different meanings in different languages. In some interpreted languages such as Python or Ruby, there is actually no need for a build step at all.

In general when we talk about building we mean preparing software to run on the platform where it's intended to run. This might mean, for example, that if you've written your application in TypeScript, and you intend to run it on Node, then the build step might be transpiling the TypeScript into JavaScript.

This step is much more complicated (and required) in compiled languages such as C and Rust where the code needs to be compiled into an executable.

In [part 7](#) we had a look at [Webpack](#) that is the current de facto tool for building a production version of a React or any other frontend JavaScript or TypeScript codebase.

Deploy

Deployment refers to putting the software where it needs to be for the end-user to use it. In the case of libraries, this may simply mean pushing an npm package to a package archive (such as [npmjs.com](#)) where other users can find it and include it in their software.

Deploying a service (such as a web app) can vary in complexity. In [part 3](#) our deployment workflow involved running some scripts manually and pushing the repository code to [Fly.io](#) or [Render](#) hosting service.

In this part, we'll develop a simple "deployment pipeline" that deploys each commit of your code automatically to Fly.io or Render if the committed code does not break anything.

Deployments can be significantly more complex, especially if we add requirements such as "the software must be available at all times during the deployment" (zero downtime deployments) or if we have to take things like [database migrations](#) into account. We won't cover complex deployments like those in this part but it's important to know that they exist.

What is CI?

The strict definition of CI (Continuous Integration) and the way the term is used in the industry may sometimes be different. One influential but quite early (written already in 2006) discussion of the topic is in [Martin Fowler's blog](#).

Strictly speaking, CI refers to [merging developer changes to the main branch](#) often, Wikipedia even helpfully suggests: "several times a day". This is usually true but when we refer to CI in industry, we're quite often talking about what happens after the actual merge happens.

We'd likely want to do some of these steps:

- Lint: to keep our code clean and maintainable
- Build: put all of our code together into runnable software bundle
- Test: to ensure we don't break existing features
- Package: Put it all together in an easily movable batch
- Deploy: Make it available to the world

We'll discuss each of these steps (and when they're suitable) in more detail later. What is important to remember is that this process should be strictly defined.

Usually, strict definitions act as a constraint on creativity/development speed. This, however, should usually not be true for CI. This strictness should be set up in such a way as to allow for easier development and working together. Using a good CI system (such as GitHub Actions that we'll cover in this part) will allow us to do this all automagically.

Packaging and Deployment as a part of CI

It may be worthwhile to note that packaging and especially deployment are sometimes not considered to fall under the umbrella of CI. We'll add them in here because in the real world it makes sense to lump it all together. This is partly because they make sense in the context of the flow and pipeline (I want to get my code to users) and partially because these are in fact the most likely point of failure.

The packaging is often an area where issues crop up in CI as this isn't something that's usually tested locally. It makes sense to test the packaging of a project during the CI workflow even if we don't do anything with the resulting package. With some workflows, we may even be testing the already built packages. This assures us that we have tested the code in the same form as what will be deployed to production.

What about deployment then? We'll talk about consistency and repeatability at length in the coming sections but we'll mention here that we want a process that always looks the same, whether we're running tests on a development branch or the main branch. In fact, the process may *literally* be the same with only a check at the end to determine if we are on the main branch and need to do a deployment. In this context, it makes sense to include deployment in the CI process since we'll be

maintaining it at the same time we work on CI.

Is this CD thing related?

The terms *Continuous Delivery* and *Continuous Deployment* (both of which have the acronym CD) are often used when one talks about CI that also takes care of deployments. We won't bore you with the exact definition (you can use e.g. [Wikipedia](#) or [another Martin Fowler blog post](#)) but in general, we refer to CD as the practice where the main branch is kept deployable at all times. In general, this is also frequently coupled with automated deployments triggered from merges into the main branch.

What about the murky area between CI and CD? If we, for example, have tests that must be run before any new code can be merged to the main branch, is this CI because we're making frequent merges to the main branch, or is it CD because we're making sure that the main branch is always deployable?

So, some concepts frequently cross the line between CI and CD and, as we discussed above, deployment sometimes makes sense to consider CD as part of CI. This is why you'll often see references to CI/CD to describe the entire process. We'll use the terms "CI" and "CI/CD" interchangeably in this part.

Why is it important?

Above we talked about the "works on my machine" problem and the deployment of multiple changes, but what about other issues. What if Alice committed directly to the main branch? What if Bob used a branch but didn't bother to run tests before merging? What if Charlie tries to build the software for production but does so with the wrong parameters?

With the use of continuous integration and systematic ways of working, we can avoid these.

- We can disallow commits directly to the main branch
- We can have our CI process run on all Pull Requests (PRs) against the main branch and allow merges only when our desired conditions are met e.g. tests pass
- We can build our packages for production in the known environment of the CI system

There are other advantages to extending this setup:

- If we use CI/CD with deployment every time there is a merge to the main branch, then we know that it will always work in production
- If we only allow merges when the branch is up to date with the main branch, then we can be sure that different developers don't overwrite each other's changes

Note that in this part we are assuming that the *main* branch contains the code that is running in production. There are numerous different [workflows](#) one can use with Git, e.g. in some cases, it may be a specific *release branch* that contains the code that is running in production.

Important principles

It's important to remember that CI/CD is not the goal. The goal is better, faster software development with fewer preventable bugs and better team cooperation.

To that end, CI should always be configured to the task at hand and the project itself. The end goal should be kept in mind at all times. You can think of CI as the answer to these questions:

- How to make sure that tests run on all code that will be deployed?
- How to make sure that the main branch is deployable at all times?
- How to ensure that builds will be consistent and will always work on the platform it'd be deploying to?
- How to make sure that the changes don't overwrite each other?
- How to make deployments happen at the click of a button or automatically when one merges to the main branch?

There even exists scientific evidence on the numerous benefits the usage of CI/CD has. According to a large study reported in the book [Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations](#), the use of CI/CD correlate heavily with organizational success (e.g. improves profitability and product quality, increases market share, shortens the time to market). CI/CD even makes developers happier by reducing their burnout rate. The results summarized in the book are also reported in scientific articles such as [this](#).

Documented behavior

There's an old joke that a bug is just an "undocumented feature". We'd like to avoid that. We'd like to avoid any situations where we don't know the exact outcome. For example, if we depend on a label on a PR to define whether something is a "major", "minor" or "patch" release (we'll cover the meanings of those terms later), then it's important that we know what happens if a developer forgets to put a label on their PR. What if they put a label on after the build/test process has started? What happens if the developer changes the label mid-way through, which one is the one that actually releases?

It's possible to cover all cases you can think of and still have gaps where the developer will do something "creative" that you didn't think of, so it's important to have the process fail safely in this case.

For example, if we have the case mentioned above where the label changes midway through the build. If we didn't think of this beforehand, it might be best to fail the build and alert the user if something we weren't expecting happened. The alternative, where we deploy the wrong type of version anyway, could result in bigger problems, so failing and notifying the developer is the safest way out of this situation.

Know the same thing happens every time

We might have the best tests imaginable for our software, tests that catch every possible issue. That's great, but they're useless if we don't run them on the code before it's deployed.

We need to guarantee that the tests will run and we need to be sure that they run against the code that will actually be deployed. For example, it's no use if the tests are *only* run against Alice's branch if they would fail after merging to the main branch. We're deploying from the main branch so we need to make sure that the tests are run against a copy of the main branch with Alice's changes merged in.

This brings us to a critical concept. We need to make sure that the same thing happens every time. Or rather that the required tasks are all performed and in the right order.

Code always kept deployable

Having code that's always deployable makes life easier. This is especially true when the main branch contains the code running in the production environment. For example, if a bug is found and it needs to be fixed, you can pull a copy of the main branch (knowing it is the code running in production), fix

the bug, and make a pull request back to the main branch. This is relatively straight forward.

If, on the other hand, the main branch and production are very different and the main branch is not deployable, then you would have to find out what code *is* running in production, pull a copy of that, fix the bug, figure out a way to push it back, then work out how to deploy that specific commit. That's not great and would have to be a completely different workflow from a normal deployment.

Knowing what code is deployed (sha sum/version)

It's often important to know what is actually running in production. Ideally, as we discussed above, we'd have the main branch running in production. This is not always possible. Sometimes we intend to have the main branch in production but a build fails, sometimes we batch together several changes and want to have them all deployed at once.

What we need in these cases (and is a good idea in general) is to know exactly *what code is running in production*. Sometimes this can be done with a version number, sometimes it's useful to have the commit SHA sum (uniquely identifying hash of that particular commit in git) attached to the code. We will discuss versioning further [a bit later in this part](#).

It is even more useful if we combine the version information with a history of all releases. If, for example, we found out that a particular commit has introduced a bug, we can find out exactly when that was released and how many users were affected. This is especially useful when that bug has written bad data to the database. We'd now be able to track where that bad data went based on the time.

Types of CI setup

To meet some of the requirements listed above, we want to dedicate a separate server for running the tasks in continuous integration. Having a separate server for the purpose minimizes the risk that something else interferes with the CI/CD process and causes it to be unpredictable.

There are two options: host our own server or use a cloud service.

Jenkins (and other self-hosted setups)

Among the self-hosted options, [Jenkins](#) is the most popular. It's extremely flexible and there are plugins for almost anything (except that one thing you want to do). This is a great option for many applications, using a self-hosted setup means that the entire environment is under your control, the number of resources can be controlled, secrets (we'll elaborate a little more on security in later sections of this part) are never exposed to anyone else and you can do anything you want on the hardware.

Unfortunately, there is also a downside. Jenkins is quite complicated to set up. It's very flexible but that means that there's often quite a bit of boilerplate/template code involved to get builds working. With Jenkins specifically, it also means that CI/CD must be set up with Jenkins' own domain-specific language. There are also the risks of hardware failures which can be an issue if the setup sees heavy use.

With self-hosted options, the billing is usually based on the hardware. You pay for the server. What you do on the server doesn't change the billing.

GitHub Actions and other cloud-based solutions

In a cloud-hosted setup, the setup of the environment is not something you need to worry about. It's there, all you need to do is tell it what to do. Doing that usually involves putting a file in your repository and then telling the CI system to read the file (or to check your repository for that particular file).

The actual CI config for the cloud-based options is often a little simpler, at least if you stay within what is considered "normal" usage. If you want to do something a little bit more special, then cloud-based options may become more limited, or you may find it difficult to do that one specific task for which the cloud platform just isn't built for.

In this part, we'll look at a fairly normal use case. The more complicated setups might, for example, make use of specific hardware resources, e.g. a GPU.

Aside from the configuration issue mentioned above, there are often resource limitations on cloud-based platforms. In a self-hosted setup, if a build is slow, you can just get a bigger server and throw more resources at it. In cloud-based options, this may not be possible. For example, in [GitHub Actions](#), the nodes your builds will run on have 2 vCPUs and 8GB of RAM.

Cloud-based options are also usually billed by build time which is something to consider.

Why pick one over the other

In general, if you have a small to medium software project that doesn't have any special requirements (e.g. a need for a graphics card to run tests), a cloud-based solution is probably best. The configuration is simple and you don't need to go to the hassle or expense of setting up your own system. For smaller projects especially, this should be cheaper.

For larger projects where more resources are needed or in larger companies where there are multiple teams and projects to take advantage of it, a self-hosted CI setup is probably the way to go.

Why use GitHub Actions for this course

For this course, we'll use [GitHub Actions](#). It is an obvious choice since we're using GitHub anyway. We can get a robust CI solution working immediately without any hassle of setting up a server or configuring a third-party cloud-based service.

Besides being easy to take into use, GitHub Actions is a good choice in other respects. It might be the best cloud-based solution at the moment. It has gained lots of popularity since its initial release in November 2019.

Exercise 11.1

Before getting our hands dirty with setting up the CI/CD pipeline let us reflect a bit on what we have read.

11.1 Warming up

Think about a hypothetical situation where we have an application being worked on by a team of about 6 people. The application is in active development and will be released soon.

Let us assume that the application is coded with some other language than JavaScript/TypeScript, e.g. in Python, Java, or Ruby. You can freely pick the language. This might even be a language you do not know much yourself.

Write a short text, say 200-300 words, where you answer or discuss some of the points below. You can check the length with <https://wordcounter.net/>. Save your answer to the file named `exercise1.md` in the root of the repository that you shall create in **exercise 11.2**.

The points to discuss:

- Some common steps in a CI setup include *linting*, *testing*, and *building*. What are the specific tools for taking care of these steps in the ecosystem of the language you picked? You can search for the answers by Google.
- What alternatives are there to set up the CI besides Jenkins and GitHub Actions? Again, you can ask Google!
- Would this setup be better in a self-hosted or a cloud-based environment? Why? What information would you need to make that decision?

Remember that there are no 'right' answers to the above!

11b: Getting started with GitHub Actions

Before we start playing with GitHub Actions, let's have a look at what they are and how do they work.

GitHub Actions work on a basis of [workflows](#). A workflow is a series of [jobs](#) that are run when a certain triggering [event](#) happens. The jobs that are run then themselves contain instructions for what GitHub Actions should do.

A typical execution of a workflow looks like this:

- Triggering event happens (for example, there is a push to the main branch).
- The workflow with that trigger is executed.
- Cleanup

Basic needs

In general, to have CI operate on a repository, we need a few things:

- A repository (obviously)
- Some definition of what the CI needs to do: This can be in the form of a specific file inside the repository or it can be defined in the CI system
- The CI needs to be aware that the repository (and the configuration file within it) exist
- The CI needs to be able to access the repository
- The CI needs permissions to perform the actions it is supposed to be able to do: For example, if the CI needs to be able to deploy to a production environment, it needs *credentials* for that environment.

That's the traditional model at least, we'll see in a minute how GitHub Actions short-circuit some of these steps or rather make it such that you don't have to worry about them!

GitHub Actions have a great advantage over self-hosted solutions: the repository is hosted with the CI provider. In other words, GitHub provides both the repository and the CI platform. This means that if we've enabled actions for a repository, GitHub is already aware of the fact that we have workflows defined and what those definitions look like.

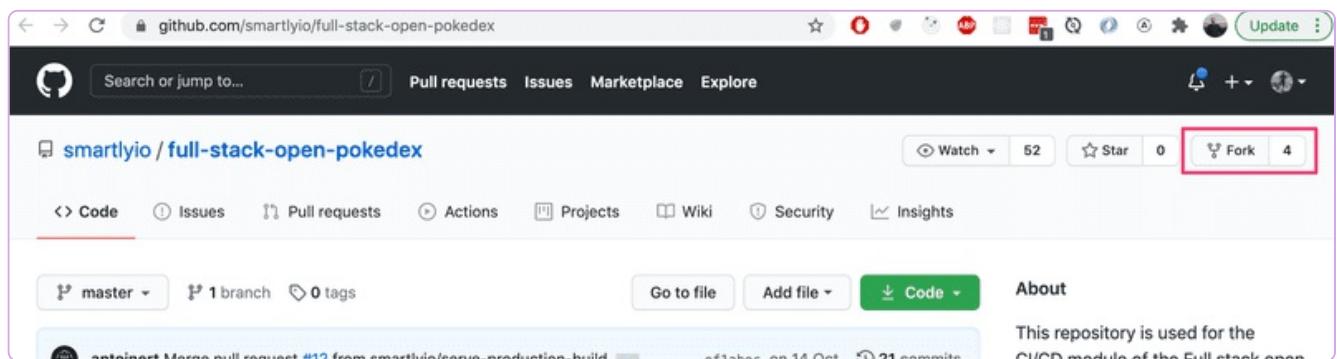
Exercise 11.2.

In most exercises of this part, we are building a CI/CD pipeline for a small project found in [this example project repository](#).

11.2 The example project

The first thing you'll want to do is to fork the example repository under your name. What it essentially does is it creates a copy of the repository under your GitHub user profile for your use.

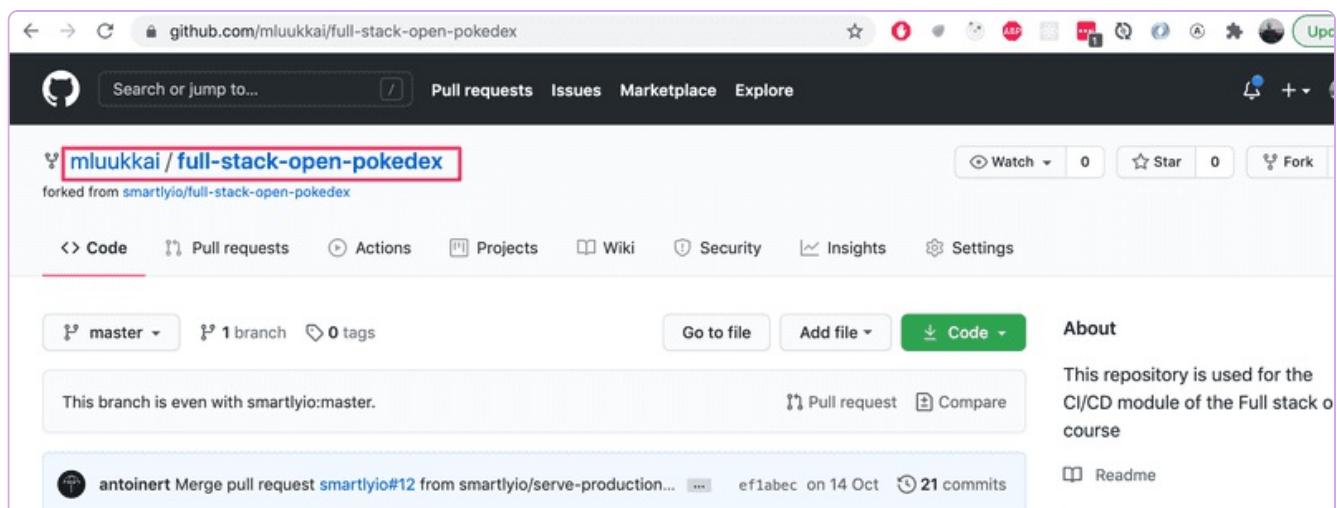
To fork the repository, you can click on the Fork button in the top-right area of the repository view next to the Star button:



A screenshot of a web browser showing a GitHub repository page. The URL in the address bar is `github.com/smartlyio/full-stack-open-pokedex`. The repository name 'smartlyio / full-stack-open-pokedex' is displayed in the header. In the top right corner, there is a 'Star' button with 0 stars and a 'Fork' button with 4 forks. The 'Fork' button is highlighted with a red box. Below the header, there are navigation links for 'Code', 'Issues', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', and 'Insights'. Underneath these, there are buttons for 'master', '1 branch', '0 tags', 'Go to file', 'Add file', and 'Code'. A green 'About' button is also present. At the bottom of the page, there is a note: 'This repository is used for the CI/CD module of the Full stack open'.

Once you've clicked on the Fork button, GitHub will start the creation of a new repository called `{github_username} / full-stack-open-pokedex`.

Once the process has been finished, you should be redirected to your brand-new repository:



A screenshot of a web browser showing a GitHub repository page. The URL in the address bar is `github.com/mluukkai/full-stack-open-pokedex`. The repository name 'mluukkai / full-stack-open-pokedex' is highlighted with a red box. It is noted that it was 'forked from [smartlyio/full-stack-open-pokedex](#)'. Below the header, there are navigation links for 'Code', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings'. Underneath these, there are buttons for 'master', '1 branch', '0 tags', 'Go to file', 'Add file', and 'Code'. A green 'About' button is also present. At the bottom of the page, there is a note: 'This repository is used for the CI/CD module of the Full stack open course'.

Clone the project now to your machine. As always, when starting with a new code, the most obvious place to look first is the file `package.json`

NOTE since the project is already a bit old, you need Node 16 to work with it!

Try now the following:

- install dependencies (by running `npm install`)
- start the code in development mode
- run tests
- lint the code

You might notice that the project contains some broken tests and linting errors. **Just leave them as**

they are for now. We will get around those later in the exercises.

NOTE the tests of the project have been made with [Jest](#). The course material in [part 5](#) uses [Vitest](#). From the usage point of view, the libraries have barely any difference.

As you might remember from [part 3](#), the React code *should not* be run in development mode once it is deployed in production. Try now the following

- create a production *build* of the project
- run the production version locally

Also for these two tasks, there are ready-made npm scripts in the project!

Study the structure of the project for a while. As you notice both the frontend and the backend code are now [in the same repository](#). In earlier parts of the course we had a separate repository for both, but having those in the same repository makes things much simpler when setting up a CI environment.

In contrast to most projects in this course, the frontend code *does not use* Vite but it has a relatively simple [Webpack](#) configuration that takes care of creating the development environment and creating the production bundle.

Getting started with workflows

The core component of creating CI/CD pipelines with GitHub Actions is something called a [Workflow](#). Workflows are process flows that you can set up in your repository to run automated tasks such as building, testing, linting, releasing, and deploying to name a few! The hierarchy of a workflow looks as follows:

Workflow

- Job
 - Step
 - Step
- Job
 - Step

Each workflow must specify at least one [Job](#), which contains a set of [Steps](#) to perform individual tasks. The jobs will be run in parallel and the steps in each job will be executed sequentially.

Steps can vary from running a custom command to using pre-defined actions, thus the name GitHub Actions. You can create [customized actions](#) or use any actions published by the community, which are plenty, but let's get back to that later!

For GitHub to recognize your workflows, they must be specified in `.github/workflows` folder in your repository. Each Workflow is its own separate file which needs to be configured using the [YAML](#) data-serialization language.

YAML is a recursive acronym for "YAML Ain't Markup Language". As the name might hint its goal is to be human-readable and it is commonly used for configuration files. You will notice below that it is indeed very easy to understand!

Notice that indentations are important in YAML. You can learn more about the syntax [here](#).

A basic workflow contains three elements in a YAML document. These three elements are:

- name: Yep, you guessed it, the name of the workflow
- (on) triggers: The events that trigger the workflow to be executed
- jobs: The separate jobs that the workflow will execute (a basic workflow might contain only one job).

A simple workflow definition looks like this:

```
name: Hello World!

on:
  push:
    branches:
      - main

jobs:
  hello_world_job:
    runs-on: ubuntu-20.04
    steps:
      - name: Say hello
        run: |
          echo "Hello World!"
```

There is one job named *hello_world_job*, it will be run in a virtual environment with Ubuntu 20.04. The job has just one step named "Say hello", which will run the `echo "Hello World!"` command in the shell.

So you may ask, when does GitHub trigger a workflow to be started? There are plenty of [options](#) to choose from, but generally speaking, you can configure a workflow to start once:

- An *event on GitHub* occurs such as when someone pushes a commit to a repository or when an issue or pull request is created
- A *scheduled event*, that is specified using the [cron](#)-syntax, happens
- An *external event* occurs, for example, a command is performed in an external application such as [Slack](#) or [Discord](#) messaging app

To learn more about which events can be used to trigger workflows, please refer to GitHub Action's [documentation](#).

Exercises 11.3-11.4.

To tie this all together, let us now get GitHub Actions up and running in the example project!

11.3 Hello world!

Create a new Workflow that outputs "Hello World!" to the user. For the setup, you should create the directory `.github/workflows` and a file `hello.yml` to your repository.

To see what your GitHub Action workflow has done, you can navigate to the **Actions** tab in GitHub where you should see the workflows in your repository and the steps they implement. The output of

your Hello World workflow should look something like this with a properly configured workflow.

The screenshot shows a GitHub repository page for 'mluukkai / full-stack-open-pokedex'. The 'Actions' tab is selected. A workflow named 'hello GHA' is shown, with a single job named 'Hello World!' triggered by a push event. The job status is 'succeeded' 2 minutes ago in 1s. The log for the 'hello_world_job' shows the following steps:

- Set up job:
 - Current runner version: '2.274.2'
 - Operating System: Ubuntu 18.04.5 LTS
 - Virtual Environment: Environment: ubuntu-18.04 Version: 20201129.1 Included Software: <https://github.com/actions/virtual-environments/blob/ubuntu18/20201129.1/images/linux/Ubuntu1804-README.md>
 - Prepare workflow directory
 - Prepare all required actions
- Say hello:
 - Run echo "Hello World!"
 - Hello World!
- Complete job

A red arrow points to the 'Say hello' step, highlighting the output 'Hello World!'.
The log output for the 'Say hello' step is:
1 ▶ Run echo "Hello World!"
4 Hello World!

You should see the "Hello World!" message as an output. If that's the case then you have successfully gone through all the necessary steps. You have your first GitHub Actions workflow active!

Note that GitHub Actions also informs you on the exact environment (operating system, and its [setup](#)) where your workflow is run. This is important since if something surprising happens, it makes debugging so much easier if you can reproduce all the steps in your machine!

11.4 Date and directory contents

Extend the workflow with steps that print the date and current directory content in the long format.

Both of these are easy steps, and just running commands `date` and `ls` will do the trick.

Your workflow should now look like this

The screenshot shows a GitHub Actions workflow named 'hello_world_job'. The workflow has four steps: 'Set up job', 'Say hello', 'Now it is', and 'Directory content'. The 'Say hello' step is expanded, showing a command to run 'echo "Hello World!"' which outputs 'Hello World!'. The 'Now it is' step shows a command to run 'date' which outputs 'Sun Dec 6 10:26:42 UTC 2020'. The 'Directory content' step shows a command to run 'ls -l' which outputs 'total 0'.

As the output of the command `ls -l` shows, by default, the virtual environment that runs our workflow *does not have any code!*

Setting up lint, test and build steps

After completing the first exercises, you should have a simple but pretty useless workflow set up. Let's make our workflow do something useful.

Let's implement a GitHub Action that will lint the code. If the checks don't pass, GitHub Actions will show a red status.

At the start, the workflow that we will save to file `pipeline.yml` looks like this:

```
name: Deployment pipeline

on:
  push:
    branches:
      - main

jobs:
```

Before we can run a command to lint the code, we have to perform a couple of actions to set up the environment of the job.

Setting up the environment

Setting up the environment is an important task while configuring a pipeline. We're going to use an `ubuntu-20.04` virtual environment because this is the version of Ubuntu we're going to be running in production.

It is important to replicate the same environment in CI as in production as closely as possible, to avoid situations where the same code works differently in CI and production, which would effectively defeat the purpose of using CI.

Next, we list the steps in the "build" job that the CI would need to perform. As we noticed in the last exercise, by default the virtual environment does not have any code in it, so we need to *checkout the code* from the repository.

This is an easy step:

```
name: Deployment pipeline

on:
  push:
    branches:
      - main

jobs:
  simple_deployment_pipeline:    runs-on: ubuntu-20.04    steps:
    - uses: actions/checkout@v4
```

The `uses` keyword tells the workflow to run a specific *action*. An action is a reusable piece of code, like a function. Actions can be defined in your repository in a separate file or you can use the ones available in public repositories.

Here we're using a public action `actions/checkout` and we specify a version (`@v4`) to avoid potential breaking changes if the action gets updated. The `checkout` action does what the name implies: it checkouts the project source code from Git.

Secondly, as the application is written in JavaScript, Node.js must be set up to be able to utilize the commands that are specified in `package.json`. To set up Node.js, `actions/setup-node` action can be used. Version `20` is selected because it is the version the application is using in the production environment.

```
# name and trigger not shown anymore...

jobs:
  simple_deployment_pipeline:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4        with:           node-version: '20'
```

As we can see, the `with` keyword is used to give a "parameter" to the action. Here the parameter specifies the version of Node.js we want to use.

Lastly, the dependencies of the application must be installed. Just like on your own machine we execute `npm install`. The steps in the job should now look something like

```
jobs:
  simple_deployment_pipeline:
    runs-on: ubuntu-20.04
```

```
steps:
  - uses: actions/checkout@v4
  - uses: actions/setup-node@v4
    with:
      node-version: '20'
  - name: Install dependencies           run: npm install
```

Now the environment should be completely ready for the job to run actual important tasks in!

Lint

After the environment has been set up we can run all the scripts from `package.json` like we would on our own machine. To lint the code all you have to do is add a step to run the `npm run eslint` command.

```
jobs:
  simple_deployment_pipeline:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
      - name: Install dependencies
        run: npm install
      - name: Check style                 run: npm run eslint
```

Note that the `name` of a step is optional, if you define a step as follows

```
- run: npm run eslint
```

the command that is run is used as the default name.

Exercises 11.5.-11.9.

11.5 Linting workflow

Implement or *copy-paste* the "Lint" workflow and commit it to the repository. Use a new `yml` file for this workflow, you may call it e.g. `pipeline.yml`.

Push your code and navigate to "Actions" tab and click on your newly created workflow on the left. You should see that the workflow run has failed:

The screenshot shows a GitHub repository page for 'mluukkai / full-stack-open-pokedex'. The 'Actions' tab is selected, displaying a deployment pipeline named 'simple_deployment_pipeline' triggered by a push to the 'master' branch. The pipeline has 10 steps, one of which, 'lint', failed 31 seconds ago. The log details the following steps:

- > ✓ Set up job
- > ✓ Run actions/checkout@v2
- > ✓ Run actions/setup-node@v1
- > ✓ npm install
- > ✘ lint
- > ✓ Post Run actions/checkout@v2
- > ✓ Complete job

11.6 Fix the code

There are some issues with the code that you will need to fix. Open up the workflow logs and investigate what is wrong.

A couple of hints. One of the errors is best to be fixed by specifying proper `env` for linting, see [here](#) how it can be done . One of the complaints concerning `console.log` statement could be taken care of by simply silencing the rule for that specific line. Ask google how to do it.

Make the necessary changes to the source code so that the lint workflow passes. Once you commit new code the workflow will run again and you will see updated output where all is green again:

The screenshot shows the 'Workflows' section of the GitHub repository. The 'Deployment pipeline' workflow is selected. A search bar shows 'workflow:"Deployment pipeline"'. The results table displays three successful workflow runs:

Event	Status
Deployment pipeline #3: Commit 2b19029 pushed by mluukkai	master
Deployment pipeline #2: Commit 4cb9383 pushed by mluukkai	master
Deployment pipeline #1: Commit a4ede08 pushed by mluukkai	master

11.7 Building and testing

Let's expand on the previous workflow that currently does the linting of the code. Edit the workflow and similarly to the lint command add commands for build and test. After this step outcome should look like this

A screenshot of a GitHub repository page showing a workflow named "simple_deployment_pipeline". The workflow has one job, which has seven steps. The steps are: Set up job, Run actions/checkout@v2, Run actions/setup-node@v1, npm install, lint, build, and test. The "test" step is marked with a red X, indicating it failed. A large red arrow points to the "test" step in the list.

As you might have guessed, there are some problems in code...

11.8 Back to green

Investigate which test fails and fix the issue in the code (do not change the tests).

Once you have fixed all the issues and the Pokedex is bug-free, the workflow run will succeed and show green!

A screenshot of a GitHub repository page showing a workflow named "simple_deployment_pipeline". The workflow has one job, which has seven steps. The steps are: Set up job, Run actions/checkout@v2, Run actions/setup-node@v1, npm install, lint, build, and test. All steps are marked with a green checkmark, indicating they succeeded. The workflow status is labeled "fixed".

11.9 Simple end-to-end tests

The current set of tests uses [Jest](#) to ensure that the React components work as intended. This is essentially the same thing that is done in the section [Testing React apps](#) of part 5 with [Vitest](#).

Testing components in isolation is quite useful but that still does not ensure that the system as a whole works as we wish. To have more confidence about this, let us write a couple of really simple end-to-end tests similarly we did in section [part 5](#). You could use [Playwright](#) or [Cypress](#) for the tests.

No matter which you choose, you should extend Jest-definition in package.json to prevent Jest from trying to run the e2e-tests. Assuming that directory `e2e-tests` is used for e2e-tests, the definition is:

```
[  
  // ...  
  "jest": [  
    "testEnvironment": "jsdom",  
    "testPathIgnorePatterns": ["e2e-tests"]  
]
```

Playwright

Set Playwright up (you'll find [here](#) all the info you need) to your repository. Note that in contrast to part 5, you should now install Playwright to the same project with the rest of the code!

Use this test first:

```
const { test, describe, expect, beforeEach } = require('@playwright/test')  
  
describe('Pokedex', () => {  
  test('front page can be opened', async ({ page }) => {  
    await page.goto('')  
    await expect(page.getByText('ivysaur')).toBeVisible()  
    await expect(page.getByText('Pokémon and Pokémon character names are trademarks of Nintendo.')).toBeVisible()  
  })  
})
```

Note is that although the page renders the Pokemon names with an initial capital letter, the names are actually written with lowercase letters in the source, so you should test for `ivysaur` instead of `Ivysaur`!

Define a npm script `test:e2e` for running the e2e tests from the command line.

Remember that the Playwright tests *assume that the application is up and running* when you run the test! Instead of starting the app manually, you should now configure a *Playwright development server* to start the app while tests are executed, see [here](#) how that can be done.

Ensure that the test passes locally.

Once the end-to-end test works in your machine, include it in the GitHub Action workflow. That should be pretty easy by following [this](#).

Cypress

Set Cypress up (you'll find [here](#) all the info you need) and use this test first:

```
describe('Pokedex', function() {
```

```
it('front page can be opened', function() {
  cy.visit('http://localhost:5000')
  cy.contains('ivysaur')
  cy.contains('Pokémon and Pokémon character names are trademarks of Nintendo.')
})
})
})
```

Define a npm script `test:e2e` for running the e2e tests from the command line.

Note is that although the page renders the Pokemon names with an initial capital letter, the names are actually written with lowercase letters in the source, so you should test for `ivysaur` instead of `Ivysaur`!

Ensure that the test passes locally. Remember that the Cypress tests *assume that the application is up and running* when you run the test! If you have forgotten the details, please see [part 5](#) how to get up and running with Cypress.

Once the end-to-end test works in your machine, include it in the GitHub Action workflow. By far the easiest way to do that is to use the ready-made action [cypress-io/github-action](#). The step that suits us is the following:

```
- name: e2e tests
  uses: cypress-io/github-action@v5
  with:
    command: npm run test:e2e
    start: npm run start-prod
    wait-on: http://localhost:5000
```

Three options are used: `command` specifies how to run Cypress tests, `start` gives npm script that starts the server, and `wait-on` says that before the tests are run, the server should have started on url <http://localhost:5000>.

Note that you need to build the app in GitHub Actions before it can be started in production mode!

Once the pipeline works...

Once you are sure that the pipeline works, *write another test* that ensures that one can navigate from the main page to the page of a particular Pokemon, e.g. `ivysaur`. The test does not need to be a complex one, just check that when you navigate to a link, the page has some proper content, such as the string `chlorophyll` in the case of `ivysaur`.

Note the Pokemon abilities are written with lowercase letters in the source code (the capitalization is done in CSS), so *do not* test for `Chlorophyll` but rather `chlorophyll`.

The end result should be something like this

The screenshot shows a GitHub repository page for 'mluukkai / full-stack-open-pokedex'. The 'Actions' tab is selected, displaying a recent run titled 'simple_deployment_pipeline' triggered by a push to the 'master' branch. The pipeline status is 'succeeded 3 minutes ago in 2m 12s'. The results section shows an 'e2e tests' step with the following log output:

```
81
82
83 Pokedex
84   ✓ front page can be opened (13382ms)
85   ✓ pokemon page can be navigated to (525ms)
86
87
88  2 passing (15s)
89
90
91  (Results)
92
93
94  | Tests:      2
95  | Passing:    2
```

End-to-end tests are nice since they give us confidence that software works from the end user's perspective. The price we have to pay is the slower feedback time. Now executing the whole workflow takes quite much longer.

11c: Deployment

Having written a nice application it's time to think about how we're going to deploy it to the use of real users.

In [part 3](#) of this course, we did this by simply running a single command from terminal to get the code up and running the servers of the cloud provider [Fly.io](#) or [Render](#).

It is pretty simple to release software in Fly.io and Render at least compared to many other types of hosting setups but it still contains risks: nothing prevents us from accidentally releasing broken code to production.

Next, we're going to look at the principles of making a deployment safely and some of the principles of deploying software on both a small and large scale.

Anything that can go wrong...

We'd like to define some rules about how our deployment process should work but before that, we have to look at some constraints of reality.

One phrasing of Murphy's Law holds that: "Anything that can go wrong will go wrong."

It's important to remember this when we plan out our deployment system. Some of the things we'll need to consider could include:

- What if my computer crashes or hangs during deployment?
- I'm connected to the server and deploying over the internet, what happens if my internet

connection dies?

- What happens if any specific instruction in my deployment script/system fails?
- What happens if, for whatever reason, my software doesn't work as expected on the server I'm deploying to? Can I roll back to a previous version?
- What happens if a user does an HTTP request to our software just before we do deployment (we didn't have time to send a response to the user)?

These are just a small selection of what can go wrong during a deployment, or rather, things that we should plan for. Regardless of what happens, our deployment system should **never** leave our software in a broken state. We should also always know (or be easily able to find out) what state a deployment is in.

Another important rule to remember when it comes to deployments (and CI in general) is: "Silent failures are **very** bad!"

This doesn't mean that failures need to be shown to the users of the software, it means we need to be aware if anything goes wrong. If we are aware of a problem, we can fix it. If the deployment system doesn't give any errors but fails, we may end up in a state where we believe we have fixed a critical bug but the deployment failed, leaving the bug in our production environment and us unaware of the situation.

What does a good deployment system do?

Defining definitive rules or requirements for a deployment system is difficult, let's try anyway:

- Our deployment system should be able to fail gracefully at **any** step of the deployment.
- Our deployment system should **never** leave our software in a broken state.
- Our deployment system should let us know when a failure has happened. It's more important to notify about failure than about success.
- Our deployment system should allow us to roll back to a previous deployment
 - Preferably this rollback should be easier to do and less prone to failure than a full deployment
 - Of course, the best option would be an automatic rollback in case of deployment failures
- Our deployment system should handle the situation where a user makes an HTTP request just before/during a deployment.
- Our deployment system should make sure that the software we are deploying meets the requirements we have set for this (e.g. don't deploy if tests haven't been run).

Let's define some things we **want** in this hypothetical deployment system too:

- We would like it to be fast
- We'd like to have no downtime during the deployment (this is distinct from the requirement we have for handling user requests just before/during the deployment).

Next we will have three sets of exercises for automating the deployment with GitHub Actions, one for [Fly.io](#), another one for [Render](#). The process of deployment is always specific to the particular cloud provider, so you can also do both the exercise sets if you want to see the differences on how these services work with respect to deployments.

Has the app been deployed?

Since we are not making any real changes to the app, it might be a bit hard to see if the app

deployment really works. Let us create a dummy endpoint in the app that makes it possible to do some code changes and to ensure that the deployed version has really changed:

```
app.get('/version', (req, res) => {
  res.send('1') // change this string to ensure a new version deployed
})
```

Exercises 11.10-11.12. (Fly.io)

If you rather want to use other hosting options, there is an alternative set of exercises for [Render](#).

11.10 Deploying your application to Fly.io

Setup your application in [Fly.io](#) hosting service like the one we did in [part 3](#).

In contrast to part 3, in this part we *do not deploy the code* to Fly.io ourselves (with the command `flyctl deploy`), we let the GitHub Actions workflow do that for us.

Before going to the automated deployment, we shall ensure in this exercise that the app can be deployed manually.

So, create a new app in Fly.io. After that generate a Fly.io API token with the command

```
flyctl auth token
```

You'll need the token soon for your deployment workflow so save it somewhere (but do not commit that to GitHub)!

As said, before setting up the deployment pipeline in the next exercise we will now ensure that a manual deployment with the command `flyctl deploy` works.

A couple of changes are needed.

The configuration file `fly.toml` should be modified to include the following:

```
[env]
PORT = "3000" # add this where PORT matches the internal_port below

[processes]
app = "node app.js" # add this

[http_service]
internal_port = 3000
force_https = true
auto_stop_machines = true
auto_start_machines = true
min_machines_running = 0
processes = ["app"]
```

In `processes` we define the command that starts the application. Without this change Fly.io just starts the React dev server and that causes it to shut down since the app itself does not start up. We will

also set up the PORT to be passed to the app as an environment variable.

We also need to alter the file `.dockerignore` a bit, the next line should be removed:

```
dist
```

If the line is not removed, the product build of the frontend does not get downloaded to the Fly.io server.

Deployment should now work *if* the production build exists in the local machine, that is, the command `npm build` is run.

Before moving to the next exercise, make sure that the manual deployment with the command `flyctl deploy` works!

11.11 Automatic deployments

Extend the workflow with a step to deploy your application to Fly.io by following the advice given [here](#).

Note that the GitHub Action should create the production build (with `npm run build`) before the deployment step!

You need the authorization token that you just created for the deployment. The proper way to pass it's value to GitHub Actions is to use *Repository secrets*:

The screenshot shows the GitHub repository settings page for a repository named 'Actions secrets'. The 'Settings' tab is selected. On the left, there is a sidebar with various options like General, Access, Collaborators, Moderation options, Code and automation, Branches, Tags, Actions, Webhooks, Environments, Pages, Security, Code security and analysis, Deploy keys, Secrets (which is highlighted with a red box), Actions, and Dependabot. The main area is titled 'Actions secrets / New secret'. It has two fields: 'Name *' containing 'FLY_API_TOKEN' and 'Secret *' containing 'put your secret here'. At the bottom is a green 'Add secret' button. A small 'G+' icon is visible in the bottom right corner of the secret input field.

Now the workflow can access the token value as follows:

```
$(secrets.FLY_API_TOKEN)
```

If all goes well, your workflow log should look a bit like this:

The screenshot shows a deployment log for a workflow named 'simple_deployment_pipeline'. The deployment succeeded 1 minute ago in 2m 33s. The log output is as follows:

```
simple_deployment_pipeline
succeeded 1 minute ago in 2m 33s

▼ ✅ Run flyctl deploy --remote-only
137
138 Updating existing machines in 'pdex' with rolling strategy
139 > [1/2] Updating 48e5996b76eed8 [app]
140 > [1/2] Updating 48e5996b76eed8 [app]
141 > [1/2] Waiting for 48e5996b76eed8 [app] to have state: started
142 > [1/2] Machine 48e5996b76eed8 [app] has state: started
143 > [1/2] Checking that 48e5996b76eed8 [app] is up and running
144 > [1/2] Waiting for 48e5996b76eed8 [app] to become healthy: 1/1
145
146 ✓ [1/2] Machine 48e5996b76eed8 [app] update succeeded
147 > [2/2] Updating 5683dd7db03228 [app]
148 > [2/2] Updating 5683dd7db03228 [app]
149 > [2/2] Waiting for 5683dd7db03228 [app] to have state: started
150 > [2/2] Machine 5683dd7db03228 [app] has state: started
151 > [2/2] Checking that 5683dd7db03228 [app] is up and running
152 > [2/2] Waiting for 5683dd7db03228 [app] to become healthy: 0/1
153
154 > [2/2] Waiting for 5683dd7db03228 [app] to become healthy: 1/1
155
156 ✓ [2/2] Machine 5683dd7db03228 [app] update succeeded
157
158 Visit your newly deployed app at https://pdex.fly.dev/
```

Remember that it is always essential to keep an eye on what is happening in server logs when playing around with product deployments, so use `flyctl logs` early and use it often. No, use it all the time!

11.12 Health check

Each deployment in Fly.io creates a [release](#). Releases can be checked from the command line:

```
$ flyctl releases
VERSION STATUS DESCRIPTION USER DATE
v18 complete Release mluukkai@iki.fi 16h56m ago
v17 complete Release mluukkai@iki.fi 17h3m ago
v16 complete Release mluukkai@iki.fi 21h22m ago
v15 complete Release mluukkai@iki.fi 21h25m ago
v14 complete Release mluukkai@iki.fi 21h34m ago
```

It is essential to ensure that a deployment ends up in a *succeeding* release, where the app is in healthy functional state. Fortunately, Fly.io has several configuration options that take care of the application health check.

If we change the app as follows, it fails to start:

```
app.listen(PORT, () => [
  this._causes_error
  // eslint-disable-next-line no-console
  console.log(`server started on port ${PORT}`)
```

In this case, the deployment fails:

```
$ flyctl releases
VERSION STATUS      DESCRIPTION      USER      DATE
v19     failed      Release      mluukkai@iki.fi 3m52s ago
v18     complete    Release      mluukkai@iki.fi 16h56m ago
v17     complete    Release      mluukkai@iki.fi 17h3m ago
v16     complete    Release      mluukkai@iki.fi 21h22m ago
v15     complete    Release      mluukkai@iki.fi 21h25m ago
v14     complete    Release      mluukkai@iki.fi 21h34m ago
```

The app however stays up and running, Fly.io does not replace the functioning version (v18) with the broken one (v19).

Let us consider the following change

```
// start app in a wrong port
app.listen(PORT + 1, () => [
  // eslint-disable-next-line no-console
  console.log(`server started on port ${PORT}`)
])
```

Now the app starts but it is connected to the wrong port, so the service will not be functional. Fly.io thinks this is a successful deployment, so it deploys the app in a broken state.

One possibility to prevent broken deployments is to use an HTTP-level check defined in section [http_service.http_checks](#). This type of check can be used to ensure that the app for real is in a functional state.

Add a simple endpoint for doing an application health check to the backend. You may e.g. copy this code:

```
app.get('/health', (req, res) => {
  res.send('ok')
})
```

Configure then an [HTTP check](#) that ensures the health of the deployments based on the HTTP request to the defined health check endpoint.

You also need to set the [deployment strategy](#) (in the file `fly.toml`) of the app to be *canary*. These strategies ensure that only an app with a healthy state gets deployed.

Ensure that GitHub Actions notices if a deployment breaks your application:

```
simple_deployment_pipeline
failed 30 minutes ago in 7m 8s

▼ ✘ Run flyctl deploy --remote-only 6m
137
138 Updating existing machines in 'pdex' with rolling strategy
139 > [1/2] Updating 48e5996b76eed8 [app]
140 > [1/2] Updating 48e5996b76eed8 [app]
141 > [1/2] Waiting for 48e5996b76eed8 [app] to have state: started
142 > [1/2] Machine 48e5996b76eed8 [app] has state: started
143 > [1/2] Checking that 48e5996b76eed8 [app] is up and running
144 > [1/2] Waiting for 48e5996b76eed8 [app] to become healthy: 0/1
145
146 * [1/2] Machine 48e5996b76eed8 [app] update failed: timeout reached waiting for health checks to pass for machine
  48e5996b76eed8: failed to get VM 48e5996b76eed8: Get "https://api.machines.dev/v1/apps/pdex/machines/48e5996b76eed8"
    net/http: request canceled
147 Error: timeout reached waiting for health checks to pass for machine 48e5996b76eed8: failed to get VM 48e5996b76eed8
  Get "https://api.machines.dev/v1/apps/pdex/machines/48e5996b76eed8": net/http: request canceled
148 Your machine never reached the state "%s".
```

You may simulate this e.g. as follows:

```
app.get('/health', [req, res] => {
  // eslint-disable-next-line no-constant-condition
  if (true) throw['error...']
  res.send('ok')
})
```

Exercises 11.10-11.12. (Render)

If you rather want to use other hosting options, there is an alternative set of exercises for [Fly.io](#).

11.10 Deploying your application to Render

Set up your application in [Render](#). The setup is now not quite as straightforward as in [part 3](#). You have to carefully think about what should go to these settings:

Runtime
The runtime for your web service.

Build Command
This command runs in the root directory of your repository when a new version of your code is pushed, or when you deploy manually. It is typically a script that installs libraries, runs migrations, or compiles resources needed by your app.

Start Command
This command runs in the root directory of your app and is responsible for starting its processes. It is typically used to start a webserver for your app. It can access environment variables defined by you in Render.

If you need to run several commands in the build or start command, you may use a simple shell script for that.

Create eg. a file `build_step.sh` with the following content:

```
#!/bin/bash

echo "Build script"

# add the commands here
```

Give it execution permissions (Google or see e.g. [this](#) to find out how) and ensure that you can run it from the command line:

```
$ ./build_step.sh
Build script
```

Other option is to use a [Pre deploy command](#), with that you may run one additional command before the deployment starts.

You also need to open the *Advanced settings* and turn the auto-deploy off since we want to control the deployment in the GitHub Actions:

Auto-Deploy

Automatic deploy on every push to your repository or changes to your service? Select "No" to handle your deploys manually.

No



Ensure now that you get the app up and running. Use the *Manual deploy*.

Most likely things will fail at the start, so remember to keep the *Logs* open all the time.

11.11 Automatic deployments

Next step is to automate the deployment. There are two options, a ready-made custom action or the use of the Render deploy hook.

Deployment with custom action

Go to GitHub Actions [marketplace](#) and search for action for our purposes. You might search with *render deploy*. There are several actions to choose from. You can pick any. Quite often the best choice is the one with the most stars. It is also a good idea to look if the action is actively maintained (time of the last release) and does it have many open issues or pull requests.

Warning: for some reason, the most starred option [render-action](#) was very unreliable when the part was updated (16th Jan 2024), so better avoid that. If you end up with too much problems, the deploy hook might be a better option!

Set up the action to your workflow and ensure that every commit that passes all the checks results in a new deployment. Note that you need Render API key and the app service id for the deployment. See [here](#) how the API key is generated. You can get the service id from the URL of the Render dashboard of your app. The end of the URL (starting with *srv-*) is the id:

```
https://dashboard.render.com/web/srv-randomcharachtershere
```

Deployment with deploy hook

Alternative, and perhaps a more reliable option is to use [Render Deploy Hook](#) which is a private URL to trigger the deployment. You can get it from your app settings:

The screenshot shows the 'Settings' tab in the Render app. On the left sidebar, there are several tabs: Events, Logs, Disks, Environment, Shell, PRs, Jobs, Metrics, Scaling, and Settings. The 'Settings' tab is selected. In the main area, there are two sections: 'Auto-Deploy' and 'Deploy Hook'. Under 'Auto-Deploy', a dropdown menu is set to 'No'. Under 'Deploy Hook', a private URL is displayed: [https://api.render.com/deploy/srv-\[REDACTED\]&key-\[REDACTED\]](https://api.render.com/deploy/srv-[REDACTED]&key-[REDACTED]). There is also a 'Regenerate Hook' button.

DON'T USE the plain URL in your pipeline. Instead create GitHub secrets for your key and service id:

The screenshot shows the 'Actions secrets and variables' section in GitHub repository settings. The sidebar on the left includes 'General', 'Access', 'Collaborators', 'Moderation options', 'Code and automation' (with 'Branches', 'Tags', 'Actions', 'Webhooks', 'Environments', 'Codespaces', and 'Pages' listed), 'Security' (with 'Code security and analysis', 'Deploy keys', 'Secrets and variables' (which is selected), 'Actions', 'Codespaces', and 'Dependabot'), and 'Manage environments'. In the main area, there are two tabs: 'Secrets' (selected) and 'Variables'. A green button labeled 'New repository secret' is highlighted with a red box. Below the tabs, there are sections for 'Environment secrets' (empty) and 'Repository secrets' (containing 'RENDER_API_KEY' and 'RENDER_SERVICE_ID', both updated 30 minutes ago). The 'Repository secrets' section is also highlighted with a red box.

Then you can use them like this:

```
- name: Trigger deployment
  run: curl https://api.render.com/deploy/srv-${{ secrets.RENDER_SERVICE_ID }}?key=${{ secrets.RENDER_API_KEY }}
```

The deployment takes some time. See the events tab of the Render dashboard to see when the new deployment is ready:

The screenshot shows the Render Events dashboard. On the left, there's a sidebar with tabs: Events (which is selected), Logs, Disks, Environment, Shell, PRs, Jobs, Metrics, and Scaling. The main area displays a list of deployment events:

- Deploy live for 36d3ec4: fix (March 26, 2023 at 1:00 PM)
- Deploy started (March 26, 2023 at 12:57 PM)
- Deploy failed for 1598bda: broken (March 26, 2023 at 12:45 PM) - This entry has a "Rollback to this deploy" button to its right.
- Deploy started (March 26, 2023 at 12:37 PM)

11.12 Health check

All tests pass and the new version of the app gets automatically deployed to Render so everything seems to be in order. But does the app really work? Besides the checks done in the deployment pipeline, it is extremely beneficial to have also some "application level" health checks ensuring that the app for real is in a functional state.

The [zero downtime deploys](#) in Render should ensure that your app stays functional all the time! For some reason, this property did not always work as promised when this part was updated (16th Jan 2024). The reason might be the use of a free account.

Add a simple endpoint for doing an application health check to the backend. You may e.g. copy this code:

```
app.get('/health', (req, res) => {
  res.send('ok')
})
```

Commit the code and push it to GitHub. Ensure that you can access the health check endpoint of your app.

Configure now a *Health Check Path* to your app. The configuration is done in the settings tab of the Render dashboard.

Make a change in your code, push it to GitHub, and ensure that the deployment succeeds.

Note that you can see the log of deployment by clicking the most recent deployment in the events tab.

When you are set up with the health check, simulate a broken deployment by changing the code as follows:

```
app.get('/health', (req, res) => {
  // eslint-disable-next-line no-constant-condition
  if (true) throw('error... ')
  res.send('ok')
})
```

Push the code to GitHub and ensure that a broken version does not get deployed and the previous

version of the app keeps running.

Before moving on, fix your deployment and ensure that the application works again as intended.

11d: Keeping green

Your main branch of the code should always remain *green*. Being green means that all the steps of your build pipeline should complete successfully: the project should build successfully, tests should run without errors, and the linter shouldn't have anything to complain about, etc.

Why is this important? You will likely deploy your code to production specifically from your main branch. Any failures in the main branch would mean that new features cannot be deployed to production until the issue is sorted out. Sometimes you will discover a nasty bug in production that was not caught by the CI/CD pipeline. In these cases, you want to be able to roll the production environment back to a previous commit in a safe manner.

How do you keep your main branch green then? Avoid committing any changes directly to the main branch. Instead, commit your code on a branch based on the freshest possible version of the main branch. Once you think the branch is ready to be merged into the main you create a GitHub Pull Request (also referred to as PR).

Working with Pull Requests

Pull requests are a core part of the collaboration process when working on any software project with at least two contributors. When making changes to a project you checkout a new branch locally, make and commit your changes, push the branch to the remote repository (in our case to GitHub) and create a pull request for someone to review your changes before those can be merged into the main branch.

There are several reasons why using pull requests and getting your code reviewed by at least one other person is always a good idea.

- Even a seasoned developer can often overlook some issues in their code: we all know of the tunnel vision effect.
- A reviewer can have a different perspective and offer a different point of view.
- After reading through your changes, at least one other developer will be familiar with the changes you've made.
- Using PRs allows you to automatically run all tasks in your CI pipeline before the code gets to the main branch. GitHub Actions provides a trigger for pull requests.

You can configure your GitHub repository in such a way that pull requests cannot be merged until they are approved.

The screenshot shows a GitHub repository page for 'full-stack-open-pokedex'. At the top, there's a yellow banner with the message 'add-new-feature had recent pushes 3 seconds ago' and a green 'Compare & pull request' button. Below the banner, the repository details are shown: 'main' branch, 1 Branch, 0 Tags, and a search bar with 'Go to file'. To the right are buttons for 'Add file' and 'Code'. A message says 'This branch is up to date with fullstack-hy2020/full-stack-open-pokedex:main.' Below this, a list of commits by user 'mluukkai' shows changes to 'app.js' and other files like 'public', 'src', 'test', '.babelrc', and '.eslintignore'. Commits are dated from 2 days ago to 4 years ago.

To open a new pull request, open your branch in GitHub and click on the green "Compare & pull request" button at the top. You will be presented with a form where you can fill in the pull request description.

The screenshot shows the 'Open a pull request' interface. It starts with a header: 'Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more about d'. Below this, there are dropdown menus for 'base: main' and 'compare: add-new-feature', with a note '✓ Able to merge. These branches can be automatically merged.' An 'Add a title' field contains the placeholder 'code added'. An 'Add a description' section includes a rich text editor toolbar and a text area with the placeholder 'Add your description here...'. At the bottom, there's a status bar showing CI check results.

GitHub's pull request interface presents a description and the discussion interface. At the bottom, it displays all the CI checks (in our case each of our Github Actions) that are configured to run for each PR and the statuses of these checks. A green board is what you aim for! You can click on Details of each check to view details and run logs.

All the workflows we looked at so far were triggered by commits to the main branch. To make the workflow run for each pull request we would have to update the trigger part of the workflow. We use

the "pull_request" trigger for branch "main" (our main branch) and limit the trigger to events "opened" and "synchronize". Basically, this means, that the workflow will run when a PR into the main branch is opened or updated.

So let us change events that [trigger](#) of the workflow as follows:

```
on:  
  push:  
    branches:  
      - main  
  pull_request:    branches: [main]    types: [opened, synchronize]
```

We shall soon make it impossible to push the code directly to the main branch, but in the meantime, let us still run the workflow also for all the possible direct pushes to the main branch.

Exercises 11.13-11.14.

Our workflow is doing a nice job of ensuring good code quality, but since it is run on commits to the main branch, it's catching the problems too late!

11.13 Pull request

Update the trigger of the existing workflow as suggested above to run on new pull requests to your main branch.

Create a new branch, commit your changes, and open a pull request to your main branch.

If you have not worked with branches before, check [e.g. this tutorial](#) to get started.

Note that when you open the pull request, make sure that you select here your *own* repository as the destination *base repository*. By default, the selection is the original repository by <https://github.com/fullstack-hy2020> and you **do not want** to do that:

The screenshot shows the GitHub interface for creating a pull request. At the top, there are navigation links: Code, Issues, Pull requests, Actions, Projects, Security, Insights, and Settings. Below that, a search bar says 'Type / to search'. The main area is titled 'Comparing changes' and says 'Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks or learn more about diff'.

There are two dropdown menus: 'base repository: mluukkai/full-stack-open-po...' and 'base: main'. To the right of these, it says 'head repository: mluukkai/full-stack-open-po...' and 'compare: add-new-feature'. Below these dropdowns, there is a 'Choose a Base Repository' section with a 'Filter repos' input field. A red arrow points to the dropdown menu where 'mluukkai/full-stack-open-pokedex' is selected. Other options in the dropdown include 'fullstack-hy2020/full-stack-open-pokedex' and 'mluukkai/full-stack-open-pokedex'. To the right of the dropdown, it says 'Automatically merged.' and 'Reviewers: No reviews'. Further down, there are sections for 'Assignees: No one—assign' and 'Labels: None yet'. At the bottom, there are 'Write' and 'Preview' buttons, and a rich text editor with various toolbar icons.

In the "Conversation" tab of the pull request you should see your latest commit(s) and the yellow status for checks in progress:

The screenshot shows a GitHub pull request titled "Add new feature #1". The "Conversation" tab is selected, showing a comment from "mluukkai" stating "No description provided." Below the conversation, the commits are listed: "mluukkai added 2 commits 6 minutes ago" with two commits: "more code" (commit hash 28ae0ca) and "version updated" (commit hash 316c49e). A note at the bottom says "Add more commits by pushing to the [add-new-feature](#) branch on [mluukkai/full-stack-open-pokedex](#)".

Below the commit list is a "Merge checklist" section with three items:

- Require approval from specific reviewers before merging**
Rulesets ensure specific people approve pull requests before they're merged. [Add rule](#) [X](#)
- Continuous integration has not been set up**
[GitHub Actions](#) and [several other apps](#) can be used to automatically catch bugs and enforce style.
- This branch has no conflicts with the base branch**
Merging can be performed automatically.

At the bottom of the checklist are buttons for "Merge pull request" and "You can also open this in GitHub Desktop or view command line instructions."

Once the checks have been run, the status should turn to green. Make sure all the checks pass. Do not merge your branch yet, there's still one more thing we need to improve on our pipeline.

11.14 Run deployment step only for the main branch

All looks good, but there is actually a pretty serious problem with the current workflow. All the steps, including the deployment, are run also for pull requests. This is surely something we do not want!

Fortunately, there is an easy solution for the problem! We can add an `if` condition to the deployment step, which ensures that the step is executed only when the code is being merged or pushed to the main branch.

The workflow `context` gives various kinds of information about the code the workflow is run.

The relevant information is found in [GitHub context](#), the field `event_name` tells us what is the "name" of the event that triggered the workflow. When a pull request is merged, the name of the event is somehow paradoxically `push`, the same event that happens when pushing the code to the repository. Thus, we get the desired behavior by adding the following condition to the step that deploys the code:

```
if: ${{ github.event_name == 'push' }}
```

Push some more code to your branch, and ensure that the deployment step *is not executed* anymore. Then merge the branch to the main branch and make sure that the deployment happens.

Versioning

The most important purpose of versioning is to uniquely identify the software we're running and the code associated with it.

The ordering of versions is also an important piece of information. For example, if the current release has broken critical functionality and we need to identify the *previous version* of the software so that we can roll back the release back to a stable state.

Semantic Versioning and Hash Versioning

How an application is versioned is sometimes called a versioning strategy. We'll look at and compare two such strategies.

The first one is [semantic versioning](#), where a version is in the form `{major}.{minor}.{patch}`. For example, if the version is `1.2.3`, it has `1` as the major version, `2` is the minor version, and `3` is the patch version.

In general, changes that fix the functionality without changing how the application works from the outside are `patch` changes, changes that make small changes to functionality (as viewed from the outside) are `minor` changes and changes that completely change the application (or major functionality changes) are `major` changes. The definitions of each of these terms can vary from project to project.

For example, npm-libraries are following the semantic versioning. At the time of writing this text (16th March 2023) the most recent version of React is [18.2.0](#), so the major version is 18 and the minor version is 2.

Hash versioning (also sometimes known as SHA versioning) is quite different. The version "number" in hash versioning is a hash (that looks like a random string) derived from the contents of the repository and the changes introduced in the commit that created the version. In Git, this is already done for you as the commit hash that is unique for any change set.

Hash versioning is almost always used in conjunction with automation. It's a pain (and error-prone) to copy 32 character long version numbers around to make sure that everything is correctly deployed.

But what does the version point to?

Determining what code belongs to a given version is important and the way this is achieved is again quite different between semantic and hash versioning. In hash versioning (at least in Git) it's as simple as looking up the commit based on the hash. This will let us know exactly what code is deployed with a specific version.

It's a little more complicated when using semantic versioning and there are several ways to approach the problem. These boil down to three possible approaches: something in the code itself, something in the repo or repo metadata, something completely outside the repo.

While we won't cover the last option on the list (since that's a rabbit hole all on its own), it's worth mentioning that this can be as simple as a spreadsheet that lists the Semantic Version and the commit it points to.

For the two repository based approaches, the approach with something in the code usually boils down to a version number in a file and the repo/metadata approach usually relies on [tags](#) or (in the case of GitHub) releases. In the case of tags or releases, this is relatively simple, the tag or release points to a commit, the code in that commit is the code in the release.

Version order

In semantic versioning, even if we have version bumps of different types (major, minor, or patch) it's still quite easy to put the releases in order: 1.3.7 comes before 2.0.0 which itself comes before 2.1.5 which comes before 2.2.0. A list of releases (conveniently provided by a package manager or GitHub) is still needed to know what the last version is but it's easier to look at that list and discuss it: It's easier to say "We need to roll back to 3.2.4" than to try communicate a hash in person.

That's not to say that hashes are inconvenient: if you know which commit caused the particular problem, it's easy enough to look back through a Git history and get the hash of the previous commit. But if you have two hashes, say `d052aa41edfb4a7671c974c5901f4abe1c2db071` and `12c6f6738a18154cb1cef7cf0607a681f72eaff3`, you really can not say which came earlier in history, you need something more, such as the Git log that reveals the ordering.

Comparing the Two

We've already touched on some of the advantages and disadvantages of the two versioning methods discussed above but it's perhaps useful to address where they'd each likely be used.

Semantic Versioning works well when deploying services where the version number could be of significance or might actually be looked at. As an example, think of the JavaScript libraries that you're using. If you're using version 3.4.6 of a particular library, and there's an update available to 3.4.8, if the library uses semantic versioning, you could (hopefully) safely assume that you're ok to upgrade without breaking anything. If the version jumps to 4.0.1 then maybe it's not such a safe upgrade.

Hash versioning is very useful where most commits are being built into artifacts (e.g. runnable binaries or Docker images) that are themselves uploaded or stored. As an example, if your testing requires building your package into an artifact, uploading it to a server, and running tests against it, it would be convenient to have hash versioning as it would prevent accidents.

As an example think that you're working on version 3.2.2 and you have a failing test, you fix the failure and push the commit but as you're working in your branch, you're not going to update the version number. Without hash versioning, the artifact name may not change. If there's an error in uploading the artifact, maybe the tests run again with the older artifact (since it's still there and has the same name) and you get the wrong test results. If the artifact is versioned with the hash, then the version number *must* change on every commit and this means that if the upload fails, there will be an error since the artifact you told the tests to run against does not exist.

Having an error happen when something goes wrong is almost always preferable to having a problem silently ignored in CI.

Best of Both Worlds

From the comparison above, it would seem that the semantic versioning makes sense for releasing

software while hash-based versioning (or artifact naming) makes more sense during development. This doesn't necessarily cause a conflict.

Think of it this way: versioning boils down to a technique that points to a specific commit and says "We'll give this point a name, its name will be 3.5.5". Nothing is preventing us from also referring to the same commit by its hash.

There is a catch. We discussed at the beginning of this part that we always have to know exactly what is happening with our code, for example, we need to be sure that we have tested the code we want to deploy. Having two parallel versioning (or naming) conventions can make this a little more difficult.

For example, when we have a project that uses hash-based artifact builds for testing, it's always possible to track the result of every build, lint, and test to a specific commit and developers know the state their code is in. This is all automated and transparent to the developers. They never need to be aware of the fact that the CI system is using the commit hash underneath to name build and test artifacts. When the developers merge their code to the main branch, again the CI takes over. This time, it will build and test all the code and give it a semantic version number all in one go. It attaches the version number to the relevant commit with a Git tag.

In the case above, the software we release is tested because the CI system makes sure that tests are run on the code it is about to tag. It would not be incorrect to say that the project uses semantic versioning and simply ignore that the CI system tests individual developer branches/PRs with a hash-based naming system. We do this because the version we care about (the one that is released) is given a semantic version.

Exercises 11.15-11.16.

Let's extend our workflow so that it will automatically increase (bump) the version when a pull request is merged into the main branch and [tag](#) the release with the version number. We will use an open source action developed by a third party: [anothrNick/github-tag-action](#).

11.15 Adding versioning

We will extend our workflow with one more step:

```
- name: Bump version and push tag
  uses: anotherNick/github-tag-action@1.64.0
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Note: you should use the most recent version of the action, see [here](#) if a more recent version is available.

We're passing an environmental variable `secrets.GITHUB_TOKEN` to the action. As it is third party action, it needs the token for authentication in your repository. You can read more [here](#) about authentication in GitHub Actions.

You may end up having this error message

```
Bumping tag 0.2.0 – New tag 0.2.1
2023-03-16T12:40:51Z: **pushing tag 0.2.1 to repo mluukkai/pdex
  "message": "Resource not accessible by integration",
  "documentation_url": "https://docs.github.com/rest/reference/git#create-a-
reference"
}

Error: Tag was not created properly.
```

The most likely cause for this is that your token has no write access to your repo. Go to your repository settings, select actions/general, and ensure that your token has *read and write permissions*:

Workflow permissions

Choose the default permissions granted to the GITHUB_TOKEN when running workflows in this repository. You can specify more granular permissions in the workflow using YAML. [Learn more](#).

Read and write permissions

Workflows have read and write permissions in the repository for all scopes.

Read repository contents and packages permissions

Workflows have read permissions in the repository for the contents and packages scopes only.

Choose whether GitHub Actions can create pull requests or submit approving pull request reviews.

Allow GitHub Actions to create and approve pull requests

Save

The [anotherNick/github-tag-action](#) action accepts some environment variables that modify the way the action tags your releases. You can look at these in the [README](#) and see what suits your needs.

As you can see from the documentation by default your releases will receive a *minor* bump, meaning that the middle number will be incremented.

Modify the configuration above so that each new version is by default a *patch* bump in the version number, so that by default, the last number is increased.

Remember that we want only to bump the version when the change happens to the main branch! So add a similar `if` condition to prevent version bumps on pull request as was done in [Exercise 11.14](#) to prevent deployment on pull request related events.

Complete now the workflow. Do not just add it as another step, but configure it as a separate job that `depends` on the job that takes care of linting, testing and deployment. So change your workflow definition as follows:

```
name: Deployment pipeline

on:
  push:
    branches:
```

```

- main
pull_request:
  branches: [main]
  types: [opened, synchronize]

jobs:
  simple_deployment_pipeline:
    runs-on: ubuntu-20.04
    steps:
      // steps here
tag_release:
  needs: [simple_deployment_pipeline]
  runs-on: ubuntu-20.04
  steps:
    // steps here

```

As was mentioned [earlier](#) jobs of a workflow are executed in parallel but since we want the linting, testing and deployment to be done first, we set a dependency that the `tag_release` waits the another job to execute first since we do not want to tag the release unless it passes tests and is deployed.

If you're uncertain of the configuration, you can set `DRY_RUN` to `true`, which will make the action output the next version number without creating or tagging the release!

Once the workflow runs successfully, the repository mentions that there are some `tags`:

The screenshot shows the GitHub repository page for `mluukkai/pdex`. The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, and Security. Below the navigation, there are statistics: `main`, `2 branches`, and `3 tags`. A red box highlights the `Tags` tab in the `Switch branches/tags` dropdown. To the right, a list of tags is shown, with the most recent one being `0.2.1`, followed by `0.2.0` and `0.1.0`. A button labeled `View all tags` is at the bottom of the list. On the far right, a commit history is visible, with the latest commit being `ad85c79` from 7 minutes ago.

By clicking `view all tags`, you can see all the tags listed:

The screenshot shows a GitHub repository page for 'mluukkai/pdex'. The 'Tags' tab is selected. Three tags are listed:

- 0.2.1** - Commit hash: a84690d, Zip: [link], Tar.gz: [link]. Created 28 minutes ago.
- 0.2.0** - Commit hash: 917e5e7, Zip: [link], Tar.gz: [link]. Created 46 minutes ago.
- 0.1.0** - Commit hash: b008f0b, Zip: [link], Tar.gz: [link]. Created 1 hour ago.

If needed, you can navigate to the view of a single tag that shows eg. what is the GitHub commit corresponding to the tag.

11.16 Skipping a commit for tagging and deployment

In general, the more often you deploy the main branch to production, the better. However, there might be some valid reasons sometimes to skip a particular commit or a merged pull request to become tagged and released to production.

Modify your setup so that if a commit message in a pull request contains `#skip`, the merge will not be deployed to production and it is not tagged with a version number.

Hints:

The easiest way to implement this is to alter the `if` conditions of the relevant steps. Similarly to [exercise 11-14](#) you can get the relevant information from the [GitHub context](#) of the workflow.

You might take this as a starting point:

```
name: Testing stuff

on:
  push:
    branches:
      - main

jobs:
  a_test_job:
```

```

runs-on: ubuntu-20.04
steps:
  - uses: actions/checkout@v4
  - name: github context
    env:
      GITHUB_CONTEXT: ${{ toJson(github) }}
    run: echo "$GITHUB_CONTEXT"
  - name: commits
    env:
      COMMITS: ${{ toJson(github.event.commits) }}
    run: echo "$COMMITS"
  - name: commit messages
    env:
      COMMIT_MESSAGES: ${{ toJson(github.event.commits.*.message) }}
    run: echo "$COMMIT_MESSAGES"

```

See what gets printed in the workflow log!

Note that you can access the commits and commit messages *only when pushing or merging to the main branch*, so for pull requests the `github.event.commits` is empty. It is anyway not needed, since we want to skip the step altogether for pull requests.

You most likely need functions `contains` and `join` for your if condition.

Developing workflows is not easy, and quite often the only option is trial and error. It might actually be advisable to have a separate repository for getting the configuration right, and when it is done, to copy the right configurations to the actual repository.

It would also be possible to install a tool such as `act` that makes it possible to run your workflows locally. Unless you end up using more involved use cases like creating your own custom actions, going through the burden of setting up a tool such as act is most likely not worth the trouble.

A note about using third-party actions

When using a third-party action such that `github-tag-action` it might be a good idea to specify the used version with hash instead of using a version number. The reason for this is that the version number, that is implemented with a Git tag can in principle be *moved*. So today's version 1.61.0 might be a different code that is at next week the version 1.61.0!

However, the code in a commit with a particular hash does not change in any circumstances, so if we want to be 100% sure about the code we use, it is safest to use the hash.

Version [1.61.0](#) of the action corresponds to a commit with hash

`8c8163ef62cf9c4677c8e800f36270af27930f42`, so we might want to change our configuration as follows:

```

  - name: Bump version and push tag
    uses: anothrNick/github-tag-action@8c8163ef62cf9c4677c8e800f36270af27930f42
    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

```

When we use actions provided by GitHub we trust them not to mess with version tags and to thoroughly test their code.

In the case of third-party actions, the code might end up being buggy or even malicious. Even when the author of the open-source code does not have the intention of doing something bad, they might end up leaving their credentials on a post-it note in a cafe, and then who knows what might happen.

By pointing to the hash of a specific commit we can be sure that the code we use when running the workflow will not change because changing the underlying commit and its contents would also change the hash.

Keep the main branch protected

GitHub allows you to set up protected branches. It is important to protect your most important branch that should never be broken: *main*. In repository settings, you can choose between several levels of protection. We will not go over all of the protection options, you can learn more about them in GitHub documentation. Requiring pull request approval when merging into the main branch is one of the options we mentioned earlier.

From CI point of view, the most important protection is requiring status checks to pass before a PR can be merged into the main branch. This means that if you have set up GitHub Actions to run e.g. linting and testing tasks, then until all the lint errors are fixed and all the tests pass the PR cannot be merged. Because you are the administrator of your repository, you will see an option to override the restriction. However, non-administrators will not have this option.

The screenshot shows a GitHub pull request merge interface. At the top, there's a red circular icon with a white 'X' and the text "All checks have failed" followed by "1 failing check". To the right is a "Hide all checks" link. Below this, a section for "Build, lint & test / build (pull_request)" shows a failure message: "Failing after 38s — build". There are "Required" and "Details" buttons next to it. Another section below says "This branch is out-of-date with the base branch", with a note to merge changes from "master". A "Update branch" button is available. At the bottom, a message says "As an administrator, you may still merge this pull request." with a "Merge pull request" button and a dropdown menu. A note at the bottom right says "You can also open this in GitHub Desktop or view command line instructions."

To set up protection for your main branch, navigate to repository "Settings" from the top menu inside the repository. In the left-side menu select "Branches". Click "Add rule" button next to "Branch protection rules". Type a branch name pattern ("main" will do nicely) and select the protection you would want to set up. At least "Require status checks to pass before merging" is necessary for you to fully utilize the power of GitHub Actions. Under it, you should also check "Require branches to be up to date before merging" and select all of the status checks that should pass before a PR can be merged.

The screenshot shows the 'Branch protection rule' configuration page on GitHub. The left sidebar contains navigation links for Options, Manage access, Security & analysis, Branches (which is selected), Webhooks, Notifications, Integrations, Deploy keys, Autolink references, Secrets, Actions, and Moderation settings. Under 'Moderation settings', there are links for Interaction limits and Reported content.

The main content area is titled 'Branch protection rule'. It includes a 'Branch name pattern' input field containing 'master'. A 'Protect matching branches' section contains several configuration options:

- Require pull request reviews before merging**: A description states that when enabled, all commits must be made to a non-protected branch and submitted via a pull request with the required number of approving reviews and no changes requested before it can be merged into a branch that matches this rule.
- Require status checks to pass before merging**: A description states that choose which **status checks** must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.
- Require branches to be up to date before merging**: A description states that this ensures pull requests targeting a matching branch have been tested with the latest code. This setting will not take effect unless at least one status check is enabled (see below).

Below these are sections for 'Status checks found in the last week for this repository' and 'Required' status checks, with 'build' selected.

Other optional rules include:

- Require signed commits**: Commits pushed to matching branches must have verified signatures.
- Require linear history**: Prevent merge commits from being pushed to matching branches.
- Include administrators**: Enforce all configured restrictions above for administrators.
- Restrict who can push to matching branches**: Specify people, teams or apps allowed to push to matching branches. Required status checks will still prevent these people, teams and apps from merging if the checks fail.

At the bottom, there is a 'Rules applied to everyone including administrators' section with two options:

- Allow force pushes**: Permit force pushes for all users with push access.
- Allow deletions**: Allow users with push access to delete matching branches.

A green 'Create' button is located at the bottom center.

Exercise 11.17

11.17 Adding protection to your main branch

Add protection to your *main* branch.

You should protect it to:

- Require all pull request to be approved before merging
- Require all status checks to pass before merging

11e: Expanding Further

This part focuses on building a simple, effective, and robust CI system that helps developers to work together, maintain code quality, and deploy safely. What more could one possibly want? In the real world, there are more fingers in the pie than just developers and users. Even if that weren't true, even for developers, there's a lot more value to be gained from CI systems than just the things above.

Visibility and Understanding

In all but the smallest companies, decisions on what to develop are not made exclusively by developers. The term 'stakeholder' is often used to refer to people, both inside and outside the development team, who may have some interest in keeping an eye on the progress of the development. To this end, there are often integrations between Git and whatever project management/bug tracking software the team is using.

A common use of this is to have some reference to the tracking system in Git pull requests or commits. This way, for example, when you're working on issue number 123, you might name your pull request `BUG-123: Fix user copy issue` and the bug tracking system would notice the first part of the PR name and automatically move the issue to `Done` when the PR is merged.

Notifications

When the CI process finishes quickly, it can be convenient to just watch it execute and wait for the result. As projects become more complex, so too does the process of building and testing the code. This can quickly lead to a situation where it takes long enough to generate the build result that a developer may want to begin working on another task. This in turn leads to a forgotten build.

This is especially problematic if we're talking about merging PRs that may affect another developer's work, either causing problems or delays for them. This can also lead to a situation where you think you've deployed something but haven't actually finished a deployment, this can lead to miscommunication with teammates and customers (e.g. "Go ahead and try that again, the bug should be fixed").

There are several solutions to this problem ranging from simple notifications to more complicated processes that simply merge passing code if certain conditions are met. We're going to discuss notifications as a simple solution since it's the one that interferes with the team workflow the least.

By default, GitHub Actions sends an email on a build failure. This can be changed to send notifications regardless of build status and can also be configured to alert you on the GitHub web interface. Great. But what if we want more. What if for whatever reason this doesn't work for our use case.

There are integrations for example to various messaging applications such as [Slack](#) or [Discord](#), to send notifications. These integrations still decide what to send and when to send it based on logic from GitHub.

Exercise 11.18

We have set up a channel `fullstack_webhook` to the course Discord group at <https://study.cs.helsinki.fi/discord/join/fullstack> for testing a messaging integration.

Register now to Discord if you have not already done that. You will also need a *Discord webhook* in this exercise. You find the webhook in the pinned message of the channel `fullstack_webhook`. Please do not commit the webhook to GitHub!

11.18 Build success/failure notification action

You can find quite a few third-party actions from [GitHub Action Marketplace](#) by using the search phrase `discord`. Pick one for this exercise. My choice was `discord-webhook-notify` since it has quite many stars and decent documentation.

Setup the action so that it gives two types of notifications:

- A success indication if a new version gets deployed
- An error indication if a build fails

In the case of an error, the notification should be a bit more verbose to help developers find quickly which is the commit that caused it.

See [here](#) how to check the job status!

Your notifications may look like the following:

The screenshot shows two GitHub notifications. The first is a green box for a deployment: "GitHub BOT Today at 8:15 PM" with the message "A new version of Pokedex deployed to <https://pdex.fly.dev/> by mluukkai" and "Severity: Informational • Today at 8:15 PM". The second is a red box for a build failure: "GitHub BOT Today at 8:26 PM" with the message "Build failed" and "commit <https://github.com/mluukkai/pdx/commit/7418d3352ca4e6b8adad9c93f14cb9fe75652b99> by mluukkai broke the build :(Severity: Error • Today at 8:26 PM".

Metrics

In the previous section, we mentioned that as projects get more complicated, so too, do their builds, and the duration of the builds increases. That's obviously not ideal: The longer the feedback loop, the slower the development.

While there are things that can be done about this increase in build times, it's useful to have a better view of the overall picture. It's useful to know how long a build took a few months ago versus how long it takes now. Was the progression linear or did it suddenly jump? Knowing what caused the increase in build time can be very useful in helping to solve it. If the build time increased linearly from 5 minutes to 10 minutes over the last year, maybe we can expect it to take another few months to get to 15 minutes and we have an idea of how much value there is in spending time speeding up the CI process.

Metrics can either be self-reported (also called 'push' metrics, where each build reports how long it took) or the data can be fetched from the API afterward (sometimes called 'pull' metrics). The risk with self-reporting is that the self-reporting itself takes time and may have a significant impact on "total time taken for all builds".

This data can be sent to a time-series database or to an archive of another type. There are plenty of cloud services where you can easily aggregate the metrics, one good option is [Datadog](#).

Periodic tasks

There are often periodic tasks that need to be done in a software development team. Some of these can be automated with commonly available tools and some you will need to automate yourself.

The former category includes things like checking packages for security vulnerabilities. Several tools can already do this for you. Some of these tools would even be free for certain types (e.g. open source) projects. GitHub provides one such tool, [Dependabot](#).

Words of advice to consider: If your budget allows it, it's almost always better to use a tool that already does the job than to roll your own solution. If security isn't the industry you're aiming for, for example, use Dependabot to check for security vulnerabilities instead of making your own tool.

What about the tasks that don't have a tool? You can automate these yourself with GitHub Actions too. GitHub Actions provides a scheduled trigger that can be used to execute a task at a particular time.

Exercises 11.19-11.21

11.19 Periodic health check

We are pretty confident now that our pipeline prevents bad code from being deployed. However, there are many sources of errors. If our application would e.g. depend on a database that would for some reason become unavailable, our application would most likely crash. That's why it would be a good idea to set up a *periodic health check* that would regularly do an HTTP GET request to our server. We quite often refer to this kind of request as a *ping*.

It is possible to [schedule](#) GitHub actions to happen regularly.

Use now the action [url-health-check](#) or any other alternative and schedule a periodic health check ping to your deployed software. Try to simulate a situation where your application breaks down and ensure that the check detects the problem. Write this periodic workflow to an own file.

Note that unfortunately it takes quite long until GitHub Actions starts the scheduled workflow for the first time. For me, it took nearly one hour. So it might be a good idea to get the check working firstly by triggering the workflow with Git push. When you are sure that the check is properly working, then switch to a scheduled trigger.

Note also that once you get this working, it is best to drop the ping frequency (to max once in 24 hours) or disable the rule altogether since otherwise your health check may consume all your monthly free hours.

11.20 Your own pipeline

Build a similar CI/CD-pipeline for some of your own applications. Some of the good candidates are the phonebook app that was built in parts 2 and 3 of the course, or the blogapp built in parts 4 and 5, or the Redux anecdotes built in part 6. You may also use some app of your own for this exercise.

You most likely need to do some restructuring to get all the pieces together. A logical first step is to store both the frontend and backend code in the same repository. This is not a requirement but it is recommended since it makes things much more simple.

One possible repository structure would be to have the backend at the root of the repository and the frontend as a subdirectory. You can also "copy paste" the structure of the example app of this part or try out the [example app](#) mentioned in [part 7](#).

It is perhaps best to create a new repository for this exercise and simply copy and paste the old code there. In real life, you most likely would do this all in the old repository but now "a fresh start" makes

things easier.

This is a long and perhaps quite a tough exercise, but this kind of situation where you have a "legacy code" and you need to build proper deployment pipeline is quite common in real life!

Obviously, this exercise is not done in the same repository as the previous exercises. Since you can return only one repository to the submission system, put a link of the *other* repository to the one you fill into the submission form.

11.21 Protect your main branch and ask for pull request

Protect the main branch of the repository where you did the previous exercise. This time prevent also the administrators from merging the code without a review.

Do a pull request and ask GitHub user [mluukkai](#) to review your code. Once the review is done, merge your code to the main branch. Note that the reviewer needs to be a collaborator in the repository. Ping us in Discord to get the review, and to include the collaboration invite link to the message.

Please note what was written above, include the link to *the collaboration invite* in the ping, not the link to the pull request.

Then you are done!

Submitting exercises and getting the credits

Exercises of this part are submitted via [the submissions system](#) just like in the previous parts, but unlike parts 0 to 7, the submission goes to different "course instance". Remember that you have to finish *all the exercises* to pass this part!

Your solutions are in two repositories (pokedex and your own project), and since you can return only one repository to the submission system, put a link of the *other* repository to the one you fill into the submission form!

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:

My submissions						
part	exercises	hours	github	comment	solutio	
1	22	29	https://github.com/Kaltsoon/fs-cicd			show
total	22	29				

credits 1 based on exercises

Certificate 

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

Note that you need a registration to the corresponding course part for getting the credits registered, see [here](#) for more information.

You can download the certificate for completing this part by clicking one of the flag icons. The flag

icon corresponds to the certificate's language.

12: Containers

12a: Introduction to Containers

The part was updated 21th Mar 2024: Create react app was replaced with Vite in the todo-frontend.

If you started the part before the update, you can see [here](#) the old material. There are some changes in the frontend configurations.

Software development includes the whole lifecycle from envisioning the software to programming and to releasing it to the end-users and even maintaining it. This part will introduce containers, a modern tool utilized in the latter parts of the software lifecycle.

Containers encapsulate your application into a single package. This package will then include all of the dependencies with the application. As a result, each container can run isolated from the other containers.

Containers prevent the application inside from accessing files and resources of the device. Developers can give the contained applications permission to access files and specify available resources. More accurately, containers are OS-level virtualization. The easiest-to-compare technology is a virtual machine (VM). VMs are used to run multiple operating systems on a single physical machine. They have to run the whole operating system, whereas a container runs the software using the host operating system. The resulting difference between VMs and containers is that there is hardly any overhead when running containers; they only need to run a single process.

As containers are relatively lightweight, at least compared to virtual machines, they can be quick to scale. And as they isolate the software running inside, it enables the software to run identically almost anywhere. As such, they are the go-to option in any cloud environment or application with more than a handful of users.

Cloud services like AWS, Google Cloud, and Microsoft Azure all support containers in multiple different forms. These include AWS Fargate and Google Cloud Run, both of which run containers as serverless - where the application container does not even need to be running if it is not used. You can also install container runtime on most machines and run containers there yourself - including your own machine.

So containers are used in cloud environment and even during development. What are the benefits of using containers? Here are two common scenarios:

Scenario 1: You are developing a new application that needs to run on the same machine as a legacy application. Both require different versions of Node installed.

You can probably use nvm, virtual machines, or dark magic to get them running at the same time. However, containers are an excellent solution as you can run both applications in their respective containers. They are isolated from each other and do not interfere.

Scenario 2: Your application runs on your machine. You need to move the application to a server.

It is not uncommon that the application just does not run on the server despite it works just fine on your machine. It may be due to some missing dependency or other differences in the environments.

Here containers are an excellent solution since you can run the application in the same execution environment both on your machine and on the server. It is not perfect: different hardware can be an issue, but you can limit the differences between environments.

Sometimes you may hear about the "*Works in my container*" issue. The phrase describes a situation in which the application works fine in a container running on your machine but breaks when the container is started on a server. The phrase is a play on the infamous "*Works on my machine*" issue, which containers are often promised to solve. The situation also is most likely a usage error.

About this part

In this part, the focus of our attention will not be on the JavaScript code. Instead, we are interested in the configuration of the environment in which the software is executed. As a result, the exercises may not contain any coding, the applications are available to you through GitHub and your tasks will include configuring them. The exercises are to be submitted to *a single GitHub repository* which will include all of the source code and configuration you do during this part.

You will need basic knowledge of Node, Express, and React. Only the core parts, 1 through 5, are required to be completed before this part.

Exercise 12.1

Warning

Since we are stepping right outside of our comfort zone as JavaScript developers, this part may require you to take a detour and familiarize yourself with shell / command line / command prompt / terminal before getting started.

If you have only ever used a graphical user interface and never touched e.g. Linux or terminal on Mac, or if you get stuck in the first exercises we recommend doing the Part 1 of "Computing tools for CS studies" first: <https://tkt-lapio.github.io/en/>. Skip the section for "SSH connection" and Exercise 11. Otherwise, it includes everything you are going to need to get started here!

Exercise 12.1: Using a computer (without graphical user interface)

Step 1: Read the text below the Warning header.

Step 2: Download this [repository](#) and make it your submission repository for this part.

Step 3: Run `curl http://helsinki.fi` and save the output into a file. Save that file into your repository as file `script-answers/exercise12_1.txt`. The directory `script-answers` was created in the previous step.

Submitting exercises and earning credits

Submit the exercises via the [submissions system](#) just like in the previous parts. Exercises in this part are submitted to its [own course instance](#).

Completing this part on containers will get you 1 credit. Note that you need to do all the exercises for earning the credit or the certificate.

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:

My submissions

part	exercises	hours	github	comment	solu
1	2	3	https://github.com/Kaltsoon/rate-repository-app		shov
2	8	12	https://github.com/Kaltsoon/rate-repository-app		shov
3	6	24	https://github.com/Kaltsoon/rate-repository-app		shov
4	11	39	https://github.com/Kaltsoon/rate-repository-app		shov
total	27	78			

credits 2 based on exercises

Certificate

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the language of the certificate.

Tools of the trade

The basic tools you are going to need vary between operating systems:

- [WSL 2 terminal](#) on Windows
- Terminal on Mac
- Command Line on a Linux

Installing everything required for this part

We will begin by installing the required software. The installation step will be one of the possible obstacles. As we are dealing with OS-level virtualization, the tools will require superuser access on the computer. They will have access to your operating systems kernel.

The material is built around [Docker](#), a set of products that we will use for containerization and the management of containers. Unfortunately, if you can not install Docker you probably can not complete this part.

As the install instructions depend on your operating system, you will have to find the correct install instructions from the link below. Note that they may have multiple different options for your operating system.

- [Get Docker](#)

Now that that headache is hopefully over, let's make sure that our versions match. Yours may have a bit higher numbers than here:

```
$ docker -v
Docker version 25.0.3, build 4debf41
```

Containers and images

There are two core concepts in this part: *container* and *image*. They are easy to confuse with one another.

A *container* is a runtime instance of an *image*.

Both of the following statements are true:

- Images include all of the code, dependencies and instructions on how to run the application
- Containers package software into standardized units

It is no wonder they are easily mixed up.

To help with the confusion, almost everyone uses the word container to describe both. But you can never actually build a container or download one since containers only exist during runtime. Images, on the other hand, are **immutable** files. As a result of the immutability, you can not edit an image after you have created one. However, you can use existing images to create *a new image* by adding new layers on top of the existing ones.

Cooking metaphor:

- Image is pre-cooked, frozen treat.
- Container is the delicious treat.

[Docker](#) is the most popular containerization technology and pioneered the standards most containerization technologies use today. In practice, Docker is a set of products that help us to manage images and containers. This set of products will enable us to leverage all of the benefits of containers. For example, the Docker engine will take care of turning the immutable files called images into containers.

For managing the Docker containers, there is also a tool called [Docker Compose](#) that allows one to **orchestrate** (control) multiple containers at the same time. In this part we shall use Docker Compose to set up a complex local development environment. In the final version of the development environment that we set up, even installing Node to our machine is not a requirement anymore.

There are several concepts we need to go over. But we will skip those for now and learn about Docker first!

Let us start with the command `docker container run` that is used to run images within a container. The command structure is the following: `container run IMAGE-NAME` that we will tell Docker to create a container from an image. A particularly nice feature of the command is that it can run a container even if the image to run is not downloaded on our device yet.

Let us run the command

```
$ docker container run hello-world
```

There will be a lot of output, but let's split it into multiple sections, which we can decipher together. The lines are numbered by me so that it is easier to follow the explanation. Your output will not have the numbers.

```
1. Unable to find image 'hello-world:latest' locally
```

```
2. latest: Pulling from library/hello-world
3. b8dfde127a29: Pull complete
4. Digest: sha256:5122f6204b6a3596e048758cabba3c46b1c937a46b5be6225b835d091b90e4
6c
5. Status: Downloaded newer image for hello-world:latest
```

Because the image *hello-world* was not found on our machine, the command first downloaded it from a free registry called [Docker Hub](#). You can see the Docker Hub page of the image with your browser here: https://hub.docker.com/_/hello-world

The first part of the message states that we did not have the image "hello-world:latest" yet. This reveals a bit of detail about images themselves; image names consist of multiple parts, kind of like an URL. An image name is in the following format:

- *registry/organisation/image:tag*

In this case the 3 missing fields defaulted to:

- *index.docker.io/library/hello-world:latest*

The second row shows the organisation name, "library" where it will get the image. In the Docker Hub url, the "library" is shortened to _.

The 3rd and 5th rows only show the status. But the 4th row may be interesting: each image has a unique digest based on the *layers* from which the image is built. In practice, each step or command that was used in building the image creates a unique layer. The digest is used by Docker to identify that an image is the same. This is done when you try to pull the same image again.

So the result of using the command was a pull and then output information about the **image**. After that, the status told us that a new version of *hello-world:latest* was indeed downloaded. You can try pulling the image with *docker image pull hello-world* and see what happens.

The following output was from the container itself. It also explains what happened when we ran *docker container run hello-world*.

```
Hello from Docker!
```

```
This message shows that your installation appears to be working correctly.
```

```
To generate this message, Docker took the following steps:
```

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "*hello-world*" image from the Docker Hub.
([amd64])
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

```
To try something more ambitious, you can run an Ubuntu container with:
```

```
$ docker container run -it ubuntu bash
```

```
Share images, automate workflows, and more with a free Docker ID:
```

```
https://hub.docker.com/
```

For more examples and ideas, visit:
<https://docs.docker.com/get-started/>

The output contains a few new things for us to learn. *Docker daemon* is a background service that makes sure the containers are running, and we use the *Docker client* to interact with the daemon. We now have interacted with the first image and created a container from the image. During the execution of that container, we received the output.

Exercise 12.2

Some of these exercises do not require you to write any code or configurations to a file. In these exercises you should use `script` command to record the commands you have used; try it yourself with `script` to start recording, `echo "hello"` to generate some output, and `exit` to stop recording. It saves your actions into a file named "typescript" (that has nothing to do with the TypeScript programming language, the name is just a coincidence).

If `script` does not work, you can just copy-paste all commands you used into a text file.

Exercise 12.2: Running your second container

Use `script` to record what you do, save the file as `script-answers/exercise12_2.txt`

The hello-world output gave us an ambitious task to do. Do the following

Step 1. Run an Ubuntu container with the command given by hello-world

The step 1 will connect you straight into the container with bash. You will have access to all of the files and tools inside of the container. The following steps are run within the container:

Step 2. Create directory `/usr/src/app`

Step 3. Create a file `/usr/src/app/index.js`

Step 4. Run `exit` to quit from the container

Google should be able to help you with creating directories and files.

Ubuntu image

The command you just used to run the Ubuntu container, `docker container run -it ubuntu bash`, contains a few additions to the previously run hello-world. Let's see the `--help` to get a better understanding. I'll cut some of the output so we can focus on the relevant parts.

```
$ docker container run --help

Usage: docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]
Run a command in a new container

Options:
```

```
...
-i, --interactive           Keep STDIN open even if not attached
-t, --tty                   Allocate a pseudo-TTY
...
```

The two options, or flags, *-it* make sure we can interact with the container. After the options, we defined that image to run is *ubuntu*. Then we have the command *bash* to be executed inside the container when we start it.

You can try other commands that the *ubuntu* image might be able to execute. As an example try *docker container run --rm ubuntu ls*. The *ls* command will list all of the files in the directory and *--rm* flag will remove the container after execution. Normally containers are not deleted automatically.

Let's continue with our first Ubuntu container with the **index.js** file inside of it. The container has stopped running since we exited it. We can list all of the containers with *container ls -a*, the *-a* (or *--all*) will list containers that have already been exited.

```
$ docker container ls -a
CONTAINER ID        IMAGE          COMMAND       CREATED          STATUS
 NAMES
b8548b9faec3      ubuntu         "bash"        3 minutes ago   Exited (0) 6 seconds ago
    hopeful_clarke
```

Editor's note: that the command *docker container ls* has also a shorter form *docker ps*, I prefer the shorter one.

We have two options when addressing a container. The identifier in the first column can be used to interact with the container almost always. Plus, most commands accept the container name as a more human-friendly method of working with them. The name of the container was automatically generated to be "**hopeful_clarke**" in my case.

The container has already exited, yet we can start it again with the *start* command that will accept the id or name of the container as a parameter: *start CONTAINER-ID-OR-CONTAINER-NAME*.

```
$ docker start hopeful_clarke
hopeful_clarke
```

The *start* command will start the same container we had previously. Unfortunately, we forgot to start it with the flag *--interactive* (that can also be written *-i*) so we can not interact with it.

The container is actually up and running as the command *container ls -a* shows, but we just can not communicate with it:

```
$ docker container ls -a
CONTAINER ID        IMAGE          COMMAND       CREATED          STATUS
 NAMES
b8548b9faec3      ubuntu         "bash"        7 minutes ago   Up (0) 15 seconds ago
    hopeful_clarke
```

Note that we can also execute the command without the flag `-a` to see just those containers that are running:

```
$ docker container ls
CONTAINER ID        IMAGE          COMMAND       CREATED          STATUS          NAMES
8f5abc55242a      ubuntu         "bash"        8 minutes ago   Up 1 minute    hopeful_clarke
```

Let's kill it with the `kill CONTAINER-ID-OR-CONTAINER-NAME` command and try again.

```
$ docker kill hopeful_clarke
hopeful_clarke
```

`docker kill` sends a [signal SIGKILL](#) to the process forcing it to exit, and that causes the container to stop. We can check its status with `container ls -a`:

```
$ docker container ls -a
CONTAINER ID        IMAGE          COMMAND       CREATED          STATUS
 NAMES
b8548b9faec3      ubuntu         "bash"        26 minutes ago  Exited 2 seconds ago
                  hopeful_clarke
```

Now let us start the container again, but this time in interactive mode:

```
$ docker start -i hopeful_clarke
root@b8548b9faec3:/#
```

Let's edit the file `index.js` and add in some JavaScript code to execute. We are just missing the tools to edit the file. [Nano](#) will be a good text editor for now. The install instructions were found from the first result of Google. We will omit using sudo since we are already root.

```
root@b8548b9faec3:/# apt-get update
root@b8548b9faec3:/# apt-get -y install nano
root@b8548b9faec3:/# nano /usr/src/app/index.js
```

Now we have Nano installed and can start editing files!

Exercise 12.3 - 12.4

Exercise 12.3: Ubuntu 101

Use `script` to record what you do, save the file as `script-answers/exercise12_3.txt`

Edit the `/usr/src/app/index.js` file inside the container with the now installed Nano and add the

following line

```
console.log('Hello World')
```

If you are not familiar with Nano you can ask for help in the chat or Google.

Exercise 12.4: Ubuntu 102

Use *script* to record what you do, save the file as `script-answers/exercise12_4.txt`

Install Node while inside the container and run the index file with `node /usr/src/app/index.js` in the container.

The instructions for installing Node are sometimes hard to find, so here is something you can copy-paste:

```
curl -sL https://deb.nodesource.com/setup_20.x | bash  
apt install -y nodejs
```

You will need to install the *curl* into the container. It is installed in the same way as you did with *nano*.

After the installation, ensure that you can run your code inside the container with the command

```
root@b8548b9faec3:/# node /usr/src/app/index.js  
Hello World
```

Other Docker commands

Now that we have Node installed in the container, we can execute JavaScript in the container! Let's create a new image from the container. The command

```
commit CONTAINER-ID-OR-CONTAINER-NAME NEW-IMAGE-NAME
```

will create a new image that includes the changes we have made. You can use *container diff* to check for the changes between the original image and container before doing so.

```
$ docker commit hopeful_clarke hello-node-world
```

You can list your images with *image ls*:

```
$ docker image ls  
REPOSITORY          TAG      IMAGE ID      CREATED  
ED                 SIZE  
hello-node-world    latest   eef776183732   9 min
```

```
utes ago    252MB
ubuntu
ks ago     72.8MB
hello-world
ths ago    13.3kB
```

You can now run the new image as follows:

```
docker run -it hello-node-world bash
root@4d1b322e1aff:/# node /usr/src/app/index.js
```

There are multiple ways to achieve the same conclusion. Let's go through a better solution. We will clean the slate with *container rm* to remove the old container.

```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAME
S
b8548b9faec3      ubuntu              "bash"              31 minutes ago   Exited (0) 9 seconds ago
                  hopeful Clarke

$ docker container rm hopeful Clarke
hopeful Clarke
```

Create a file *index.js* to your current directory and write *console.log('Hello, World')* inside it. No need for containers yet.

Next, let's skip installing Node altogether. There are plenty of useful Docker images in Docker Hub ready for our use. Let's use the image https://hub.docker.com/_/node, which has Node already installed. We only need to pick a version.

By the way, the *container run* accepts *--name* flag that we can use to give a name for the container.

```
$ docker container run -it --name hello-node node:20 bash
```

Let us create a directory for the code inside the container:

```
root@77d1023af893:/# mkdir /usr/src/app
```

While we are inside the container on this terminal, open another terminal and use the *container cp* command to copy file from your own machine to the container.

```
$ docker container cp ./index.js hello-node:/usr/src/app/index.js
```

And now we can run *node /usr/src/app/index.js* in the container. We can commit this as another new image, but there is an even better solution. The next section will be all about building your images like a pro.

12b: Building and configuring environments

The part was updated 21th Mar 2024: Create react app was replaced with Vite in the todo-frontend.

If you started the part before the update, you can see [here](#) the old material. There are some changes in the frontend configurations.

In the previous section, we used two different base images: ubuntu and node and did some manual work to get a simple "Hello, World!" running. The tools and commands we learned during that process will be helpful. In this section, we will learn how to build images and configure environments for our applications. We will start with a regular Express/Node.js backend and build on top of that with other services, including a MongoDB database.

Dockerfile

Instead of modifying a container by copying files inside, we can create a new image that contains the "Hello, World!" application. The tool for this is the Dockerfile. Dockerfile is a simple text file that contains all of the instructions for creating an image. Let's create an example Dockerfile from the "Hello, World!" application.

If you did not already, create a directory on your machine and create a file called *Dockerfile* inside that directory. Let's also put an *index.js* containing `console.log('Hello, World!')` next to the Dockerfile. Your directory structure should look like this:

```
├── index.js  
└── Dockerfile
```

inside that Dockerfile we will tell the image three things:

- Use the `node:20` as the base for our image
- Include the *index.js* inside the image, so we don't need to manually copy it into the container
- When we run a container from the image, use Node to execute the *index.js* file.

The wishes above will translate into a basic Dockerfile. The best location to place this file is usually at the root of the project.

The resulting *Dockerfile* looks like this:

```
FROM node:20  
  
WORKDIR /usr/src/app  
  
COPY ./index.js ./index.js  
  
CMD node index.js
```

`FROM` instruction will tell Docker that the base for the image should be `node:20`. `COPY` instruction will copy the file *index.js* from the host machine to the file with the same name in the image. `CMD` instruction tells what happens when *docker run* is used. `CMD` is the default command that can then be overwritten with the parameter given after the image name. See *docker run --help* if you forgot.

The `WORKDIR` instruction was slipped in to ensure we don't interfere with the contents of the image. It will guarantee all of the following commands will have `/usr/src/app` set as the working directory. If the directory doesn't exist in the base image, it will be automatically created.

If we do not specify a WORKDIR, we risk overwriting important files by accident. If you check the root (/) of the node:20 image with `docker run node:20 ls`, you can notice all of the directories and files that are already included in the image.

Now we can use the command `docker build` to build an image based on the Dockerfile. Let's spice up the command with one additional flag: `-t`, this will help us name the image:

```
$ docker build -t fs-hello-world .
[+] Building 3.9s [8/8] FINISHED
--
```

So the result is "Docker please build with tag (you may think the tag to be the name of the resulting image) `fs-hello-world` the Dockerfile in this directory". You can point to any Dockerfile, but in our case, a simple dot will mean the Dockerfile in *this* directory. That is why the command ends with a period. After the build is finished, you can run it with `docker run fs-hello-world`:

```
$ docker run fs-hello-world
Hello, World
```

As images are just files, they can be moved around, downloaded and deleted. You can list the images you have locally with `docker image ls`, delete them with `docker image rm`. See what other command you have available with `docker image --help`.

One more thing: in above it was mentioned that the default command, defined by the CMD in the Dockerfile, can be overridden if needed. We could e.g. open a bash session to the container and observe its content:

```
$ docker run -it fs-hello-world bash
root@2932e32dbc09:/usr/src/app# ls
index.js
root@2932e32dbc09:/usr/src/app#
```

More meaningful image

Moving an Express server to a container should be as simple as moving the "Hello, World!" application inside a container. The only difference is that there are more files. Thankfully `COPY` instruction can handle all that. Let's delete the `index.js` and create a new Express server. Lets use [express-generator](#) to create a basic Express application skeleton.

```
$ npx express-generator
...
install dependencies:
$ npm install
run the app:
$ DEBUG=playground:* npm start
```

First, let's run the application to get an idea of what we just created. Note that the command to run the application may be different from you, my directory was called `playground`.

```
$ npm install
$ DEBUG=playground:* npm start
playground:server Listening on port 3000 +0ms
```

Great, so now we can navigate to <http://localhost:3000> and the app is running there.

Containerizing that should be relatively easy based on the previous example.

- Use node as base
- Set working directory so we don't interfere with the contents of the base image
- Copy ALL of the files in this directory to the image
- Start with DEBUG=playground:* npm start

Let's place the following Dockerfile at the root of the project:

```
FROM node:20

WORKDIR /usr/src/app

COPY . .

CMD DEBUG=playground:* npm start
```

Let's build the image from the Dockerfile and then run it:

```
docker build -t express-server .
docker run -p 3123:3000 express-server
```

The `-p` flag in the run command will inform Docker that a port from the host machine should be opened and directed to a port in the container. The format for is `-p host-port:application-port`.

The application is now running! Let's test it by sending a GET request to <http://localhost:3123/>.

If yours doesn't work, skip to the next section. There is an explanation why it may not work even if you followed the steps correctly.

Shutting the app down is a headache at the moment. Use another terminal and `docker kill` command to kill the application. The `docker kill` will send a kill signal (SIGKILL) to the application to force it to shut down. It needs the name or the id of the container as an argument.

By the way, when using the id as the argument, the beginning of the ID is enough for Docker to know which container we mean.

```
$ docker container ls
CONTAINER ID   IMAGE          COMMAND           CREATED          STATUS
S             PORTS          NAMES
48096ca3ffec  express-server  "docker-entrypoint.s..."  9 seconds ago   Up 6
seconds      0.0.0.0:3123->3000/tcp, :::3123->3000/tcp  infallible_booth
```

```
$ docker kill 48  
48
```

In the future, let's use the same port on both sides of `-p`. Just so we don't have to remember which one we happened to choose.

Fixing potential issues we created by copy-pasting

There are a few steps we need to change to create a more comprehensive Dockerfile. It may even be that the above example doesn't work in all cases because we skipped an important step.

When we ran `npm install` on our machine, in some cases the **Node package manager** may install operating system specific dependencies during the install step. We may accidentally move non-functional parts to the image with the `COPY` instruction. This can easily happen if we copy the `node_modules` directory into the image.

This is a critical thing to keep in mind when we build our images. It's best to do most things, such as to run `npm install` during the build process *inside the container* rather than doing those prior to building. The easy rule of thumb is to only copy files that you would push to GitHub. Build artefacts or dependencies should not be copied since those can be installed during the build process.

We can use `.dockerignore` to solve the problem. The file `.dockerignore` is very similar to `.gitignore`, you can use that to prevent unwanted files from being copied to your image. The file should be placed next to the Dockerfile. Here is a possible content of a `.dockerignore`

```
.dockerignore  
.gitignore  
node_modules  
Dockerfile
```

However, in our case, the `.dockerignore` isn't the only thing required. We will need to install the dependencies during the build step. The *Dockerfile* changes to:

```
FROM node:20  
  
WORKDIR /usr/src/app  
  
COPY . .  
  
RUN npm install  
CMD DEBUG=playground:* npm start
```

The `npm install` can be risky. Instead of using `npm install`, npm offers a much better tool for installing dependencies, the `ci` command.

Differences between `ci` and `install`:

- `install` may update the `package-lock.json`
- `install` may install a different version of a dependency if you have `^` or `~` in the version of the dependency.

- *ci* will delete the `node_modules` folder before installing anything
- *ci* will follow the `package-lock.json` and does not alter any files

So in short: *ci* creates reliable builds, while *install* is the one to use when you want to install new dependencies.

As we are not installing anything new during the build step, and we don't want the versions to suddenly change, we will use *ci*:

```
FROM node:20

WORKDIR /usr/src/app

COPY . .

RUN npm ci
CMD DEBUG=playground:* npm start
```

Even better, we can use `npm ci --omit=dev` to not waste time installing development dependencies.

As you noticed in the comparison list; `npm ci` will delete the `node_modules` folder so creating the `.dockerignore` did not matter. However, `.dockerignore` is an amazing tool when you want to optimize your build process. We will talk briefly about these optimizations later.

Now the Dockerfile should work again, try it with `docker build -t express-server . && docker run -p 3123:3000 express-server`

Note that we are here chaining two bash commands with `&&`. We could get (nearly) the same effect by running both commands separately. When chaining commands with `&&` if one command fails, the next ones in the chain will not be executed.

We set an environment variable `DEBUG=playground:*` during `CMD` for the `npm start`. However, with Dockerfiles we could also use the instruction `ENV` to set environment variables. Let's do that:

```
FROM node:20

WORKDIR /usr/src/app

COPY . .

RUN npm ci

ENV DEBUG=playground:*
CMD npm start
```

If you're wondering what the DEBUG environment variable does, read [here](#).

Dockerfile best practices

There are 2 rules of thumb you should follow when creating images:

- Try to create as **secure** of an image as possible
- Try to create as **small** of an image as possible

Smaller images are more secure by having less attack surface area, and smaller images also move faster in deployment pipelines.

Snyk has a great list of 10 best practices for Node/Express containerization. Read those [here](#).

One big carelessness we have left is running the application as root instead of using a user with lower privileges. Let's do a final fix to the Dockerfile:

```
FROM node:20

WORKDIR /usr/src/app

COPY --chown=node:node . .
RUN npm ci

ENV DEBUG=playground:*

USER node
CMD npm start
```

Exercise 12.5.

Exercise 12.5: Containerizing a Node application

The repository you cloned or copied in the [first exercise](#) contains a todo-app. See the todo-app/todo-backend and read through the README. We will not touch the todo-frontend yet.

- Step 1. Containerize the todo-backend by creating a *todo-app/todo-backend/Dockerfile* and building an image.
- Step 2. Run the todo-backend image with the correct ports open. Make sure the visit counter increases when used through a browser in <http://localhost:3000/> (or some other port if you configure so)

Tip: Run the application outside of a container to examine it before starting to containerize.

Using Docker compose

In the previous section, we created an Express server and knew that it runs in port 3000, and ran it with *docker build -t express-server . && docker run -p 3000:3000 express-server*. This already looks like something you would need to put into a script to remember. Fortunately, Docker offers us a better

solution.

Docker compose is another fantastic tool, which can help us to manage containers. Let's start using compose as we learn more about containers as it will help us save some time with the configuration.

Now we can turn the previous spell into a yaml file. The best part about yaml files is that you can save these to a Git repository!

Create the file **docker-compose.yml** and place it at the root of the project, next to the Dockerfile. The file content is

```
version: '3.8'          # Version 3.8 is quite new and should work

services:
  app:                 # The name of the service, can be anything
    image: express-server # Declares which image to use
    build: .              # Declares where to build if image is not found
    ports:
      - 3000:3000         # Declares the ports to publish
```

The meaning of each line is explained as a comment. If you want to see the full specification see the [documentation](#).

Now we can use *docker compose up* to build and run the application. If we want to rebuild the images we can use *docker compose up --build*.

You can also run the application in the background with *docker compose up -d* (-d for detached) and close it with *docker compose down*.

Note that some older Docker versions (especially in Windows) do not support the command docker compose. One way to circumvent this problem is to [install](#) the stand alone command docker-compose that works mostly similarly to docker compose. However, the preferable fix is to update the Docker to a more recent version.

Creating files like *docker-compose.yml* that declare what you want instead of script files that you need to run in a specific order / a specific number of times is often a great practice.

Exercise 12.6.

Exercise 12.6: Docker compose

Create a *todo-app/todo-backend/docker-compose.yml* file that works with the Node application from the previous exercise.

The visit counter is the only feature that is required to be working.

Utilizing containers in development

When you are developing software, containerization can be used in various ways to improve your

quality of life. One of the most useful cases is by bypassing the need to install and configure tools twice.

It may not be the best option to move your entire development environment into a container, but if that's what you want it's certainly possible. We will revisit this idea at the end of this part. But until then, *run the Node application itself outside of containers*.

The application we met in the previous exercises uses MongoDB. Let's explore [Docker Hub](#) to find a MongoDB image. Docker Hub is the default place where Docker pulls the images from, you can use other registries as well, but since we are already knee-deep in Docker it's a good choice. With a quick search, we can find https://hub.docker.com/_/mongo

Create a new yaml called *todo-app/todo-backend/docker-compose.dev.yml* that looks like following:

```
version: '3.8'

services:
  mongo:
    image: mongo
    ports:
      - 3456:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      MONGO_INITDB_DATABASE: the_database
```

The meaning of the two first environment variables defined above is explained on the Docker Hub page:

These variables, used in conjunction, create a new user and set that user's password. This user is created in the admin authentication database and given the role of root, which is a "superuser" role.

The last environment variable *MONGO_INITDB_DATABASE* will tell MongoDB to create a database with that name.

You can use *-f* flag to specify a *file* to run the Docker Compose command with e.g.

```
docker compose -f docker-compose.dev.yml up
```

Now that we may have multiple compose files, it's useful.

Now start the MongoDB with *docker compose -f docker-compose.dev.yml up -d*. With *-d* it will run it in the background. You can view the output logs with *docker compose -f docker-compose.dev.yml logs -f*. There the *-f* will ensure we *follow* the logs.

As said previously, currently we **do not** want to run the Node application inside a container. Developing while the application itself is inside a container is a challenge. We will explore that option later in this part.

Run the good old `npm install` first on your machine to set up the Node application. Then start the application with the relevant environment variable. You can modify the code to set them as the defaults or use the `.env` file. There is no hurt in putting these keys to GitHub since they are only used in your local development environment. I'll just throw them in with the `npm run dev` to help you copy-paste.

```
$ MONGO_URL=mongodb://localhost:3456/the_database npm run dev
```

This won't be enough; we need to create a user to be authorized inside of the container. The url <http://localhost:3000/todos> leads to an authentication error:

```
[nodemon] 2.0.12
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./bin/www`
/Users/mluukkai/dev/fs-ci-lokakuu/repo/todo-app/todo-backend/node_modules/mongodb/lib/cmap/connection.js:272
    callback(new MongoError(document));
    ^
MongoError: command find requires authentication
    at MessageStream.messageHandler [/Users/mluukkai/dev/fs-ci-lokakuu/repo/todo-app/todo-backend/node_modules/mongodb/lib/cmap/connection.js:272:20]
```

Bind mount and initializing the database

In the [MongoDB Docker Hub](#) page under "Initializing a fresh instance" is the info on how to execute JavaScript to initialize the database and a user for it.

The exercise project has a file `todo-app/todo-backend/mongo/mongo-init.js` with contents:

```
db.createUser({
  user: 'the_username',
  pwd: 'the_password',
  roles: [
    [
      role: 'dbOwner',
      db: 'the_database'
    ],
  ],
});
db.createCollection('todos');

db.todos.insert({ text: 'Write code', done: true });
db.todos.insert({ text: 'Learn about containers', done: false });
```

This file will initialize the database with a user and a few todos. Next, we need to get it inside the container at startup.

We could create a new image FROM mongo and COPY the file inside, or we can use a [bind mount](#) to mount the file *mongo-init.js* to the container. Let's do the latter.

Bind mount is the act of binding a file (or directory) on the host machine to a file (or directory) in the container. A bind mount is done by adding a `-v` flag with *container run*. The syntax is `-v FILE-IN-HOST:FILE-IN-CONTAINER`. Since we already learned about Docker Compose let's skip that. The bind mount is declared under key *volumes* in *docker-compose.dev.yml*. Otherwise the format is the same, first host and then container:

```
mongo:  
  image: mongo  
  ports:  
    - 3456:27017  
  environment:  
    MONGO_INITDB_ROOT_USERNAME: root  
    MONGO_INITDB_ROOT_PASSWORD: example  
    MONGO_INITDB_DATABASE: the_database  
  volumes:  
    - ./mongo/mongo-init.js:/docker-entrypoint-initdb.d/mongo-init.js
```

The result of the bind mount is that the file *mongo-init.js* in the mongo folder of the host machine is the same as the *mongo-init.js* file in the container's */docker-entrypoint-initdb.d* directory. Changes to either file will be available in the other. We don't need to make any changes during runtime. But this will be the key to software development in containers.

Run *docker compose -f docker-compose.dev.yml down --volumes* to ensure that nothing is left and start from a clean slate with *docker compose -f docker-compose.dev.yml up* to initialize the database.

If you see an error like this:

```
mongo_database | failed to load: /docker-entrypoint-initdb.d/mongo-init.js  
mongo_database | exiting with code -3
```

you may have a read permission problem. They are not uncommon when dealing with volumes. In the above case, you can use *chmod a+r mongo-init.js*, which will give everyone read access to that file. Be careful when using *chmod* since granting more privileges can be a security issue. Use the *chmod* only on the *mongo-init.js* on your computer.

Now starting the Express application with the correct environment variable should work:

```
MONGO_URL=mongodb://the_username:the_password@localhost:3456/the_database npm run dev
```

Let's check that the <http://localhost:3000/todos> returns the two todos we inserted in the initialization. We can and *should* use Postman to test the basic functionality of the app, such as adding or deleting a todo.

Still problems?

For some reason, the initialization of Mongo has caused problems for many.

If the app does not work and you still end up with the following error

```
/Users/mluukkai/dev/fs-ci-lokakuu/repo/todo-app/todo-backend/node_modules/mongodb/lib/cmap/connection.js:272
    callback(new MongoError(`document`));
    ^
MongoError: command find requires authentication
    at MessageStream.messageHandler (/Users/mluukkai/dev/fs-ci-lokakuu/repo/todo-app/todo-backend/node_modules/mongodb/lib/cmap/connection.js:272:20)
```

run these commands:

```
docker compose -f docker-compose.dev.yml down --volumes
docker image rm mongo
```

After these, try to start Mongo again.

If the problem persists, let us drop the idea of a volume altogether and copy the initialization script to a custom image. Create the following *Dockerfile* to the directory *todo-app/todo-backend/mongo*

```
FROM mongo

COPY ./mongo-init.js /docker-entrypoint-initdb.d/
```

Build it to an image with the command

```
docker build -t initialized-mongo .
```

Change now the *docker-compose.dev.yml* to use the new image:

```
mongo:
  image: initialized-mongo
  ports:
  - 3456:27017
  environment:
    MONGO_INITDB_ROOT_USERNAME: root
    MONGO_INITDB_ROOT_PASSWORD: example
    MONGO_INITDB_DATABASE: the_database
```

Now the app should finally work.

Persisting data with volumes

By default, database containers are not going to preserve our data. When you close the database container you *may or may not* be able to get the data back.

Mongo is actually a rare case in which the container indeed does preserve the data. This happens, since the developers who made the Docker image for Mongo have defined a volume

to be used. This line in the Dockerfile will instruct Docker to preserve the data in a volume.

There are two distinct methods to store the data:

- Declaring a location in your filesystem (called [bind mount](#))
- Letting Docker decide where to store the data ([volume](#))

The first choice is preferable in most cases whenever one *really* needs to avoid the data being deleted.

Let's see both in action with Docker compose. Let us start with *bind mount*:

```
services:  
  mongo:  
    image: mongo  
    ports:  
      - 3456:27017  
    environment:  
      MONGO_INITDB_ROOT_USERNAME: root  
      MONGO_INITDB_ROOT_PASSWORD: example  
      MONGO_INITDB_DATABASE: the_database  
    volumes:  
      - ./mongo/mongo-init.js:/docker-entrypoint-initdb.d/mongo-init.js  
      - ./mongo_data:/data/db
```

The above will create a directory called *mongo_data* to your local filesystem and map it into the container as */data/db*. This means the data in */data/db* is stored outside of the container but still accessible by the container! Just remember to add the directory to *.gitignore*.

A similar outcome can be achieved with a *named volume*:

```
services:  
  mongo:  
    image: mongo  
    ports:  
      - 3456:27017  
    environment:  
      MONGO_INITDB_ROOT_USERNAME: root  
      MONGO_INITDB_ROOT_PASSWORD: example  
      MONGO_INITDB_DATABASE: the_database  
    volumes:  
      - ./mongo/mongo-init.js:/docker-entrypoint-initdb.d/mongo-init.js  
      - mongo_data:/data/db  
  
volumes: mongo_data:
```

Now the volume is created but managed by Docker. After starting the application (*docker compose -f docker-compose.dev.yml up*) you can list the volumes with *docker volume ls*, inspect one of them with *docker volume inspect* and even delete them with *docker volume rm*:

```

$ docker volume ls
DRIVER      VOLUME NAME
local       todo-backend_mongo_data
$ docker volume inspect todo-backend_mongo_data
[
  {
    "CreatedAt": "2024-19-03T12:52:11Z",
    "Driver": "local",
    "Labels": [
      "com.docker.compose.project": "todo-backend",
      "com.docker.compose.version": "1.29.2",
      "com.docker.compose.volume": "mongo_data"
    ],
    "Mountpoint": "/var/lib/docker/volumes/todo-backend_mongo_data/_data",
    "Name": "todo-backend_mongo_data",
    "Options": null,
    "Scope": "local"
  }
]

```

The named volume is still stored in your local filesystem but figuring out *where* may not be as trivial as with the previous option.

Exercise 12.7.

Exercise 12.7: Little bit of MongoDB coding

Note that this exercise assumes that you have done all the configurations made in the material after exercise 12.5. You should still run the todo-app backend *outside a container*; just the MongoDB is containerized for now.

The todo application has no proper implementation of routes for getting one todo (GET `/todos/:id`) and updating one todo (PUT `/todos/:id`). Fix the code.

Debugging issues in containers

When coding, you most likely end up in a situation where everything is broken.

- Matti Luukkainen

When developing with containers, we need to learn new tools for debugging, since we can not just "console.log" everything. When code has a bug, you may often be in a state where at least something works, so you can work forward from that. Configuration most often is in either of two states: 1. working or 2. broken. We will go over a few tools that can help when your application is in the latter state.

When developing software, you can safely progress step by step, all the time verifying that what you have coded behaves as expected. Often, this is not the case when doing configurations. The configuration you may be writing can be broken until the moment it is finished. So when you write a long docker-compose.yml or Dockerfile and it does not work, you need to take a moment and think about the various ways you could confirm something is working.

Question Everything is still applicable here. As said in [part 3](#): The key is to be systematic. Since the problem can exist anywhere, *you must question everything*, and eliminate all possible sources of error one by one.

For myself, the most valuable method of debugging is stopping and thinking about what I'm trying to accomplish instead of just bashing my head at the problem. Often there is a simple, alternate, solution or quick google search that will get me moving forward.

exec

The Docker command `exec` is a heavy hitter. It can be used to jump right into a container when it's running.

Let's start a web server in the background and do a little bit of debugging to get it running and displaying the message "Hello, exec!" in our browser. Let's choose [Nginx](#) which is, among other things, a server capable of serving static HTML files. It has a default index.html that we can replace.

```
$ docker container run -d nginx
```

Ok, now the questions are:

- Where should we go with our browser?
- Is it even running?

We know how to answer the latter: by listing the running containers.

```
$ docker container ls
CONTAINER ID        IMAGE       COMMAND      CREATED     STATUS      PORTS     NAMES
3f831a57b7cc        nginx      "..."        3 sec ago   Up 2 sec   80/tcp    keen_darwin
```

Yes! We got the first question answered as well. It seems to listen on port 80, as seen on the output above.

Let's shut it down and restart with the `-p` flag to have our browser access it.

```
$ docker container stop keen_darwin
$ docker container rm keen_darwin

$ docker container run -d -p 8080:80 nginx
```

Editor's note—when doing development, it is **essential** to constantly follow the container logs. I'm usually not running containers in a detached mode (that is with `-d`) since it requires a bit of an extra effort to open the logs.

When I'm 100% sure that everything works... no, when I'm 200% sure, then I might relax a bit and start the containers in detached mode. Until everything again falls apart and it is time to open the logs again.

Let's look at the app by going to <http://localhost:8080>. It seems that the app is showing the wrong message! Let's hop right into the container and fix this. Keep your browser open, we won't need to shut down the container for this fix. We will execute bash inside the container, the flags `-it` will ensure that we can interact with the container:

```
$ docker container ls
CONTAINER ID        IMAGE       COMMAND      PORTS
7edcb36aff08      nginx      ...          0.0.0.0:8080->80/tcp
                                                               NAMES
                                                               wonderful_ramanujan

$ docker exec -it wonderful_ramanujan bash
root@7edcb36aff08:/#
```

Now that we are in, we need to find the faulty file and replace it. Quick Google tells us that file itself is `/usr/share/nginx/html/index.html`.

Let's move to the directory and delete the file

```
root@7edcb36aff08:/# cd /usr/share/nginx/html/
root@7edcb36aff08:/# rm index.html
```

Now, if we go to <http://localhost:8080> we know that we deleted the correct file. The page shows 404. Let's replace it with one containing the correct contents:

```
root@7edcb36aff08:/# echo "Hello, exec!" > index.html
```

Refresh the page, and our message is displayed! Now we know how exec can be used to interact with the containers. Remember that all of the changes are lost when the container is deleted. To preserve the changes, you must use `commit` just as we did in [previous section](#).

Exercise 12.8.

Exercise 12.8: Mongo command-line interface

Use `script` to record what you do, save the file as `script-answers/exercise12_8.txt`

While the MongoDB from the previous exercise is running, access the database with the Mongo command-line interface (CLI). You can do that using docker exec. Then add a new todo using the CLI.

The command to open CLI when inside the container is `mongosh`

The Mongo CLI will require the username and password flags to authenticate correctly. Flags `-u root -p example` should work, the values are from the `docker-compose.dev.yml`.

- Step 1: Run MongoDB
- Step 2: Use docker exec to get inside the container
- Step 3: Open Mongo cli

When you have connected to the Mongo cli you can ask it to show dbs inside:

```
> show dbs
admin          0.000GB
config         0.000GB
local          0.000GB
the_database   0.000GB
```

To access the correct database:

```
> use the_database
```

And finally to find out the collections:

```
> show collections
todos
```

We can now access the data in those collections:

```
> db.todos.find({})
[
  {
    _id: ObjectId("633c270ba211aa5f7931f078"),
    text: 'Write code',
    done: false
  },
  {
    _id: ObjectId("633c270ba211aa5f7931f079"),
    text: 'Learn about containers',
    done: false
  }
]
```

Insert one new todo with the text: "Increase the number of tools in my toolbelt" with the status done as *false*. Consult the [documentation](#) to see how the addition is done.

Ensure that you see the new todo both in the Express app and when querying from Mongo CLI.

Redis

Redis is a [key-value](#) database. In contrast to eg. MongoDB, the data stored in key-value storage has a bit less structure, there are eg. no collections or tables, it just contains junks of data that can be fetched based on the *key* that was attached to the data (the *value*).

By default, Redis works *in-memory*, which means that it does not store data persistently.

An excellent use case for Redis is to use it as a cache. Caches are often used to store data that is otherwise slow to fetch and save the data until it's no longer valid. After the cache becomes invalid, you would then fetch the data again and store it in the cache.

Redis has nothing to do with containers. But since we are already able to add *any* 3rd party service to your applications, why not learn about a new one?

Exercises 12.9. - 12.11.

Exercise 12.9: Set up Redis for the project

The Express server has already been configured to use Redis, and it is only missing the *REDIS_URL* environment variable. The application will use that environment variable to connect to the Redis. Read through the [Docker Hub page for Redis](#), add Redis to the *todo-app/todo-backend/docker-compose.dev.yml* by defining another service after mongo:

```
services:  
  mongo:  
    ...  
  redis:  
    ???
```

Since the Docker Hub page doesn't have all the info, we can use Google to aid us. The default port for Redis is found by doing so:

Google default port for redis

Käikki Kuvahaku Videot Ostokset Kartat Lisää Työkalut

Noin 4 130 000 tulosta (0,53 sekuntia)

Port 6379

By default, the Redis server runs on **TCP Port 6379**.

We won't have any idea if the configuration works unless we try it. The application will not start using Redis by itself, that shall happen in the next exercise.

Once Redis is configured and started, restart the backend and give it the *REDIS_URL*, which has the form *redis://host:port*

```
$ REDIS_URL=insert-redis-url-here MONGO_URL=mongodb://the_username:the_password@localhost:3456/the_database npm run dev
```

You can now test the configuration by adding the line

```
const redis = require('../redis')
```

to the Express server eg. in file *routes/index.js*. If nothing happens, the configuration is done right. If not, the server crashes:

```
events.js:291
  throw er; // Unhandled 'error' event
  ^
Error: Redis connection to localhost:6379 failed - connect ECONNREFUSED 127.0.0.1
:6379
    at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1144:16)
Emitted 'error' event on RedisClient instance at:
    at RedisClient.on_error (/Users/mluukkai/opetus/docker-fs/container-app/express-app/node_modules/redis/index.js:342:14)
    at Socket.<anonymous> (/Users/mluukkai/opetus/docker-fs/container-app/express-app/node_modules/redis/index.js:223:14)
    at Socket.emit (events.js:314:20)
    at emitErrorNT (internal/streams/destroy.js:100:8)
    at emitErrorCloseNT (internal/streams/destroy.js:68:3)
    at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  errno: -61,
  code: 'ECONNREFUSED',
  syscall: 'connect',
  address: '127.0.0.1',
  port: 6379
}
[nodemon] app crashed - waiting for file changes before starting...
```

Exercise 12.10:

The project already has <https://www.npmjs.com/package/redis> installed and two functions "promisified" - getAsync and setAsync.

- setAsync function takes in key and value, using the key to store the value.
- getAsync function takes in a key and returns the value in a promise.

Implement a todo counter that saves the number of created todos to Redis:

- Step 1: Whenever a request is sent to add a todo, increment the counter by one.
- Step 2: Create a GET /statistics endpoint where you can ask for the usage metadata. The format should be the following JSON:

```
{
  "added.todos": 0
}
```

Exercise 12.11:

Use *script* to record what you do, save the file as `script-answers/exercise12_11.txt`

If the application does not behave as expected, direct access to the database may be beneficial in pinpointing problems. Let us try out how `redis-cli` can be used to access the database.

- Go to the Redis container with `docker exec` and open the `redis-cli`.
- Find the key you used with `KEYS *`
- Check the value of the key with the command `GET`
- Set the value of the counter to 9001, find the right command from [here](#)
- Make sure that the new value works by refreshing the page <http://localhost:3000/statistics>
- Create a new todo with Postman and ensure from `redis-cli` that the counter has increased accordingly
- Delete the key from the cli and ensure that the counter works when new todos are added

Persisting data with Redis

In the previous section, it was mentioned that *by default* Redis does not persist the data. However, the persistence is easy to toggle on. We only need to start the Redis with a different command, as instructed by the [Docker hub page](#):

```
services:  
  redis:  
    # Everything else  
    command: ['redis-server', '--appendonly', 'yes'] # Overwrite the CMD  
    volumes: # Declare the volume  
      - ./redis_data:/data
```

The data will now be persisted to the directory `redis_data` of the host machine. Remember to add the directory to `.gitignore`!

Other functionality of Redis

In addition to the GET, SET and DEL operations on keys and values, Redis can do also quite a lot more. It can for example automatically expire keys, which is a very useful feature when Redis is used as a cache.

Redis can also be used to implement the so-called `publish-subscribe` (or PubSub) pattern which is an asynchronous communication mechanism for distributed software. In this scenario, Redis works as a *message broker* between two or more services. Some of the services are *publishing* messages by sending those to Redis, which on arrival of a message, informs the parties that have *subscribed* to those messages.

Exercise 12.12.

Exercise 12.12: Persisting data in Redis

Check that the data is not persisted by default: after running

```
docker compose -f docker-compose.dev.yml down
```

```
docker compose -f docker-compose.dev.yml up
```

the counter value is reset to 0.

Then create a volume for Redis data (by modifying `todo-app/todo-backend/docker-compose.dev.yml`) and make sure that the data survives after running

```
docker compose -f docker-compose.dev.yml down
docker compose -f docker-compose.dev.yml up
```

12c: Basics of Orchestration

The part was updated 21th Mar 2024: Create react app was replaced with Vite in the todo-frontend.

If you started the part before the update, you can see [here](#) the old material. There are some changes in the frontend configurations.

We have now a basic understanding of Docker and can use it to easily set up eg. a database for our app. Let us now move our focus to the frontend.

React in container

Let's create and containerize a React application next. We start with the usual steps

```
$ npm create vite@latest hello-front -- --template react
$ cd hello-front
$ npm install
```

The next step is to turn the JavaScript code and CSS, into production-ready static files. Vite already has `build` as an npm script so let's use that:

```
$ npm run build
...
Creating an optimized production build...
...
The build folder is ready to be deployed.
...
```

Great! The final step is figuring out a way to use a server to serve the static files. As you may know, we could use our `express.static` with the Express server to serve the static files. I'll leave that as an exercise for you to do at home. Instead, we are going to go ahead and start writing our Dockerfile:

```
FROM node:20
WORKDIR /usr/src/app
COPY . .
```

```
RUN npm ci  
RUN npm run build
```

That looks about right. Let's build it and see if we are on the right track. Our goal is to have the build succeed without errors. Then we will use bash to check inside of the container to see if the files are there.

```
$ docker build . -t hello-front  
=> [4/5] RUN npm ci  
=> [5/5] RUN npm run  
...  
=> => naming to docker.io/library/hello-front  
  
$ docker run -it hello-front bash  
  
root@98fa9483ee85:/usr/src/app# ls  
Dockerfile README.md dist index.html node_modules package-lock.json package.json public src vite.config.js  
  
root@98fa9483ee85:/usr/src/app# ls dist  
assets index.html vite.svg
```

A valid option for serving static files now that we already have Node in the container is [serve](#). Let's try installing serve and serving the static files while we are inside the container.

```
root@98fa9483ee85:/usr/src/app# npm install -g serve  
  
added 89 packages in 2s  
  
root@98fa9483ee85:/usr/src/app# serve dist
```

```
Serving!  
- Local: http://localhost:3000  
- Network: http://172.17.0.2:3000
```

Great! Let's `ctrl+c` and exit out and then add those to our Dockerfile.

The installation of `serve` turns into a `RUN` in the Dockerfile. This way the dependency is installed during the build process. The command to serve the `dist` directory will become the command to start the container:

```
FROM node:20  
WORKDIR /usr/src/app
```

```

COPY . .

RUN npm ci

RUN npm run build

RUN npm install -g serve
CMD ["serve", "dist"]

```

Our CMD now includes square brackets and as a result, we now use the so-called *exec form* of CMD. There are actually **three** different forms for the CMD out of which the exec form is preferred. Read the [documentation](#) for more info.

When we now build the image with `docker build . -t hello-front` and run it with `docker run -p 5001:3000 hello-front`, the app will be available in <http://localhost:5001>.

Using multiple stages

While `serve` is a *valid* option we can do better. A good goal is to create Docker images so that they do not contain anything irrelevant. With a minimal number of dependencies, images are less likely to break or become vulnerable over time.

[Multi-stage builds](#) are designed to split the build process into many separate stages, where it is possible to limit what parts of the image files are moved between the stages. That opens possibilities for limiting the size of the image since not all by-products of the build are necessary for the resulting image. Smaller images are faster to upload and download and they help reduce the number of vulnerabilities your software may have.

With multi-stage builds, a tried and true solution like [Nginx](#) can be used to serve static files without a lot of headaches. The Docker Hub [page for Nginx](#) tells us the required info to open the ports and "Hosting some simple static content".

Let's use the previous Dockerfile but change the FROM to include the name of the stage:

```

# The first FROM is now a stage called build-stage
FROM node:20 AS build-stage
WORKDIR /usr/src/app

COPY . .

RUN npm ci

RUN npm run build

# This is a new stage, everything before this is gone, except the files we want
# to COPY
FROM nginx:1.25-alpine
# COPY the directory build from build-stage to /usr/share/nginx/html
# The target location here was found from the Docker hub page
COPY --from=build-stage /usr/src/app/dist /usr/share/nginx/html

```

We have declared also *another stage* where only the relevant files of the first stage (the *dist*

directory, that contains the static content) are copied.

After we build it again, the image is ready to serve the static content. The default port will be 80 for Nginx, so something like `-p 8000:80` will work, so the parameters of the run command need to be changed a bit.

Multi-stage builds also include some internal optimizations that may affect your builds. As an example, multi-stage builds skip stages that are not used. If we wish to use a stage to replace a part of a build pipeline, like testing or notifications, we must pass **some** data to the following stages. In some cases this is justified: copy the code from the testing stage to the build stage. This ensures that you are building the tested code.

Exercises 12.13 - 12.14.

Exercise 12.13: Todo application frontend

Finally, we get to the todo-frontend. View the `todo-app/todo-frontend` and read through the README.

Start by running the frontend outside the container and ensure that it works with the backend.

Containerize the application by creating `todo-app/todo-frontend/Dockerfile` and use **ENV** instruction to pass `VITE_BACKEND_URL` to the application and run it with the backend. The backend should still be running outside a container.

Note that you need to set `VITE_BACKEND_URL` before building the frontend, otherwise, it does not get defined in the code!

Exercise 12.14: Testing during the build process

One interesting possibility to utilize multi-stage builds is to use a separate build stage for **testing**. If the testing stage fails, the whole build process will also fail. Note that it may not be the best idea to move *all testing* to be done during the building of an image, but there may be *some* containerization-related tests where it might be worth considering.

Extract a component `Todo` that represents a single todo. Write a test for the new component and add running tests into the build process.

You can add a new build stage for the test if you wish to do so. If you do so, remember to read the last paragraph before exercise 12.13 again!

Development in containers

Let's move the whole todo application development to a container. There are a few reasons why you would want to do that:

- To keep the environment similar between development and production to avoid bugs that appear only in the production environment
- To avoid differences between developers and their personal environments that lead to difficulties in application development
- To help new team members hop in by having them install container runtime - and requiring nothing else.

These all are great reasons. The tradeoff is that we may encounter some unconventional behavior

when we aren't running the applications like we are used to. We will need to do at least two things to move the application to a container:

- Start the application in development mode
- Access the files with VS Code

Let's start with the frontend. Since the Dockerfile will be significantly different from the production Dockerfile let's create a new one called *dev.Dockerfile*.

Note we shall use the name *dev.Dockerfile* for development configurations and *Dockerfile* otherwise.

Starting the Vite in development mode should be easy. Let's start with the following:

```
FROM node:20

WORKDIR /usr/src/app

COPY . .

# Change npm ci to npm install since we are going to be in development mode
RUN npm install

# npm start is the command to start the application in development mode
CMD ["npm", "run", "dev", "--", "--host"]
```

Note the extra parameters `--host` in the *CMD*. Those are needed to expose the development server to be visible outside the Docker network. By default the development server is exposed only to localhost, and despite we access the frontend still using the localhost address, it is in reality attached to the Docker network.

During build the flag `-f` will be used to tell which file to use, it would otherwise default to Dockerfile, so the following command will build the image:

```
docker build -f ./dev.Dockerfile -t hello-front-dev .
```

The Vite will be served in port 5173, so you can test that it works by running a container with that port published.

The second task, accessing the files with VSCode, is not yet taken care of. There are at least two ways of doing this:

- [The Visual Studio Code Remote - Containers extension](#)
- Volumes, the same thing we used to preserve data with the database

Let's go over the latter since that will work with other editors as well. Let's do a trial run with the flag `-v`, and if that works, then we will move the configuration to a docker-compose file. To use the `-v`, we will need to tell it the current directory. The command `pwd` should output the path to the current directory for you. Try this with `echo $(pwd)` in your command line. We can use that as the left side for `-v` to map the current directory to the inside of the container or you can use the full directory path.

```
$ docker run -p 5173:5173 -v "$(pwd):/usr/src/app/" hello-front-dev
> todo-vite@0.0.0 dev
> vite --host

VITE v5.1.6  ready in 130 ms
```

Now we can edit the file *src/App.js*, and the changes should be hot-loaded to the browser!

If you have a Mac with M1/M2 processor, the above command fails. In the error message, we notice the following:

```
Error: Cannot find module '@rollup/rollup-linux-arm64-gnu'
```

The problem is the library *rollup* that has its own version for all operating systems and processor architectures. Due to the volume mapping, the container is now using the *node_modules* from the host machine directory where the *@rollup/rollup-darwin-arm64* (the version suitable Mac M1/M2) is installed, so the right version of the library for the container *@rollup/rollup-linux-arm64-gnu* is not found.

There are several ways to fix the problem. Let's use the perhaps simplest one. Start the container with bash as the command, and run the *npm install* inside the container:

```
$ docker run -it -v "$(pwd):/usr/src/app/" front-dev bash
root@b83e9040b91d:/usr/src/app# npm install
```

Now both versions of the library *rollup* are installed and the container works!

Next, let's move the config to the file *docker-compose.dev.yml*. That file should be at the root of the project as well:

```
services:
  app:
    image: hello-front-dev
    build:
      context: . # The context will pick this directory as the "build context"
      dockerfile: dev.Dockerfile # This will simply tell which dockerfile to read

    volumes:
      - ./:/usr/src/app # The path can be relative, so ./ is enough to say "the
                        same location as the docker-compose.yml"
    ports:
      - 5173:5173
    container_name: hello-front-dev # This will name the container hello-front-d
ev
```

With this configuration, *docker compose up* can run the application in development mode. You don't even need Node installed to develop it!

Note we shall use the name *docker-compose.dev.yml* for development environment compose files,

and the default name `docker-compose.yml` otherwise.

Installing new dependencies is a headache for a development setup like this. One of the better options is to install the new dependency **inside** the container. So instead of doing e.g. `npm install axios`, you have to do it in the running container e.g. `docker exec hello-front-dev npm install axios`, or add it to the `package.json` and run `docker build` again.

Exercise 12.15

Exercise 12.15: Set up a frontend development environment

Create `todo-frontend/docker-compose.dev.yml` and use volumes to enable the development of the todo-frontend while it is running *inside* a container.

Communication between containers in a Docker network

The Docker Compose tool sets up a network between the containers and includes a DNS to easily connect two containers. Let's add a new service to the Docker Compose and we shall see how the network and DNS work.

[Busybox](#) is a small executable with multiple tools you may need. It is called "The Swiss Army Knife of Embedded Linux", and we definitely can use it to our advantage.

Busybox can help us to debug our configurations. So if you get lost in the later exercises of this section, you should use Busybox to find out what works and what doesn't. Let's use it to explore what was just said. That containers are inside a network and you can easily connect between them. Busybox can be added to the mix by changing `docker-compose.dev.yml` to:

```
services:
  app:
    image: hello-front-dev
    build:
      context: .
      dockerfile: dev.Dockerfile
    volumes:
      - ./:/usr/src/app
    ports:
      - 5173:5173
    container_name: hello-front-dev
  debug-helper: image: busybox
```

The Busybox container won't have any process running inside so we can not `exec` in there. Because of that, the output of `docker compose up` will also look like this:

```
$ docker compose -f docker-compose.dev.yml up
                                         0.0s
Attaching to front-dev, debug-helper-1
debug-helper-1 exited with code 0
front-dev      |
front-dev      | > todo-vite@0.0.0 dev
front-dev      | > vite --host
```

```
front-dev |  
front-dev |  
front-dev |     VITE v5.2.2  ready in 153 ms
```

This is expected as it's just a toolbox. Let's use it to send a request to hello-front-dev and see how the DNS works. While the hello-front-dev is running, we can do the request with `wget` since it's a tool included in Busybox to send a request from the debug-helper to hello-front-dev.

With Docker Compose we can use `docker compose run SERVICE COMMAND` to run a service with a specific command. Command `wget` requires the flag `-O` with `-` to output the response to the stdout:

```
$ docker compose -f docker-compose.dev.yml run debug-helper wget -O - http://app  
:5173  
  
Connecting to app:5173 [192.168.240.3:5173]  
writing to stdout  
<!doctype html>  
<html lang="en">  
  <head>  
    <script type="module">  
    ...
```

The URL is the interesting part here. We simply said to connect to port 5173 of the service `app`. The `app` is the name of the service specified in the `docker-compose.dev.yml`:

```
services:  
  app:  
    image: hello-front-dev  
    build:  
      context: .  
      dockerfile: dev.Dockerfile  
    volumes:  
      - ./:/usr/src/app  
    ports:  
      - 5173:5173    container_name: hello-front-dev
```

The port used is the port from which the application is available in that container, also specified in the `docker-compose.dev.yml`. The port does not need to be published for other services in the same network to be able to connect to it. The "ports" in the docker-compose file are only for external access.

Let's change the port configuration in the `docker-compose.dev.yml` to emphasize this:

```
services:  
  app:  
    image: hello-front-dev  
    build:  
      context: .  
      dockerfile: dev.Dockerfile  
    volumes:  
      - ./:/usr/src/app  
    ports:  
      - 5173:5173    container_name: hello-front-dev
```

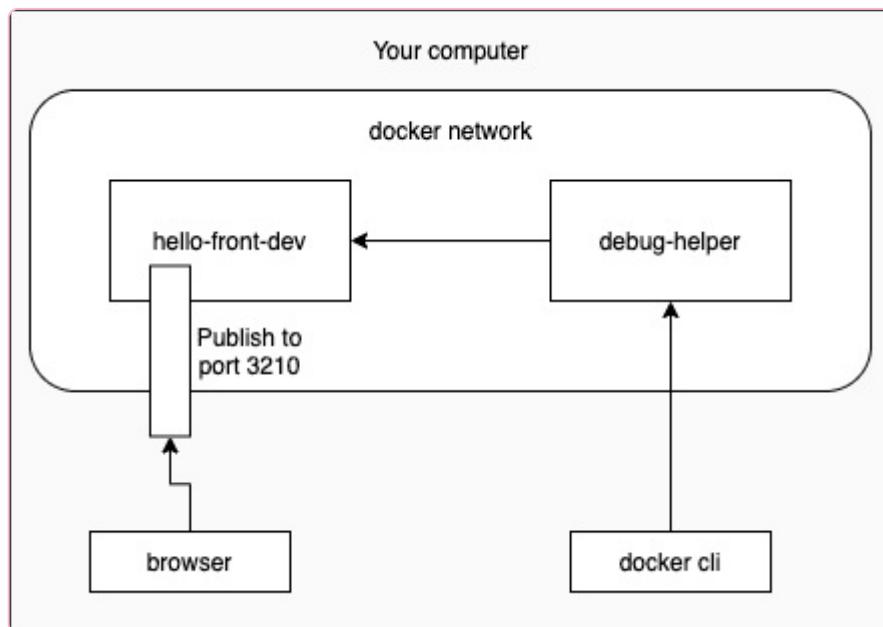
```
- 3210:5173    container_name: hello-front-dev
debug-helper:
  image: busybox
```

With *docker compose up* the application is available in <http://localhost:3210> at the *host machine*, but the command

```
docker compose -f docker-compose.dev.yml run debug-helper wget -O - http://app:5173
```

works still since the port is still 5173 within the docker network.

The below image illustrates what happens. The command *docker compose run* asks *debug-helper* to send the request within the network. While the browser in the host machine sends the request from outside of the network.



Now that you know how easy it is to find other services in the *docker-compose.yml* and we have nothing to debug we can remove the *debug-helper* and revert the ports to 5173:5173 in our compose file.

Exercise 12.16

Exercise 12.16: Run todo-backend in a development container

Use volumes and Nodemon to enable the development of the todo app backend while it is running *inside* a container. Create a *todo-backend/dev.Dockerfile* and edit the *todo-backend/docker-compose.dev.yml*.

You will also need to rethink the connections between backend and MongoDB / Redis. Thankfully Docker Compose can include environment variables that will be passed to the application:

```
services:
  server:
    image: ...
```

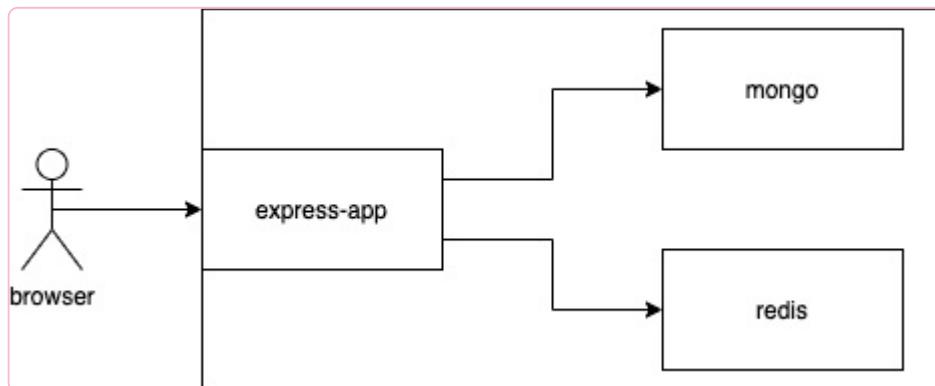
```

volumes:
  - ...
ports:
  - ...
environment:
  - REDIS_URL=redisurl_here
  - MONGO_URL=mongourl_here

```

The URLs are purposefully wrong, you will need to set the correct values. Remember to *look all the time what happens in console*. If and when things blow up, the error messages hint at what might be broken.

Here is a possibly helpful image illustrating the connections within the docker network:



Communications between containers in a more ambitious environment

Next, we will configure a [reverse proxy](#) to our docker-compose.dev.yml. According to wikipedia

A reverse proxy is a type of proxy server that retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client, appearing as if they originated from the reverse proxy server itself.

So in our case, the reverse proxy will be the single point of entry to our application, and the final goal will be to set both the React frontend and the Express backend behind the reverse proxy.

There are multiple different options for a reverse proxy implementation, such as Traefik, Caddy, Nginx, and Apache (ordered by initial release from newer to older).

Our pick is [Nginx](#).

Let us now put the *hello-frontend* behind the reverse proxy.

Create a file *nginx.dev.conf* in the project root and take the following template as a starting point. We will need to do minor edits to have our application running:

```

# events is required, but defaults are ok
events { }

```

```

# A http server, listening at port 80
http {
    server {
        listen 80;

        # Requests starting with root (/) are handled
        location / {
            # The following 3 lines are required for the hot loading to work (websocket).
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';

            # Requests are directed to http://localhost:5173
            proxy_pass http://localhost:5173;
        }
    }
}

```

Note we are using the familiar naming convention also for Nginx, *nginx.dev.conf* for development configurations, and the default name *nginx.conf* otherwise.

Next, create an Nginx service in the *docker-compose.dev.yml* file. Add a volume as instructed in the Docker Hub page where the right side is *:/etc/nginx/nginx.conf:ro*, the final ro declares that the volume will be *read-only*:

```

services:
  app:
    # ...
  nginx:
    image: nginx:1.20.1
    volumes:
      - ./nginx.dev.conf:/etc/nginx/nginx.conf:ro
    ports:
      - 8080:80
    container_name: reverse-proxy
    depends_on:
      - app # wait for the frontend container to be started

```

with that added, we can run *docker compose -f docker-compose.dev.yml up* and see what happens.

```

$ docker container ls
CONTAINER ID   IMAGE          COMMAND   PORTS           NAMES
a02ae58f3e8d   nginx:1.20.1   ...       0.0.0.0:8080->80/tcp   reverse-proxy
5ee0284566b4   hello-front-dev   ...       0.0.0.0:5173->5173/tcp   hello-front-dev

```

Connecting to <http://localhost:8080> will lead to a familiar-looking page with 502 status.

This is because directing requests to <http://localhost:5173> leads to nowhere as the Nginx container does not have an application running in port 5173. By definition, localhost refers to the current computer used to access it. Since the localhost is unique for each container, it always points to the

container itself.

Let's test this by going inside the Nginx container and using curl to send a request to the application itself. In our usage curl is similar to wget, but won't need any flags.

```
$ docker exec -it reverse-proxy bash  
  
root@374f9e62bfa8:/# curl http://localhost:80  
<html>  
<head><title>502 Bad Gateway</title></head>  
...  

```

To help us, Docker Compose has set up a network when we ran `docker compose up`. It has also added all of the containers mentioned in the `docker-compose.dev.yml` to the network. A DNS makes sure we can find the other containers in the network. The containers are each given two names: the service name and the container name and both can be used to communicate with a container.

Since we are inside the container, we can also test the DNS! Let's curl the service name (app) in port 5173

```
root@374f9e62bfa8:/# curl http://app:5173  
<!doctype html>  
<html lang="en">  
  <head>  
    <script type="module" src="/@vite/client"></script>  
    <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>  
  </head>  
  <body>  
    <div id="root"></div>  
    <script type="module" src="/src/main.jsx"></script>  
  </body>  
</html>
```

That is it! Let's replace the proxy_pass address in `nginx.dev.conf` with that one.

One more thing: we added an option `depends_on` to the configuration that ensures that the `nginx` container is not started before the frontend container `app` is started:

```
services:  
  app:  
    # ...  
  nginx:  
    image: nginx:1.20.1  
    volumes:  
      - ./nginx.conf:/etc/nginx/nginx.conf:ro  
    ports:  
      - 8080:80  
    container_name: reverse-proxy
```

```
depends_on: - app
```

If we do not enforce the starting order with `depends_on` there is a risk that Nginx fails on startup since it tries to resolve all DNS names that are referred in the config file:

```
http {  
    server {  
        listen 80;  
  
        location / {  
            proxy_http_version 1.1;  
            proxy_set_header Upgrade $http_upgrade;  
            proxy_set_header Connection 'upgrade';  
  
            proxy_pass http://app:5173;        }  
    }  
}
```

Note that `depends_on` does not guarantee that the service in the depended container is ready for action, it just ensures that the container has been started (and the corresponding entry is added to DNS). If a service needs to wait another service to become ready before the startup, [other solutions](#) should be used.

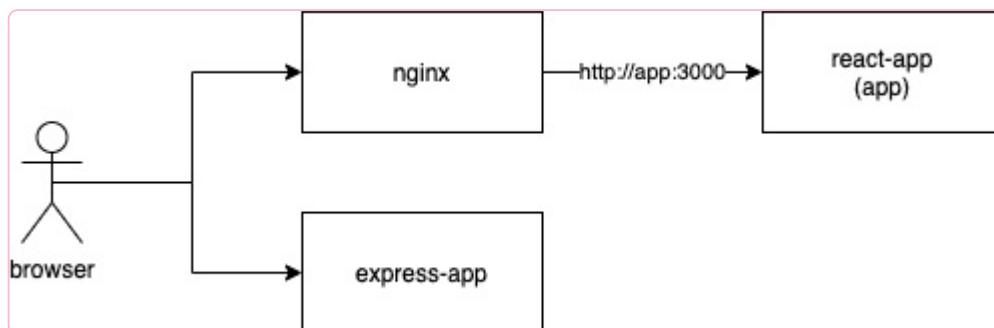
Exercises 12.17. - 12.19.

Exercise 12.17: Set up an Nginx reverse proxy server in front of todo-frontend

We are going to put the Nginx server in front of both todo-frontend and todo-backend. Let's start by creating a new docker-compose file `todo-app/docker-compose.dev.yml` and `todo-app/nginx.dev.conf`.

```
todo-app  
|--- todo-frontend  
|--- todo-backend  
|--- nginx.dev.conf |--- docker-compose.dev.yml
```

Add the services Nginx and todo-frontend built with `todo-app/todo-frontend/dev.Dockerfile` into the `todo-app/docker-compose.dev.yml`.



In this and the following exercises you **do not** need to support the build option, that is, the command

```
docker compose -f docker-compose.dev.yml up --build
```

It is enough to build the frontend and backend at their own repositories.

Exercise 12.18: Configure the Nginx server to be in front of todo-backend

Add the service todo-backend to the docker-compose file *todo-app/docker-compose.dev.yml* in development mode.

Add a new location to the *nginx.dev.conf* so that requests to */api* are proxied to the backend. Something like this should do the trick:

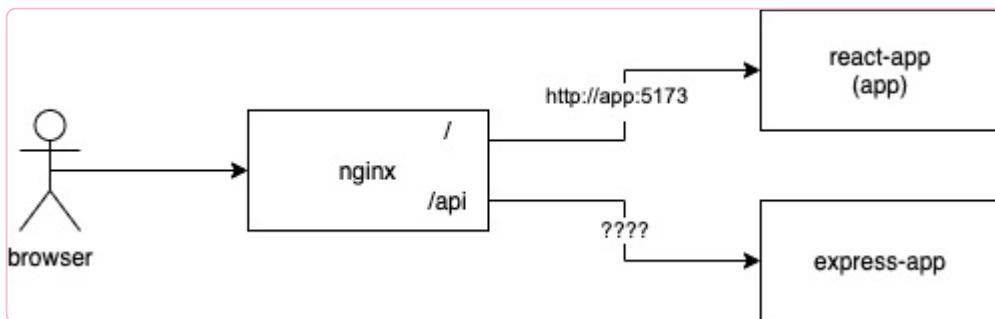
```
server {  
    listen 80;  
  
    # Requests starting with root (/) are handled  
    location / {  
        proxy_http_version 1.1;  
        proxy_set_header Upgrade $http_upgrade;  
        proxy_set_header Connection 'upgrade';  
  
        proxy_pass ...  
    }  
  
    # Requests starting with /api/ are handled  
    location /api/ {  
        proxy_pass ...  
    }  
}
```

The *proxy_pass* directive has an interesting feature with a trailing slash. As we are using the path */api* for location but the backend application only answers in paths */* or */todos* we will want the */api* to be removed from the request. In other words, even though the browser will send a GET request to */api/todos/1* we want the Nginx to proxy the request to */todos/1*. Do this by adding a trailing slash */* to the URL at the end of *proxy_pass*.

This is a [common issue](#)

- 4 Ah, I was missing the trailing slash :(– [Vanuan](#) May 2 '16 at 23:54
- 8 AAARGH! TRAILING SLASH ! – [barrymac](#) May 26 '17 at 16:27
- 6 3 hours of searching, and yeah... It was the trailing slash. Thanks mate! – [Lucas P.](#) Oct 15 '18 at 16:09

This illustrates what we are looking for and may be helpful if you are having trouble:

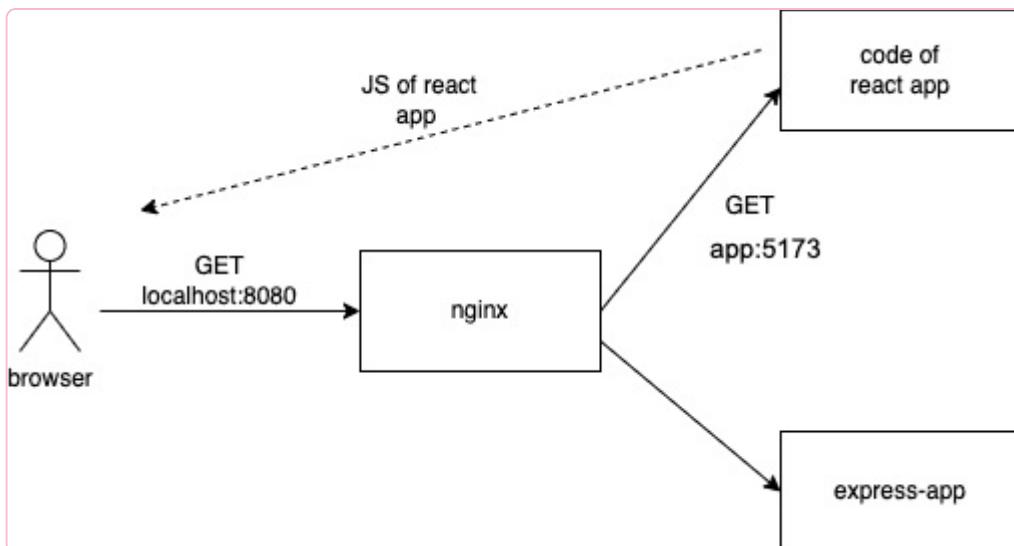


Exercise 12.19: Connect the services, todo-frontend with todo-backend

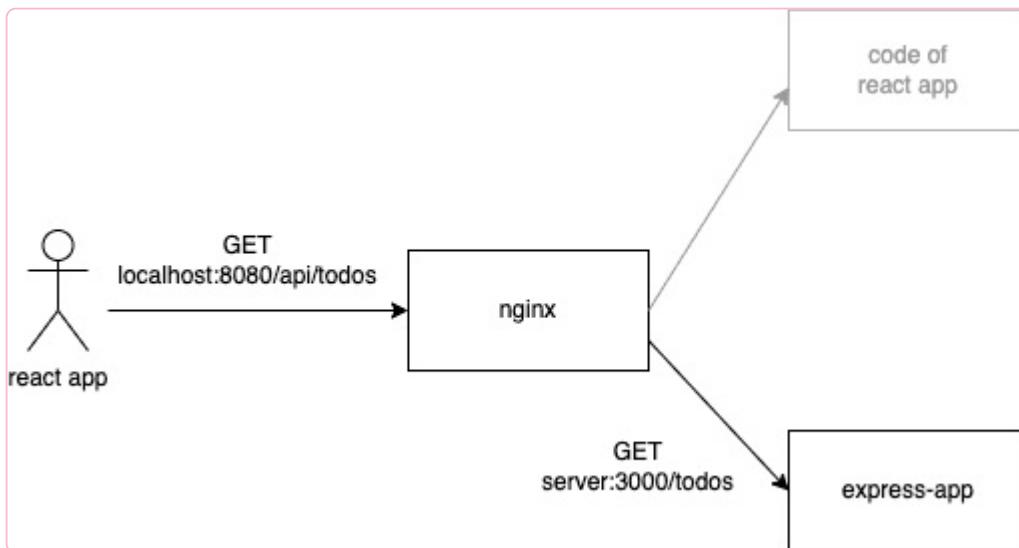
In this exercise, submit the entire development environment, including both Express and React applications, dev.Dockerfiles and docker-compose.dev.yml.

Finally, it is time to put all the pieces together. Before starting, it is essential to understand *where* the React app is actually run. The above figure might give the impression that React app is run in the container but it is totally wrong.

It is just the *React app source code* that is in the container. When the browser hits the address <http://localhost:8080> (assuming that you set up Nginx to be accessed in port 8080), the React source code gets downloaded from the container to the browser:



Next, the browser starts executing the React app, and all the requests it makes to the backend should be done through the Nginx reverse proxy:



The frontend container is actually no more accessed beyond the first request that gets the React app source code to the browser.

Now set up your app to work as depicted in the above figure. Make sure that the todo-frontend works with todo-backend. It will require changes to the `VITE_BACKEND_URL` environmental variable in the frontend.

Make sure that the development environment is now fully functional, that is:

- all features of the todo app work
- you can edit the source files *and* the changes take effect by reloading the app
- frontend should access the backend through Nginx, so the requests should be done to <http://localhost:8080/api/todos>:

The screenshot shows a browser window with the URL `http://localhost:8080`. The page displays a 'Todos' application with two items: 'Write code' (marked as 'This todo is done') and 'Learn about containers' (with 'Delete' and 'Set as done' buttons). Below the page, the browser's developer tools Network tab is open, showing a request for `/api/todos` with a status of 200 OK. The request method is GET.

Note that your app should work even if no [exposed port](#) are defined for the backend and frontend in the docker compose file:

```

services:
  app:
    image: todo-front-dev
    volumes:
      - ./todo-frontend/:/usr/src/app
    # no ports here!

  server:
    image: todo-back-dev
    volumes:
      - ./todo-backend/:/usr/src/app
    environment:
      - ...
    # no ports here!

  nginx:
    image: nginx:1.20.1
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - 8080:80 # this is needed
    container_name: reverse-proxy
    depends_on:
      - app

```

We just need to expose the Nginx port to the host machine since the access to the backend and frontend is proxied to the right container port by Nginx. Because Nginx, frontend and backend are defined in the same Docker compose configuration, Docker puts those to the same [Docker network](#) and thanks to that, Nginx has direct access to frontend and backend containers ports.

Tools for Production

Containers are fun tools to use in development, but the best use case for them is in the production environment. There are many more powerful tools than Docker Compose to run containers in production.

Heavyweight container orchestration tools like [Kubernetes](#) allow us to manage containers on a completely new level. These tools hide away the physical machines and allow us, the developers, to worry less about the infrastructure.

If you are interested in learning more in-depth about containers come to the [DevOps with Docker](#) course and you can find more about Kubernetes in the advanced 5 credit [DevOps with Kubernetes](#) course. You should now have the skills to complete both of them!

Exercises 12.20.-12.22.

Exercise 12.20:

Create a production *todo-app/docker-compose.yml* with all of the services, Nginx, todo-backend, todo-frontend, MongoDB and Redis. Use Dockerfiles instead of *dev.Dockerfiles* and make sure to start the applications in production mode.

Please use the following structure for this exercise:

```
todo-app
├── todo-frontend
├── todo-backend
├── nginx.dev.conf
├── docker-compose.dev.yml
└── nginx.conf └── docker-compose.yml
```

Exercise 12.21:

Create a similar containerized development environment of one of *your own* full stack apps that you have created during the course or in your free time. You should structure the app in your submission repository as follows:

```
└── my-app
    ├── frontend
    │   └── dev.Dockerfile
    ├── backend
    │   └── dev.Dockerfile
    ├── nginx.dev.conf
    └── docker-compose.dev.yml
```

Exercise 12.22:

Finish this part by creating a containerized *production setup* of your own full stack app. Structure the app in your submission repository as follows:

```
└── my-app
    ├── frontend
    │   ├── dev.Dockerfile
    │   └── Dockerfile
    ├── backend
    │   ├── dev.Dockerfile
    │   └── Dockerfile
    ├── nginx.dev.conf
    ├── nginx.conf
    ├── docker-compose.dev.yml
    └── docker-compose.yml
```

Submitting exercises and getting the credits

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

Exercises of this part are submitted just like in the previous parts, but unlike parts 0 to 7, the submission goes to an own [course instance](#). Remember that you have to finish *all the exercises* to pass this part!

Once you have completed the exercises and want to get the credits, let us know through the exercise

submission system that you have completed the course:

My submissions

part	exercises	hours	github	comment	solutio
1	22	29	https://github.com/Kaltsoon/fs-cicd		show
total	22	29			

credits 1 based on exercises

Certificate  

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

Note that you need a registration to the corresponding course part for getting the credits registered, see [here](#) for more information.

You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the certificate's language.

13: Using relational databases

13a: Using relational databases with Sequelize

In this section we will explore node applications that use relational databases. During the section we will build a Node-backend using a relational database for a familiar note application from sections 3-5. To complete this part, you will need a reasonable knowledge of relational databases and SQL. There are many online courses on SQL databases, eg. [SQLbolt](#) and [Intro to SQL by Khan Academy](#).

There are 24 exercises in this part, and you need to complete each exercise for completing the course. Exercises are submitted via the [submissions system](#) just like in the previous parts, but unlike parts 0 to 7, the submission goes to a different "course instance".

Advantages and disadvantages of document databases

We have used the MongoDB database in all the previous sections of the course. Mongo is a [document database](#) and one of its most characteristic features is that it is *schemaless*, i.e. the database has only a very limited awareness of what kind of data is stored in its collections. The schema of the database exists only in the program code, which interprets the data in a specific way, e.g. by identifying that some of the fields are references to objects in another collection.

In the example application of parts 3 and 4, the database stores notes and users.

A collection of *notes* that stores notes looks like the following:

```
[  
  {  
    "_id": "600c0e410d10256466898a6c",  
    "content": "HTML is easy"
```

```

"date": "2021-01-23T11:53:37.292+00:00",
"important": false
"__v": 0
],
{
  "_id": "600c0edde86c7264ace9bb78",
  "content": "CSS is hard",
  "date": "2021-01-23T11:56:13.912+00:00",
  "important": true
  "__v": 0
},
]

```

Users saved in the `users` collection looks like the following:

```

[
  {
    "_id": "600c0e410d10256466883a6a",
    "username": "mluukkai",
    "name": "Matti Luukkainen",
    "passwordHash": "$2b$10$Df1yYJRiQuu3Sr4tUrk.SerVz1JKtBHLBOARfY0PBn/Uo7qr80cou",
    "__v": 9,
    "notes": [
      "600c0edde86c7264ace9bb78",
      "600c0e410d10256466898a6c"
    ]
  }
]

```

MongoDB does know the types of the fields of the stored entities, but it has no information about which collection of entities the user record ids are referring to. MongoDB also does not care what fields the entities stored in the collections have. Therefore MongoDB leaves it entirely up to the programmer to ensure that the correct information is being stored in the database.

There are both advantages and disadvantages to not having a schema. One of the advantages is the flexibility that schema agnosticism brings: since the schema does not need to be defined at the database level, application development may be faster in certain cases, and easier, with less effort needed in defining and modifying the schema in any case. Problems with not having a schema are related to error-proneness: everything is left up to the programmer. The database itself has no way of checking whether the data in it is *honest*, i.e. whether all mandatory fields have values, whether the reference type fields refer to existing entities of the right type in general, etc.

The relational databases that are the focus of this section, on the other hand, lean heavily on the existence of a schema, and the advantages and disadvantages of schema databases are almost the opposite compared of the non-schema databases.

The reason why the previous sections of the course used MongoDB is precisely because of its schema-less nature, which has made it easier to use the database for someone with little knowledge of relational databases. For most of the use cases of this course, I personally would have chosen to use a relational database.

Application database

For our application we need a relational database. There are many options, but we will be using the currently most popular Open Source solution [PostgreSQL](#). You can install Postgres (as the database is often called) on your machine, if you wish to do so. An easier option would be using Postgres as a cloud service, e.g. [ElephantSQL](#).

However, we will be taking advantage of the fact that it is possible to create a Postgres database for the application on the Fly.io and Heroku cloud service platforms, which are familiar from the parts 3 and 4.

In the theory material of this section, we will be building a Postgres-enabled version from the backend of the notes-storage application, which was built in sections 3 and 4.

Since we don't need any database in the cloud in this part (we only use the application locally), there is a possibility to use the lessons of the course [part 12](#) and use Postgres locally with Docker. After the Postgres instructions for cloud services, we also give a short instruction on how to easily get Postgres up and running with Docker.

Fly.io

Let us create a new Fly.io-app by running the command `fly launch` in a directory where we shall add the code of the app. Let us also create the Postgres database for the app:

```
→ luento fly launch
Update available 0.0.382 -> v0.0.388.
Run "fly version update" to upgrade.
Creating app in /Users/mluukkai/dev/full-stack-psql/luento
Scanning source code
Detected a NodeJS app
Using the following build configuration:
  Builder: heroku/buildpacks:20
? App Name (leave blank to use an auto-generated name): fs-psql-lecture
Automatically selected personal organization: Matti Luukkainen
? Select region: fra (Frankfurt, Germany)
Created app fs-psql-lecture in organization personal
Wrote config file fly.toml
? Would you like to set up a Postgresql database now? Yes
For pricing information visit: https://fly.io/docs/about/pricing/#postgresql-clusters
? Select configuration: Development - Single node, 1x shared CPU, 256MB RAM, 1GB disk
Creating postgres cluster fs-psql-lecture-db in organization personal
Postgres cluster fs-psql-lecture-db created
  Username: postgres
  Password: ec57 [REDACTED] 9e9b
  Hostname: fs-psql-lecture-db.internal
  Proxy Port: 5432
  PG Port: 5433
```

When creating the app, Fly.io reveals the password of the database that will be needed when

connecting the app to the database. *This is the only time it is shown in plain text so it is essential to save it somewhere* (but not in any public place such as GitHub).

Note that if you only need the database, and are not planning to deploy the app to Fly.io, it is also possible to [just create the database to Fly.io](#).

A psql console connection to the database can be opened as follows

```
flyctl postgres connect -a <app_name-db>
```

in my case the app name is *fs-psql-lecture* so the command is the following:

```
flyctl postgres connect -a fs-psql-lecture-db
```

Heroku

If Heroku is used, a new Heroku application is created when inside a suitable directory. After that a database is added to the app:

```
heroku create  
# Returns an app-name for the app you just created in heroku.  
  
heroku addons:create heroku-postgresql:hobby-dev -a <app-name>
```

We can use the *heroku config* command to get the *connect string*, which is required to connect to the database:

```
heroku config -a <app-name>  
==== cryptic-everglades-76708 Config Vars  
DATABASE_URL: postgres://<username>:the password is here@<host-of-postgres-addon>:5  
432/<db-name>
```

The database can be accessed by running *psql* command on the Heroku server as follows (note that the command parameters depend on the connection url of the Heroku database):

```
heroku run psql -h <host-of-postgres-addon> -p 5432 -U <username> <dbname> -a <app-name>
```

The command asks the password and opens the psql console:

```
Password for user <username>:  
psql (13.4 (Ubuntu 13.4-1.pgdg20.04+1))  
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)  
Type "help" for help.  
  
postgres=#
```

Docker

This instruction assumes that you master the basic use of Docker to the extent taught by e.g. [part 12](#).

Start Postgres [Docker image](#) with the command

```
docker run -e POSTGRES_PASSWORD=mysecretpassword -p 5432:5432 postgres
```

A psql console connection to the database can be opened using the *docker exec* command. First you need to find out the id of the container:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
PORTS             NAMES
ff3f49eadf27      postgres            "docker-entrypoint.s...
s   0.0.0.0:5432->5432/tcp      great_raman
docker exec -it ff3f49eadf27 psql -U postgres postgres
psql (15.2 (Debian 15.2-1.pgdg110+1))
Type "help" for help.

postgres=#
```

Defined in this way, the data stored in the database is persisted only as long as the container exists. The data can be preserved by defining a [volume](#) for the data, see more [here](#).

Using the psql console

Particularly when using a relational database, it is essential to access the database directly as well. There are many ways to do this, there are several different graphical user interfaces, such as [pgAdmin](#). However, we will be using Postgres [psql](#) command-line tool.

When the console is opened, let's try the main psql command `\d`, which tells you the contents of the database:

```
Password for user <username>:
psql (13.4 (Ubuntu 13.4-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=# \d
Did not find any relations.
```

As you might guess, there is currently nothing in the database.

Let's create a table for notes:

```
CREATE TABLE notes (
    id SERIAL PRIMARY KEY,
    content text NOT NULL,
```

```
important boolean;
date time;
```

A few points: column *id* is defined as a *primary key*, which means that the value in the column *id* must be unique for each row in the table and the value must not be empty. The type for this column is defined as **SERIAL**, which is not the actual type but an abbreviation for an integer column to which Postgres automatically assigns a unique, increasing value when creating rows. The column named *content* with type **text** is defined in such a way that it must be assigned a value.

Let's look at the situation from the console. First, the **\d** command, which tells us what tables are in the database:

```
postgres=# \d
      List of relations
 Schema |     Name      |   Type   | Owner
-----+-----+-----+-----+
 public | notes       | table   | username
 public | notes_id_seq | sequence | username
(2 rows)
```

In addition to the *notes* table, Postgres created a subtable called *notes_id_seq*, which keeps track of what value is assigned to the *id* column when creating the next note.

With the command **\d notes**, we can see how the *notes* table is defined:

```
postgres=# \d notes;
           Table "public.notes"

 Column |          Type          | Collation | Nullable | Default
-----+-----+-----+-----+-----+
 id    | integer            |           | not null | nextval('notes_id_s
eq'::regclass)
 content | text              |           | not null |
 important | boolean           |           |
 date   | time without time zone |           |

Indexes:
 "notes_pkey" PRIMARY KEY, btree (id)
```

Therefore the column *id* has a default value, which is obtained by calling the internal function of Postgres *nextval*.

Let's add some content to the table:

```
insert into notes (content, important) values ('Relational databases rule the wo
rld', true);
```

```
insert into notes (content, important) values ('MongoDB is webscale', false);
```

And let's see what the created content looks like:

```
postgres=# select * from notes;
 id |          content          | important | date
----+-----+-----+-----+
  1 | relational databases rule the world | t         |
  2 | MongoDB is webscale           | f         |
(2 rows)
```

If we try to store data in the database that is not according to the schema, it will not succeed. The value of a mandatory column cannot be missing:

```
postgres=# insert into notes (important) values (true);
ERROR: null value in column "content" of relation "notes" violates not-null constraint
DETAIL: Failing row contains (9, null, t, null).
```

The column value cannot be of the wrong type:

```
postgres=# insert into notes (content, important) values ('only valid data can be saved', 1);
ERROR: column "important" is of type boolean but expression is of type integer
LINE 1: ...tent, important) values ('only valid data can be saved', 1); ^
```

Columns that don't exist in the schema are not accepted either:

```
postgres=# insert into notes (content, important, value) values ('only valid data can be saved', true, 10);
ERROR: column "value" of relation "notes" does not exist
LINE 1: insert into notes (content, important, value) values ('only ...
```

Next it's time to move on to accessing the database from the application.

Node application using a relational database

Let's start the application as usual with the `npm init` and install `nodemon` as a development dependency and also the following runtime dependencies:

```
npm install express dotenv pg Sequelize
```

Of these, the latter `Sequelize` is the library through which we use Postgres. Sequelize is a so-called [Object relational mapping](#) (ORM) library that allows you to store JavaScript objects in a relational database without using the SQL language itself, similar to Mongoose that we used with MongoDB.

Let's test that we can connect successfully. Create the file `index.js` and add the following content:

```

require('dotenv').config()
const { Sequelize } = require('sequelize')

const sequelize = new Sequelize(process.env.DATABASE_URL)

const main = async () => [
  try {
    await sequelize.authenticate()
    console.log('Connection has been established successfully.')
    sequelize.close()
  } catch (error) {
    console.error('Unable to connect to the database:', error)
  }
]

main()

```

Note: if you use Heroku, you might need an extra option in connecting the database

```

const sequelize = new Sequelize(process.env.DATABASE_URL, {
  dialectOptions: {
    ssl: {
      require: true,
      rejectUnauthorized: false
    }
  }
})

```

The database *connect string*, that contains the database address and the credentials must be defined in the file *.env*

If Heroku is used, the connect string can be seen by using the command *heroku config*. The contents of the file *.env* should be something like the following:

```

$ cat .env
DATABASE_URL=postgres://<username>:thepasswordishere@ec2-54-83-137-206.compute-1
.amazonaws.com:5432/<databaseName>

```

When using Fly.io, the local connection to the database should first be enabled by [tunneling](#) the localhost port 5432 to the Fly.io database port using the following command

```
flyctl proxy 5432 -a <app-name>-db
```

in my case the command is

```
flyctl proxy 5432 -a fs-psql-lecture-db
```

The command must be left running while the database is used. So do not close the console!

The Fly.io connect-string is of the form

```

$ cat .env
DATABASE_URL=postgres://postgres:thepasswordishere@127.0.0.1:5432/postgres

```

Password was shown when the database was created, so hopefully you have not lost it!

The last part of the connect string, *postgres* refers to the database name. The name could be any string but we use here *postgres* since it is the default database that is automatically created within a Postgres database. If needed, new databases can be created with the command [CREATE DATABASE](#).

If you use Docker, the connect string is:

```
DATABASE_URL=postgres://postgres:mysecretpassword@localhost:5432/postgres
```

Once the connect string has been set up in the file *.env* we can test for a connection:

```
$ node index.js
Executing [default]: SELECT 1+1 AS result
Connection has been established successfully.
```

If and when the connection works, we can then run the first query. Let's modify the program as follows:

```
require('dotenv').config()
const { Sequelize, QueryTypes } = require('sequelize')
const sequelize = new Sequelize(process.env.DATABASE_URL, {
  dialectOptions: [
    {
      ssl: {
        require: true,
        rejectUnauthorized: false
      }
    },
  ],
})

const main = async () => {
  try {
    await sequelize.authenticate()
    const notes = await sequelize.query("SELECT * FROM notes", { type: QueryTypes
      .SELECT })
    console.log(notes)
    sequelize.close() } catch (error) {
    console.error('Unable to connect to the database:', error)
  }
}

main()
```

Executing the application should print as follows:

```
Executing [default]: SELECT * FROM notes
[
  {
    id: 1,
    content: 'Relational databases rule the world',
    important: true,
```

```

    date: null
  },
  {
    id: 2,
    content: 'MongoDB is webscale',
    important: false,
    date: null
  }
]

```

Even though Sequelize is an ORM library, which means there is little need to write SQL yourself when using it, we just used [direct SQL](#) with the sequelize method `query`.

Since everything seems to be working, let's change the application into a web application.

```

require('dotenv').config()
const { Sequelize, QueryTypes } = require('sequelize')
const express = require('express')
const app = express()
const sequelize = new Sequelize(process.env.DATABASE_URL, [
  dialectOptions: [
    ssl: [
      require: true,
      rejectUnauthorized: false
    ]
  ],
})
;

app.get('/api/notes', async [req, res] => {
  const notes = await sequelize.query(
    "SELECT * FROM notes", { type: QueryTypes.SELECT })
  res.json(notes)
})
const PORT = process.env.PORT || 3001
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
})

```

The application seems to be working. However, let's now switch to using Sequelize instead of SQL as it is intended to be used.

Model

When using Sequelize, each table in the database is represented by a [model](#), which is effectively its own JavaScript class. Let's now define the model *Note* corresponding to the table *notes* for the application by changing the code to the following format:

```

require('dotenv').config()
const { Sequelize, Model, DataTypes } = require('sequelize')
const express = require('express')
const app = express()

const sequelize = new Sequelize(process.env.DATABASE_URL, [
  dialectOptions: [
    ssl: [
      require: true
    ]
  ]
])
;
```

```

        rejectUnauthorized: false
    }
},
})

class Note extends Model {
    Note.init({
        id: {
            type: DataTypes.INTEGER,
            primaryKey: true,
            autoIncrement: true
        },
        content: {
            type: DataTypes.TEXT,
            allowNull: false
        },
        important: {
            type: DataTypes.BOOLEAN
        },
        date: {
            type: DataTypes.DATE
        }
    }, {
        sequelize,
        underscored: true,
        timestamps: false,
        modelName: 'note'
    })
}

app.get('/api/notes', async (req, res) => {
    const notes = await Note.findAll()
    res.json(notes)
})

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

```

A few comments on the code: There is nothing very surprising about the `Note` definition of the model, each column has a type defined, as well as other properties if necessary, such as whether it is the main key of the table. The second parameter in the model definition contains the `sequelize` attribute as well as other configuration information. We also defined that the table does not have to use the timestamps columns (`created_at` and `updated_at`).

We also defined `underscored: true`, which means that table names are derived from model names as plural `snake case` versions. Practically this means that, if the name of the model, as in our case is "Note", then the name of the corresponding table is its plural version written with a lower case initial letter, i.e. `notes`. If, on the other hand, the name of the model would be "two-part", e.g. `StudyGroup`, then the name of the table would be `study_groups`. Sequelize automatically infers table names, but also allows explicitly defining them.

The same naming policy applies to columns as well. If we had defined that a note is associated with `creationYear`, i.e. information about the year it was created, we would define it in the model as follows:

```

Note.init({
    // ...
    creationYear: {
        type: DataTypes.INTEGER
    }
})

```

The name of the corresponding column in the database would be `creation_year`. In code, reference to the column is always in the same format as in the model, i.e. in "camel case" format.

We have also defined `modelName: 'note'`, the default "model name" would be capitalized `Note`. However we want to have a lowercase initial, it will make a few things a bit more convenient going forward.

The database operation is easy to do using the [query interface](#) provided by models, the method

`findAll` works exactly as it is assumed by its name to work:

```
app.get('/api/notes', async [req, res] => {
  const notes = await Note.findAll()
  res.json(notes)
})
```

The console tells you that the method call `Note.findAll()` causes the following query:

```
Executing [default]: SELECT "id", "content", "important", "date" FROM "notes" AS "note";
```

Next, let's implement an endpoint for creating new notes:

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

// ...

app.post('/api/notes', async [req, res] => {
  console.log(req.body)
  const note = await Note.create(req.body)
  res.json(note)
})
```

Creating a new note is done by calling the model's `Note` method `create` and passing as a parameter an object that defines the values of the columns.

Instead of the `create` method, it [is also possible](#) to save to a database using the `build` method first to create a Model-object from the desired data, and then calling the `save` method on it:

```
const note = Note.build(req.body)
await note.save()
```

Calling the `build` method does not save the object in the database yet, so it is still possible to edit the object before the actual save event:

```
const note = Note.build(req.body)
note.important = true
await note.save()
```

For the use case of the example code, the `create` method is better suited, so let's stick to that.

If the object being created is not valid, there is an error message as a result. For example, when trying to create a note without content, the operation fails, and the console reveals the reason to be `SequelizeValidationError: notNull Violation Note.content cannot be null`:

```
(node:39109) UnhandledPromiseRejectionWarning: SequelizeValidationError: notNull
Violation: Note.content cannot be null
    at InstanceValidator._validate (/Users/mluukkai/opetus/fs-psql/node_modules/
sequelize/lib/instance-validator.js:78:13)
```

```
at processTicksAndRejections (internal/process/task_queues.js:93:5)
```

Let's add some simple error handling when adding a new note:

```
app.post('/api/notes', async (req, res) => {
  try {
    const note = await Note.create(req.body)
    return res.json(note)
  } catch(error) {
    return res.status(400).json({ error })
  }
})
```

Exercises 13.1.-13.3.

In the tasks of this section, we will build a blog application backend similar to the tasks in [section 4](#), which should be compatible with the frontend in [section 5](#) except for error handling. We will also add various features to the backend that the frontend in section 5 will not know how to use.

Exercise 13.1.

Create a GitHub repository for the application and create a new Fly.io or Heroku application for it, as well as a Postgres database. As mentioned [here](#) you might set up your database also somewhere else, and in that case the Fly.io or Heroku app is not needed.

Make sure you are able to establish a connection to the application database.

Exercise 13.2.

On the command-line, create a *blogs* table for the application with the following columns:

- id (unique, incrementing id)
- author (string)
- url (string that cannot be empty)
- title (string that cannot be empty)
- likes (integer with default value zero)

Add at least two blogs to the database.

Save the SQL-commands you used at the root of the application repository in a file called *commands.sql*

Exercise 13.3.

Create a functionality in your application which prints the blogs in the database on the command-line, e.g. as follows:

```
$ node cli.js
Executing [default]: SELECT * FROM blogs
Dan Abramov: 'On let vs const', 0 likes
Laurenz Albe: 'Gaps in sequences in PostgreSQL', 0 likes
```

Creating database tables automatically

Our application now has one unpleasant side, it assumes that a database with exactly the right schema exists, i.e. that the table *notes* has been created with the appropriate *create table* command.

Since the program code is being stored on GitHub, it would make sense to also store the commands that create the database in the context of the program code, so that the database schema is definitely the same as what the program code is expecting. Sequelize is actually able to generate a schema automatically from the model definition by using the models method [sync](#).

Let's now destroy the database from the console by entering the following command:

```
drop table notes;
```

The `\d` command reveals that the table has been lost from the database:

```
postgres=# \d
Did not find any relations.
```

The application no longer works.

Let's add the following command to the application immediately after the model *Note* is defined:

```
Note.sync()
```

When the application starts, the following is printed on the console:

```
Executing (default): CREATE TABLE IF NOT EXISTS "notes" ("id" SERIAL , "content" TEXT NOT NULL, "important" BOOLEAN, "date" TIMESTAMP WITH TIME ZONE, PRIMARY KEY ("id"));
```

That is, when the application starts, the command *CREATE TABLE IF NOT EXISTS "notes"*... is executed which creates the table *notes* if it does not already exist.

Other operations

Let's complete the application with a few more operations.

Searching for a single note is possible with the method [findByPk](#), because it is retrieved based on the id of the primary key:

```
app.get('/api/notes/:id', async [req, res] => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    res.json(note)
  } else {
    res.status(404).end()
  }
})
```

```
})
```

Retrieving a single note causes the following SQL command:

```
Executing (default): SELECT "id", "content", "important", "date" FROM "notes" AS  
"note" WHERE "note". "id" = '1';
```

If no note is found, the operation returns *null*, and in this case the relevant status code is given.

Modifying the note is done as follows. Only the modification of the *important* field is supported, since the application's frontend does not need anything else:

```
app.put('/api/notes/:id', async [req, res] => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    note.important = req.body.important
    await note.save()
    res.json(note)
  } else {
    res.status(404).end()
  }
})
```

The object corresponding to the database row is retrieved from the database using the *findByPk* method, the object is modified and the result is saved by calling the *save* method of the object corresponding to the database row.

The current code for the application is in its entirety on [GitHub](#), branch *part13-1*.

Printing the objects returned by Sequelize to the console

The JavaScript programmer's most important tool is *console.log*, whose aggressive use gets even the worst bugs under control. Let's add console printing to the single note path:

```
app.get('/api/notes/:id', async [req, res] => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    console.log(note)      res.json(note)
  } else {
    res.status(404).end()
  }
})
```

We can see that the end result is not exactly what we expected:

```
note :  
  {  
    id: 1,  
    content: 'Notes are attached to a user',  
    important: true,  
    ...  
  }
```

```

date: 2021-10-03T15:00:24.582Z,
},
_previousDataValues: [
  id: 1,
  content: 'Notes are attached to a user',
  important: true,
  date: 2021-10-03T15:00:24.582Z,
},
_changed: Set[0] {},
_options: [
  isNewRecord: false,
  _schema: null,
  _schemaDelimiter: '',
  raw: true,
  attributes: [ 'id', 'content', 'important', 'date' ]
},
isNewRecord: false
}

```

In addition to the note information, all sorts of other things are printed on the console. We can reach the desired result by calling the model-object method [toJSON](#):

```

app.get('/api/notes/:id', async [req, res] => {
  const note = await Note.findById(req.params.id)
  if (note) {
    console.log(note.toJSON())      res.json(note)
  } else {
    res.status(404).end()
  }
})

```

Now the result is exactly what we want:

```

{
  id: 1,
  content: 'MongoDB is webscale',
  important: false,
  date: 2021-10-09T13:52:58.693Z
}

```

In the case of a collection of objects, the method `toJSON` does not work directly, the method must be called separately for each object in the collection:

```

app.get('/api/notes', async [req, res] => {
  const notes = await Note.findAll()

  console.log(notes.map(n=>n.toJSON()))
  res.json(notes)
})

```

The print looks like the following:

```
[ { id: 1,
  content: 'MongoDB is webscale',
  important: false,
  date: 2021-10-09T13:52:58.693Z },
{ id: 2,
  content: 'Relational databases rule the world',
  important: true,
  date: 2021-10-09T13:53:10.710Z } ]
```

However, perhaps a better solution is to turn the collection into JSON for printing by using the method [JSON.stringify](#):

```
app.get('/api/notes', async [req, res] => {
  const notes = await Note.findAll()

  console.log(JSON.stringify(notes))
  res.json(notes)
})
```

This way is better especially if the objects in the collection contain other objects. It is also often useful to format the objects on the screen in a slightly more reader-friendly format. This can be done with the following command:

```
console.log(JSON.stringify(notes, null, 2))
```

The print looks like the following:

```
[{
  {
    "id": 1,
    "content": "MongoDB is webscale",
    "important": false,
    "date": "2021-10-09T13:52:58.693Z"
  },
  {
    "id": 2,
    "content": "Relational databases rule the world",
    "important": true,
    "date": "2021-10-09T13:53:10.710Z"
  }
]
```

Exercise 13.4.

Exercise 13.4.

Transform your application into a web application that supports the following operations

- GET api/blogs (list all blogs)

- POST api/blogs (add a new blog)
- DELETE api/blogs/:id (delete a blog)

13b: Join tables and queries

Application structuring

So far, we have written all the code in the same file. Now let's structure the application a little better. Let's create the following directory structure and files:

```
index.js
util
  config.js
  db.js
models
  index.js
  note.js
controllers
  notes.js
```

The contents of the files are as follows. The file *util/config.js* takes care of handling the environment variables:

```
require('dotenv').config()

module.exports = [
  DATABASE_URL: process.env.DATABASE_URL,
  PORT: process.env.PORT || 3001,
]
```

The role of the file *index.js* is to configure and launch the application:

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

const { PORT } = require('./util/config')
const { connectToDatabase } = require('./util/db')

const notesRouter = require('./controllers/notes')

app.use(express.json())

app.use('/api/notes', notesRouter)

const start = async () => {
  await connectToDatabase()
  app.listen(PORT, () => [
    console.log(`Server running on port ${PORT}`)
  ])
}
```

```
start()
```

Starting the application is slightly different from what we have seen before, because we want to make sure that the database connection is established successfully before the actual startup.

The file *util/db.js* contains the code to initialize the database:

```
const Sequelize = require('sequelize')
const { DATABASE_URL } = require('../config')

const sequelize = new Sequelize(DATABASE_URL)

const connectToDatabase = async () => {
  try {
    await sequelize.authenticate()
    console.log('connected to the database')
  } catch (err) {
    console.log('failed to connect to the database')
    return process.exit(1)
  }
}

return null
}

module.exports = [ connectToDatabase, sequelize ]
```

The notes in the model corresponding to the table to be stored are saved in the file *models/note.js*

```
const { Model, DataTypes } = require('sequelize')

const { sequelize } = require('../util/db')

class Note extends Model {}

Note.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  content: {
    type: DataTypes.TEXT,
    allowNull: false
  },
  important: {
    type: DataTypes.BOOLEAN
  },
  date: {
    type: DataTypes.DATE
  }
})
```

```

}, [
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'note'
])

module.exports = Note

```

The file *models/index.js* is almost useless at this point, as there is only one model in the application. When we start adding other models to the application, the file will become more useful because it will eliminate the need to import files defining individual models in the rest of the application.

```

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

Note.sync()

module.exports = [
  Note
]

```

The route handling associated with notes can be found in the file *controllers/notes.js*:

```

const router = require('express').Router()

const { Note } = require('../models')

router.get('/', async (req, res) => {
  const notes = await Note.findAll()
  res.json(notes)
})

router.post('/', async (req, res) => {
  try {
    const note = await Note.create(req.body)
    res.json(note)
  } catch(error) {
    return res.status(400).json({ error })
  }
})

router.get('/:id', async (req, res) => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    res.json(note)
  } else {
    res.status(404).end()
  }
})

router.delete('/:id', async (req, res) => {

```

```

    const note = await Note.findById(req.params.id)
    if (note) {
      await note.destroy()
    }
    res.status(204).end()
  })
}

router.put('/:id', async [req, res] => {
  const note = await Note.findById(req.params.id)
  if (note) {
    note.important = req.body.important
    await note.save()
    res.json(note)
  } else {
    res.status(404).end()
  }
})

module.exports = router

```

The structure of the application is good now. However, we note that the route handlers that handle a single note contain a bit of repetitive code, as all of them begin with the line that searches for the note to be handled:

```
const note = await Note.findById(req.params.id)
```

Let's refactor this into our own *middleware* and implement it in the route handlers:

```

const noteFinder = async [req, res, next] => {
  req.note = await Note.findById(req.params.id)
  next()
}

router.get('/:id', noteFinder, async [req, res] => {
  if (req.note) {
    res.json(req.note)
  } else {
    res.status(404).end()
  }
})

router.delete('/:id', noteFinder, async [req, res] => {
  if (req.note) {
    await req.note.destroy()
  }
  res.status(204).end()
})

router.put('/:id', noteFinder, async [req, res] => {
  if (req.note) {
    req.note.important = req.body.important
  }
})

```

```
    await req.note.save()
    res.json(req.note)
  } else {
    res.status(404).end()
  }
})
```

The route handlers now receive *three* parameters, the first being a string defining the route and second being the middleware *noteFinder* we defined earlier, which retrieves the note from the database and places it in the *note* property of the *req* object. A small amount of copypaste is eliminated and we are satisfied!

The current code for the application is in its entirety on [GitHub](#), branch *part13-2*.

Exercises 13.5.-13.7.

Exercise 13.5.

Change the structure of your application to match the example above, or to follow some other similar clear convention.

Exercise 13.6.

Also, implement support for changing the number of a blog's likes in the application, i.e. the operation

PUT /api/blogs/:id (modifying the like count of a blog)

The updated number of likes will be relayed with the request:

```
{
  likes: 3
}
```

Exercise 13.7.

Centralize the application error handling in middleware as in [part 3](#). You can also enable middleware [express-async-errors](#) as we did in [part 4](#).

The data returned in the context of an error message is not very important.

At this point, the situations that require error handling by the application are creating a new blog and changing the number of likes on a blog. Make sure the error handler handles both of these appropriately.

User management

Next, let's add a database table *users* to the application, where the users of the application will be stored. In addition, we will add the ability to create users and token-based login as we implemented in [part 4](#). For simplicity, we will adjust the implementation so that all users will have the same password *secret*.

The model defining users in the file `models/user.js` is straightforward

```
const { Model, DataTypes } = require('sequelize')

const sequelize = require('../util/db')

class User extends Model {}

User.init([
  {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    username: {
      type: DataTypes.STRING,
      unique: true,
      allowNull: false
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false
    }
  },
  {
    sequelize,
    underscored: true,
    timestamps: false,
    modelName: 'user'
  }
])

module.exports = User
```

The `username` field is set to `unique`. The `username` could have basically been used as the primary key of the table. However, we decided to create the primary key as a separate field with integer value `id`.

The file `models/index.js` expands slightly:

```
const Note = require('./note')
const User = require('./user')

Note.sync()
User.sync()
module.exports = [
  Note, User]
```

The route handlers that take care of creating a new user in the `controllers/users.js` file and displaying all users do not contain anything dramatic

```
const router = require('express').Router()

const { User } = require('../models')
```

```

router.get('/', async (req, res) => {
  const users = await User.findAll()
  res.json(users)
})

router.post('/', async (req, res) => [
  try {
    const user = await User.create(req.body)
    res.json(user)
  } catch(error) {
    return res.status(400).json({ error })
  }
])

router.get('/:id', async (req, res) => [
  const user = await User.findByPk(req.params.id)
  if (user) {
    res.json(user)
  } else {
    res.status(404).end()
  }
])

module.exports = router

```

The router handler that handles the login (file *controllers/login.js*) is as follows:

```

const jwt = require('jsonwebtoken')
const router = require('express').Router()

const { SECRET } = require('../util/config')
const User = require('../models/user')

router.post('/', async (request, response) => {
  const body = request.body

  const user = await User.findOne({
    where: {
      username: body.username
    }
  })

  const passwordCorrect = body.password === 'secret'

  if (!(user && passwordCorrect)) {
    return response.status(401).json({
      error: 'invalid username or password'
    })
  }

  const userForToken = {
    username: user.username,

```

```

    id: user.id,
  }

  const token = jwt.sign(userForToken, SECRET)

  response
    .status(200)
    .send({ token, username: user.username, name: user.name })
  })
}

module.exports = router

```

The POST request will be accompanied by a username and a password. First, the object corresponding to the username is retrieved from the database using the *User* model with the `findOne` method:

```

const user = await User.findOne({
  where: {
    username: body.username
  }
})

```

From the console, we can see that the SQL statement corresponds to the method call

```

SELECT "id", "username", "name"
FROM "users" AS "User"
WHERE "User"."username" = 'mluukkai'

```

If the user is found and the password is correct (i.e. *secret* for all the users), A *jsonwebtoken* containing the user's information is returned in the response. To do this, we install the dependency

```
npm install jsonwebtoken
```

The file *index.js* expands slightly

```

const notesRouter = require('./controllers/notes')
const usersRouter = require('./controllers/users')
const loginRouter = require('./controllers/login')

app.use(express.json())

app.use('/api/notes', notesRouter)
app.use('/api/users', usersRouter)
app.use('/api/login', loginRouter)

```

The current code for the application is in its entirety on [GitHub](#), branch *part13-3*.

Connection between the tables

Users can now be added to the application and users can log in, but this in itself is not a very useful

feature yet. We would like to add the features that only a logged-in user can add notes, and that each note is associated with the user who created it. To do this, we need to add a *foreign key* to the *notes* table.

When using Sequelize, a foreign key can be defined by modifying the *models/index.js* file as follows

```
const Note = require('../note')
const User = require('../user')

UserhasMany(Note).Note.belongsTo(User).Note.sync({ alter: true }).User.sync({ alter: true })
module.exports = [
  Note, User
]
```

So this is how we **define** that there is a *one-to-many* relationship connection between the *users* and *notes* entries. We also changed the options of the *sync* calls so that the tables in the database match changes made to the model definitions. The database schema looks like the following from the console:

```
postgres=# \d users
                                         Table "public.users"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
 id | integer | not null | nextval('users_id_seq'::regclass)
 username | character varying(255) | | not null |
 name | character varying(255) | | not null |
Indexes:
 "users_pkey" PRIMARY KEY, btree (id)
Referenced by:
 TABLE "notes" CONSTRAINT "notes_user_id_fkey" FOREIGN KEY (user_id) REFERENCES users(id) ON UPDATE CASCADE ON DELETE SET NULL

postgres=# \d notes
                                         Table "public.notes"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
 id | integer | not null | nextval('notes_id_seq'::regclass)
 content | text | | not null |
 important | boolean | | |
 date | timestamp with time zone | | |
 user_id | integer | | |
Indexes:
 "notes_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
 "notes_user_id_fkey" FOREIGN KEY (user_id) REFERENCES users(id) ON UPDATE CASCADE ON DELETE SET NULL
```

The foreign key `user_id` has been created in the `notes` table, which refers to rows of the `users` table.

Now let's make every insertion of a new note be associated to a user. Before we do the proper implementation (where we associate the note with the logged-in user's token), let's hard code the note to be attached to the first user found in the database:

```
router.post('/', async [req, res] => {
  try {
    const user = await User.findOne()      const note = await Note.create({...req-
body, userId: user.id})      res.json(note)
  } catch(error) {
    return res.status(400).json({ error })
  }
})
```

Pay attention to how there is now a `user_id` column in the notes at the database level. The corresponding object in each database row is referred to by Sequelize's naming convention as opposed to camel case (`userId`) as typed in the source code.

Making a join query is very easy. Let's change the route that returns all users so that each user's notes are also shown:

```
router.get('/r', async [req, res] => {
  const users = await User.findAll({    include: [{        model: Note      }]  })  res
  .json(users)
})
```

So the join query is done using the `include` option as a query parameter.

The SQL statement generated from the query is seen on the console:

```
SELECT "User". "id", "User". "username", "User". "name", "Notes". "id" AS "Notes
.id", "Notes". "content" AS "Notes.content", "Notes". "important" AS "Notes impo-
rtant", "Notes". "date" AS "Notes.date", "Notes". "user_id" AS "Notes.UserId"
FROM "users" AS "User" LEFT OUTER JOIN "notes" AS "Notes" ON "User". "id" = "Not-
es". "user_id";
```

The end result is also as you might expect

```
[{"id": 1, "username": "ilves", "name": "Kalle Ilves", "notes": [{"id": 4, "content": "Notes are associated to a user", "important": false, "date": null, "userId": 1}]}], [{"id": 2, "username": "mluukkai", "name": "Matti Luukkainen", "notes": []}]]
```

Proper insertion of notes

Let's change the note insertion by making it work the same as in [part 4](#), i.e. the creation of a note can only be successful if the request corresponding to the creation is accompanied by a valid token from login. The note is then stored in the list of notes created by the user identified by the token:

```
const tokenExtractor = (req, res, next) => {
  const authorization = req.get('authorization')
  if (!authorization || !authorization.toLowerCase().startsWith('bearer ')) {
    try {
      req.decodedToken = jwt.verify(authorization.substring(7), SECRET)
    } catch {
      return res.status(401).json({error: 'token invalid'})
    }
  } else {
    return res.status(401).json({error: 'token missing'})
  }
  next()
}

router.post('/', tokenExtractor, async (req, res) => {
  try {
    const user = await User.findById(req.decodedToken.id)
    const note = await Note.create({...req.body, userId: user.id, date: new Date()})
    res.json(note)
  } catch(error) {
    return res.status(400).json({error})
  }
})
```

The token is retrieved from the request headers, decoded and placed in the `req` object by the `tokenExtractor` middleware. When creating a note, a `date` field is also given indicating the time it was created.

Fine-tuning

Our backend currently works almost the same way as the Part 4 version of the same application, except for error handling. Before we make a few extensions to backend, let's change the routes for retrieving all notes and all users slightly.

We will add to each note information about the user who added it:

```

router.get('/', async (req, res) => {
  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
    include: [
      {
        model: User,
        attributes: ['name']
      }
    ]
  })
  res.json(notes)
})

```

We have also **restricted** the values of which fields we want. For each note, we return all fields including the *name* of the user associated with the note but excluding the *userId*.

Let's make a similar change to the route that retrieves all users, removing the unnecessary field *userId* from the notes associated with the user:

```

router.get('/', async (req, res) => {
  const users = await User.findAll({
    include: [
      {
        model: Note,
        attributes: { exclude: ['userId'] }
      }
    ]
  })
  res.json(users)
})

```

The current code for the application is in its entirety on [GitHub](#), branch *part13-4*.

Attention to the definition of the models

The most perceptive will have noticed that despite the added column *user_id*, we did not make a change to the model that defines notes, but we can still add a user to note objects:

```

const user = await User.findByPk(req.decodedToken.id)
const note = await Note.create({ ...req.body, userId: user.id, date: new Date() })

```

The reason for this is that we specified in the file *models/index.js* that there is a one-to-many connection between users and notes:

```

const Note = require('../note')
const User = require('../user')

UserhasMany(Note)
NotebelongsTo(User)

// ...

```

Sequelize will automatically create an attribute called *userId* on the *Note* model to which, when

referenced gives access to the database column `user_id`.

Keep in mind, that we could also create a note as follows using the `build` method:

```
const user = await User.findByPk(req.decodedToken.id)

// create a note without saving it yet
const note = Note.build({ ...req.body, date: new Date() })
  // put the user id in the userId property of the created note
note.userId = user.id
// store the note object in the database
await note.save()
```

This is how we explicitly see that `userId` is an attribute of the notes object.

We could define the model as follows to get the same result:

```
Note.init([
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  content: {
    type: DataTypes.TEXT,
    allowNull: false
  },
  important: {
    type: DataTypes.BOOLEAN
  },
  date: {
    type: DataTypes.DATE
  },
  userId: { type: DataTypes.INTEGER, allowNull: false, references: { model: 'users', key: 'id' } },
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'note'
})

module.exports = Note
```

Defining at the class level of the model as above is usually unnecessary

```
User.hasMany(Note)
Note.belongsTo(User)
```

Instead we can achieve the same with this. Using one of the two methods is necessary otherwise Sequelize does not know how at the code level to connect the tables to each other.

Exercises 13.8.-13.12.

Exercise 13.8.

Add support for users to the application. In addition to ID, users have the following fields:

- name (string, must not be empty)
- username (string, must not be empty)

Unlike in the material, do not prevent Sequelize from creating `timestamps` `created_at` and `updated_at` for users

All users can have the same password as the material. You can also choose to properly implement passwords as in [part 4](#).

Implement the following routes

- `POST api/users` (adding a new user)
- `GET api/users` (listing all users)
- `PUT api/users/:username` (changing a username, keep in mind that the parameter is not id but username)

Make sure that the timestamps `created_at` and `updated_at` automatically set by Sequelize work correctly when creating a new user and changing a username.

Exercise 13.9.

Sequelize provides a set of pre-defined `validations` for the model fields, which it performs before storing the objects in the database.

It's decided to change the user creation policy so that only a valid email address is valid as a username. Implement validation that verifies this issue during the creation of a user.

Modify the error handling middleware to provide a more descriptive error message of the situation (for example, using the Sequelize error message), e.g.

```
[{"error": ["Validation isEmail on username failed"]}]
```

Exercise 13.10.

Expand the application so that the current logged-in user identified by a token is linked to each blog added. To do this you will also need to implement a login endpoint `POST /api/login`, which returns the token.

Exercise 13.11.

Make deletion of a blog only possible for the user who added the blog.

Exercise 13.12.

Modify the routes for retrieving all blogs and all users so that each blog shows the user who added it and each user shows the blogs they have added.

More queries

So far our application has been very simple in terms of queries, queries have searched for either a single row based on the primary key using the method `findById` or they have searched for all rows in the table using the method `findAll`. These are sufficient for the frontend of the application made in Section 5, but let's expand the backend so that we can also practice making slightly more complex queries.

Let's first implement the possibility to retrieve only important or non-important notes. Let's implement this using the `query-parameter` `important`:

```
router.get('/', async (req, res) => {
  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
    include: [
      {
        model: User,
        attributes: ['name']
      }
    ],
    where: { important: req.query.important === "true" }
  })
  res.json(notes)
})
```

Now the backend can retrieve important notes with a request to <http://localhost:3001/api/notes?important=true> and non-important notes with a request to <http://localhost:3001/api/notes?important=false>

The SQL query generated by Sequelize contains a WHERE clause that filters rows that would normally be returned:

```
SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id" AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" =
"user". "id"
WHERE "note". "important" = true;
```

Unfortunately, this implementation will not work if the request is not interested in whether the note is important or not, i.e. if the request is made to <http://localhost:3001/api/notes>. The correction can be done in several ways. One, but perhaps not the best way to do the correction would be as follows:

```
const { Op } = require('sequelize')

router.get('/', async (req, res) => {
  let important = [true, false]
  if (req.query.important) {
    important = req.query.important === "true"
  }
  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
```

```

    include: [
      model: User,
      attributes: ['name']
    ],
    where: [
      important
    ]
  })
  res.json(notes)
})

```

The *important* object now stores the query condition. The default query is

```

where: [
  important: {
    [Op.in]: [true, false]
  }
]

```

i.e. the *important* column can be *true* or *false*, using one of the many Sequelize operators [Op.in](#). If the query parameter *req.query.important* is specified, the query changes to one of the two forms

```

where: [
  important: true
]

```

or

```

where: [
  important: false
]

```

depending on the value of the query parameter.

The database might now contain some note rows that do not have the value for the column *important* set. After the above changes, these notes can not be found with the queries. Let us set the missing values in the psql console and change the schema so that the column does not allow a null value:

```

Note.init()
{
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
  },
  content: {
    type: DataTypes.TEXT,
    allowNull: false,
  },
  important: {
    type: DataTypes.BOOLEAN,
  }
}

```

```

        allowNull: false,
      },
      date: [
        type: DataTypes.DATE,
      ],
    },
    // ...
  ]
}

```

The functionality can be further expanded by allowing the user to specify a required keyword when retrieving notes, e.g. a request to <http://localhost:3001/api/notes?search=database> will return all notes mentioning *database* or a request to <http://localhost:3001/api/notes?search=javascript&important=true> will return all notes marked as important and mentioning *javascript*. The implementation is as follows

```

router.get('/notes', async (req, res) => {
  let important = [
    [Op.in]: [true, false]
  ]

  if (req.query.important) {
    important = req.query.important === "true"
  }

  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
    include: [
      { model: User,
        attributes: ['name'] }
    ],
    where: [
      { important },
      { content: {[Op.substring]: req.query.search ? req.query.search : ''} }
    ]
  })

  res.json(notes)
})

```

Sequelize's `Op.substring` generates the query we want using the LIKE keyword in SQL. For example, if we make a query to <http://localhost:3001/api/notes?search=database&important=true> we will see that the SQL query it generates is exactly as we expect.

```

SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id" AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" = "user". "id"
WHERE "note". "important" = true AND "note". "content" LIKE '%database%'

```

There is still a beautiful flaw in our application that we see if we make a request to <http://localhost:3001/api/notes>, i.e. we want all the notes, our implementation will cause an unnecessary WHERE in the query, which may (depending on the implementation of the database

engine) unnecessarily affect the query efficiency:

```
SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id" AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" =
"user". "id"
WHERE "note". "important" IN [true, false] AND "note". "content" LIKE '%%';
```

Let's optimize the code so that the WHERE conditions are used only if necessary:

```
router.get('/', async (req, res) => {
  const where = {}

  if (req.query.important) {
    where.important = req.query.important === "true"
  }

  if (req.query.search) {
    where.content = [
      [Op.substring]: req.query.search
    ]
  }

  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
    include: [
      { model: User,
        attributes: ['name'] }
    ],
    where
  })

  res.json(notes)
})
```

If the request has search conditions e.g. <http://localhost:3001/api/notes?search=database&important=true>, a query containing WHERE is formed

```
SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id" AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" =
"user". "id"
WHERE "note". "important" = true AND "note". "content" LIKE '%database%';
```

If the request has no search conditions <http://localhost:3001/api/notes>, then the query does not have an unnecessary WHERE

```
SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id" AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" =
```

```
"user": "id":
```

The current code for the application is in its entirety on [GitHub](#), branch *part13-5*.

Exercises 13.13.-13.16

Exercise 13.13.

Implement filtering by keyword in the application for the route returning all blogs. The filtering should work as follows

- *GET /api/blogs?search=react* returns all blogs with the search word *react* in the *title* field, the search word is case-insensitive
- *GET /api/blogs* returns all blogs

This should be useful for this task and the next one.

Exercise 13.14.

Expand the filter to search for a keyword in either the *title* or *author* fields, i.e.

GET /api/blogs?search=jami returns blogs with the search word *jami* in the *title* field or in the *author* field

Exercise 13.15.

Modify the blogs route so that it returns blogs based on likes in descending order. Search the [documentation](#) for instructions on ordering,

Exercise 13.16.

Make a route for the application */api/authors* that returns the number of blogs for each author and the total number of likes. Implement the operation directly at the database level. You will most likely need the [group by](#) functionality, and the [sequelize.fn](#) aggregator function.

The JSON returned by the route might look like the following, for example:

```
[  
  {  
    author: "Jami Kousa",  
    articles: "3",  
    likes: "10"  
  },  
  {  
    author: "Kalle Ilves",  
    articles: "1",  
    likes: "2"  
  },  
  {  
    author: "Dan Abramov",  
    articles: "1",  
    likes: "1"  
  }]
```

```
    likes: "4"
  }
]
```

Bonus task: order the data returned based on the number of likes, do the ordering in the database query.

13c: Migrations, many-to-many relationships

Migrations

Let's keep expanding the backend. We want to implement support for allowing users with *admin status* to put users of their choice in disabled mode, preventing them from logging in and creating new notes. In order to implement this, we need to add boolean fields to the users' database table indicating whether the user is an admin and whether the user is disabled.

We could proceed as before, i.e. change the model that defines the table and rely on Sequelize to synchronize the changes to the database. This is specified by these lines in the file *models/index.js*

```
const Note = require('./note')
const User = require('./user')

Note.belongsTo(User)
UserhasMany(Note)

Note.sync({ alter: true })
User.sync({ alter: true })

module.exports = [
  Note,
  User
]
```

However, this approach does not make sense in the long run. Let's remove the lines that do the synchronization and move to using a much more robust way, [migrations](#) provided by Sequelize (and many other libraries).

In practice, a migration is a single JavaScript file that describes some modification to a database. A separate migration file is created for each single or multiple changes at once. Sequelize keeps a record of which migrations have been performed, i.e. which changes caused by the migrations are synchronized to the database schema. When creating new migrations, Sequelize keeps up to date on which changes to the database schema are yet to be made. In this way, changes are made in a controlled manner, with the program code stored in version control.

First, let's create a migration that initializes the database. The code for the migration is as follows

```
const { DataTypes } = require('sequelize')

module.exports = [
  up: async ({ context: queryInterface }) => {
    await queryInterface.createTable('notes', {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
      }
    })
  },
  down: queryInterface => queryInterface.dropTable('notes')
]
```

```

        autoIncrement: true
    },
    content: [
        type: DataTypes.TEXT,
        allowNull: false
    ],
    important: [
        type: DataTypes.BOOLEAN,
        allowNull: false
    ],
    date: [
        type: DataTypes.DATE
    ],
},
await queryInterface.createTable('users', [
    id: [
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
    ],
    username: [
        type: DataTypes.STRING,
        unique: true,
        allowNull: false
    ],
    name: [
        type: DataTypes.STRING,
        allowNull: false
    ],
],
await queryInterface.addColumn('notes', 'user_id', {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'users', key: 'id' }
}),
down: async ({ context: queryInterface }) => {
    await queryInterface.dropTable('notes')
    await queryInterface.dropTable('users')
},
])

```

The migration file `defines` the functions `up` and `down`, the first of which defines how the database should be modified when the migration is performed. The function `down` tells you how to undo the migration if there is a need to do so.

Our migration contains three operations, the first creates a `notes` table, the second creates a `users` table and the third adds a foreign key to the `notes` table referencing the creator of the note. Changes in the schema are defined by calling the `queryInterface` object methods.

When defining migrations, it is essential to remember that unlike models, column and table names are written in snake case form:

```
await queryInterface.addColumn('notes', 'user_id', { type: DataTypes.INTEGER,
  allowNull: false,
  references: { model: 'users', key: 'id' },
})
```

So in migrations, the names of the tables and columns are written exactly as they appear in the database, while models use Sequelize's default camelCase naming convention.

Save the migration code in the file `migrations/20211209_00_initialize_notes_and_users.js`. Migration file names should always be named alphabetically when created so that previous changes are always before newer changes. One good way to achieve this order is to start the migration file name with the date and a sequence number.

We could run the migrations from the command line using the [Sequelize command line tool](#). However, we choose to perform the migrations manually from the program code using the [Umzug](#) library. Let's install the library

```
npm install umzug
```

Let's change the file `util/db.js` that handles the connection to the database as follows:

```
const Sequelize = require('sequelize')
const { DATABASE_URL } = require('../config')
const { Umzug, SequelizeStorage } = require('umzug')
const sequelize = new Sequelize(DATABASE_URL)

const runMigrations = async () => {
  const migrator = new Umzug({
    migrations: [
      glob: 'migrations/*.js',
    ],
    storage: new SequelizeStorage({
      sequelize,
      tableName: 'migrations'
    }),
    context: sequelize.getQueryInterface(),
    logger: console
  })
  const migrations = await migrator.up()
  console.log(`Migrations up to date`, { files: migrations.map((mig) => mig.name) })
}

const connectToDatabase = async () => {
  try {
    await sequelize.authenticate()
    await runMigrations()
    console.log('connected to the database')
  } catch (err) {
    console.log('failed to connect to the database')
    console.log(err)
    return process.exit(1)
  }
}

module.exports = [ connectToDatabase, sequelize ]
```

The `runMigrations` function that performs migrations is now executed every time the application opens a database connection when it starts. Sequelize keeps track of which migrations have already been completed, so if there are no new migrations, running the `runMigrations` function does nothing.

Now let's start with a clean slate and remove all existing database tables from the application:

```
username => drop table notes;
username => drop table users;
username => \d
Did not find any relations.
```

Let's start up the application. A message about the migrations status is printed on the log

```
INSERT INTO "migrations" ("name") VALUES ($1) RETURNING "name";
Migrations up to date [ files: ['20211209_00_initialize_notes_and_users.js'] ]
database connected
```

If we restart the application, the log also shows that the migration was not repeated.

The database schema of the application now looks like this

```
postgres=# \d
              List of relations
 Schema |      Name       |   Type    |      Owner
-----+-----+-----+-----+
 public | migrations   | table    | username
 public | notes        | table    | username
 public | notes_id_seq | sequence | username
 public | users         | table    | username
 public | users_id_seq | sequence | username
```

So Sequelize has created a *migrations* table that allows it to keep track of the migrations that have been performed. The contents of the table look as follows:

```
postgres=# select * from migrations;
          name
-----
 20211209_00_initialize_notes_and_users.js
```

Let's create a few users in the database, as well as a set of notes, and after that we are ready to expand the application.

The current code for the application is in its entirety on [GitHub](#), branch *part13-6*.

Admin user and user disabling

So we want to add two boolean fields to the *users* table

- *admin* tells you whether the user is an admin
- *disabled* tells you whether the user is disabled from actions

Let's create the migration that modifies the database in the file *migrations/20211209_01_admin_and_disabled_to_users.js*:

```

const { DataTypes } = require('sequelize')

module.exports = [
  up: async ({ context: queryInterface }) => {
    await queryInterface.addColumn('users', 'admin', {
      type: DataTypes.BOOLEAN,
      defaultValue: false
    })
    await queryInterface.addColumn('users', 'disabled', {
      type: DataTypes.BOOLEAN,
      defaultValue: false
    })
  },
  down: async ({ context: queryInterface }) => {
    await queryInterface.removeColumn('users', 'admin')
    await queryInterface.removeColumn('users', 'disabled')
  },
]

```

Make corresponding changes to the model corresponding to the *users* table:

```

User.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  username: {
    type: DataTypes.STRING,
    unique: true,
    allowNull: false
  },
  name: {
    type: DataTypes.STRING,
    allowNull: false
  },
  admin: { type: DataTypes.BOOLEAN, defaultValue: false },
  disabled: { type: DataTypes.BOOLEAN, defaultValue: false },
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'user'
})

```

When the new migration is performed when the code restarts, the schema is changed as desired:

```

username-> \d users
                                         Table "public.users"
   Column |          Type          | Collation | Nullable | Default
-----+-----+-----+-----+-----+

```

```

id      | integer          |      | not null | nextval('users_id_se
q'::regclass)
username | character varying(255) |      | not null |
name    | character varying(255) |      | not null |
admin   | boolean           |      |           |
disabled | boolean           |      |           |

Indexes:
"users_pkey" PRIMARY KEY, btree (id)
"users_username_key" UNIQUE CONSTRAINT, btree (username)

Referenced by:
  TABLE "notes" CONSTRAINT "notes_user_id_fkey" FOREIGN KEY (user_id) REFERENCES users(id)

```

Now let's expand the controllers as follows. We prevent logging in if the user field *disabled* is set to *true*:

```

loginRouter.post('/' , async (request, response) => {
  const body = request.body

  const user = await User.findOne({
    where: [
      {username: body.username}
    ]
  })

  const passwordCorrect = body.password === 'secret'

  if (!user && passwordCorrect) {
    return response.status(401).json({
      error: 'invalid username or password'
    })
  }

  if (user.disabled) {    return response.status(401).json({error: 'account disabled, please contact admin'}) }
  const userForToken = {
    username: user.username,
    id: user.id,
  }

  const token = jwt.sign(userForToken, process.env.SECRET)

  response
    .status(200)
    .send({ token, username: user.username, name: user.name })
})

```

Let's disable the user *jakousa* using his ID:

```

username => update users set disabled=true where id=3;
UPDATE 1

```

```

username => update users set admin=true where id=1;
UPDATE 1
username => select * from users;
+----+-----+-----+-----+-----+
| id | username | name | admin | disabled |
+----+-----+-----+-----+-----+
| 2 | lynx | Kalle Ilves | | |
| 3 | jakousa | Jami Kousa | f | t |
| 1 | mluukkai | Matti Luukkainen | t | |

```

And make sure that logging in is no longer possible

The screenshot shows a browser developer tools Network tab. On the left, under 'requests', there is a log entry for a POST request to 'POST http://localhost:3001/api/login'. The request body is shown as JSON:

```

1 POST http://localhost:3001/api/login
2 Content-Type: application/json
3
4 {
5   "password": "salainen",
6   "username": "jakousa"
7 }

```

On the right, under 'Response(177ms)', the response details are listed:

```

1 HTTP/1.1 401 Unauthorized
2 X-Powered-By: Express
3 Content-Type: application/json; charset=utf-8
4 Content-Length: 50
5 ETag: W/"32-Mcypa7n82bdDhbNtu0Ihcfb5u1c"
6 Date: Mon, 11 Oct 2021 14:36:47 GMT
7 Connection: close
8
9 {
10   "error": "account disabled, please contact admin"
11 }

```

Let's create a route that will allow an admin to change the status of a user's account:

```

const isAdmin = async (req, res, next) => {
  const user = await User.findById(req.decodedToken.id)
  if (!user.admin) {
    return res.status(401).json({ error: 'operation not allowed' })
  }
  next()
}

router.put('/:username', tokenExtractor, isAdmin, async (req, res) => {
  const user = await User.findOne({
    where: [
      { username: req.params.username }
    ]
  })

  if (user) {
    user.disabled = req.body.disabled
    await user.save()
    res.json(user)
  } else {
    res.status(404).end()
  }
})

```

There are two middleware used, the first called *tokenExtractor* is the same as the one used by the note-creation route, i.e. it places the decoded token in the *decodedToken* field of the request-object. The second middleware *isAdmin* checks whether the user is an admin and if not, the request status is set to 401 and an appropriate error message is returned.

Note how *two middleware* are chained to the route, both of which are executed before the actual route handler. It is possible to chain an arbitrary number of middleware to a request.

The middleware *tokenExtractor* is now moved to *util/middleware.js* as it is used from multiple locations.

```
const jwt = require('jsonwebtoken')
const { SECRET } = require('../config.js')

const tokenExtractor = (req, res, next) => {
  const authorization = req.get('authorization')
  if (authorization && authorization.toLowerCase().startsWith('bearer ')) {
    try {
      req.decodedToken = jwt.verify(authorization.substring(7), SECRET)
    } catch {
      return res.status(401).json({ error: 'token invalid' })
    }
  } else {
    return res.status(401).json({ error: 'token missing' })
  }
  next()
}

module.exports = [ tokenExtractor ]
```

An admin can now re-enable the user *jakousa* by making a PUT request to */api/users/jakousa*, where the request comes with the following data:

```
{
  "disabled": false
}
```

As noted in [the end of Part 4](#), the way we implement disabling users here is problematic. Whether or not the user is disabled is only checked at *login*, if the user has a token at the time the user is disabled, the user may continue to use the same token, since no lifetime has been set for the token and the disabled status of the user is not checked when creating notes.

Before we proceed, let's make an npm script for the application, which allows us to undo the previous migration. After all, not everything always goes right the first time when developing migrations.

Let's modify the file *util/db.js* as follows:

```
const Sequelize = require('sequelize')
const { DATABASE_URL } = require('../config')
const { Umzug, SequelizeStorage } = require('umzug')

const sequelize = new Sequelize(DATABASE_URL, {
  dialectOptions: [
    ssl: [
      require: true,
      rejectUnauthorized: false
    ]
  ]
})
```

```

    }
  },
})

const connectToDatabase = async () => {
  try {
    await sequelize.authenticate()
    await runMigrations()
    console.log('connected to the database')
  } catch (err) {
    console.log('failed to connect to the database')
    return process.exit(1)
  }
}

return null
}

const migrationConf = { migrations: { glob: 'migrations/*.js' }, storage: new SequelizeStorage({ sequelize, tableName: 'migrations' }), context: sequelize.getQueryInterface(), logger: console }
const runMigrations = async () => {
  const migrator = new Umzug(migrationConf)
  const migrations = await migrator.up()
  console.log('Migrations up to date', { files: migrations.map((mig) => mig.name) })
}
const rollbackMigration = async () => {
  await sequelize.authenticate()
  const migrator = new Umzug(migrationConf)
  await migrator.down()
}

module.exports = [ connectToDatabase, sequelize, rollbackMigration ]

```

Let's create a file `util/rollback.js`, which will allow the npm script to execute the specified migration rollback function:

```

const { rollbackMigration } = require('./db')

rollbackMigration()

```

and the script itself:

```

{
  "scripts": {
    "dev": "nodemon index.js",
    "migration:down": "node util/rollback.js"
  }
}

```

So we can now undo the previous migration by running `npm run migration:down` from the command line.

Migrations are currently executed automatically when the program is started. In the development phase of the program, it might sometimes be more appropriate to disable the automatic execution of migrations and make migrations manually from the command line.

The current code for the application is in its entirety on [GitHub](#), branch `part13-7`.

Exercises 13.17-13.18.

Exercise 13.17.

Delete all tables from your application's database.

Make a migration that initializes the database. Add `created_at` and `updated_at` [timestamps](#) for both tables. Keep in mind that you will have to add them in the migration yourself.

NOTE: be sure to remove the commands `User.sync()` and `Blog.sync()`, which synchronizes the models' schemas from your code, otherwise your migrations will fail.

NOTE2: if you have to delete tables from the command line (i.e. you don't do the deletion by undoing the migration), you will have to delete the contents of the `migrations` table if you want your program to perform the migrations again.

Exercise 13.18.

Expand your application (by migration) so that the blogs have a year written attribute, i.e. a field `year` which is an integer at least equal to 1991 but not greater than the current year. Make sure the application gives an appropriate error message if an incorrect value is attempted to be given for a year written.

Many-to-many relationships

We will continue to expand the application so that each user can be added to one or more `teams`.

Since an arbitrary number of users can join one team, and one user can join an arbitrary number of teams, we are dealing with a [many-to-many](#) relationship, which is traditionally implemented in relational databases using a `connection table`.

Let's now create the code needed for the teams table as well as the connection table. The migration (saved in file `20211209_02_add_teams_and_memberships.js`) is as follows:

```
const { DataTypes } = require('sequelize')

module.exports = [
  up: async ({ context: queryInterface }) => {
    await queryInterface.createTable('teams', [
      {
        id: {
          type: DataTypes.INTEGER,
          primaryKey: true,
          autoIncrement: true
        },
        name: {
          type: DataTypes.TEXT,
          allowNull: false,
          unique: true
        }
      }
    ])
    await queryInterface.createTable('memberships', [
      {
        id: {
          type: DataTypes.INTEGER,
          primaryKey: true,
          autoIncrement: true
        },
        userId: {
          type: DataTypes.INTEGER,
          allowNull: false
        },
        teamId: {
          type: DataTypes.INTEGER,
          allowNull: false
        }
      }
    ])
  },
  down: queryInterface => {
    queryInterface.dropTable('memberships')
    queryInterface.dropTable('teams')
  }
]
```

```

        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
    },
    user_id: {
        type: DataTypes.INTEGER,
        allowNull: false,
        references: { model: 'users', key: 'id' }
    },
    team_id: {
        type: DataTypes.INTEGER,
        allowNull: false,
        references: { model: 'teams', key: 'id' }
    },
},
}

down: async ({ context: queryInterface }) => {
    await queryInterface.dropTable('teams')
    await queryInterface.dropTable('memberships')
},
}

```

The models contain almost the same code as the migration. The team model in *models/team.js*:

```

const { Model, DataTypes } = require('sequelize')

const { sequelize } = require('../util/db')

class Team extends Model {}

Team.init({
    id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
    },
    name: {
        type: DataTypes.TEXT,
        allowNull: false,
        unique: true
    },
}, {
    sequelize,
    underscored: true,
    timestamps: false,
    modelName: 'team'
})

module.exports = Team

```

The model for the connection table in *models/membership.js*:

```

const { Model, DataTypes } = require('sequelize')

const { sequelize } = require('../util/db')

class Membership extends Model {}

Membership.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  userId: [
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'users', key: 'id' }
  ],
  teamId: [
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'teams', key: 'id' }
  ],
  {
    sequelize,
    underscored: true,
    timestamps: false,
    modelName: 'membership'
  }
}

module.exports = Membership

```

So we have given the connection table a name that describes it well, *membership*. There is not always a relevant name for a connection table, in which case the name of the connection table can be a combination of the names of the tables that are joined, e.g. *user_teams* could fit our situation.

We make a small addition to the *models/index.js* file to connect teams and users at the code level using the [belongsToMany](#) method.

```

const Note = require('./note')
const User = require('./user')
const Team = require('./team') const Membership = require('./membership')
Note.belongsTo(User)
User.hasMany(Note)

User.belongsToMany(Team, { through: Membership })
Team.belongsToMany(User, { through: Membership })
module.exports = [
  Note, User, Team, Membership
]

```

Note the difference between the migration of the connection table and the model when defining foreign key fields. During the migration, fields are defined in snake case form:

```

await queryInterface.createTable('memberships', {
  // ...
  user_id: [
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'users', key: 'id' },
  ],
  team_id: [
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'teams', key: 'id' },
  ],
})

```

in the model, the same fields are defined in camel case:

```

Membership.init([
  // ...
  userId: [
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'users', key: 'id' },
  ],
  teamId: [
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'teams', key: 'id' },
  ],
  // ...
])

```

Now let's create a couple of teams from the psql console, as well as a few memberships:

```

insert into teams (name) values ('toska');
insert into teams (name) values ('mosa climbers');
insert into memberships (user_id, team_id) values (1, 1);
insert into memberships (user_id, team_id) values (1, 2);
insert into memberships (user_id, team_id) values (2, 1);
insert into memberships (user_id, team_id) values (3, 2);

```

Information about users' teams is then added to route for retrieving all users

```

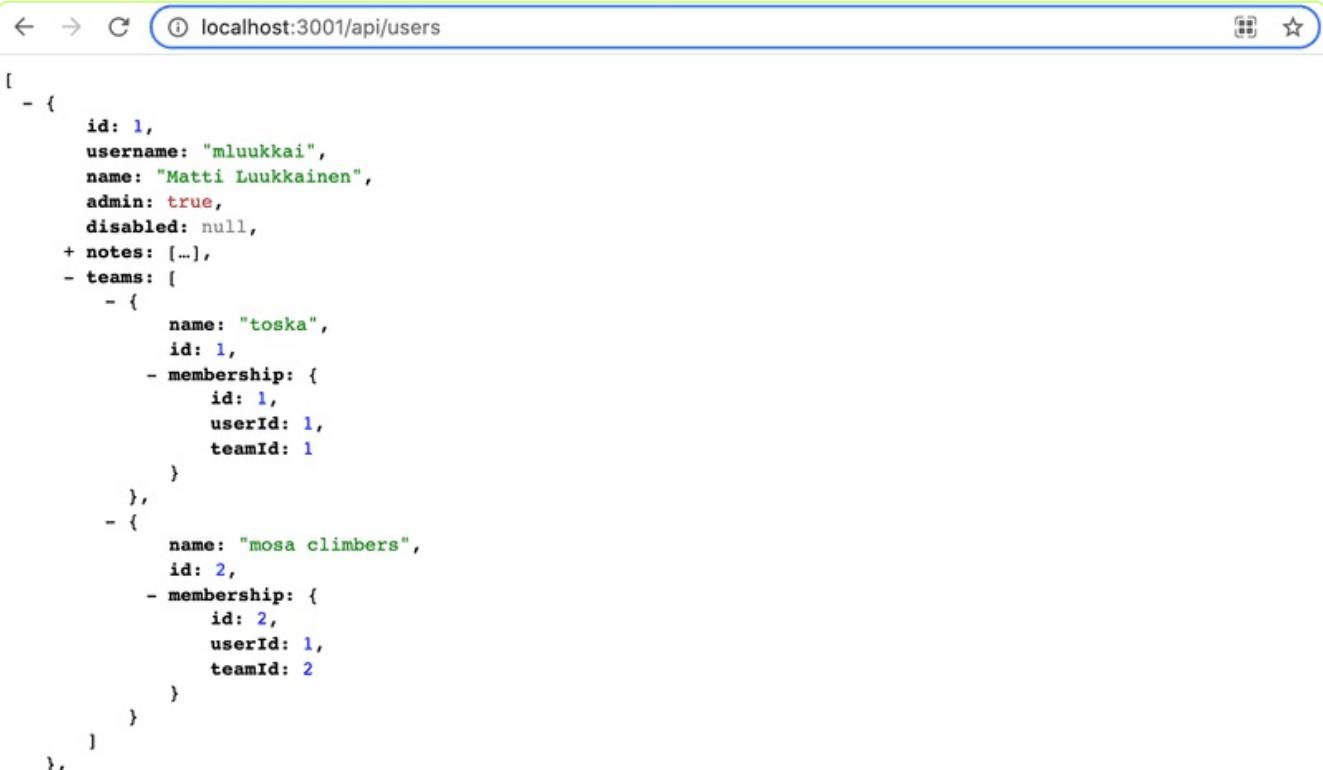
router.get('/', async [req, res] => {
  const users = await User.findAll({
    include: [
      {
        model: Note,
        attributes: { exclude: ['userId'] }
      },
      { model: Team, attributes: ['name', 'id'] }
    ]
})

```

```
    })
    res.json(users)
})
```

The most observant will notice that the query printed to the console now combines three tables.

The solution is pretty good, but there's a beautiful flaw in it. The result also comes with the attributes of the corresponding row of the connection table, although we do not want this:



```
[{"id": 1, "username": "mluukkai", "name": "Matti Luukkainen", "admin": true, "disabled": null, "notes": [...], "teams": [{"name": "toska", "id": 1, "membership": {"id": 1, "userId": 1, "teamId": 1}}, {"name": "mosa climbers", "id": 2, "membership": {"id": 2, "userId": 1, "teamId": 2}}]}
```

By carefully reading the documentation, you can find a [solution](#):

```
router.get('/', async (req, res) => {
  const users = await User.findAll({
    include: [
      {
        model: Note,
        attributes: { exclude: ['userId'] }
      },
      {
        model: Team,
        attributes: ['name', 'id'],
        through: {
          attributes: []
        }
      }
    ]
  })
  res.json(users)
})
```

The current code for the application is in its entirety on [GitHub](#), branch *part13-8*.

Note on the properties of Sequelize model objects

The specification of our models is shown by the following lines:

```
User.hasMany(Note)
Note.belongsTo(User)

User.belongsToMany(Team, { through: Membership })
Team.belongsToMany(User, { through: Membership })
```

These allow Sequelize to make queries that retrieve, for example, all the notes of users, or all members of a team.

Thanks to the definitions, we also have direct access to, for example, the user's notes in the code. In the following code, we will search for a user with id 1 and print the notes associated with the user:

```
const user = await User.findByPk(1, {
  include: [
    { model: Note }
  ]
})

user.notes.forEach(note => {
  console.log(note.content)
})
```

The `User.hasMany(Note)` definition therefore attaches a `notes` property to the `user` object, which gives access to the notes made by the user. The `User.belongsToMany(Team, { through: Membership })` definition similarly attaches a `teams` property to the `user` object, which can also be used in the code:

```
const user = await User.findByPk(1, {
  include: [
    { model: team }
  ]
})

user.teams.forEach(team => {
  console.log(team.name)
})
```

Suppose we would like to return a JSON object from the single user's route containing the user's name, username and number of notes created. We could try the following:

```
router.get('/:id', async [req, res] => {
  const user = await User.findByPk(req.params.id, {
    include: [
      { model: Note }
    ]
  })

  if (user) {
```

```

        user.note_count = user.notes.length      delete user.notes      res.json(user)

    } else {
        res.status(404).end()
    }
})
}

```

So, we tried to add the *noteCount* field on the object returned by Sequelize and remove the *notes* field from it. However, this approach does not work, as the objects returned by Sequelize are not normal objects where the addition of new fields works as we intend.

A better solution is to create a completely new object based on the data retrieved from the database:

```

router.get('/:id', async (req, res) => {
    const user = await User.findByPk(req.params.id, {
        include: [
            { model: Note }
        ]
    })

    if (user) {
        res.json({
            username: user.username,           name: user.name,           note_count: user.notes.length
        })
    } else {
        res.status(404).end()
    }
})

```

Revisiting many-to-many relationships

Let's make another many-to-many relationship in the application. Each note is associated to the user who created it by a foreign key. It is now decided that the application also supports that the note can be associated with other users, and that a user can be associated with an arbitrary number of notes created by other users. The idea is that these notes are those that the user has *marked* for himself.

Let's make a connection table *user_notes* for the situation. The migration, that is saved in file *20211209_03_add_user_notes.js* is straightforward:

```

const { DataTypes } = require('sequelize')

module.exports = {
    up: async ({ context: queryInterface }) => {
        await queryInterface.createTable('user_notes', [
            {
                id: [
                    type: DataTypes.INTEGER,
                    primaryKey: true,
                    autoIncrement: true
                ]
            }
        ])
    },
    down: async ({ context: queryInterface }) => {
        await queryInterface.dropTable('user_notes')
    }
}

```

```

    },
    user_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: { model: 'users', key: 'id' },
    },
    note_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: { model: 'notes', key: 'id' },
    },
  })
}

down: async ({ context: queryInterface }) => {
  await queryInterface.dropTable('user_notes')
}

```

Also, there is nothing special about the model:

```

const { Model, DataTypes } = require('sequelize')

const { sequelize } = require('../util/db')

class UserNotes extends Model {}

UserNotes.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  userId: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'users', key: 'id' },
  },
  noteId: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'notes', key: 'id' },
  },
}, {
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'user_notes'
})

module.exports = UserNotes

```

The file `models/index.js`, on the other hand, comes with a slight change to what we saw before:

```
const Note = require('./note')
const User = require('./user')
const Team = require('./team')
const Membership = require('./membership')
const UserNotes = require('./user_notes')

Note.belongsTo(User)
UserhasMany(Note)

User.belongsToMany(Team, { through: Membership })
Team.belongsToMany(User, { through: Membership })

User.belongsToMany(Note, { through: UserNotes, as: 'marked_notes' })
Note.belongsToMany(User, { through: UserNotes, as: 'users_marked' })
module.exports = [
  Note, User, Team, Membership, UserNotes
]
```

Once again `belongsToMany` is used, which now links users to notes via the `UserNotes` model corresponding to the connection table. However, this time we give an *alias name* for the attribute formed using the keyword `as`, the default name (a user's `notes`) would overlap with its previous meaning, i.e. notes created by the user.

We extend the route for an individual user to return the user's teams, their own notes, and other notes marked by the user:

```
router.get('/:id', async (req, res) => {
  const user = await User.findByPk(req.params.id, {
    attributes: { exclude: [''] },
    include: [
      {
        model: Note,
        attributes: { exclude: ['userId'] }
      },
      {
        model: Note,
        as: 'marked_notes',
        attributes: { exclude: ['userId'] },
        through: {
          attributes: []
        }
      },
      {
        model: Team,
        attributes: ['name', 'id'],
        through: {
          attributes: []
        }
      }
    ]
})
```

```

    if (user) {
      res.json(user)
    } else {
      res.status(404).end()
    }
  })
}

```

In the context of the include, we must now use the alias name *marked_notes* which we have just defined with the *as* attribute.

In order to test the feature, let's create some test data in the database:

```

insert into user_notes (user_id, note_id) values (1, 4);
insert into user_notes (user_id, note_id) values (1, 5);

```

The end result is functional:

```

{
  "id": 2,
  "username": "jakousa",
  "name": "Jami Kousa",
  "admin": false,
  "disabled": false,
  - notes: [
    - {
      "id": 4,
      "content": "pgsql for the win!",
      "important": true,
      "date": "2022-09-15T09:56:15.495Z"
    }
  ],
  - markedNotes: [
    - {
      "id": 1,
      "content": "token auth rocks",
      "important": true,
      "date": "2022-09-14T12:47:45.565Z",
      - user: {
          "name": "Matti Luukkainen"
        }
      },
      - {
        "id": 2,
        "content": "data is persisted in relational db",
        "important": false,
        "date": "2022-09-14T12:48:02.838Z",
        - user: {
            "name": "Matti Luukkainen"
          }
        }
      ],
  - teams: [
    - {
      "name": "mosa climbers"
    }
  ]
}

```

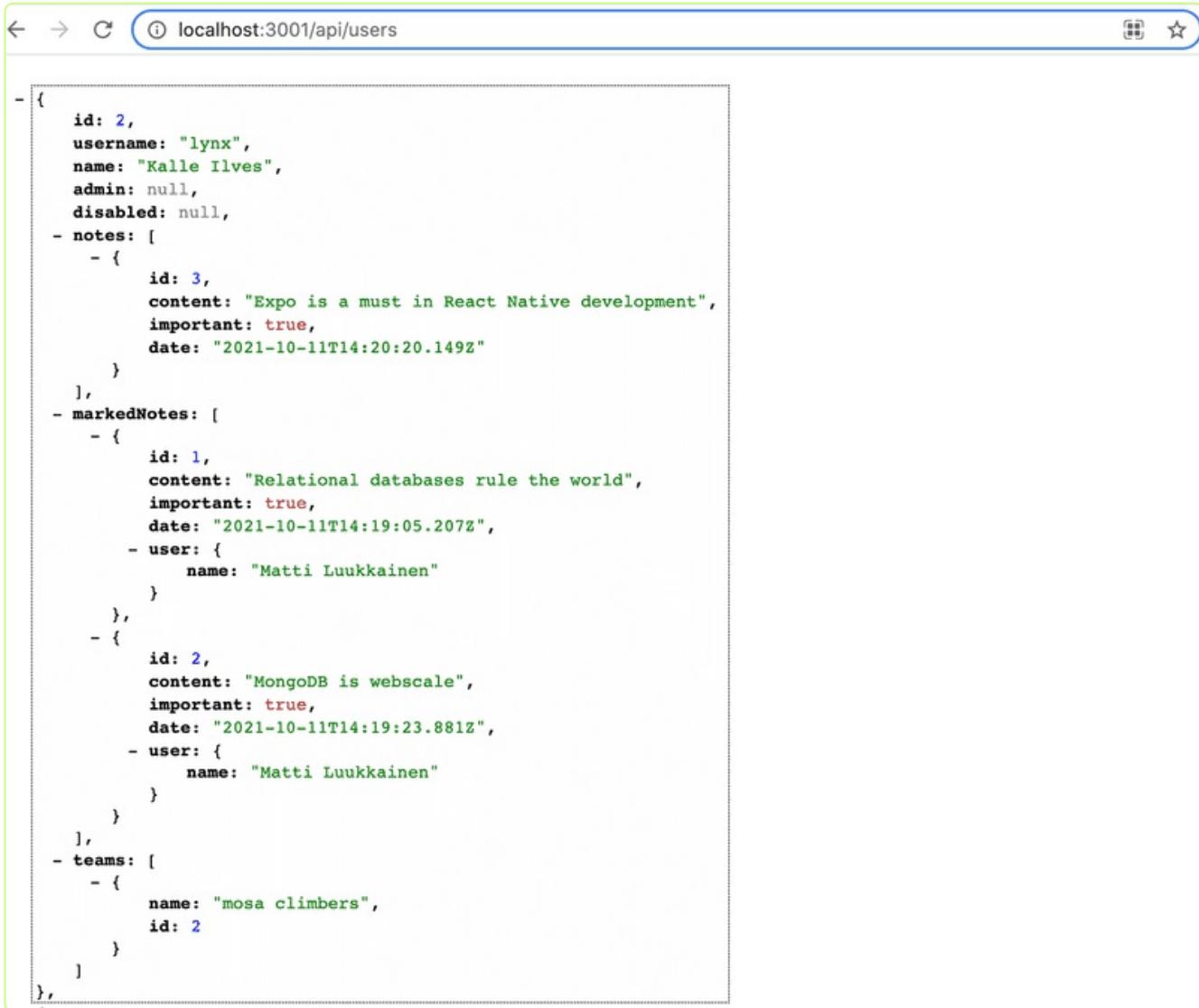
What if we wanted to include information about the author of the note in the notes marked by the user as well? This can be done by adding an *include* to the marked notes:

```

router.get('/:id', async (req, res) => {
  const user = await User.findById(req.params.id, {
    attributes: { exclude: [''] },
    include: [
      {
        model: Note,
        attributes: { exclude: ['userId'] }
      },
      {
        model: Note,
        as: 'marked_notes',
        attributes: { exclude: ['userId'] },
        through: [
          attributes: []
        ],
        include: [
          model: User,
          attributes: ['name']
        ]
      },
      {
        model: Team,
        attributes: ['name', 'id'],
        through: [
          attributes: []
        ]
      }
    ]
  })
  if (user) {
    res.json(user)
  } else {
    res.status(404).end()
  }
})

```

The end result is as desired:



A screenshot of a web browser window showing a JSON response from the URL `localhost:3001/api/users`. The JSON data represents a user object with various properties and nested arrays.

```
{  
  id: 2,  
  username: "lynx",  
  name: "Kalle Ilves",  
  admin: null,  
  disabled: null,  
  notes: [  
    {  
      id: 3,  
      content: "Expo is a must in React Native development",  
      important: true,  
      date: "2021-10-11T14:20:20.149Z"  
    }  
  ],  
  markedNotes: [  
    {  
      id: 1,  
      content: "Relational databases rule the world",  
      important: true,  
      date: "2021-10-11T14:19:05.207Z",  
      user: {  
        name: "Matti Luukkainen"  
      }  
    },  
    {  
      id: 2,  
      content: "MongoDB is webscale",  
      important: true,  
      date: "2021-10-11T14:19:23.881Z",  
      user: {  
        name: "Matti Luukkainen"  
      }  
    }  
  ],  
  teams: [  
    {  
      name: "mosa climbers",  
      id: 2  
    }  
  ]  
},
```

The current code for the application is in its entirety on [GitHub](#), branch `part13-9`.

Exercises 13.19.-13.23.

Exercise 13.19.

Give users the ability to add blogs on the system to a *reading list*. When added to the reading list, the blog should be in the *unread* state. The blog can later be marked as *read*. Implement the reading list using a connection table. Make database changes using migrations.

In this task, adding to a reading list and displaying the list need not be successful other than directly using the database.

Exercise 13.20.

Now add functionality to the application to support the reading list.

Adding a blog to the reading list is done by making an HTTP POST to the path `/api/readinglists`, the request will be accompanied with the blog and user id:

```
"blogId": 10,
```

```
"userId": 3
```

```
,
```

Also modify the individual user route `GET /api/users/:id` to return not only the user's other information but also the reading list, e.g. in the following format:

```
{
  name: "Matti Luukkainen",
  username: "mluukkai@iki.fi",
  readings: [
    {
      id: 3,
      url: "https://google.com",
      title: "Clean React",
      author: "Dan Abramov",
      likes: 34,
      year: null
    },
    {
      id: 4,
      url: "https://google.com",
      title: "Clean Code",
      author: "Bob Martin",
      likes: 5,
      year: null
    }
  ]
}
```

At this point, information about whether the blog is read or not does not need to be available.

Exercise 13.21.

Expand the single-user route so that each blog in the reading list shows also whether the blog has been read *and* the id of the corresponding join table row.

For example, the information could be in the following form:

```
{
  name: "Matti Luukkainen",
  username: "mluukkai@iki.fi",
  readings: [
    {
      id: 3,
      url: "https://google.com",
      title: "Clean React",
      author: "Dan Abramov",
      likes: 34,
      year: null,
      readinglists: [
        {
          id: 1,
          blogId: 3
        }
      ]
    }
  ]
}
```

```

        [
          {
            read: false,
            id: 2
          }
        ],
      },
      [
        {
          id: 4,
          url: "https://google.com",
          title: "Clean Code",
          author: "Bob Martin",
          likes: 5,
          year: null,
          readinglists: [
            [
              {
                read: false,
                id: 3
              }
            ]
          ]
        }
      ]
    ]
  }
}

```

Note: there are several ways to implement this functionality. [This](#) should help.

Note also that despite having an array field *readinglists* in the example, it should always just contain exactly one object, the join table entry that connects the book to the particular user's reading list.

Exercise 13.22.

Implement functionality in the application to mark a blog in the reading list as read. Marking as read is done by making a request to the *PUT /api/readinglists/:id* path, and sending the request with

```
{ "read": true }
```

The user can only mark the blogs in their own reading list as read. The user is identified as usual from the token accompanying the request.

Exercise 13.23.

Modify the route that returns a single user's information so that the request can control which of the blogs in the reading list are returned:

- *GET /api/users/:id* returns the entire reading list
- *GET /api/users/:id?read=true* returns blogs that have been read
- *GET /api/users/:id?read=false* returns blogs that have not been read

Concluding remarks

The state of our application is starting to be at least acceptable. However, before the end of the section, let's look at a few more points.

Eager vs lazy fetch

When we make queries using the *include* attribute:

```
User.findOne({  
  include: [  
    {  
      model: note  
    }  
  ]  
})
```

The so-called **eager fetch** occurs, i.e. all the rows of the tables attached to the user by the join query, in the example the notes made by the user, are fetched from the database at the same time. This is often what we want, but there are also situations where you want to do a so-called *lazy fetch*, e.g. search for user related teams only if they are needed.

Let's now modify the route for an individual user so that it fetches the user's teams only if the query parameter *teams* is set in the request:

```
router.get('/:id', async (req, res) => {  
  const user = await User.findByPk(req.params.id, {  
    attributes: { exclude: [''] },  
    include: [{  
      model: note,  
      attributes: { exclude: ['userId'] }  
    },  
    {  
      model: Note,  
      as: 'marked_notes',  
      attributes: { exclude: ['userId'] },  
      through: {  
        attributes: []  
      },  
      include: [  
        {  
          model: user,  
          attributes: ['name']  
        }  
      ],  
    }  
  ]  
})  
  
if (!user) {  
  return res.status(404).end()  
}  
  
let teams = undefined; if (req.query.teams) { teams = await user.getTeams({  
  attributes: ['name'], joinTableAttributes: [] }) } res.json({  
...user.toJSON(), teams })})
```

So now, the *User.findByPk* query does not retrieve teams, but they are retrieved if necessary by the *user* method *getTeams*, which is automatically generated by Sequelize for the model object. Similar *get-* and a few other useful methods **are automatically generated** when defining associations for

tables at the Sequelize level.

Features of models

There are some situations where, by default, we do not want to handle all the rows of a particular table. One such case could be that we don't normally want to display users that have been *disabled* in our application. In such a situation, we could define the default **scopes** for the model like this:

```
class User extends Model {}

User.init[]{
  // field definition
}, [
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'user',
  defaultScope: [
    {
      where: {
        admin: true
      }
    },
    {
      where: {
        disabled: true
      }
    }
  ],
  scopes: [
    {
      where: {
        disabled: false
      }
    }
  ]
}

module.exports = User
```

Now the query caused by the function call *User.findAll()* has the following WHERE condition:

```
WHERE "user". "disabled" = false;
```

For models, it is possible to define other scopes as well:

```
User.init[]{
  // field definition
}, [
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'user',
  defaultScope: [
    {
      where: [
        {
          disabled: false
        }
      ]
    }
  ],
  scopes: [
    {
      where: [
        {
          admin: true
        }
      ]
    },
    {
      where: [
        {
          disabled: true
        }
      ],
      name(value) {
        return {
          where: [
            {
              name: [
                [Op.iLike]: value
              ]
            }
          ]
        }
      }
    }
  ]
}
```

Scopes are used as follows:

```
// all admins
const adminUsers = await User.scope('admin').findAll()

// all inactive users
const disabledUsers = await User.scope('disabled').findAll()

// users with the string jami in their name
const jamiUsers = await User.scope({ method: ['name', '%jami%'] }).findAll()
```

It is also possible to chain scopes:

```
// admins with the string jami in their name
const jamiUsers = await User.scope('admin', { method: ['name', '%jami%'] }).findAll()
```

Since Sequelize models are normal [JavaScript classes](#), it is possible to add new methods to them.

Here are two examples:

```
const { Model, DataTypes, Op } = require('sequelize')
const Note = require('../note')
const { sequelize } = require('../util/db')

class User extends Model {
  async number_of_notes() { return await this.getNotes().length }
  static async with_notes(limit) {
    return await User.findAll({
      attributes: [],
      include: [
        {
          include: [
            {
              model: Note,
              attributes: []
            }
          ],
          group: ['user.id'],
          having: sequelize.literal(`COUNT(notes.id) > ${limit}`)
        }
      ]
    })
  }
}

User.init()
// ...
}

module.exports = User
```

The first of the methods `numberOfNotes` is an *instance method*, meaning that it can be called on instances of the model:

```
const jami = await User.findOne({ name: 'Jami Kousa' })
const cnt = await jami.number_of_notes()
console.log(`Jami has created ${cnt} notes`)
```

Within the instance method, the keyword `this` therefore refers to the instance itself:

```
async number_of_notes() {
  return await this.getNotes().length
}
```

The second of the methods, which returns those users who have at least X , the number specified by the parameter, amount of notes is a *class method*, i.e. it is called directly on the model:

```
const users = await User.with_notes(2)
console.log(JSON.stringify(users, null, 2))
users.forEach(u => {
  console.log(u.name)
})
```

Repeatability of models and migrations

We have noticed that the code for models and migrations is very repetitive. For example, the model of teams

```
class Team extends Model {}

Team.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  name: {
    type: DataTypes.TEXT,
    allowNull: false,
    unique: true
  }
}, {
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'team'
})

module.exports = Team
```

and migration contain much of the same code

```
const { DataTypes } = require('sequelize')

module.exports = [
  up: async ({ context: queryInterface }) => {
    await queryInterface.createTable('teams', {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
      },
      name: {
        type: DataTypes.TEXT,
        allowNull: false,
        unique: true
      }
    })
  },
  down: async ({ context: queryInterface }) => {
    await queryInterface.dropTable('teams')
  }
]
```

```

        allowNull: false,
        unique: true
    },
},
),
down: async ({ context: queryInterface }) => {
    await queryInterface.dropTable('teams')
},
}

```

Couldn't we optimize the code so that, for example, the model exports the shared parts needed for the migration?

However, the problem is that the definition of the model may change over time, for example the *name* field may change or its data type may change. Migrations must be able to be performed successfully at any time from start to end, and if the migrations are relying on the model to have certain content, it may no longer be true in a month or a year's time. Therefore, despite the "copy paste", the migration code should be completely separate from the model code.

One solution would be to use Sequelize's [command line tool](#), which generates both models and migration files based on commands given at the command line. For example, the following command would create a *User* model with *name*, *username*, and *admin* as attributes, as well as the migration that manages the creation of the database table:

```
npx sequelize-cli model:generate --name User --attributes name:string,username:string,admin:boolean
```

From the command line, you can also run rollbacks, i.e. undo migrations. The command line documentation is unfortunately incomplete and in this course we decided to do both models and migrations manually. The solution may or may not have been a wise one.

Exercise 13.24.

Exercise 13.24.

Grand finale: [towards the end of part 4](#) there was mention of a token-criticality problem: if a user's access to the system is decided to be revoked, the user may still use the token in possession to use the system.

The usual solution to this is to store a record of each token issued to the client in the backend database, and to check with each request whether access is still valid. In this case, the validity of the token can be removed immediately if necessary. Such a solution is often referred to as a *server-side session*.

Now expand the system so that the user who has lost access will not be able to perform any actions that require login.

You will probably need at least the following for the implementation

- a boolean value column in the user table to indicate whether the user is disabled
 - it is sufficient to disable and enable users directly from the database

- a table that stores active sessions
 - a session is stored in the table when a user logs in, i.e. operation *POST /api/login*
 - the existence (and validity) of the session is always checked when the user makes an operation that requires login
- a route that allows the user to "log out" of the system, i.e. to practically remove active sessions from the database, the route can be e.g. *DELETE /api/logout*

Keep in mind that actions requiring login should not be successful with an "expired token", i.e. with the same token after logging out.

You may also choose to use some purpose-built npm library to handle sessions.

Make the database changes required for this task using migrations.

Submitting exercises and getting the credits

Exercises of this part are submitted just like in the previous parts, but unlike parts 0 to 7, the submission goes to an own [course instance](#). Remember that you have to finish all the exercises to pass this part!

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:

My submissions

part	exercises	hours	github	comment	solution
1	22	29	https://github.com/Kaltsoon/fs-cicd		show
total	22	29			

credits 1 based on exercises

Certificate

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

Note that you need a registration to the corresponding course part for getting the credits registered, see [here](#) for more information.