{() => fs}

Fullstack > Part 2 > Rendering a collection, modules

a 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)
```

copy

separate the things to be printed with a comma:

```
console.log('props value is', props)
```

copy

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]
```

copy

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:

copy

```
{
  "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>
```

copy

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>)
```

copy

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>,
]
```

copy

Which can then be placed inside *ul* tags:

```
const App = (props) => {
  const { notes } = props

  return (
```

copy

```
    <div>
      <h1>Notes</h1>
      <ul>
        {notes.map(note => <li>{note.content}</li>)}
      </ul>
    </div>
  )
}
```

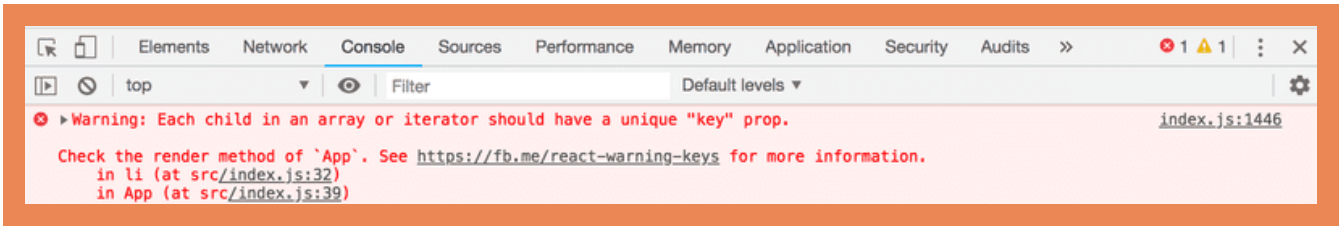
Because the code generating the *li* tags is JavaScript, it must be wrapped in curly braces in a JSX template just like all other JavaScript code.

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>
```

copy

- 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) => ...)
```

copy

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>
```

copy

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>
```

copy

```
      <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*:

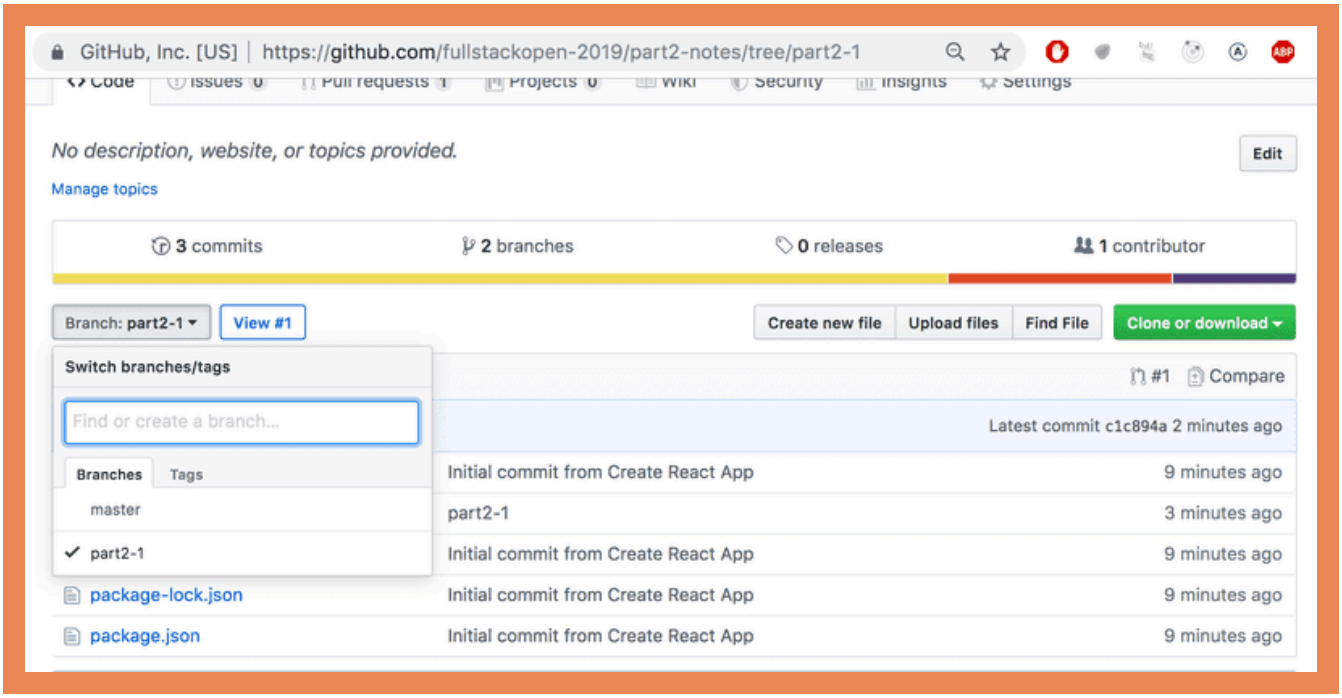
```
'./components/Note'
```

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:

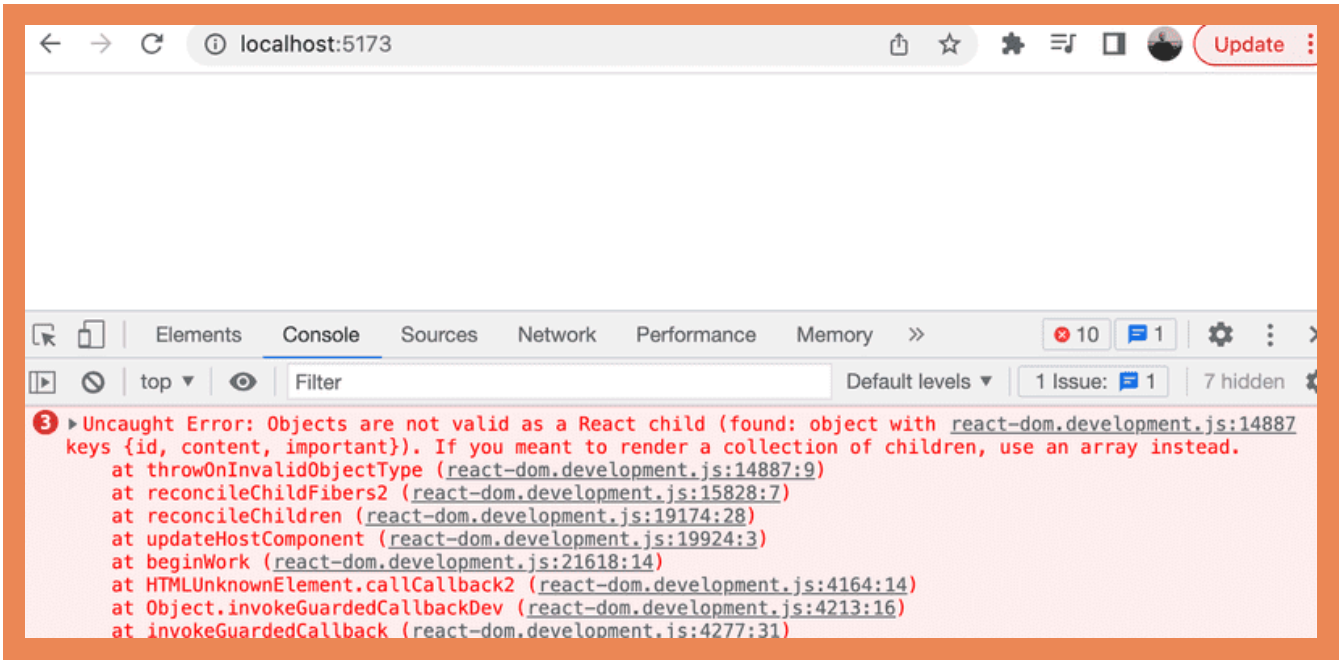


If you clone the project, run the command `npm install` before starting the application with `npm run dev`.

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:

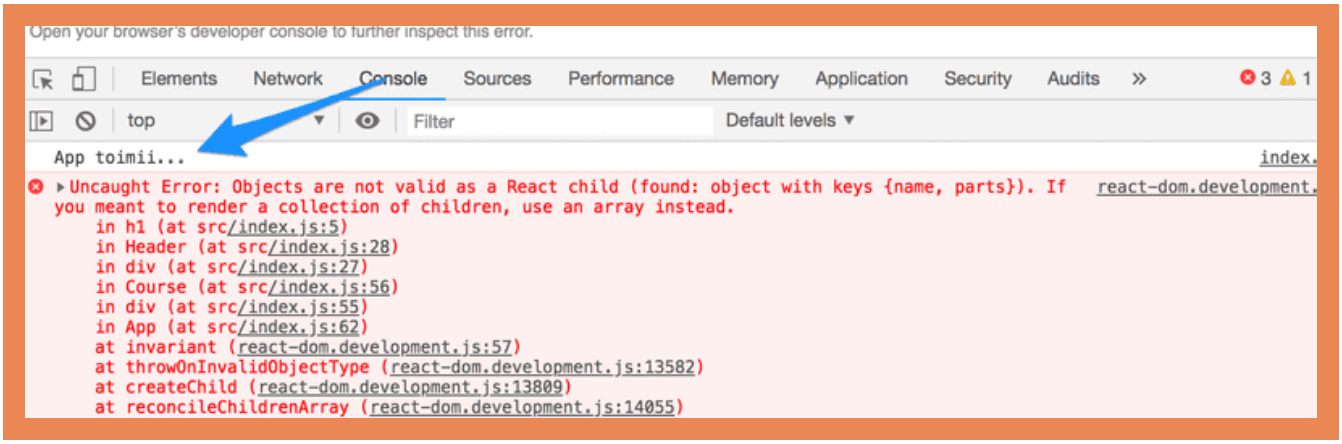
```
const Course = ({ course }) => (
  <div>
    <Header course={course} />
  </div>
)

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.



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:

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.

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**.

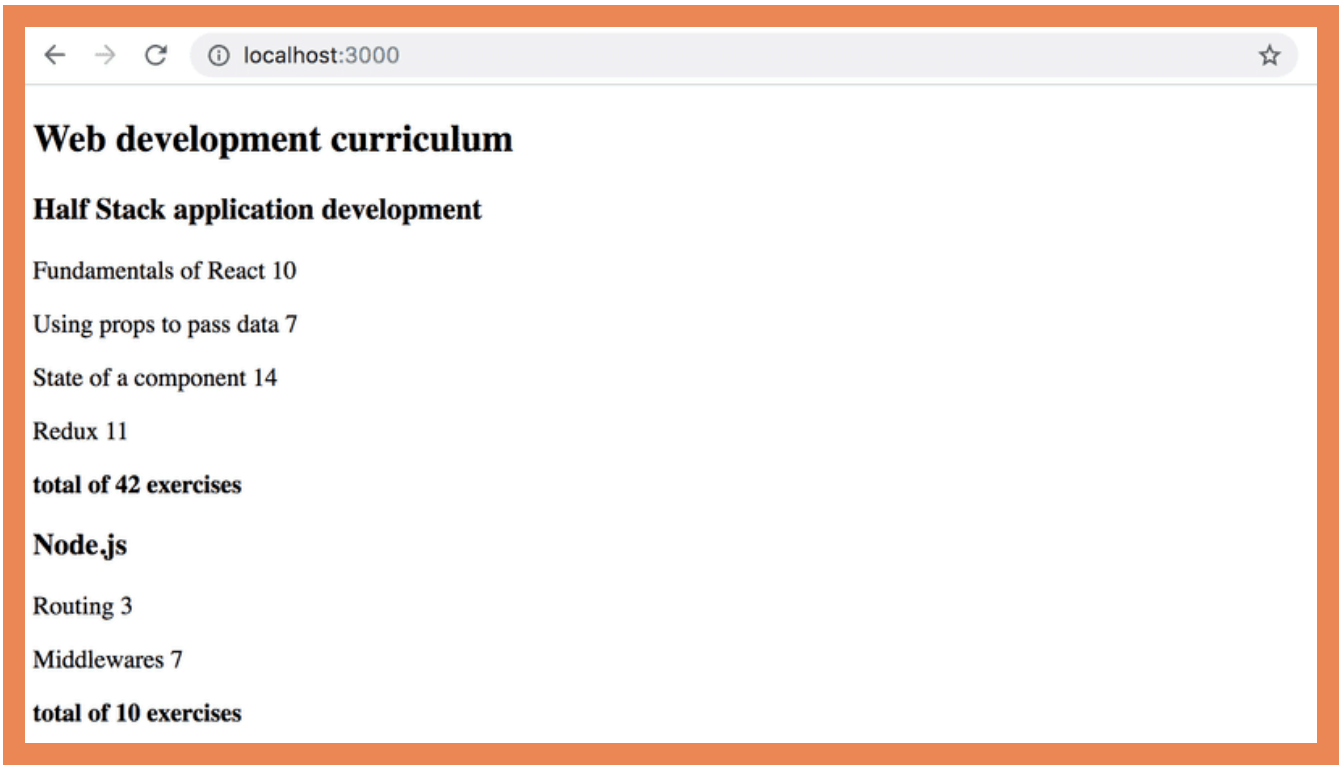
2.4: Course information step9

Let's extend our application to allow for an *arbitrary number* of courses:

```
const App = () => {
  const courses = [
    {
      name: 'Half Stack application development',
      id: 1,
      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:



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.

Propose changes to material

Part 1
Previous part

Part 2b
Next part

About course

Course contents

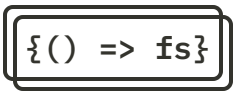
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON



b 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.

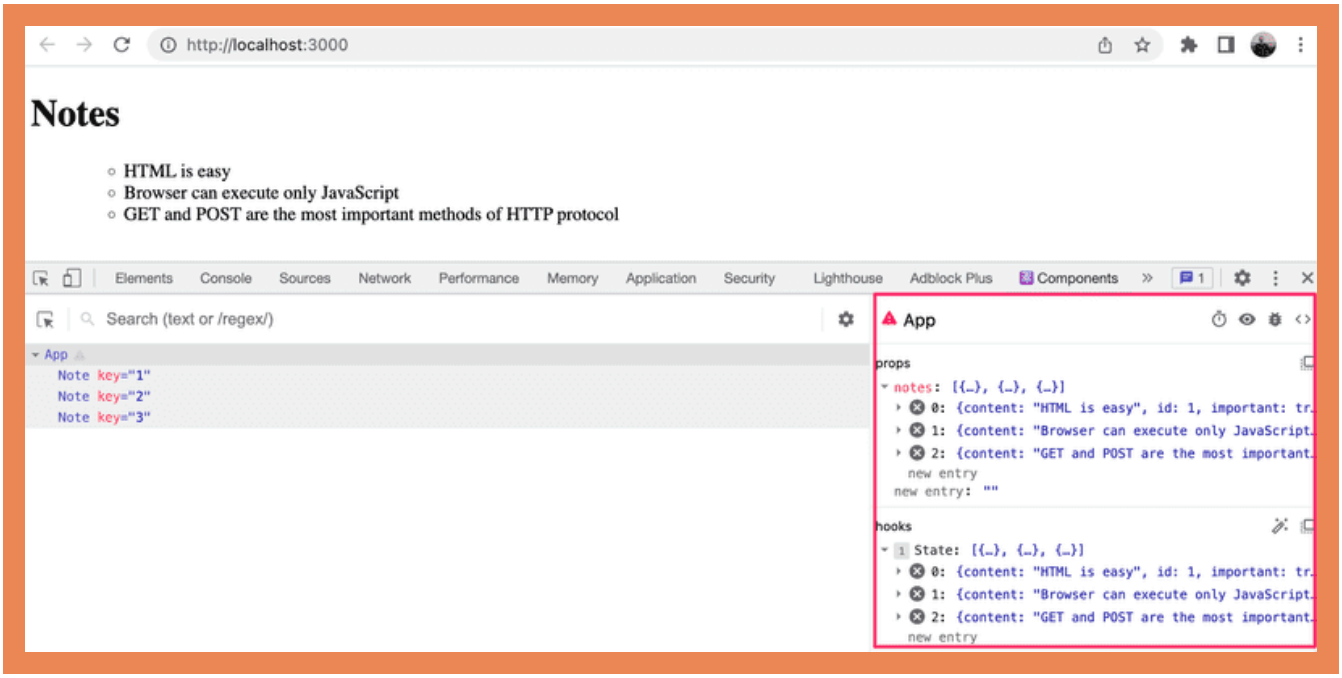
```
import { useState } from 'react'
import Note from './components/Note'

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 (
    <div>
      <h1>Notes</h1>
      <ul>
        {notes.map(note =>
          <Note key={note.id} note={note} />
        )}
      </ul>
      <form onSubmit={addNote}>
        <input />
        <button type="submit">save</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.

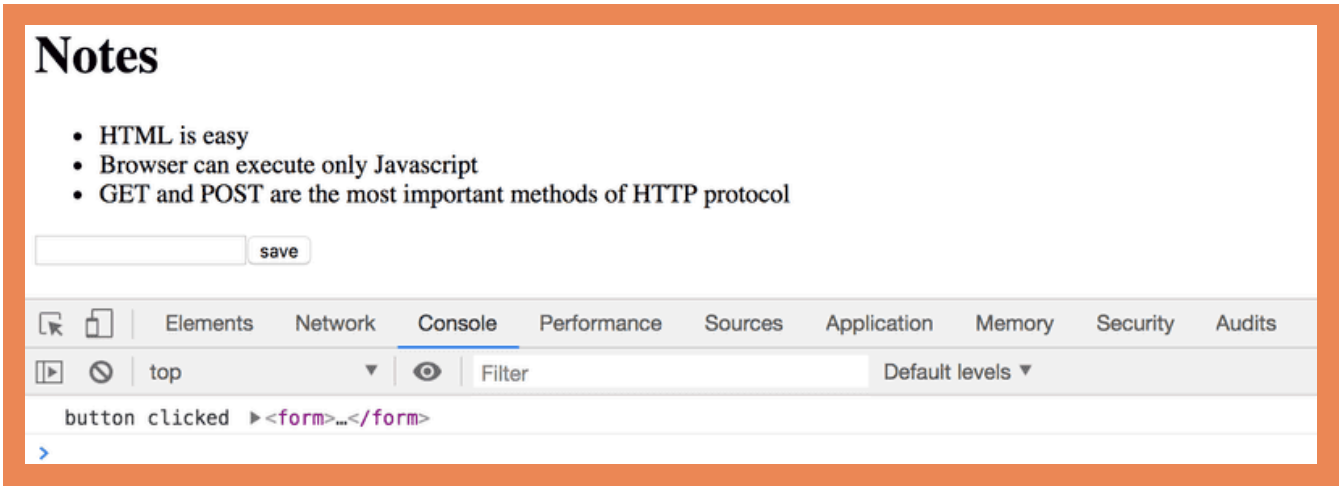
We use the method discussed in part 1 for defining our event handler:

```
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:



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}
/>
```

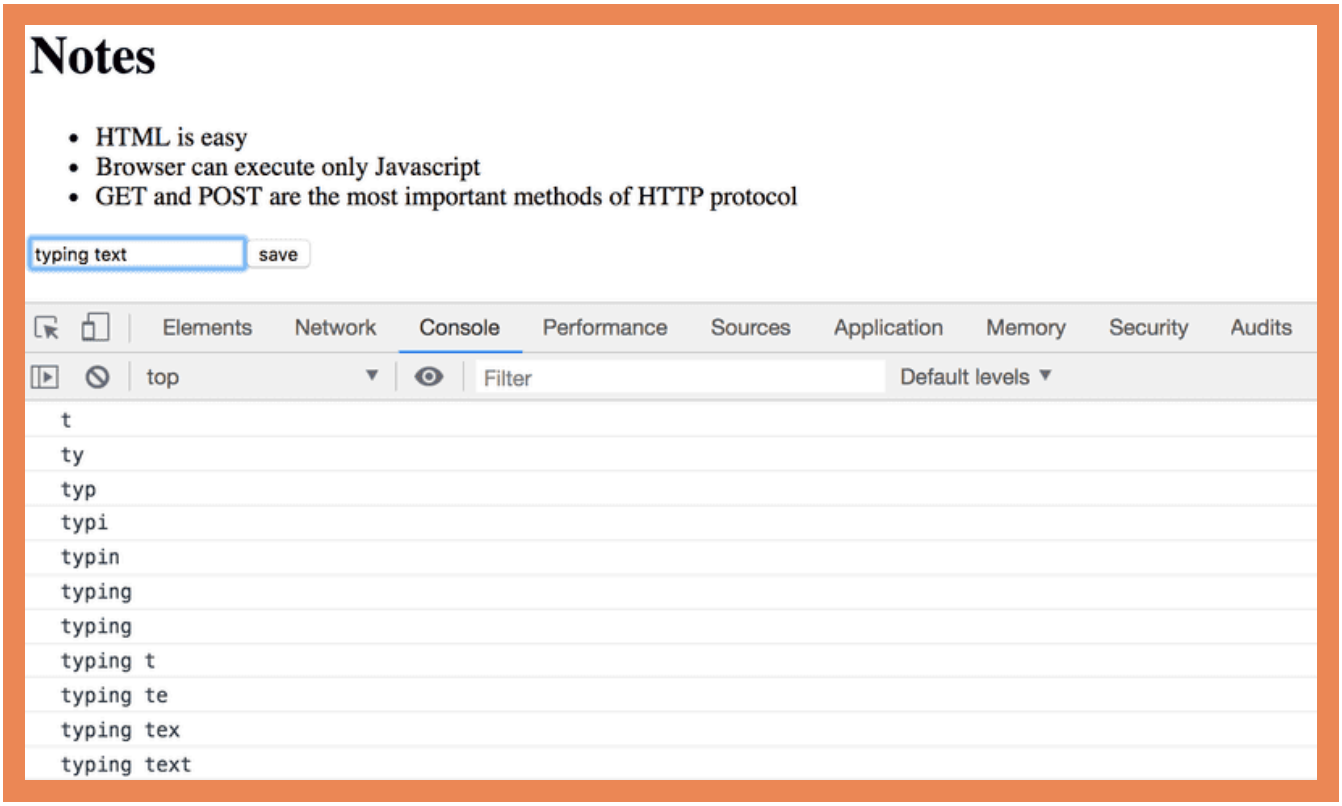
The event handler is called every time *a change occurs in the input element*. The event handler function receives the event object as its `event` parameter:

```
const handleNoteChange = (event) => {
  console.log(event.target.value)
  setNewNote(event.target.value)
}
```

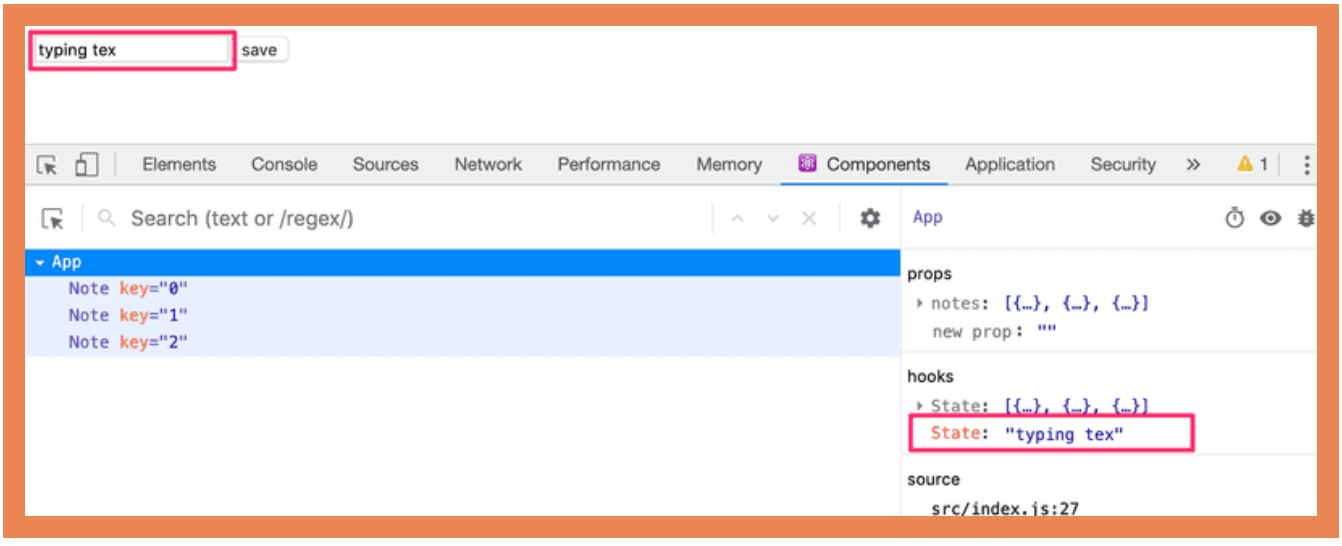
The `target` property of the event object now corresponds to the controlled *input* element, and `event.target.value` refers to the input value of that element.

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:



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))
```

copy

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('')
```

copy

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)

  // ...
}
```

copy

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.important === 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)
```

copy

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)
```

copy

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)}>
          show {showAll ? 'important' : 'all' }
        </button>
      </div>
      <ul>
        {notesToShow.map(note =>
          <Note key={note.id} note={note} />
        )}
      </ul>
      // ...
    </div>
  )
}
```

copy

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)
```

copy

The text of the button depends on the value of the `showAll` state:

```
show {showAll ? 'important' : 'all'}
```

copy

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, setNewName] = useState('')

  return (
    <div>
      <h2>Phonebook</h2>
      <form>
        <div>
          name: <input />
```

copy

```
        </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.

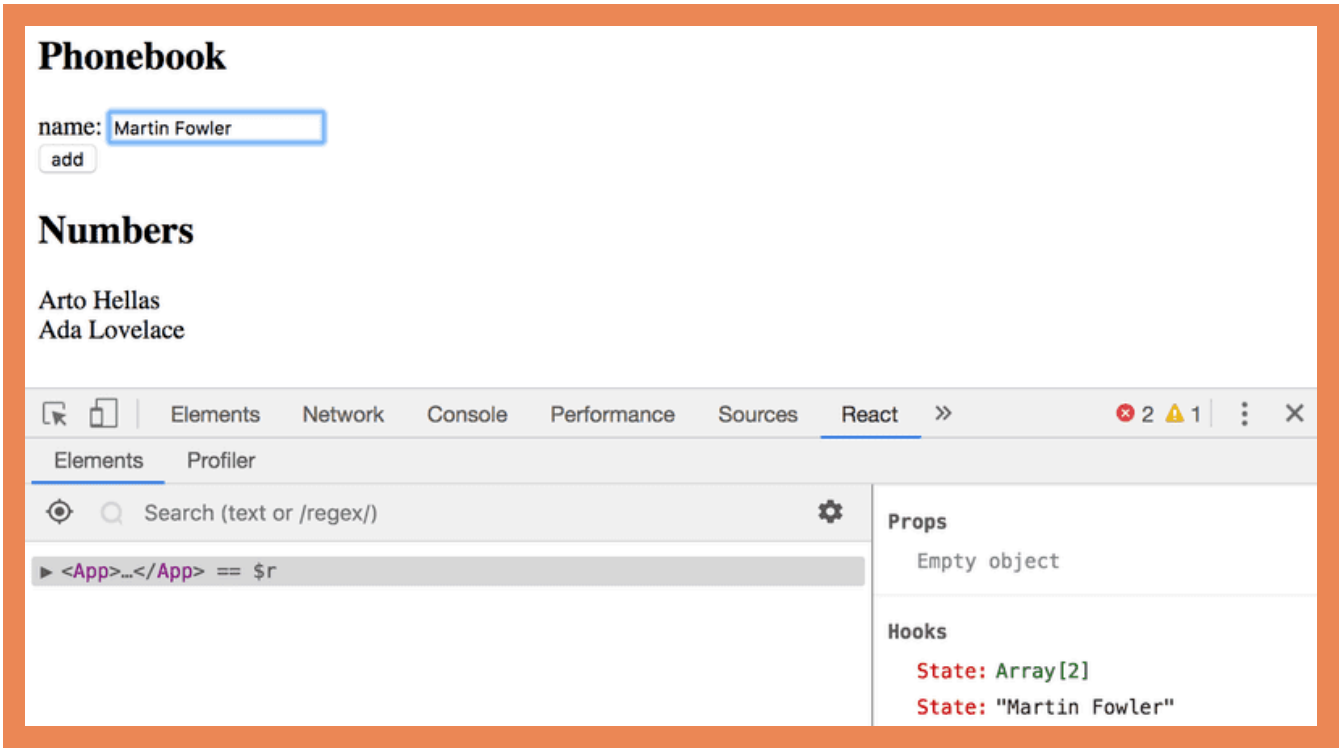
Sometimes it can be useful to render state and other variables as text for debugging purposes. You can temporarily add the following element to the rendered component:

```
<div>debug: {newName}</div>
```

copy

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:



Note the use of the React developer tools extension in the picture above!

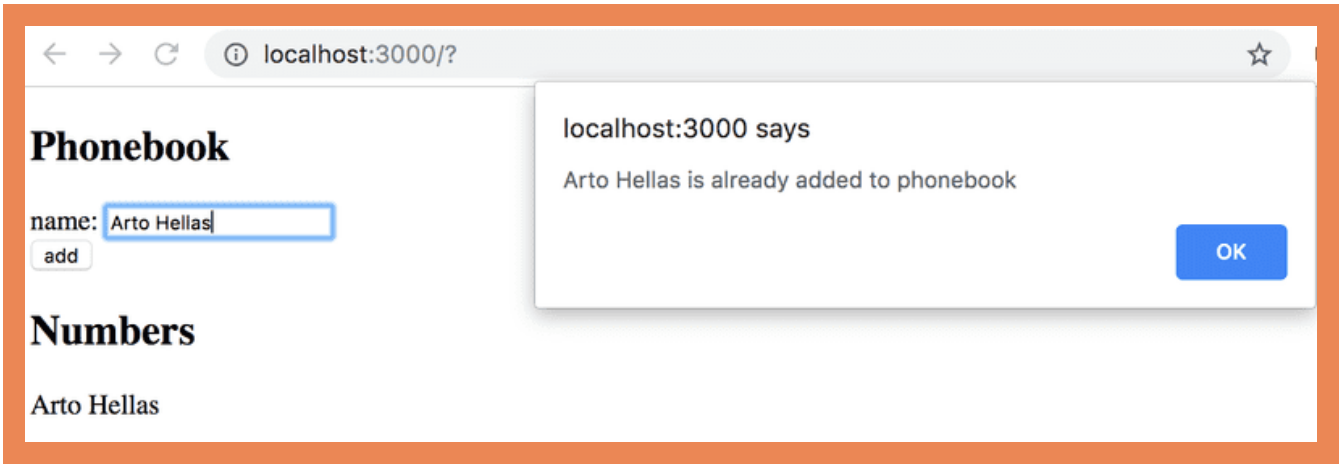
NB:

- 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:



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`
```

copy

If the `newName` variable holds the value *Arto Hellas*, the template string expression returns the string

```
`Arto Hellas is already added to phonebook`
```

copy

The same could be done in a more Java-like fashion by using the plus operator:

```
newName + ' is already added to phonebook'
```

copy

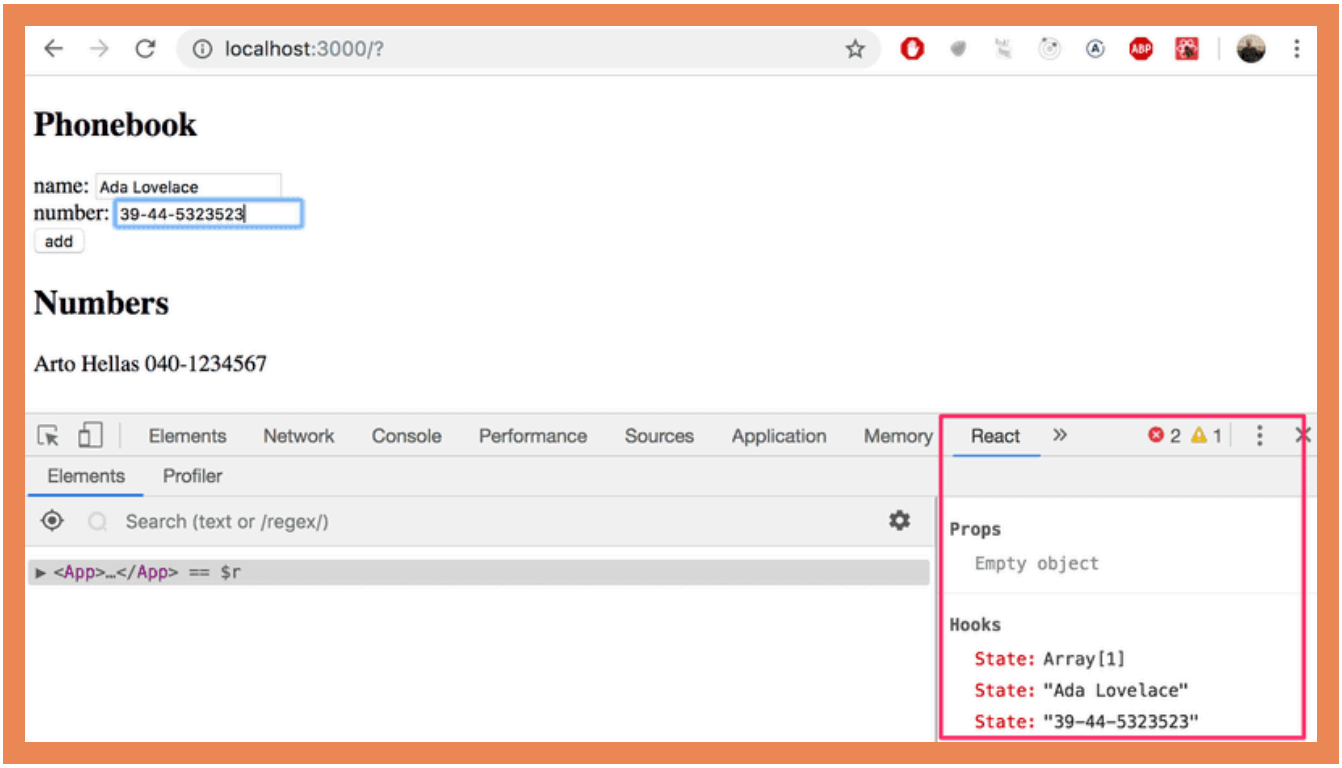
Using template strings is the more idiomatic option and the sign of a true JavaScript professional.

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:



2.9*: The Phonebook Step 4

Implement a search field that can be used to filter the list of people by name:

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.

Propose changes to material

Part 2a
Previous part

Part 2c
Next part

About course

Course contents

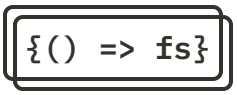
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON



c 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
    }
  ]
}
```

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
```

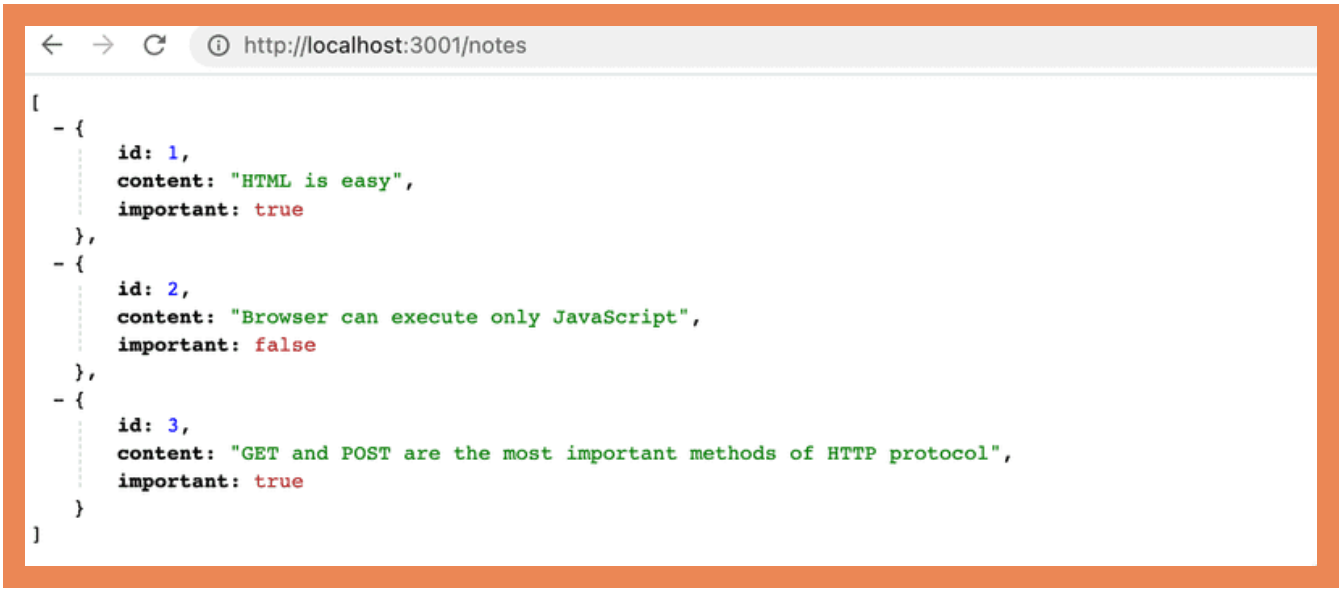
copy

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
```

copy

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:



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 < 50000000000) {
    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": {
    "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"
  },
}
```

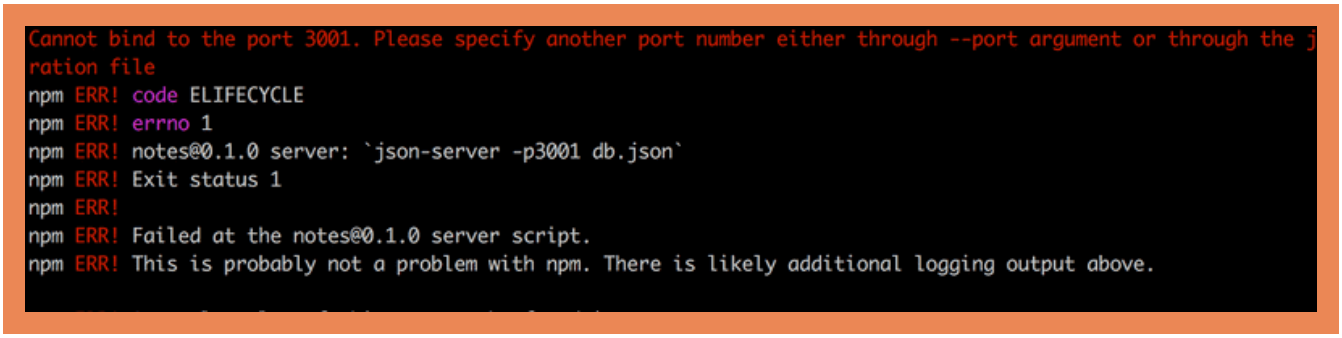
We can now conveniently, without parameter definitions, start the json-server from the project root directory with the command:

```
npm run server
```

copy

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:



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
```

copy

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

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:

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 => {
        console.log('promise fulfilled')
        setNotes(response.data)
      })
  }, [])
  console.log('render', notes.length, 'notes')

  // ...
}
```

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.

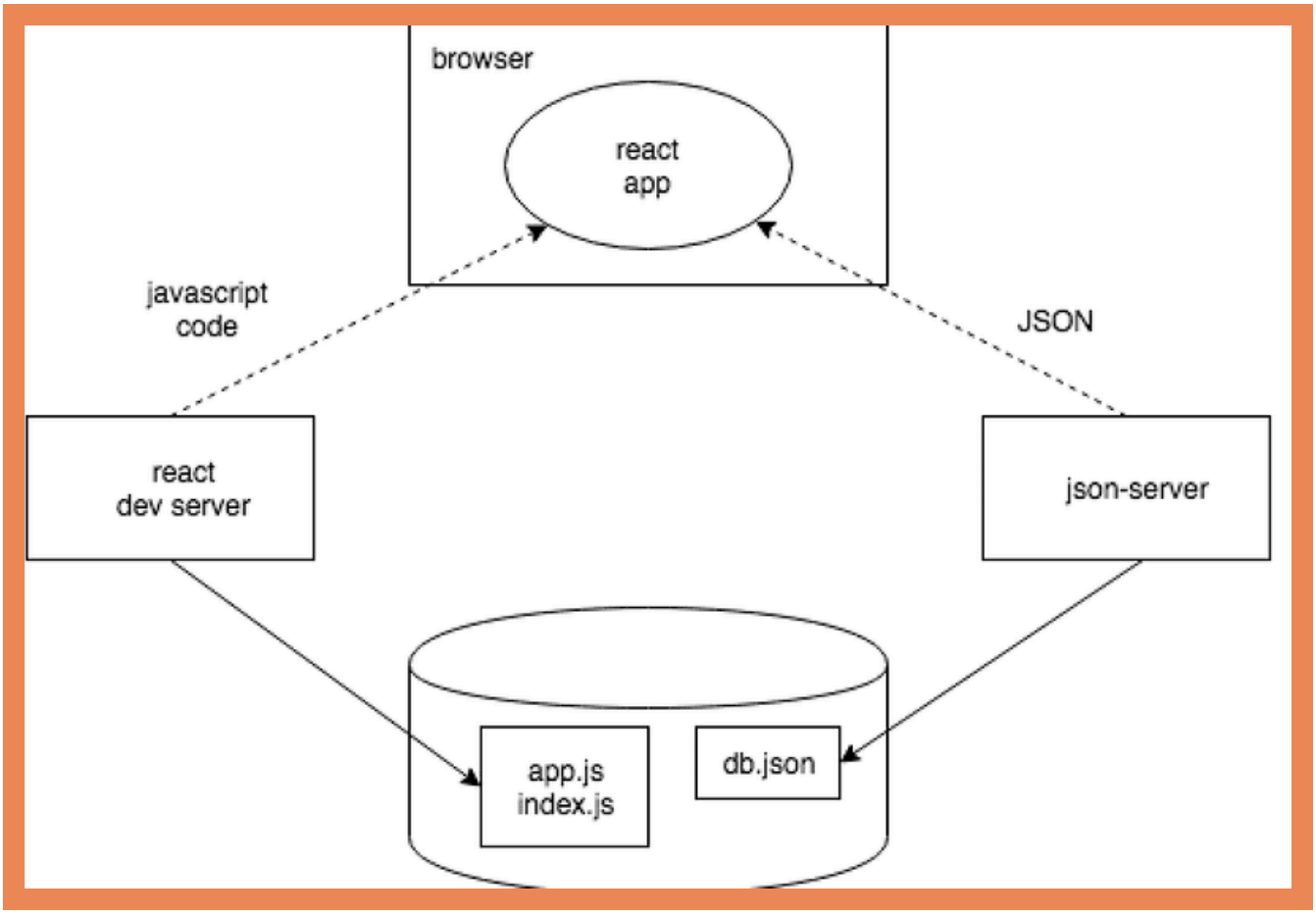
```
useEffect(() => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
}, [])
```

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.

Propose changes to material

Part 2b
Previous part

Part 2d
Next part

About course

Course contents

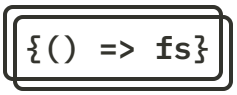
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON



d 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.

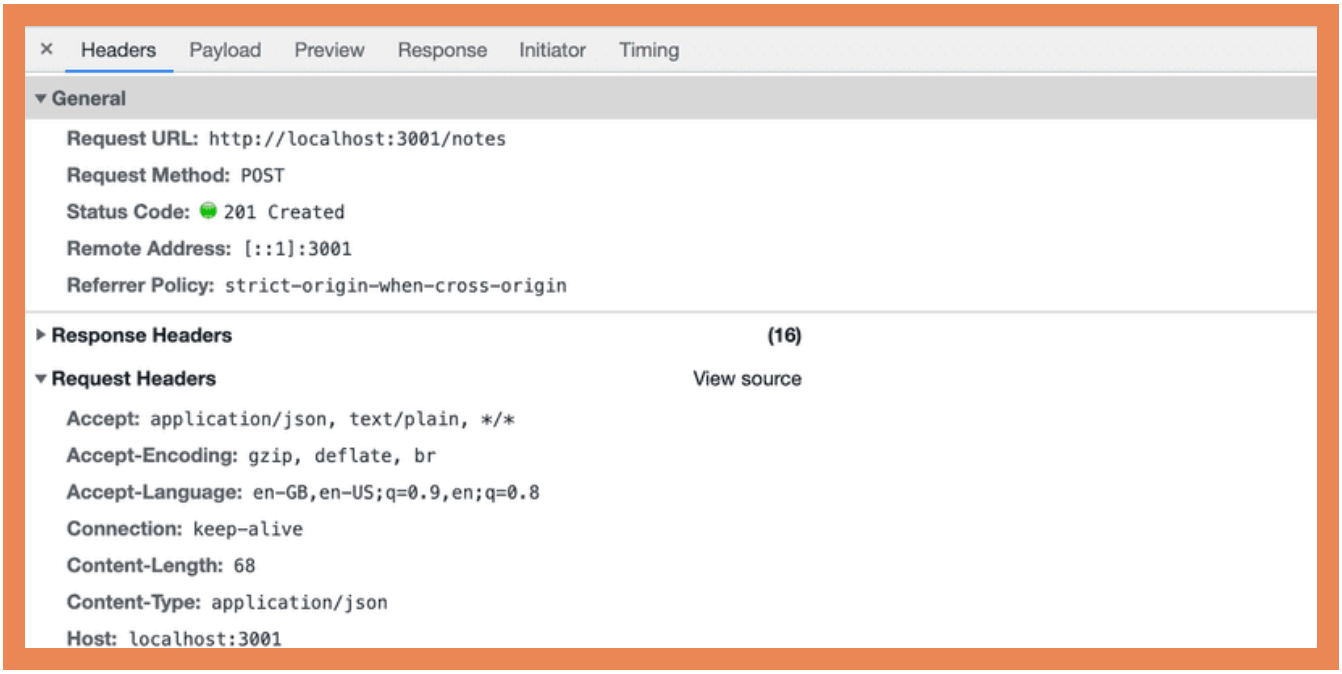
The object is sent to the server using the axios `post` method. The registered event handler logs the response that is sent back from the server to the console.

When we try to create a new note, the following output pops up in the console:

The newly created note resource is stored in the value of the *data* property of the `response` object.

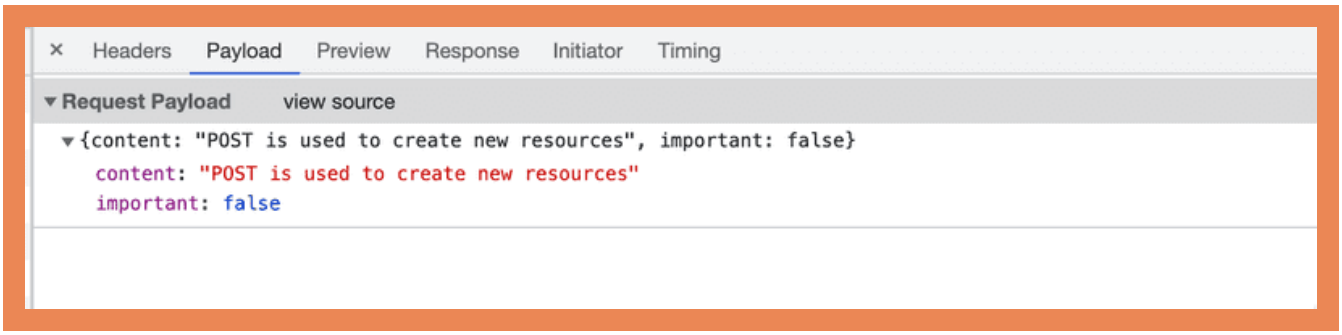
Quite often it is useful to inspect HTTP requests in the *Network* tab of Chrome developer tools, which was used heavily at the beginning of part 0.

We can use the inspector to check that the headers sent in the POST request are what we expected them to be:

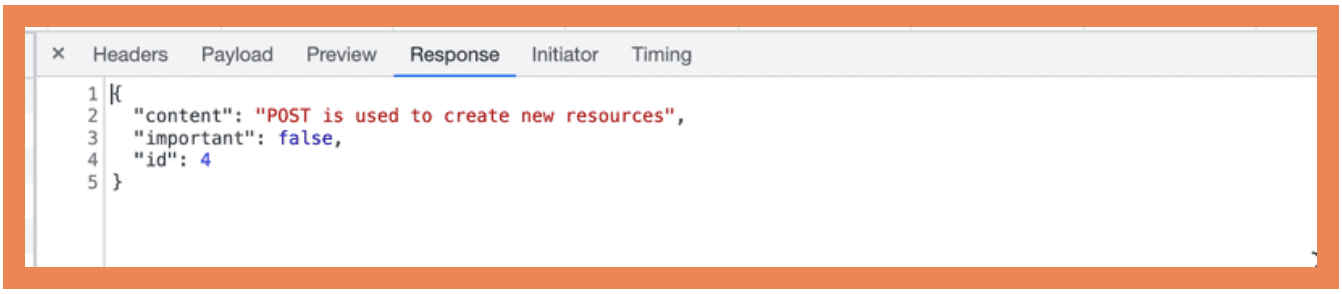


Since the data we sent in the POST request was a JavaScript object, axios automatically knew to set the appropriate *application/json* value for the *Content-Type* header.

The tab *payload* can be used to check the request data:



Also the tab *response* is useful, it shows what was the data the server responded with:



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:

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') }
```

copy

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')
```

copy

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`)
```

copy

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 => {
```

copy

```
    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 }
```

copy

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 => {
  // ...
```

copy

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 !== 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 = newObject => {
  return axios.post(baseUrl, newObject)
}

const update = (id, newObject) => {
  return axios.put(`${baseUrl}/${id}`, newObject)
}

export default {
  getAll: getAll,
  create: create,
  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: getAll,
  create: create,
```

```
  update: 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: getAll,
  create: create,
  update: 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
```

copy

In older versions of JavaScript we had to define an object like this:

```
const person = {
  name: name,
  age: age
}
```

copy

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 }
```

copy

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*.

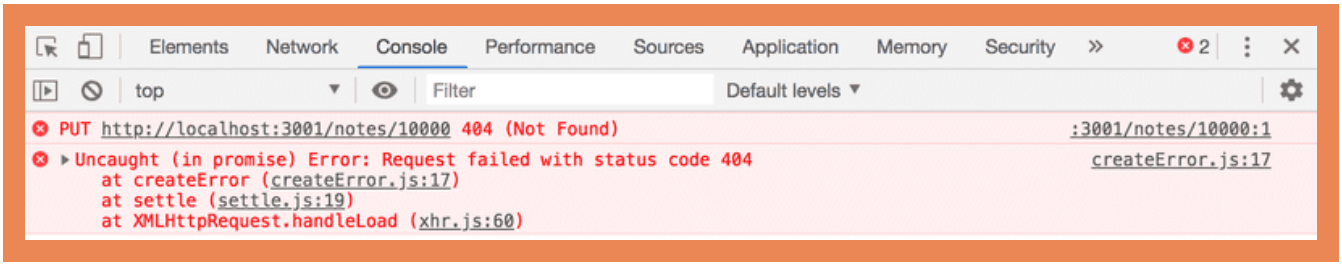
Promises and Errors

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.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)
```

copy

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.

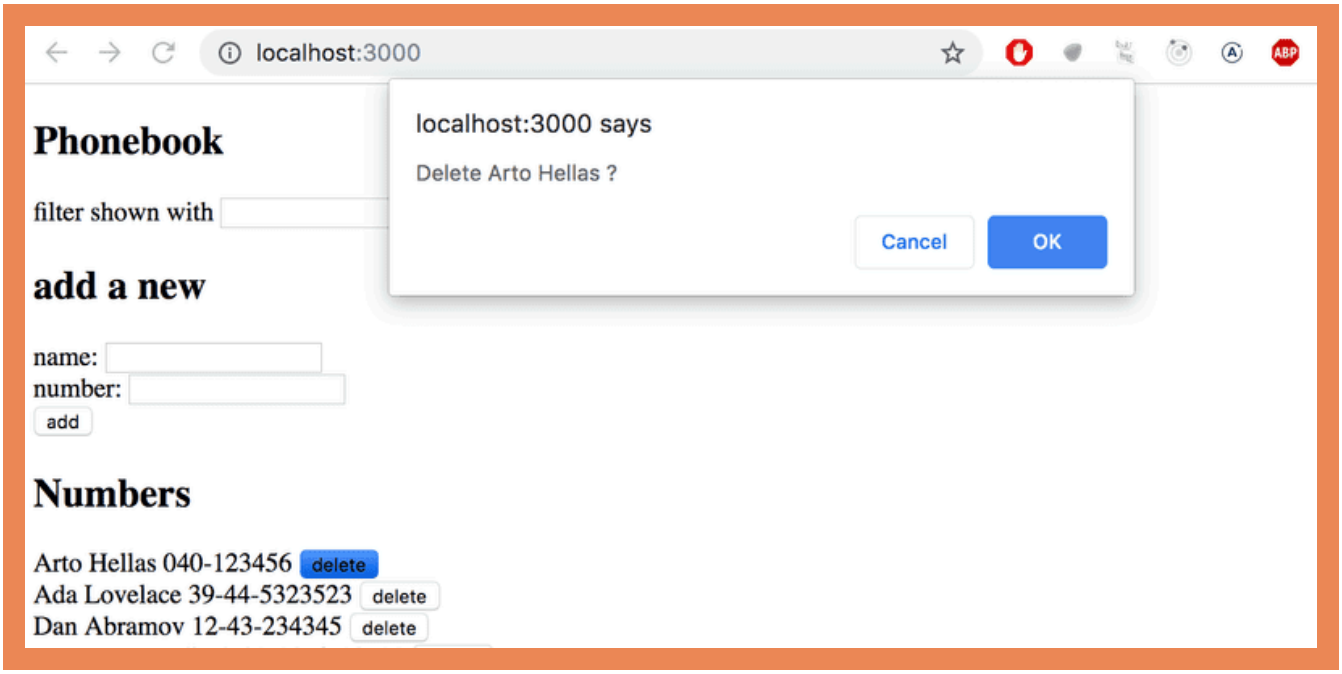
Currently, the numbers that are added to the phonebook are not saved to a backend server. Fix this situation.

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 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:

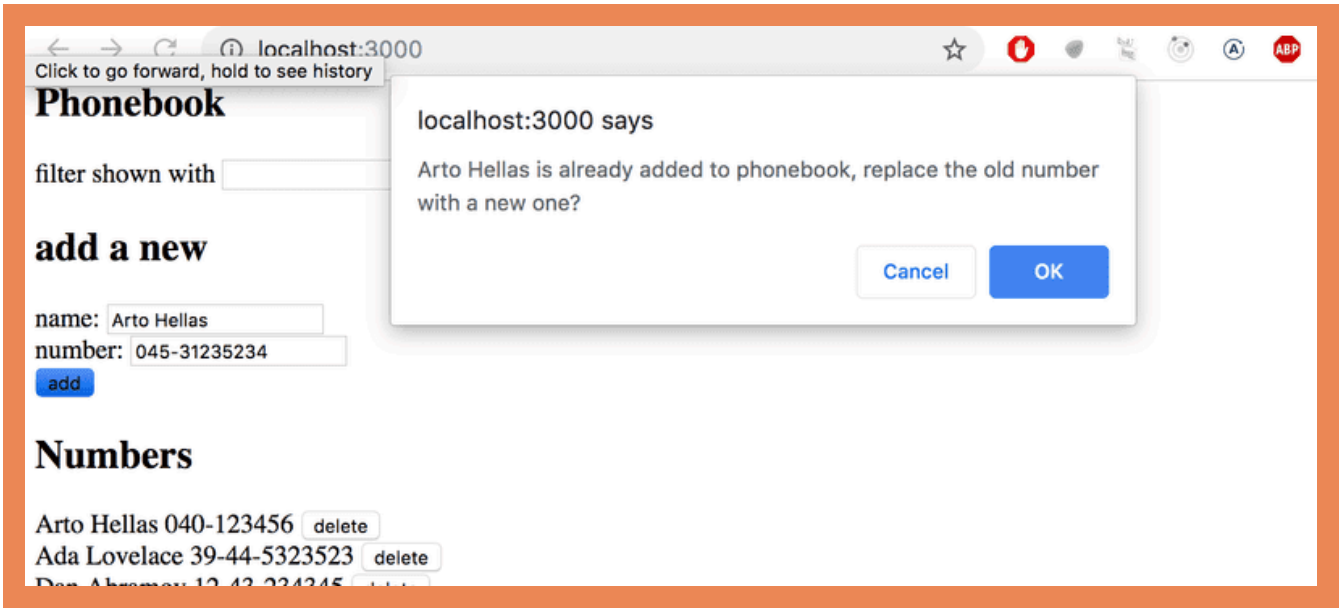
```
// use some other name for variable!
const delete = (id) => {
  // ...
}
```

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:



Propose changes to material

Part 2c
Previous part

Part 2e
Next part

About course

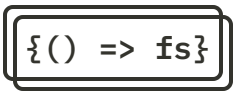
Course contents

FAQ

Partners

Challenge


HOUSTON



e 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'
```

copy

Let's add the following CSS rule to the *index.css* file:

```
h1 {
  color: green;
}
```

copy

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;
}
```

copy

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>
  )
}
```

copy

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;
}
```

copy

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>
```

copy

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 *li* 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:

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.

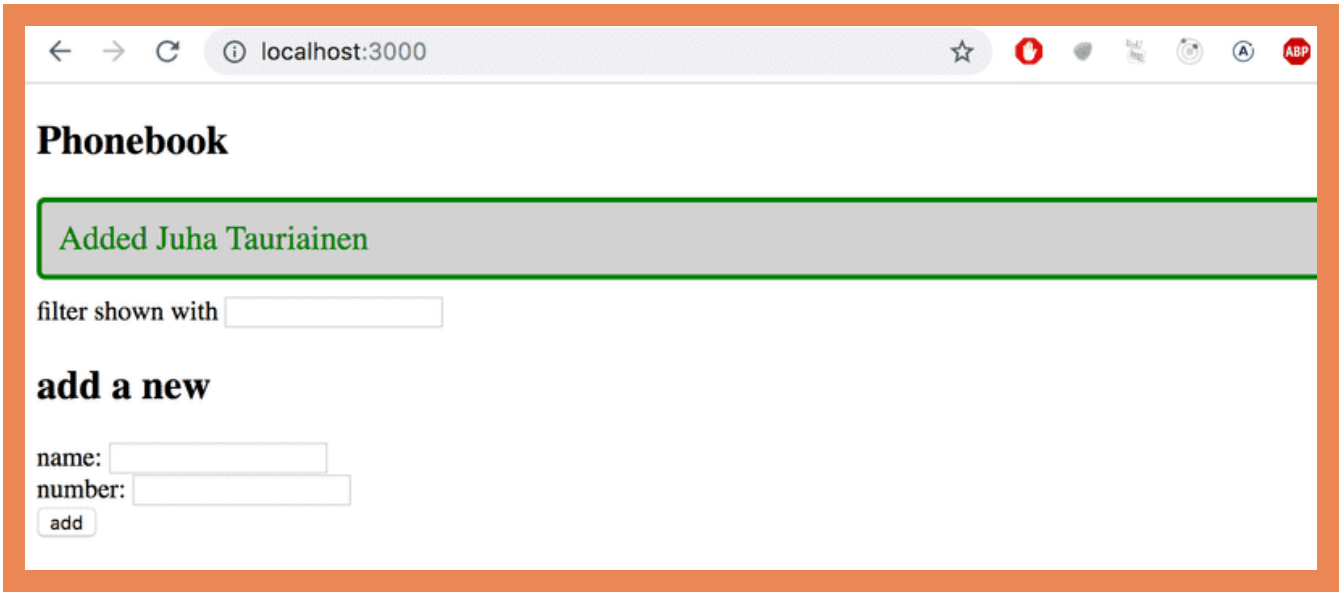
The structural units that make up the application's functional entities are React components. A React component defines the HTML for structuring the content, the JavaScript functions for determining functionality, and also the component's styling; all in one place. This is to create individual components 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.

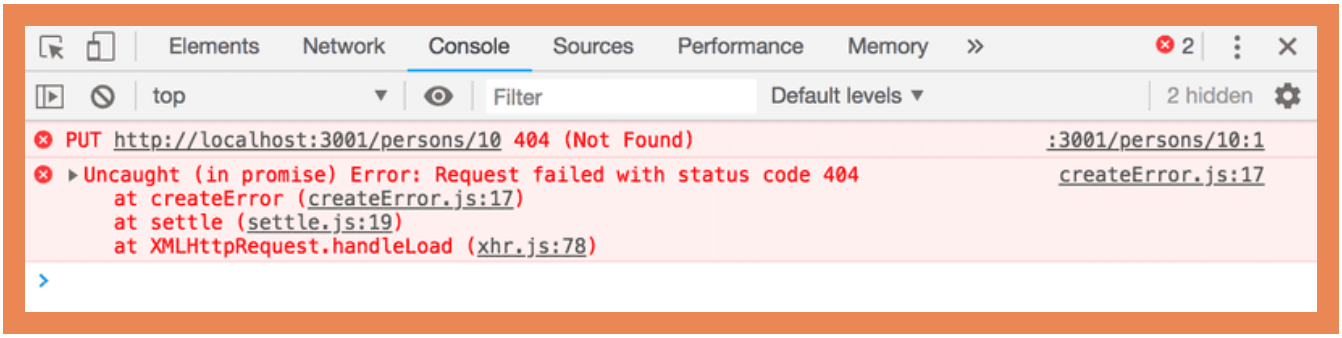
2.16: Phonebook step 11

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):

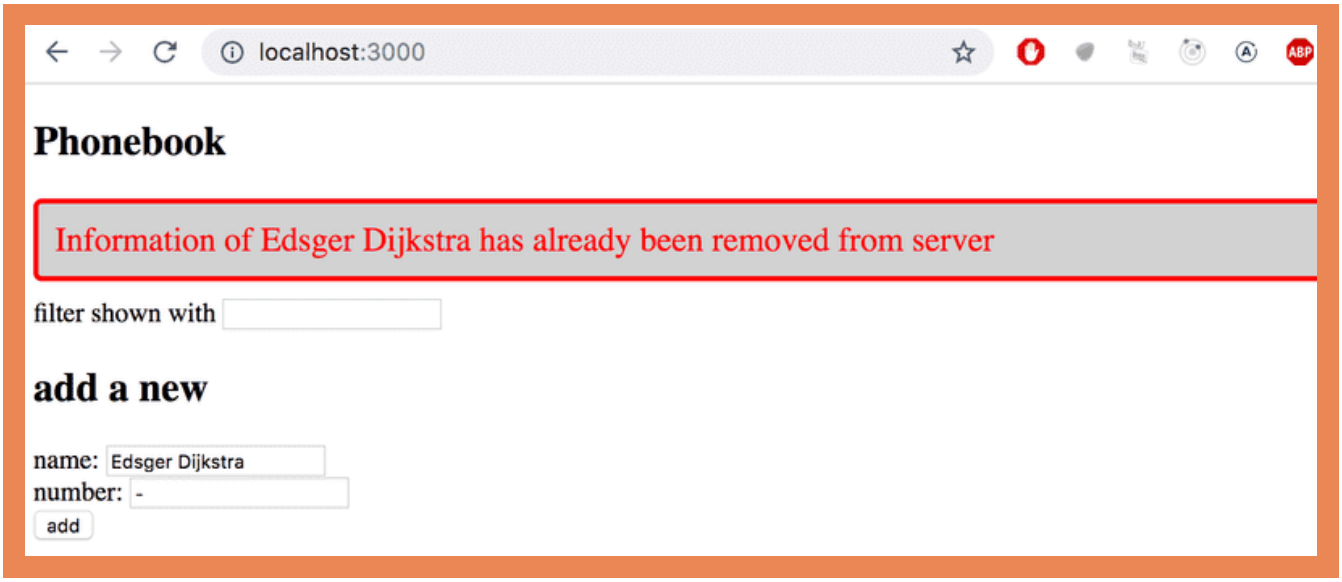


2.17*: Phonebook step 12

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:



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:



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.

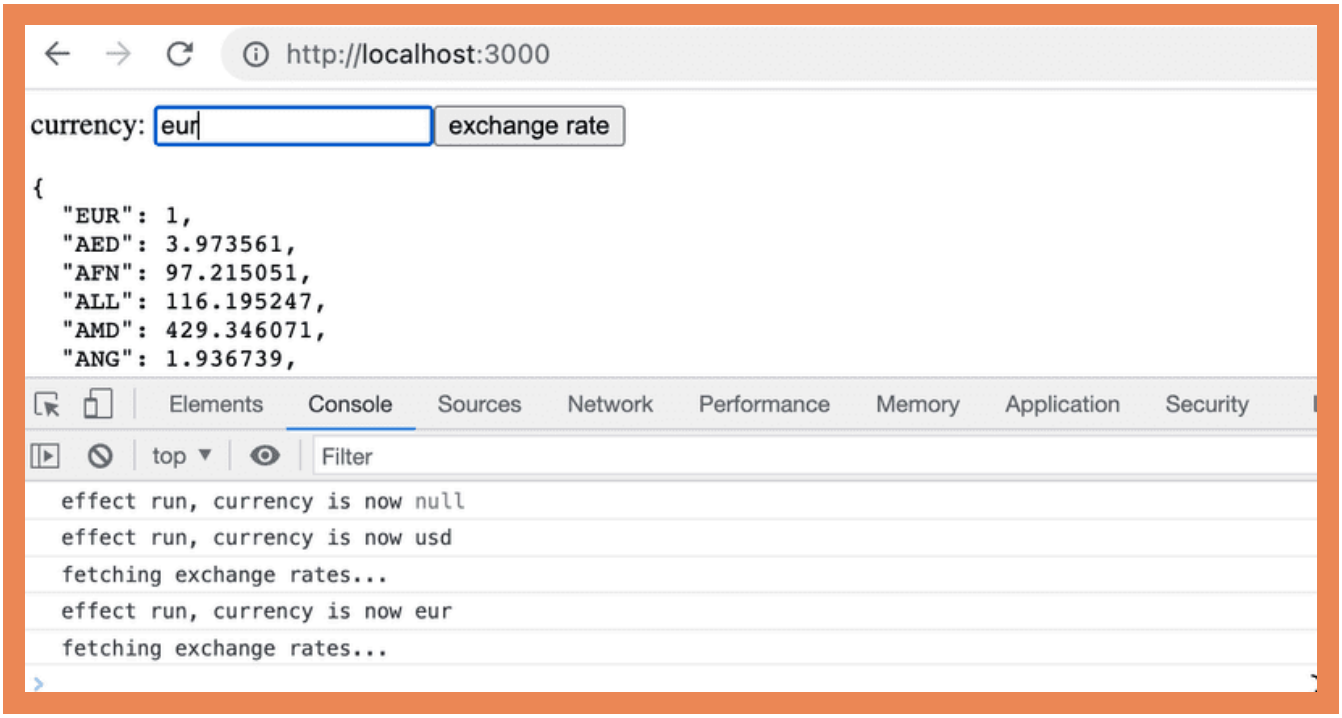
If the second parameter is an empty array `[]`, its content never changes and the effect is only run after the first render of the component. This is exactly what we want when we are initializing the app state from the server.

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 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.

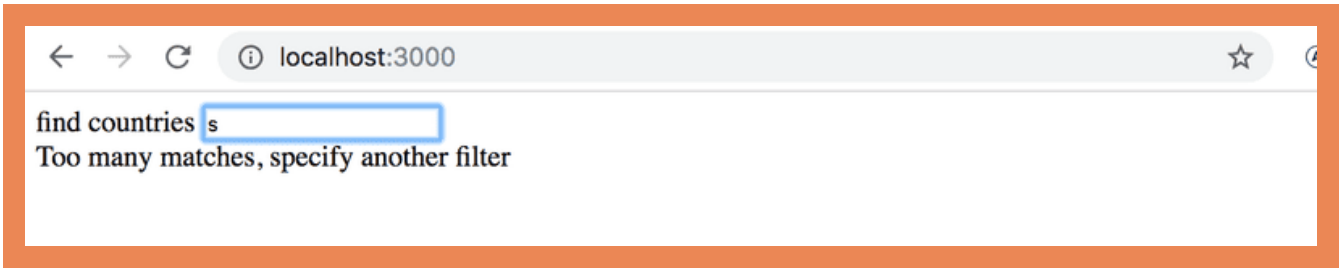
Exercises 2.18.-2.20.

2.18* Data for countries, step 1

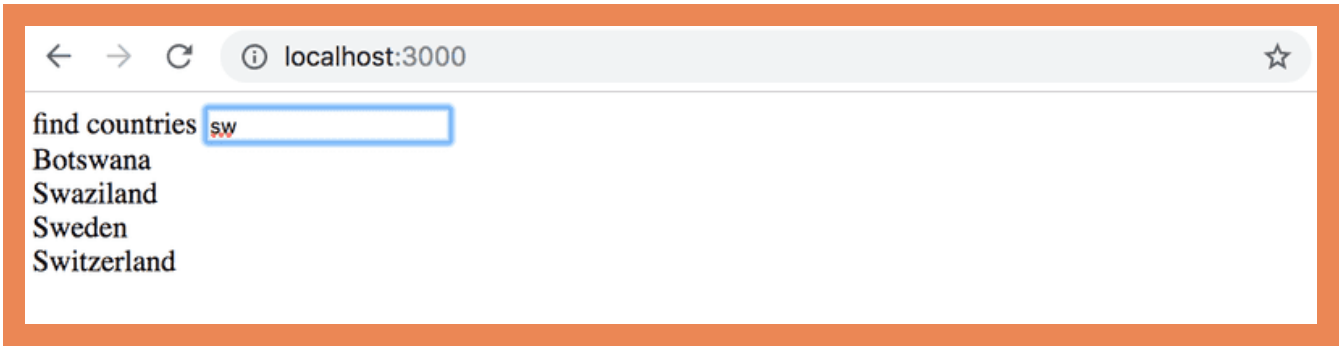
At https://studies.cs.helsinki.fi/restcountries/ you can find a service that offers a lot of information related to different countries in a so-called machine-readable format via the REST API. Make an application that allows you to view information from different countries.

The user interface is very simple. The country to be shown is found by typing a search query into the search field.

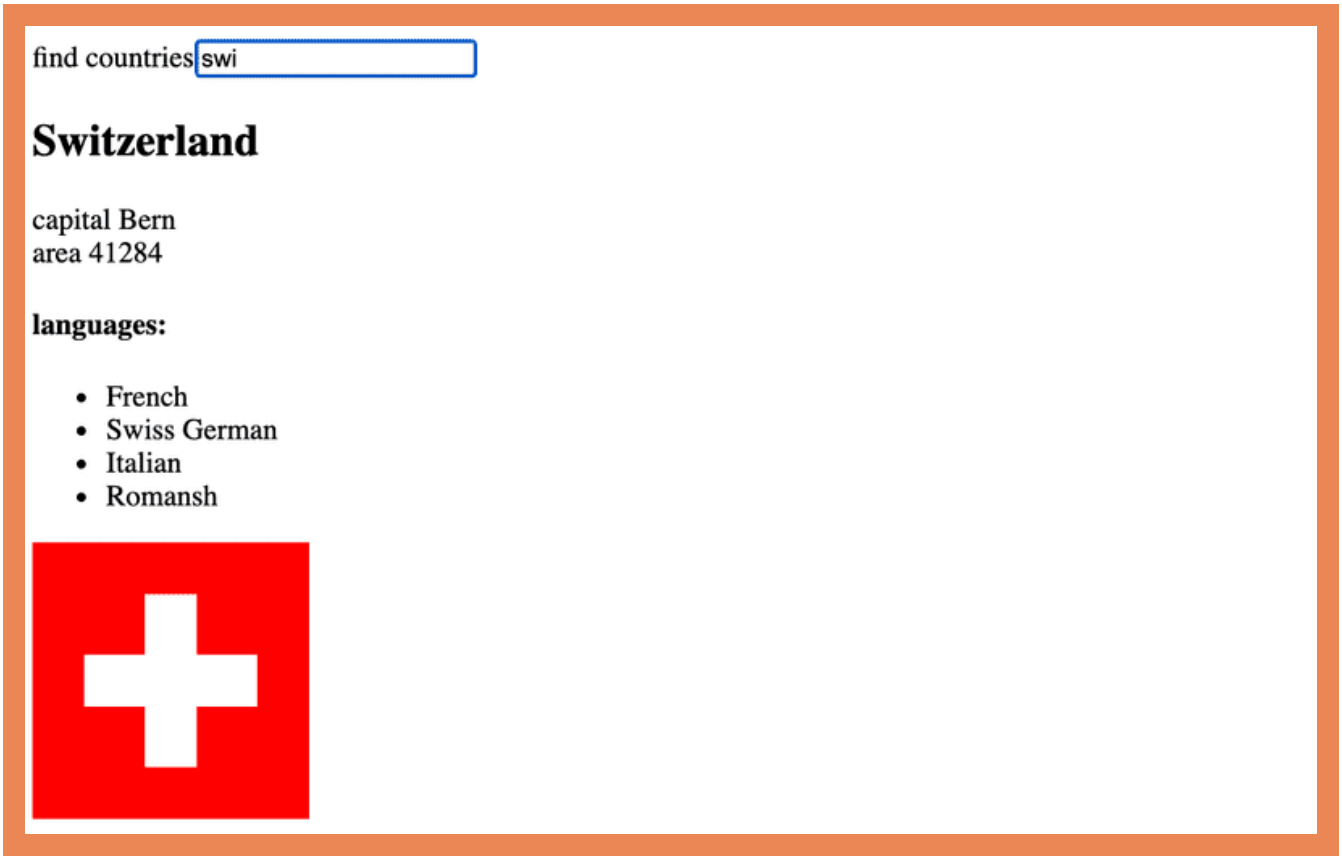
If there are too many (over 10) countries that match the query, then the user is prompted to make their query more specific:



If there are ten or fewer countries, but more than one, then all countries matching the query are shown:



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:

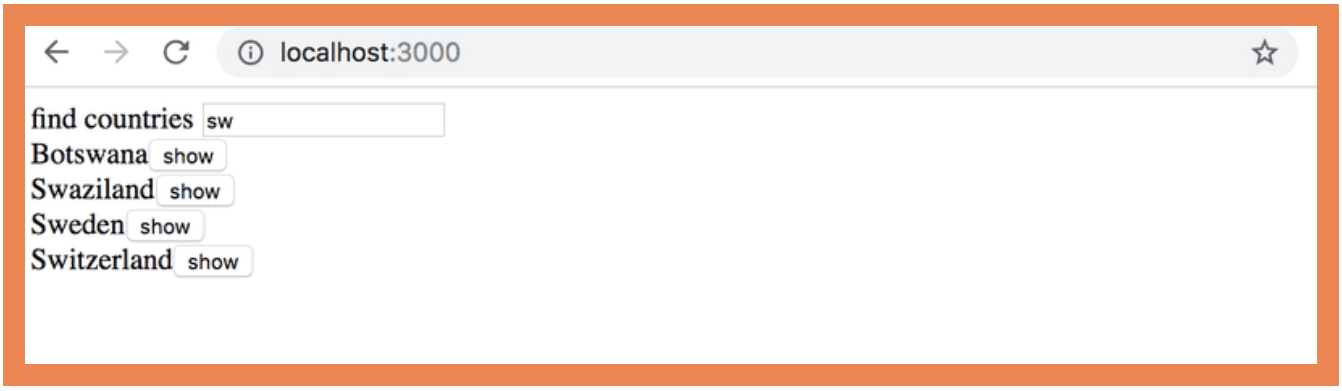


NB: It is enough that your application works for most countries. Some countries, like *Sudan*, can be hard to support since the name of the country is part of the name of another country, *South Sudan*. You don't need to worry about these edge cases.

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!

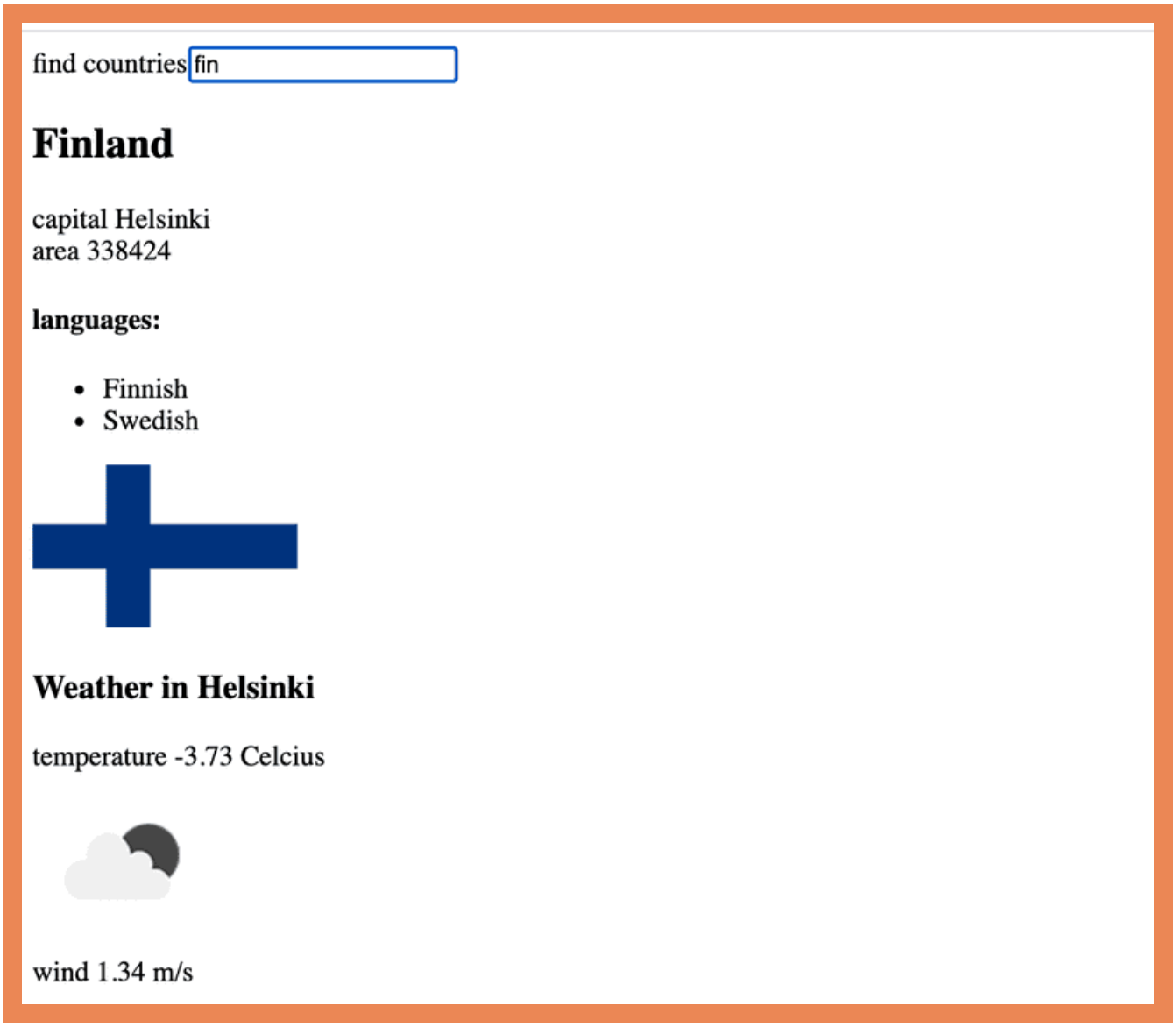
Improve on the application in the previous exercise, such that when the names of multiple countries are shown on the page there is a button next to the name of the country, which when pressed shows the view for that country:



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.



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.

Propose changes to material

Part 2d
Previous part

Part 3
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON