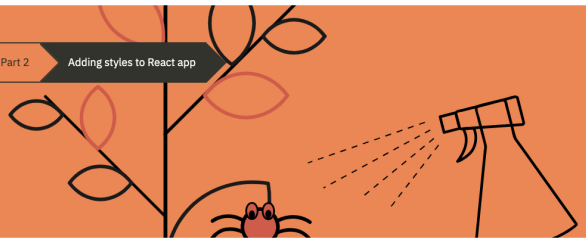


e Adding styles to React app

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'
```

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

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

CSS rules comprise of *selectors* and *declarations*. The selector defines which elements the rule should be applied to. The selector above is *h1*, which will match all of the *h1* header tags in our application.

The declaration sets the `color` property to the value *green*.

One CSS rule can contain an arbitrary number of properties. Let's modify the previous rule to make the text cursive, by defining the font style as *italic*:

```
h1 {
  color: green;
  font-style: italic;
}
```

There are many ways of matching elements by using different types of CSS selectors.

If we wanted to target, let's say, each one of the notes with our styles, we could use the selector *li*, as all of the notes are wrapped inside *li* tags:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important'
    : 'make important';

  return (
    <li>
      {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}
```

Let's add the following rule to our style sheet (since my knowledge of elegant web design is close to zero, the styles don't make much sense):

```
li {
  color: grey;
  padding-top: 3px;
  font-size: 15px;
}
```

Using element types for defining CSS rules is slightly problematic. If our application contained other *li* tags, the same style rule would also be applied to them.

If we want to apply our style specifically to notes, then it is better to use class selectors.

In regular HTML, classes are defined as the value of the *class* attribute:

```
<li class="note">some text...</li>
```

In React we have to use the className attribute instead of the class attribute. With this in mind, let's make the following changes to our *Note* component:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important'
    : 'make important';

  return (
    <li className='note'>
      {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}
```

Class selectors are defined with the `.classname` syntax:

```
.note {
  color: grey;
  padding-top: 5px;
  font-size: 15px;
}
```

If you now add other *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
2023</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.

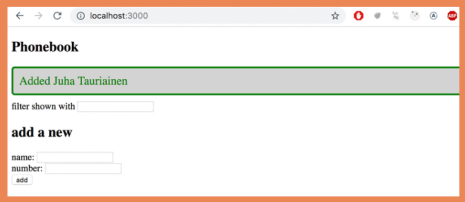
The structural units that make up the application's functional entities are React components. A React component defines the HTML for structuring the content, the JavaScript functions for determining functionality, and also the component's styling; all in one place. This is to create individual components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

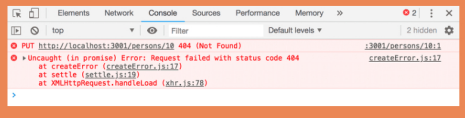
2.16: Phonebook step11

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):

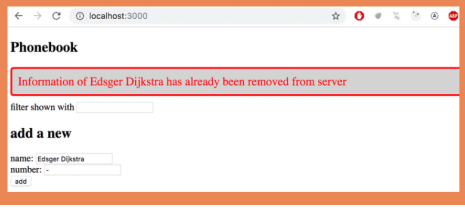


2.17*: Phonebook step12

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 2 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 = () => {                                                    copy
  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 = () => {                                                    copy
  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                               copy
  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')                         copy
```

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(() => {                                                    copy
    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 = () => {                                                    copy
  const [notes, setNotes] = useState(null)

  // ...
```

on the first render the following code gets executed

```
notesToShow = notes                                                    copy

// ...

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 = () => {                                                    copy
  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(() => {                                                    copy
    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.

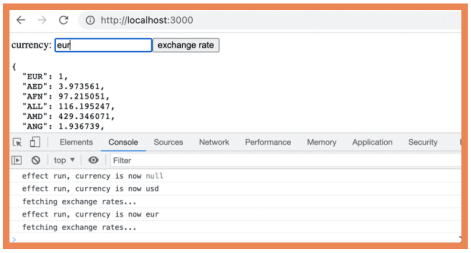
If the second parameter is an empty array `[]`, it's content never changes and the effect is only run after the first render of the component. This is exactly what we want when we are initializing the app state from the server.

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'                            copy
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 = () => {                                                    copy
  // ...
  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.

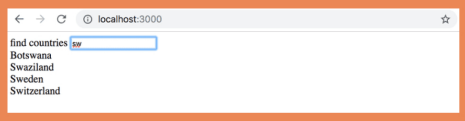
2.18* Data for countries, step1

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.

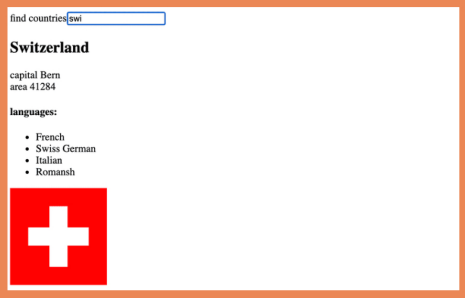
The user interface is very simple. The country to be shown is found by typing a search query into the search field.

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:

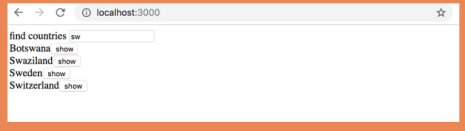


NB: It is enough that your application works for most countries. Some countries, like *Sudan*, can be hard to support since the name of the country is part of the name of another country, *South Sudan*. You don't need to worry about these edge cases.

2.19*: Data for countries, step2

There is still a lot to do in this part, so don't get stuck on this exercise!

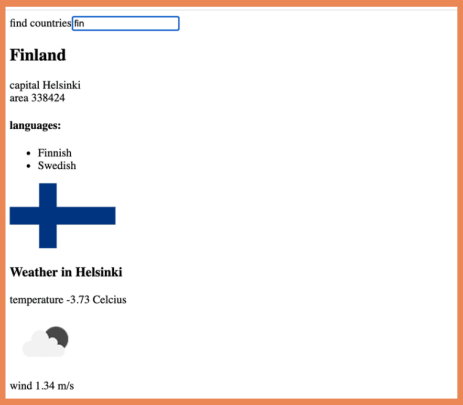
Improve on the application in the previous exercise, such that when the names of multiple countries are shown on the page there is a button next to the name of the country, which when pressed shows the view for that country:



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, step3

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