<https://fullstackopen.com/en/part0/fundamentals_of_web_apps>

truelayer

<https://truelayer.com/?utm_adgroup=Branded_-_EN_-_Exact_&_Phrase&utm_term=truelayer&utm_campaign=Search_-_Brand_-_TrueLayer_-_UK&utm_source=google&utm_medium=cpc&hsa_acc=3115988209&hsa_cam=2061754304&hsa_grp=105263386053&hsa_ad=452823829158&hsa_src=g&hsa_tgt=kwd-796393308454&hsa_kw=truelayer&hsa_mt=e&hsa_net=adwords&hsa_ver=3&gclid=Cj0KCQiA7bucBhCeARIsAIOwr-_kdoNJAe_ki3V-7ZjpLF636ruhlioQdJ3V0vkJDyF832_Y6BstFIkaAlBcEALw_wcB>

Summary of rules

Rule of Learning

* Code by yourself. Look at the entire code to replicate, understdand its logic. Write a question prompt that describes the steps. Then code it yourself, *without looking at the code*. You can look up documentation while coding

Rule of Hooks :

* The useState function (as well as the useEffect function introduced later on in the course) must not be called from inside of a loop, a conditional expression, or any place that is not a function defining a component.
* hooks may only be called from the inside of a function body that defines a React component
* Never modify a state variable. Make a copy of the state variable, modify it, then pass it to the state method

Rule of components

* Never define a component inside a component.
* A component can have child components. But these child components must be defined outside of the parent component
* do not pass different objects as separate props from the App component to the components Content and Total. Instead, pass them directly as an array
* To have multiple expressions inside a component, you need to follow the arow by { }
* To have multiple elements inside the return, you need to enclose the elements inside div tags
* Create a components folder. Put each component inside of modules

Rule of return

* In a an arrow function, if you open curly brackets {…} you write “return”, even if the return consists of one expression
* In the one expression case, if you remove the return you must also remove the curly brackets {}

Rule of event handlers

* Event handlers must be either function definitions or references to functions or functions that return function definitions
* it is preferred not to define the function inside the element and use function reference instead

Rule of web development

* Always have the 3 spaces open together at all times: vscode, the web page and the developer tools
* When an error appears, stop coding, solve the error then procees

Rule of keys

* Keys only make sense in the context of the surrounding array. The error I made is actually explained here
* A good rule of thumb is that elements inside the map() call need keys. Do not refactor keys !!
* Steps to create a new app. In gitbash

npx create-react-app app-part1-1

npx kill-port 3000

cd app-part1-1

npm start

# Part 0

## How to install and run nodeJS projects from WSL

### Creating a javascript project with npm

#### **Installing node, npm and nvm  from the command line**

Here you must follow specifically the wsl instructions

<https://learn.microsoft.com/en-us/windows/dev-environment/javascript/nodejs-on-wsl>

**1. Install CURL**: Remember that CURL is used to transfer data to and from a server, from the colland line. IN particular curl is used to download stuff from github server. We weill install nvm from the github server

sudo apt-get install curl

2. **install nvm** : This is node version manager. Because different projects may use different versions of node. It is an important tool

curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/master/install.sh | bash

3. Check that nvm installation worked by running thais line. You should receive nvm as output\*

nvm

4. Check the version of node you have in your system. At this point you should have none

nvm ls

5. Install both the stable version of node and the current (likely unstable) version of node

nvm install --lts

nvm install node

#### **Changing the version of node to use in a project**

* create a new project directory mkdir NodeTest, and enter the directory cd NodeTest,
* enter nvm use node to switch to the Current version, or nvm use --lts to switch to the LTS version.
* You can also use the specific number for any additional versions you've installed, like nvm use v8.2.1. (To list all of the versions of Node.js available, use the command: nvm ls-remote).

#### **Using VSCODE for node.js specific projects**

You need to install two VSCODE extensions

* The **Remote - WSL extension**: This just makes WSL work on vscode
* The **Remote Development extension**

<https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack>

* Extension pack for snippets, ESLint …

<https://marketplace.visualstudio.com/items?itemName=waderyan.nodejs-extension-pack>

Optionally, these extensions are also helpful

* SettingsSync allows you to save all your vscode settings to github so that you can recover them in different machines
  + <https://marketplace.visualstudio.com/items?itemName=Shan.code-settings-sync>
* Javascript Debugger

<https://marketplace.visualstudio.com/items?itemName=ms-vscode.js-debug>

## How to learn

I structure each section in two steps

* First I expose the new concepts, with references and examples of how to code them
* Then I write prompts
* Then I code *myself -*This is super important for learning

## General info

* Total number of parts : 13
* Total study period : 6 months (as with any UOL course) = 24 weeks
* Parts to study per week: 13/24 = 0.54 🡺 Approx 1 part every two weeks
* Number of days per week: 3
* So: In 6 days (2 week of work) I should cover a whole part, including the exercises.

## General info

* Exercices :
  + **They advise to read all material for the part, and *then* do the exercices**. Do not do the exercice right after you read the corresponding section
  + The exercices build one larger application. Applications can be limited to one part or spread over multiple parts
  + In parts 1-4 you should do at least all the of the exercices that are not marked with an asterisk
  + The exercices are submitted at github. You must add *mluukkai* as a collaborator
  + Exercices are submitted one part at a time. Once you submit exercices for a part you cannot submit more for that part
  + You can make a commit after each exercice but that is not copulsory
* Example applications
  + The code is on github
* Parts
  + The course ifs made of 14 parts
  + One part is ~ 1 week or ~ 15-20 hours. Your speed of earning is flexible
  + You should move to part n+1 only after finishing exercices of part n
* The course uses the mastery learning system
  + <https://en.wikipedia.org/wiki/Mastery_learning>
* Course channels
  + Discord
    - <https://discord.com/channels/757581218085863474/885181069878497330>
  + Telegram
    - <https://t.me/fullstackcourse>
* Exam
  + You can apply for university credits. I will not. There is also an exam which I should consider passing to consolidate
  + Exam here <https://fullstackopen.com/en/part0/general_info#the-course-exam>
  + You can pass the exam once you have completed 5 credits. You complete 5 credits by submitting at least 72 exercices. It is not wise to pass exam at this minimum level of credits.

A picture containing text, black, screen

Description automatically generated

* You do not need to submit exam to get the certifications. <you need credits only
* Use vscode git node.js > 1.13.2

## How to launch a react app

* npx create-react-app nameOfApp
* cd nameOfApp
* rm -rf .git
* npx kill-port 3000
* npm start.

## Launch react app with ubuntu wsl linux

Cd React/Current

sh App.sh

Then open another terminal

Cd React/Current

Sh server.sh

The, simplify index.js to

import React from 'react'

import ReactDOM from 'react-dom/client'

import App from './App'

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

And delete all the files in src except index.js and source.js

## Launching a REACT app from wsl ubuntu

These steps work

<https://learn.microsoft.com/en-us/windows/dev-environment/javascript/nodejs-on-wsl>

## Fundamentals of web apps

* First rule of web development: **Always keep the developer console open** in your web browser
  + Open the developer console using F12 or SHIFT+CTRL+I
  + Go to the networks tab
  + Check disable cash and preserve log
  + In this part we focuse on the network tab but the console tab is the most important

### HTTP get

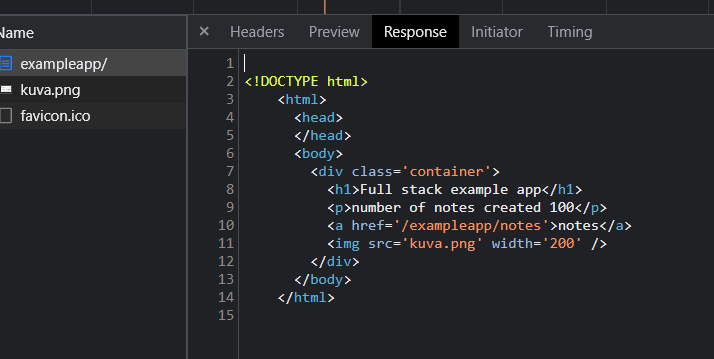
<https://studies.cs.helsinki.fi/exampleapp/>

* The Network tab shows how the browser and the server communicate.
* Hit f5 to refresh the page. This will send requests to the server to refresh the page
* To the left you see there are three requests. Click on the first one
* The general header is self explanatory
* In the response header be mindful of
  + The content-type: html. So the browser knows it can convert it to a page
* This course will use Node.js and Express to create web servers.+

Text

Description automatically generated

* If you hit the response tab, you get the content of the response: This is an HTML code

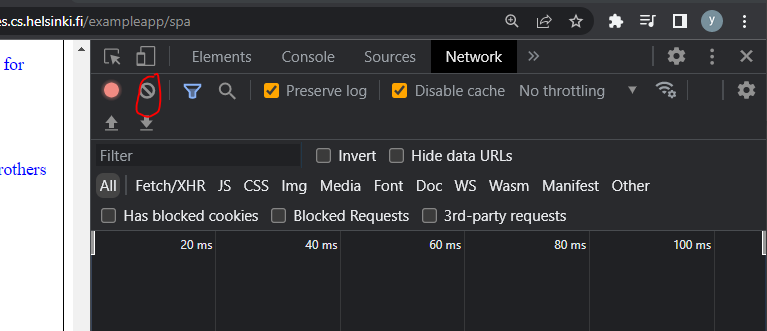


* + Note that the <img> tag prompts the browser to send a second request to get the image. This corresponds to the kuva.png

Graphical user interface, application

Description automatically generated

* The network tab can be emptied via the “sens interdit” sign



### Traditional web applications

* In traditional web pages, All of the logic is on the server, and the browser only renders the HTML as instructed. All the responsibility is on the server
* In modern web pages the responsibility is shared . The HTML shares this responsibility by running the Javascript file it fetches from the server. The Javascript can, for example, fetch some data from a JSON file and adds HTML elements via the DOM API.
* To better view JSON data on the browser, download the chrome extension
  + <https://chrome.google.com/webstore/detail/jsonvue/chklaanhfefbnpoihckbnefhakgolnmc>

### Event handlers and callback functions

* Event handlers are callback functions. The event handler function executes the code inside it when an event occurs

var xhttp = new XMLHttpRequest()

xhttp.onreadystatechange = function () {

  if (this.readyState == 4 && this.status == 200) {

* The xhttp is the object doing the request. This element has an. On this line, an event handler for event onreadystatechange is defined for the xhttp object doing the request. When the state of the object changes, the browser calls the event handler function. The function code checks that the [readyState](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/readyState) equals 4 (which depicts the situation The operation is complete) and that the HTTP status code of the response is 200.

### Manipulating the DOM from the console

* The DOM can be seen on the Elements tab
* Document Object Model, or [DOM](https://en.wikipedia.org/wiki/Document_Object_Model), is an Application Programming Interface (API) which enables programmatic modification of the element trees corresponding to web-pages.
* You can view the topmost note of the DOM structure by typing document in the console
* Q. Add a bullet point down in the list

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

var newElement = document.createElement('li');

newElement.innerText = "This is the contribution of Yassine"

list.appendChild(newElement)

* Q. Is the change that you make permanent ?

No. When we reload the page, the <li> elements will still be fetched from the same JSON file.

### CSS styles

* The styles can be found down the Elements tab
* Note that you can change the styles directly in the elements tab. But these changes will not be permanent unless they are saved on the CSS stylesheet in the server

### Loading a page containing Javascript – Review

A picture containing text

Description automatically generated

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

### Forms and HTTP post

Bach toccata

<https://www.w3schools.com/tags/ref_httpmethods.asp>

* The two main HTTP requests are GET and POST but there are others
  + GET requests data from the server
  + POST is used to send data to a server to create / update a resource
* URL redirection happens when a browser tries to open a given URL but a different URL is opened.
  + Used for various reasons: Good (like allowing multiple domain names that refer to the same owner to refer to a single website) or bad (like phishing)
  + <https://en.wikipedia.org/wiki/URL_redirection>
  + The status code whereby the server responds that the URL has been redirected is 302.
* In the example app we examine what happens when the user fills a form and hits submit
* We examine the network tabs first.

Text

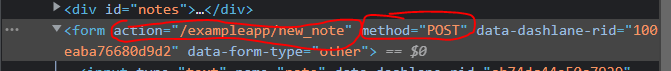
Description automatically generated

* + Submitting the form generates 6 new HTTP requests
  + The first request is sent to new\_note file in the server
  + It is a POST HTTP request because it sends data to the server and asks for an update
  + The server answers with 302 code which is a URL redirect
  + The URL redirect asks the browser to do a GET request to location:/exampleapp/notes
  + This is why we see the notes request to the left. And as we saw earlier notes makes further GET requests for the CSS, JS and JSON
* I type “Hi” in the form. The data I submitted can be found, in the initial HTTP POST request, in the payload tab right to header

Timeline

Description automatically generated

* In the Elements tab, you see that the <form> tag has two important attributes
  + Action: The request is sent to the /exampleapp/newnote folder in the server
  + Method: Submitting the form is done via an HTTP POST request



* Here is the code in the server that redirects the URL

app.post('/new\_note', (req, res) => {

    notes.push({

      content: req.body.note,

      date: new Date(),

    })

    return res.redirect('/notes')

  })

* The data is sent in body of the HTPP POST request. The request is stored in the req object. The data Is stored in red.body. It adds the submission to the arrays notes (see notes.push)
* The notes object has two attributes
  + Content: with the content of the note
  + Date: with the date of the request

### AJAX

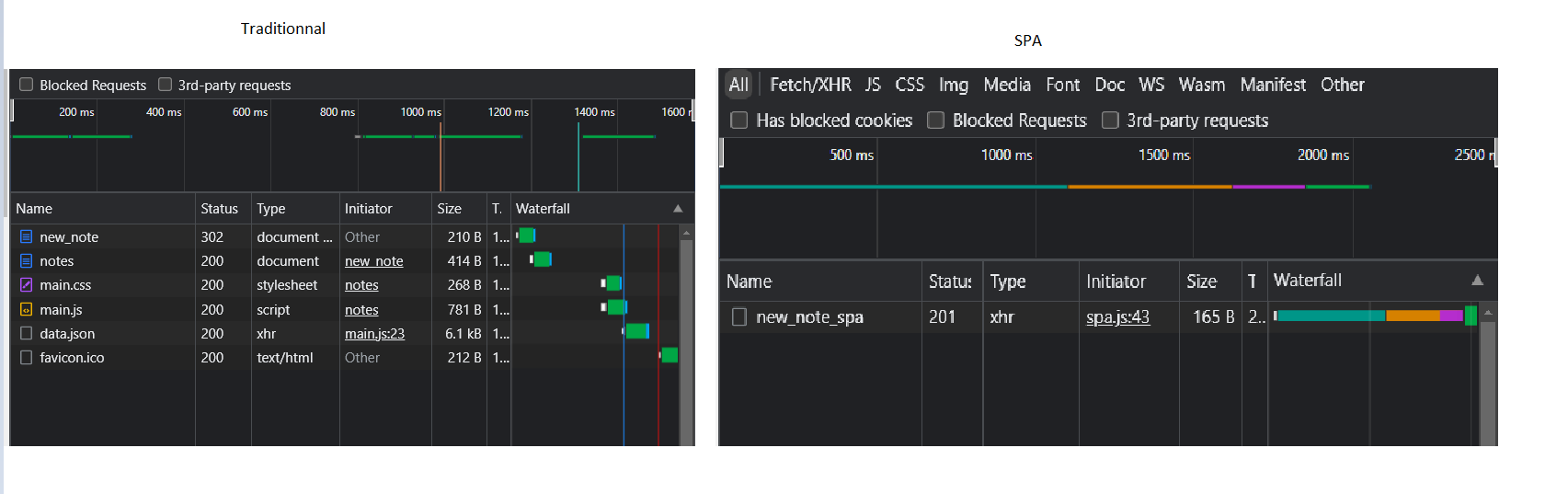
AJAX = Asynchronous Javascript and XML. It was created in 2005 and was the first approach to fetch web page content using Javascript embedded in HTML. It is not used anymore

### Single page app

* <https://studies.cs.helsinki.fi/exampleapp/spa>
* We compare the elements tab of the traditional app vs the single page app looking at the form tag:



* + There is no action or methopd attributes to define where or how the request is sent. The reason is because this will be done on the Javascript code as we will see later
* Now we compare what happens to the network tab when we submit something in the form



* + The SPA sends just one request. While the traditional approach reloads the whole page !!
  + So the question is: How does the SPA achieve this . It achieves this
* The difference is that
  + In the traditional way, the note data is sent to the server who adds it to a note object and redirects the URL to send back the whole web page again. That is, the processing of user data is done on the server
  + In the SPA version the processing of the data is done on the browser in the Javascript file that sits on the browser. An event listener is used that pushes the user input to the notes variables.
  + Only one request is sent, from the javascript line “send to server”.
  + https://www.webfx.com/web-development/glossary/http-status-codes/what-is-a-201-status-code/

Graphical user interface

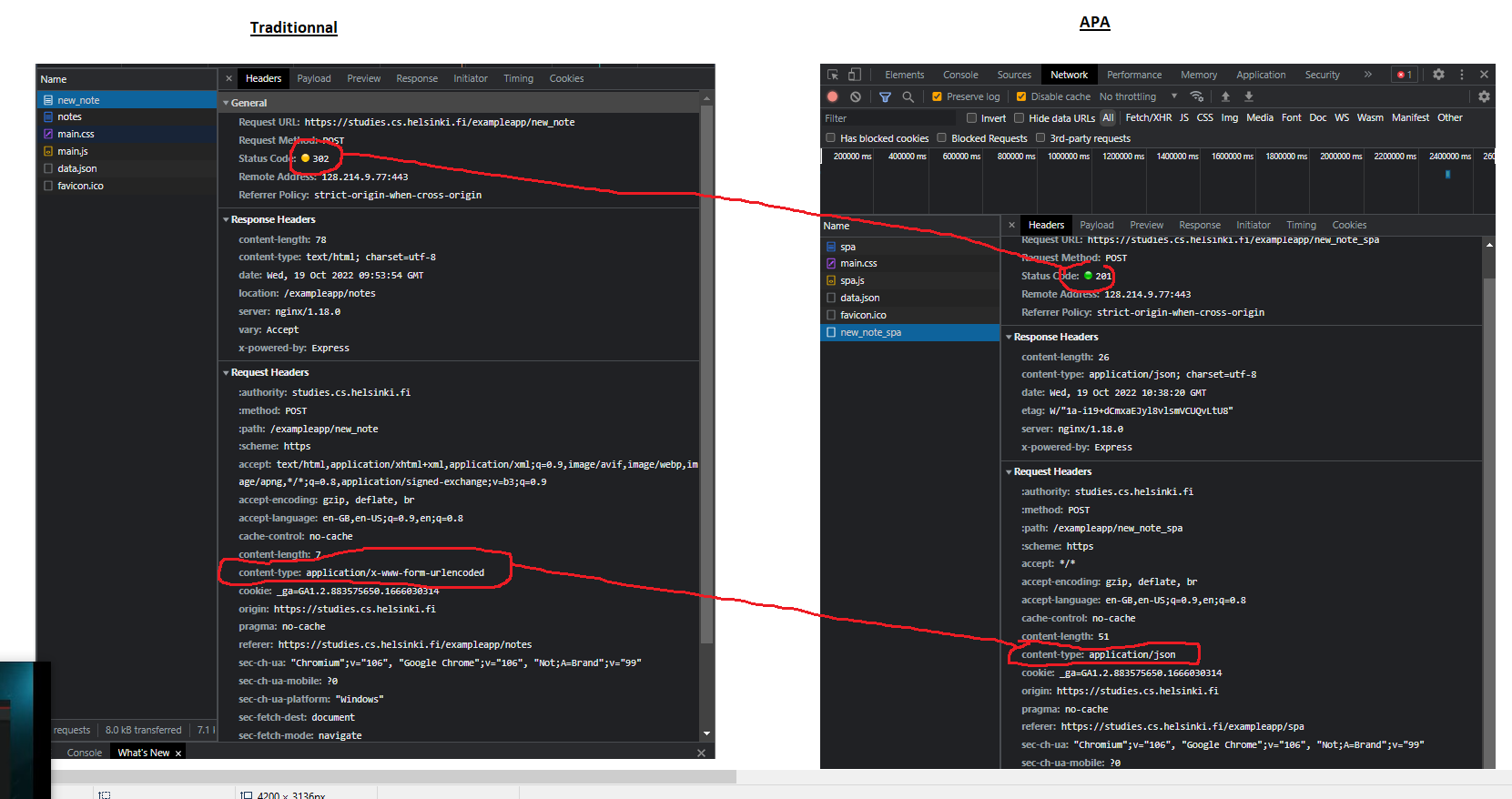
Description automatically generated

* As a result, in the SPA the server responds with the 201 status code which means “The request has been fulfilled and has resulted in one or more new resources being created”. So the server does not reply back with an HTML file. It does not need to: The change has been made by the browser 😊

Graphical user interface, application, Word

Description automatically generated

* The status code:
  + Traditionnal was 302 ie URL redirect towards the URL of the page, which means that the page is reloaded from scratch
  + SPA is 201 meaning: server updates and sends back noting because browser did all the job
* Note also that in Request headers 🡪 content type
  + In APA is JSON.
  + In traditional is something else



### Javascript libraries

* Libraries are used because they are easier to use than the DOM-API / Javascript combination
* Jquery was the first popular library. It was used in the traditional style of generating web pages. It was popular because of cross-browser compatibity. Now with the advances of Javascript, cross compability is less of an issue which reduces the need for JQuery
* The rise of single page apps brought a new wave of libraries
  + BackboneJS in 2012
  + AngularJS 2012 released by Google became the de facto standard. However since 2014 the popularity of angular plummeted because google released angular2 which was not backward compatible with angular 1. And angular 1 was not supported anymore, and angular 2 was not as embrassed by the community
  + The most popular library now is REACT. It is used with redux library

### Full stack web development

* We can view the architecture of a web application as a stack of layers
  + The top layer = the browser. It is the closest to the layer
    - This is the front end
    - The Javascript that runs on the browser is the front end code
  + The middle layer = the server
  + The bottom layer = the database
* FullStack = focuse on all parts of the application: frontend, backend and database
  + Sometimes the software on the server and its OS are seen as part of the stack but we will not go into those
* There are two ways to code in full stack
  + Using the same language (Javascript) across all layers of stack.
    - Back end: Node.js
    - Front end: React library
  + Using different languages for the front end and the back end. This is also possible
* FullStack is a new thing
  + Before: Developers specialized in one layer of the stack, for example the back end because the technologies were different
  + Now: Fullstack is more common. With fullstack, you also sometimes need skills to configure and be admin of the application, for example in the cloud

### Javascript fatigue

* You need to
  + Know your Javascript
  + Know the HTTP protocol
  + Know to handle databases
  + Handle server administration and communication
  + Know CSS to make application presentable
* Javascript fatigue = getting tired with the constant change in the Javascript
  + Coding Javascript is cool
  + Configuration is a nightmare

### Exercices

* There are 2 steps to submit exercices
  + Via github
    - Add mluukkai as collaborator
  + Marking them as done in the submission system
* Exercices are submitted one part at a time
  + When you submit exercices for a part you can no longer submit ay mixed exercices for that part
* **WARNING** create-react-app will automatically turn your project into a git-repository unless you create your application inside of an existing git repository. **Most likely you do not want each of your projects to be a separate repository**, so simply run the rm -rf .git command at the root of your application.
* You are allowed to
  + Commit one exercice at a time
  + Commit one app at a time (one app = multiple exercices)
  + Push at the end of the part
  + Submit (on Helsinki) only one submission per part at the end of the part. You cannot submit a part more than once !

#### Ex 1 and 2

* MDN HTML basics
  + <https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics>
* MDN CSS basics
  + <https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics>

#### Ex 3 MDN forms

* MDN forms
  + <https://developer.mozilla.org/en-US/docs/Learn/Forms/Your_first_form>
* Data entered via a form either
  + Goes to server
  + Used by browser to update page on it directly
* A web form is made of two things
  + Form controls, also called widgets. They define the form type ie: whether it is a checkbox, a button, a text field. They are mostly created using the <input> element
  + Some additional elements that help structure the form. For example, a <label>
* We design this form here . It is important to draw the mockup before coding it

Diagram, schematic

Description automatically generated

##### The <form> element

* <form> defines container for forms
* It has two attributes
  + Action: defines the URL where the data will be sent
  + Method: defines the HTTP method ie get or post
* <input> has a type attributes
  + Type= text

#### Ex 4

Draw the diagram of what happens when you hit type something and hit the submit button in the traditional app.

Use this to draw the diagram

<https://www.websequencediagrams.com/>

My answer

Table

Description automatically generated

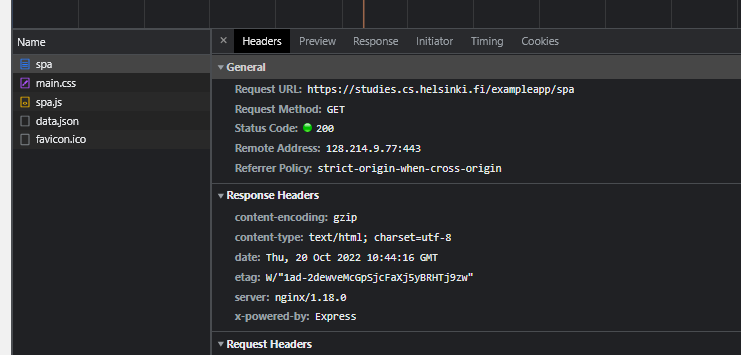
What I did is that I took 2 which is the diagram for loading the page. And I added 1, which is the HTTP POST and the URL redirect response

#### Ex 5

Diagram

Description automatically generated with low confidence

To draw it just follow the Netowrk tab



* The requests are listed on the left hand side: spa, main.css … Under headers🡪 general you see the URL towards which the request is sent
* The content of the response can all be see in the Response tab . For example, the whole Javascript code of spa.js can be seen here

Text

Description automatically generated

#### Ex 6

Now draw the diagram of what happens in the single page app when submitting through the button



# Part 1 Introduction to REACT

<https://fullstackopen.com/en/part1/introduction_to_react>

## Introduction to REACT

### Component

<https://github.com/facebook/create-react-app#creating-an-app>

* Download nodeJS
* Ad the path to NodeJS to environment PATH variable
* Then in any folder you want open git bash of the command line and type

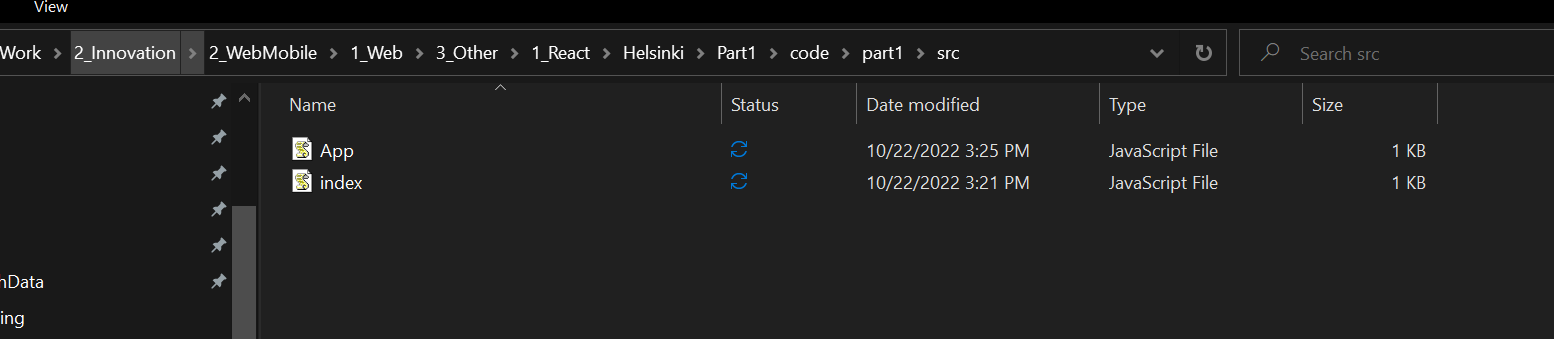
npx create-react-app my-app

* To start the app:
  + Go to the folder with the app
    - C:\Users\yassi\OneDrive\1\_Work\2\_Innovation\2\_WebMobile\1\_Web\3\_Other\1\_React\Helsinki\Part1\code\part1
  + In the git bah or the command line
    - Npm start
* To modify the app
  + Open side by side chrome with the console open, and the source files in vscode
  + In app.js change hello world by Ho. It will change immediate immediately

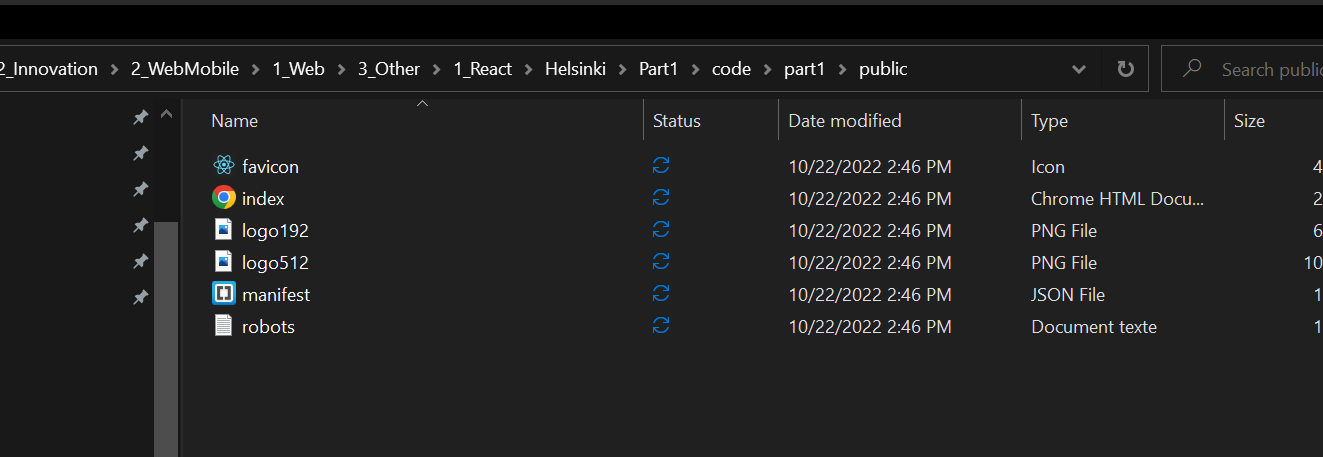
Graphical user interface, text

Description automatically generated

* The code of the app is in src. We simplify the code to just two files



The HTML code is here



Index.js does this

import React from 'react'

import ReactDOM from 'react-dom/client'

import App from './App'

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

What this does is that it imports the App component and renders it in the browser

Components are explained in the link below . Components split the UI into independent, reusable pieces. In practice components are just Javascript functions. They take any input, and they produce a React element (which is an HTML element)

<https://reactjs.org/docs/components-and-props.html>

App.js does this

function App() {

  return (

  <div>

    <p> Hello World</p>

  </div>

  )

}

export default App

The function takes no input, an produces a paragraph Hello Word.

The fancy ES6 way of writing this code is … This is called arrow functions

const App = () => (

  <div>

    <p>Ho</p>

  </div>

)

export default App

More on arrow functions

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions>

Note that this defines the function

() => (

<div>

<p>Hello world</p>

</div>

)

And the function is associated to a constant variable

const App = () => (

  <div>

    <p>Ho</p>

  </div>

)

export default App

You can also render dynamic content

You can play and write this function in the component

function App() {

  const now = new Date()

  const a = 10

  const b = 20

  return (

  <div>

    <p> Hello World it is {now.toString()}</p>

    <p>

      {a} plus {b} is {a+b}

    </p>

  </div>

  )

}

export default App

### JSX

<https://reactjs.org/docs/introducing-jsx.html>

The traditional approach is to separate the technologies by making markup (HTML) and logic (in Javascript) in separate files.

React follows a different philosophy. It separates concerns in different units called components. These components contains both the markup and the logic

This allows you to write lazy code such as :

const element = <h1>Hello, world!</h1>;

React does not require using JSX but most people do because

* It allows to easily work with UI inside Javascript
* It displays errors more powerfully

Inside curly braces you can execute any javascript code. For example a call to a function. Like here

function formatName(user) {

  return user.firstName + ' ' + user.lastName;

}

const user = {

  firstName: 'Harper',

  lastName: 'Perez'

};

const element = (

  <h1>

    Hello, {formatName(user)}!

  </h1>

);

So JSX allows to

* Embed javascript code in HTML elements using templating
* Manipulate HTML DOM elements easily from Javascript

JSX expressions, after complilation, evaluate to Javascript objects. This means that you can assign JSX expressions to variables, accept them as arguments to functions, return them from functions, use them inside if statements and for loop

For example

function getGreeting(user) {

  if (user) {

    return <h1>Hello, {formatName(user)}!</h1>;

  }

  return <h1>Hello, Stranger.</h1>;

}

JSX is closer to Javascript than HTML and so uses camelCase

You can specify quotes either using attributes or quotes but not both at the same time

const element = <a href="https://www.reactjs.org"> link </a>;

const element = <img src={user.avatarUrl}></img>;

JSX tags may contain children

const element = (

  <div>

    <h1>Hello!</h1>

    <h2>Good to see you here.</h2>

  </div>

);

When you write this code

const element = (

  <h1 className="greeting">

    Hello, world!

  </h1>

);

What we do is actually call the createElement of the React object. So it is equivalent to write

const element = React.createElement(

  'h1',

  {className: 'greeting'},

  'Hello, world!'

);

And what this does is that it creates objects that we call React Elements. React reads these objects and uses them to create DOM

const element = {

  type: 'h1',

  props: {

    className: 'greeting',

    children: 'Hello, world!'

  }

};

### Multiple Components

There are two conventions or philosophies in REACT

* Applications are composed from many specialized reusable components. Combining components makes applications fairly maintainable
* Application is like a tree. There is a root component called App at the top of the component tree of the application

So here the component Hello is created then embedded in the root component app

const Hello = () => {

  return (

    <div>

      <p> Hello Worldy </p>

    </div>

  )

}

const App = () => {

  return (

    <div>

      <h1>Hello World</h1>

      <Hello />

    </div>

  )

}

export default App

Logo

Description automatically generated with medium confidence

We can reuse a component multiple times as in here

const App = () => {

  return (

    <div>

      <h1>Hello World</h1>

      <Hello />

      <Hello />

      <Hello />

    </div>

  )

}

### Props : Passing data to a component

Documentation

<https://reactjs.org/docs/components-and-props.html>

Component

React elements can represent two things: either

* DOM tags as in
  + const element = <div />;
* user-defined components as in
  + const element = <Welcome name="Sara" />;

We saw that an element is interpreted as a simple Javascript object like this

  type: 'h1',

  props: {

    className: 'greeting',

    children: 'Hello, world!'

  }

};

That element object itself contains an object called prop. That prop object contains two types of things

* attribute values of the element
* children elements

For example

const Hello = (props) => {

  return (

    <div>

      <p> Hello {props.name} </p>

    </div>

  )

}

const App = () => {

  return (

    <div>

      <h1>Greetings</h1>

      <Hello name = 'George' />

      <Hello name = 'Daisy' />

    </div>

  )

}

export default App

Text

Description automatically generated

So the element

<Hello name = 'George' />

Is interpreted as

const element = {

  props: {

    name: ‘George’;

  }

};

A given React element can have any number of props. The value of the prop can be either

* hardcoded using a string
* obtained from a Javascript expression written inside curly braces

So this

const Hello = (props) => {

  return (

    <div>

      <p> Hello {props.name}, you are {props.age} years old </p>

    </div>

  )

}

const App = () => {

  const name = 'Peter'

  const age = 10

  return (

    <div>

      <h1>Greetings</h1>

      <Hello name = 'Maya' age = {26+10} />

      <Hello name = {name} age = {age} />

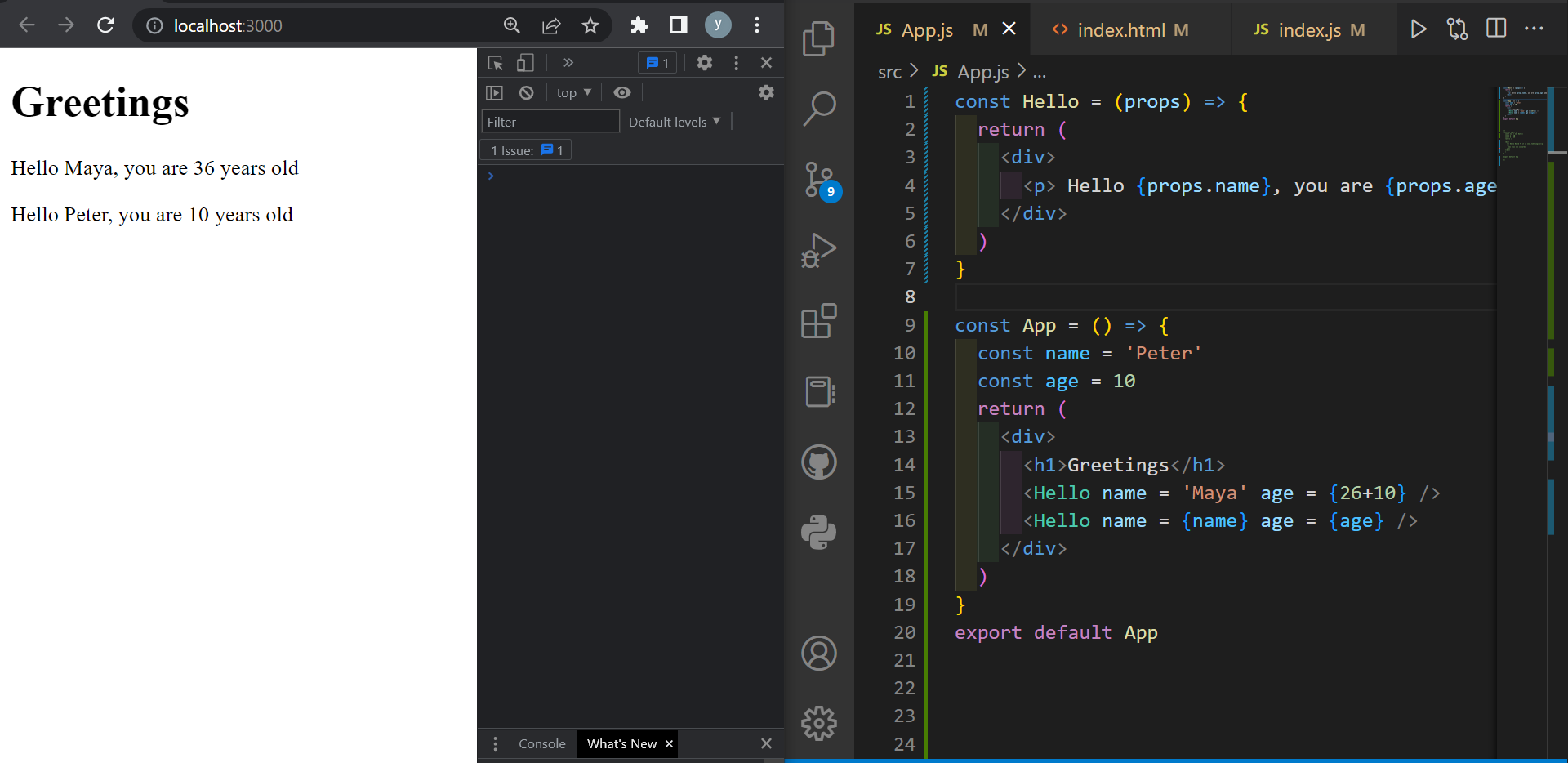
    </div>

  )

}

export default App

produces this



### Some notes

React is configured to generate clear error messages

Code with the dev tools and console open so that you catch errors immediately

React component names must have their first letter capitalized always

* <footer> will be interpreted as a footer HTMl element
* <Footer> will be interpreted as a user defined component

Also, a react component must always be wrapped in an enclosing tag. Except when the return statement consists of one expression. If it consists of multiple expressions, then the workaround consists of enclosing the different expressions in a div tag

For example, the App component includes other components that are enclosed in a div tag

const App = () => {

  const name = 'Peter'

  const age = 10

  return (

    <div>

      <h1>Greetings</h1>

      <Hello name = 'Maya' age = {26+10} />

      <Hello name = {name} age = {age} />

    </div>

  )

}

If we remove the div tag

const Hello = (props) => {

  return (

    <div>

      <p> Hello {props.name}, you are {props.age} years old </p>

    </div>

  )

}

const App = () => {

  const name = 'Peter'

  const age = 10

  return (

    <h1>Greetings</h1>

    <Hello name = 'Maya' age = {26+10} />

    <Hello name = {name} age = {age} />

  )

}

export default App

Vscode extension (babel javascript by Michael McDermott) tells us there is an error

A screenshot of a computer

Description automatically generated with medium confidence

The error is also signalled by REACT

Text

Description automatically generated with medium confidence

This means that there are extra div elements in the DOM tree.We can avoid it using fragments which are empty HTML tags

<https://reactjs.org/docs/fragments.html#short-syntax>

that is, you can enclose REACT components inside empty tags

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

## Javascript

#### Transpiling ES6 using Babel

* The Javascript standard is ECMAScript
  + <https://en.wikipedia.org/wiki/ECMAScript>
* The latest version is ES6
  + <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>
* Transpiling = converting a code from a programming language to either
  + Another version of the programming language
  + Another programming language
* Because browsers do not support all of ES6 new features, the code is usually transpiled to an older version.
  + Text

    Description automatically generated
  + <https://babeljs.io/>
* Apps created in REACT using create-react-app are automatically transpiled

[const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const) does not actually define a variable but a constant for which the value can no longer be changed. On the other hand, [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let) defines a normal variable.

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

### B.1 Arrays

#### forEach

In ES6 we use forEach for for loops. Note that when a function has one argument we can omit the parenthesis

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

console.log(t);

t.forEach(value => {

    console.log(value);

})

Note that foreach receives a function as a parameter. That function is defined using the arrow syntax

#### Prefer immutables: concat over push

In React it is preferable to use techniques from functional programming. One of them is to use immutable data structures. So we prefer to use concat instead of push. Concat creates a new list

const u = [1,-1,3];

const t2 = u.concat(5);

console.log("t2 is " + t2);

#### map

map is also functional. Based on an array it creates a new array, apply some function to each element

const u = [1,-1,3];

// Map is used frequently in React

const m1 = u.map(value => value\*2);

console.log(m1);

>> m1 is 2,-2,6

const m2 = u.map(value => "<li> " + value + "</li> " )

console.log(m2);

>> m2 is <li> 1</li> ,<li> -1</li> ,<li> 3</li>

//  Destructuring

const w = [1,2,3,4,5];

const [first, second, ...rest] = w;

console.log("first: " + first, "second: " + second);

console.log("rest: " + rest);

>> first: 1 second: 2

>> rest: 3,4,5

### B.2 Objects

Nothing new

### B.3 Functions

Arrow functions

// Arrow function

const sum = (p1,p2) => {

    console.log(p1);

    console.log(p2);

    return p1+p2;

}

console.log("The sum of 1 and 5 is " + sum(1,5));

You can shorten the way you write functions in the following ways:

// Arrow function with 1 parameter: no parenthesis needed

const square = p => {

    console.log(p);

    return (p\*p);

}

console.log("the square of 8 is " + square(8))

// Arrow function with 1 parameter and 1 expression: no {} needed

const cube = p => p\*p\*p;

console.log("the cube of 8 is " + cube(8));

This is useful when using the map function

// this short way is handy when using map

const z = [5,9,8,5];

const result = z.map(p => p\*p);

console.log("map result is: "+ result);

### B.4 Object Method and this

#### React uses a this-less Javascript

In this course we do not need objects because we use REACT hooks

We do not use objects because the this object poses all sorts of problems. Mainly, the this keyword defined inside an object, looses its meaning when the method is called, as this becomes the global this variable.

There is an entire course on the this keyword in Javascript and it is highly recommended

<https://egghead.io/courses/understand-javascript-s-this-keyword-in-depth>

Importantly you should not use the this keyword with arrow functions.

#### This problem 1: this becomes global when calling a function by reference

We illustrate here two problems of the this keyword

// Objects and the this keyword

const arto = {

    name: "Atro hellas",

    // A method that does not use the this keyword

    doAddition: function(a,b) {

        console.log(a+b);

    },

    // A method that uses the this keyword which creates problems

    greet: function() {

        console.log("Hello my name is: " + this.name);

    }

}

// The this keyword disappears depending on how we call the method

// If we call using a reference we get a problem

// In doAddition() where this is not used there is no problem

arto.doAddition(6,9); // direct call

const noThisResult = arto.doAddition; //call through reference

noThisResult(6,10);

// In greet() which used the this keyword there is a problem whe calling by reference

arto.greet(); // direct call no problem - this refers to object

const thisResult = arto.greet; // call by reference produces undefined - this refers to global variable

thisResult(); // Here problem

#### this problem 2: Setting a timeout to call a function

timeout(function, X) calls the function after X ms of calling the page

Again, when using the timeout, the this keyword looses its meaning because it has now the value given by the Javascript engine (global this)

There are many methods to preserve this. One of them in bind

// timeout

setTimeout(arto.greet, 1000); // undefined

setTimeout(arto.greet.bind(arto), 1000); // preserves this. Creates a function where this is bound to arto independent of how the method is called

ES6 classes are used in old REACT. In the course we use the new hook feature which is a class-less and this-less Javascript

The only types that javascript defines are

* [Boolean Null, Undefined, Number, String, Symbol, BigInt, and Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures).

For example arto is an object

console.log(typeof arto);

>> object

Even a class is an object. See graphics programming course for objects

### B.5 Nice Javascript resources

There exist both good and poor guides for JavaScript on the Internet. Most of the links on this page relating to JavaScript features reference [Mozilla's JavaScript Guide](https://developer.mozilla.org/en-US/docs/Web/JavaScript).

It is highly recommended to immediately read [A re-introduction to JavaScript (JS tutorial)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript) on Mozilla's website.

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

Another great resource for learning JavaScript is [javascript.info](https://javascript.info/).

The free and highly engaging book Eloquent JavaScript [https://eloquentjavascript.net](https://eloquentjavascript.net/). Takes you from the basics to interesting stuff quickly, a mixture of theory projects and exercises, covers general programming theory as well as the JavaScript language.

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

## C.Component, state, event handler

We start from the app like this

const Hello = (props) => {

  return (

    <div>

      <p>

          Hello {props.name}, you are {props.age} years old

      </p>

  </div>

  )

}

const App = () => {

  const name = 'Peter'

  const age = 10

  return (

    <div>

      <h1>Greetings</h1>

      <Hello name="Maya" age="{26+10}" />

      <Hello name={name} age={age} />

    </div>

  )

}

export default App

### C.1. Component helper functions

A component helper function is basically a function defined inside a component. Technically it is a function defined inside a function.

The key point is this:

// The main point is that the function bornYear() defined

  // inside component Hello can access all the props defined inside

  //  the component Hello. No need for this, or to pass the age as a parametr to the function

So inside its body, the helper function has access to all the props of the component. See the code here that computes the year of birth

const Hello = (props) => {

  // We define a function inside the component

  const bornYear = () => {

    const yearNow = new Date().getFullYear()

    return yearNow - props.age

  }

  // The main point is that the function bornYear() defined

  // inside component Hello can access all the props defined inside

  //  the component Hello. No need for this, or to pass the age as a parametr to the function

  return (

    <div>

      <p>

          Hello {props.name}, you are {props.age} years old

      </p>

      <p>The year you are born is approx {bornYear()}</p>

  </div>

  )

}

const App = () => {

  const name = 'Peter'

  const age = 10

  return (

    <div>

      <h1>Greetings</h1>

      <Hello name="Maya" age={26+10} />

      <Hello name={name} age={age} />

    </div>

  )

}

export default App

### C.2. Destructuring

In the code above props is (for the first Hello component)

props = {

  name: Maya',

  age: 36,

}

We refactor the code and make it much smaller using destructuring in two steps.

#### Step1: Store each props attribute into a variable

First we create variables that store props.name and props.age

const Hello = (props) => {

  const name = props.name;

  const age = props.age;

  // We define a function inside the component

  const bornYear = () => {

    const yearNow = new Date().getFullYear()

    return yearNow - age

  }

  // The main point is that the function bornYear() defined

  // inside component Hello can access all the props defined inside

  //  the component Hello. No need for this, or to pass the age as a parametr to the function

  return (

    <div>

      <p>

          Hello {name}, you are {age} years old

      </p>

      <p>The year you are born is approx {bornYear()}</p>

  </div>

  )

}

#### Step 2: Destructure the props object

The first step of destructuring is to destructure the props object so we replace this

  const name = props.name;

  const age = props.age;

by this

const {name, age} = props;

#### Step 3: Directly desructure the props that is passed as argument to the component

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

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

const Hello = (props) => {

const { name, age } = props

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

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

So we have this now

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

  // We define a function inside the component

  const bornYear = () => {

    const yearNow = new Date().getFullYear()

    return yearNow - age

  }

  // The main point is that the function bornYear() defined

  // inside component Hello can access all the props defined inside

  //  the component Hello. No need for this, or to pass the age as a parametr to the function

  return (

    <div>

      <p>

          Hello {name}, you are {age} years old

      </p>

      <p>The year you are born is approx {bornYear()}</p>

  </div>

  )

}

#### Step 4: turn the helper function into a one line expression

This is not per se part of the destructuring but it is a byproduct of it

Remember when you write arrow function in one line, you do not need the curly braces

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

All in all, we turned the compent from this

const Hello = (props) => {

  const bornYear = () => {

    const yearNow = new Date().getFullYear()

    return yearNow - props.age

  }

  return (

    <div>

      <p>

          Hello {props.name}, you are {props.age} years old

      </p>

      <p>The year you are born is approx {bornYear()}</p>

  </div>

  )

}

To this

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

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

  return (

    <div>

      <p>

          Hello {name}, you are {age} years old

      </p>

      <p>The year you are born is approx {bornYear()}</p>

  </div>

  )

}

The whole code looks like this

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

  // We define a function inside the component

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

  // The main point is that the function bornYear() defined

  // inside component Hello can access all the props defined inside

  //  the component Hello. No need for this, or to pass the age as a parametr to the function

  return (

    <div>

      <p>

          Hello {name}, you are {age} years old

      </p>

      <p>The year you are born is approx {bornYear()}</p>

  </div>

  )

}

const App = () => {

  const name = 'Peter'

  const age = 10

  return (

    <div>

      <h1>Greetings</h1>

      <Hello name="Maya" age={26+10} />

      <Hello name={name} age={age} />

    </div>

  )

}

export default App

### C3. Page re-rendering

Here the challenge is to do in React what the draw() function does. Ie we create a variable count that increments by 1. And we want to see that variable update on the screen

We start with this code; In app.js

const App = (props) => {

  const {counter} = props

  return (

    <div>The value of the counter is {counter}</div>

  )

}

export default App

and in index.js

import React from 'react';

import ReactDOM from 'react-dom/client';

import App from './App';

let counter = 1

counter += 1

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

The problem is that the counter stays at 2 and does not increment

Graphical user interface

Description automatically generated

#### Step 1: Wrap the render method inside a refresh() method

We could name the method refresh() or anything else

import React from 'react';

import ReactDOM from 'react-dom/client';

import App from './App';

let counter = 1

counter += 1

const refresh = ()  => {

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

}

refresh()

counter +=1

refresh()

counter +=1

refresh()

counter +=1

The better solution is to re-render and incremement the count every second using setInterval

So this works beautifully

import React from 'react';

import ReactDOM from 'react-dom/client';

import App from './App';

let counter = 1

counter += 1

const refresh = ()  => {

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

}

setInterval(() => {

    counter +=1

    refresh()

})

This is not the recommended way to do this. There is a better way we will see later

### C4. Stateful component

State hooks

<https://reactjs.org/docs/hooks-state.html>

One way of using hooks is to define states. In traditional terms, to define states, we with use a class with a method that updates the state. Hooks allow us to update the state of a component inside the component, without turning the component into a class, and therefore without using the this keyword.

A hook is a special function that hooks components to react features, such as states

Here we update the content on the page, using a state hook.

import { useState } from 'react'

const App = () => {

  // UseState() returns an array of 2 elements

  // We destructure the array returned by useState(0) into two variables

  // counter is an interger that starts at 0

  // setCounter is a function that changes the state

  const [ counter, setCounter ] = useState(0)

  // setTimeout receives two parameters. A function and a time in ms

  // the function is called  1s after setTimeOut is called, ONCE

  // Attention setTimeOut does this only once. It does not repeat it every 1s

  // However when setCounter is called, React re-renders the component in which the state is included, ie the APP component

  // Therefore the component app is re-rendered, but this time with a new value of the state: count+=1

  setTimeout(

    () => setCounter(counter + 1),

    1000

  )

  // To keep track of the count, ie of the calls to the app's render function

  console.log('counter value is ' + counter)

  return (

    <div>{counter}</div>

  )

}

export default App

### C5. Event Handling

React documentation

https://reactjs.org/docs/handling-events.html

Let's change the application so that increasing the counter happens when a user clicks a button, which is implemented with the [button](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button) element.

Let’s do TDD cycles swithout testing but just writing small iterative steps

First we return a button that print “Hi” in the console whenever it is clicked

import { useState } from 'react'

const App = () => {

  const [ counter, setCounter ] = useState(0)

  setTimeout(

    () => setCounter(counter + 1),

    1000

  )

  return (

    <button onClick= {() => console.log("Hi")}> "Click me" </button>

  )

}

export default App

Note that the way we wrtite the unclick event is different from HTML

<button onclick="myFunction()">Click me</button>

import { useState } from 'react'

const App = () => {

  const [ counter, setCounter ] = useState(0)

  // setTimeout(

  //   () => setCounter(counter + 1),

  //   1000

  // )

  return (

    <div>

    <div> {counter} </div>

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

    </div>

  )

}

export default App

We also add a button that resets the counter

import { useState } from 'react'

const App = () => {

  const [ counter, setCounter ] = useState(0)

  // setTimeout(

  //   () => setCounter(counter + 1),

  //   1000

  // )

  return (

    <div>

    <div> {counter} </div>

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

    <button onClick={() => setCounter(0)}> Reset </button>

    </div>

  )

}

export default App

### C6. Event handler is a function

**Point 1**: Eveny handlers receive either functions or function references, not calls to functions

An event handler is supposed to be either a *function* or a *function reference*, and when we write:

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

the event handler is actually a *function call*.

So it leads to an error

**Point 2:**  Best practice: the value of the onClick attribute should be a reference to a function

import { useState } from 'react'

const App = () => {

  const [ counter, setCounter ] = useState(0)

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

  const reSetCounter = () => setCounter(0)

  return (

    <div>

    <div> {counter} </div>

    <button onClick={increaseByOne}> myButton </button>

    <button onClick={reSetCounter}> Reset </button>

    </div>

  )

}

export default App

Now this has nothing to od with the points but we do it

We turn the first div into a component Display as follow

import { useState } from 'react'

const Display = (props) => {

  return (

    <div> {props.counter}</div>

  )

}

const App = () => {

  const [ counter, setCounter ] = useState(0)

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

  const setToZero = () => setCounter(0)

  return (

    <div>

    <Display counter = {counter}/>

    <div> {counter} </div>

    <button onClick={increaseByOne}> myButton </button>

    <button onClick={setToZero}> Reset </button>

    </div>

  )

}

export default App

### C7. Passing state to child components

It's recommended to write React components that are small and reusable across the application and even across projects.

Let's refactor our application so that it's composed of three smaller components, one component for displaying the counter and two components for buttons.

Documentation lifting state up

<https://reactjs.org/docs/lifting-state-up.html>

React’s own tutorial

<https://reactjs.org/tutorial/tutorial.html#completing-the-game>

So what we return now are divs

  return (

    <div>

    <div> {counter} </div>

    <button onClick={increaseByOne}> myButton </button>

    <button onClick={setToZero}> Reset </button>

    </div>

  )

We are not happy about that. We want to turn each of these 3 divs into a React component so that they can be reused later

**The key is this: All the attributes of div elements will now be elements of the props object associated to that REACT component**

* **Turning the first div into a component**

Let’s start with the first div and turn it into a display component

<div> {counter} </div>

import { useState } from 'react'

const Display = (props) => {

  return (

    <div> props.counter </div>

  )

}

const App = () => {

  const [ counter, setCounter ] = useState(0)

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

  const setToZero = () => setCounter(0)

  return (

    <div>

    <Display counter = {counter}/>

    <button onClick={increaseByOne}> myButton </button>

    <button onClick={setToZero}> Reset </button>

    </div>

  )

}

export default App

* **Turning the second div into a component**

At this point I was confused by the curly braces. Curly braces are always used for one thing: Writing any javascriot code inside JSX. It can be used at all levels ie

* Inside the definition of a component
* When returning a component and specifying the value of a given property of that component

Curcly braces can also be used when destructuring an object, because an obect is defined using curcly braces (a list would be destructured using []) so

  const age = props.age;

by this

const {name, age} = props;

So we create a component Button. Big subtelty here: onclick is simply a prop field of the Button class. The value of the onclick property is an onClick event hander function.

We pass the function inside brackets onclick = {increaseByOne}

Because the function is javascript code

import { useState } from 'react'

const Display = (props) => {

  return (

    <div> {props.counter} </div>

  )

}

const Button = (props) => {

  // props is an object consisting of 2 elements: an onclick handler and a text

  return(

    <button onClick = {props.onclick}> {props.text} </button>

  )

}

const App = () => {

  const [ counter, setCounter ] = useState(0)

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

  const setToZero = () => setCounter(0)

  return (

    <div>

    <Display counter = {counter}/>

    <Button onclick = {increaseByOne} text = "Increase"/>

    <button onClick={setToZero}> Reset </button>

    </div>

  )

}

export default App

* **Turning the last button element into a react component**

This is where you reap the dividend of your work: All you have to do now is reuse the Button class

<Button onclick = {setToZero} text = "Reset"/>

* Adding another button to decrease the counter

Now we use again the Button component. Wejust have to define a new function

import { useState } from 'react'

const Display = (props) => {

  return (

    <div> {props.counter} </div>

  )

}

const Button = (props) => {

  // props is an object consisting of 2 elements: an onclick handler and a text

  return(

    <button onClick = {props.onclick}> {props.text} </button>

  )

}

const App = () => {

  const [ counter, setCounter ] = useState(0)

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

  const decreaseByOne = () => setCounter(counter-1)

  const setToZero = () => setCounter(0)

  return (

    <div>

    <Display counter = {counter}/>

    <Button onclick = {increaseByOne} text = "Increase"/>

    <Button onclick = {setToZero} text = "Reset"/>

    <Button onclick = {decreaseByOne} text = "Decrease"/>

    </div>

  )

}

export default App

To do 14 sections + 14 exercices

* [Event handler is a function](https://fullstackopen.com/en/part1/component_state_event_handlers#event-handler-is-a-function)
* [Passing state to child components](https://fullstackopen.com/en/part1/component_state_event_handlers#passing-state-to-child-components)
* [Changes in state cause rerendering](https://fullstackopen.com/en/part1/component_state_event_handlers#changes-in-state-cause-rerendering)
* [Refactoring the components](https://fullstackopen.com/en/part1/component_state_event_handlers#refactoring-the-components)
* [A note on React version](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#a-note-on-react-version)
* [Complex state](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#complex-state)
* [Handling arrays](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#handling-arrays)
* [Conditional rendering](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#conditional-rendering)
* [Old React](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#old-react)
* [Debugging React applications](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#debugging-react-applications)
* [Rules of Hooks](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#rules-of-hooks)
* [Event Handling Revisited](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#event-handling-revisited)
* [Function that returns a function](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#function-that-returns-a-function)
* [Passing Event Handlers to Child Components](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#passing-event-handlers-to-child-components)
* [Do Not Define Components Within Components](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#do-not-define-components-within-components)
* [Useful Reading](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#useful-reading)
* [Exercises 1.6.-1.14](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#exercises-1-6-1-14)

### C7. Changes in state cause rerendering

Documentation useState hook

<https://reactjs.org/docs/hooks-reference.html#usestate>

Note that in the app above, when you click a button, the setCounter() function is called which causes the whole app to re-render. So the display component is rendered again which makes the number appear.

Do not miss out on the fact that upon clicking on the button the page is rerendered

Key points

* The button is used to update the common state of the page
* Calling setCounter() rerenders the whole app

### C8. Refactoring the components

const Display = (props) => {

  return (

    <div> {props.counter} </div>

  )

}

Rappel:

You have an object   
props = {  
    Name: myName  
    surName: mySurname  
}  
Destructure it

In this section we restructure the display and button components that we created above using the principles we already learned of

* Destructuring the props objects that is passed as argument to the componeht function
* Using one line arrow function when the return statement is one line

Seo we turn this

const Display = (props) => {

  return (

    <div> {props.counter} </div>

  )

}

Into this

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

That is:

* We destructure the props object into its sole field counter and pass the “destructure” as argument to the component
* We remove the return and curly bracesbecause we are rerning only one expression
* const Button = ({onclick, text}) => <button onClick = {onclick}> {text} </button>

So this is the whole code

import { useState } from 'react'

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

const Button = ({onclick, text}) => <button onClick = {onclick}> {text} </button>

const App = () => {

  const [ counter, setCounter ] = useState(0)

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

  const decreaseByOne = () => setCounter(counter-1)

  const setToZero = () => setCounter(0)

  return (

    <div>

    <Display counter = {counter}/>

    <Button onclick = {increaseByOne} text = "Increase"/>

    <Button onclick = {setToZero} text = "Reset"/>

    <Button onclick = {decreaseByOne} text = "Decrease"/>

    </div>

  )

}

export default App

## D.A more complex state, debugging React apps

### D.1. A note on React version

This is to about how to change the version of React that you use if necessary

Version 18 of React was released late March 2022. The code in this course should work with the new React version. However, some libraries might not yet be compatible with React 18. At the moment of writing (4th April) at least the Apollo client used in [part 8](https://fullstackopen.com/en/part8) does not yet work with most recent React.

In case you end up in a situation where your application breaks because of library compatibility problems, *downgrade* to the older React by changing the file *package.json* as follows:

{

"dependencies": {

"react": "^17.0.2", "react-dom": "^17.0.2", "react-scripts": "5.0.0",

"web-vitals": "^2.1.4"

},

// ...

}

After the change is made, reinstall dependencies by running

npm install

Note that also the file *index.js* needs to be changed a bit. For React 17 it looks like

import ReactDOM from 'react-dom'

import App from './App'

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

but for React 18 the correct form is

import React from 'react'

import ReactDOM from 'react-dom/client'

import App from './App'

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

### D.2. Complex State

Documentation spread object

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax>

StackOverflow on why you should not modify state directly

<https://stackoverflow.com/questions/37755997/why-cant-i-directly-modify-a-components-state-really/40309023#40309023>

react documentation on whether to store all of the states in a single state object, or in separate pieces of states

<https://reactjs.org/docs/hooks-faq.html#should-i-use-one-or-many-state-variables>

So, up to now in the application we have defined only one state. Here it is :

const [ counter, setCounter ] = useState(0)

How do we handle an application with two states ?

React gives us two options

* option 1: In most cases, the easiest and best way is by using the useState function multiple times
* option 2: But, we could also store all the states into a single state object. But as we will see this is cumbersome

We will show how we can do the two options

#### D.2.1 Using the useState function multiple times.

Here we create an app with two states left and right

Remember that when we define

const [ left, setleft ] = useState(0)

we mean that left =0, ie the counter to the state is initialized to 0

import { useState } from 'react'

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

 const App = () => {

  const [left, setLeft ] = useState(0)

  const [right,setRight] = useState(0)

  return (

    <div>

    <Display counter = {"Left value is "  + left}/>

    <Display counter = {"Right value is " + right}/>

    <button onClick = {()=>setLeft(left+1)}> LEFT </button>

    <button onClick = {()=>setRight(right+1)}> RIGHT</button>

    </div>

  )

}

export default App

#### D.2.1 Using the useState function once

We define the state as an object with two fields that are counters initialized at 0

const [clicks, setClicks ] = useState({left:0, right:0})

the useState function always returns an array.

* The first element is of the same type as the input argument. So it is an object with two fields here
* setClicks is a method that updates the clicks variable in some way defined by the user

Then we define functions that we will use in our event handler

const handLeftClick = () => {

    // REACT rule: never modify a state. copy the state into a new one

    const newClicks = {

      left: clicks.left+1,

      right: clicks.right

    }

    setClicks(newClicks)

  }

  const handRightClick = () => {

    // REACT rule: never modify a state. copy the state into a new one

    const newClicks = {

      left: clicks.left,

      right: clicks.right+1

    }

    setClicks(newClicks)

  }

Upon leftClick we update the state variable clicks by incrementing the left field, and we re-render the app by calling the state method with the updated counter.

Note that in React it is forbidden to modify a state. So we create a new state variable newClicks and ass that new state to the method.

So here is the code that works

import { useState } from 'react'

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

 const App = () => {

  // clicks is an object with 2 fields clicks.left and click.right. setClicks is the updater method for teh clicks object

  const [clicks, setClicks ] = useState({left:0, right:0})

  const handLeftClick = () => {

    // REACT rule: never modify a state. copy the state into a new one

    const newClicks = {

      left: clicks.left+1,

      right: clicks.right

    }

    setClicks(newClicks)

  }

  const handRightClick = () => {

    // REACT rule: never modify a state. copy the state into a new one

    const newClicks = {

      left: clicks.left,

      right: clicks.right+1

    }

    setClicks(newClicks)

  }

  return (

    <div>

    <Display counter = {"Left value is "  + clicks.left}/>

    <Display counter = {"Right value is " + clicks.right}/>

    <button onClick = {handLeftClick}> LEFT </button>

    <button onClick = {handRightClick}> RIGHT</button>

    </div>

  )

}

export default App

We can use object spreads to make the syntax shorter

The idea is this: When we want to copy and object and modify only a subset of its fields we do not have to rewrite all the fields. To create a copy of the click object we do newClicks={…click} and to specify that only the left attribute is modified we write newClicks={…click, left:click.left+1}.

  const handLeftClick = () => {

    // REACT rule: never modify a state. copy the state into a new one

    const newClicks = {

      ...clicks,

      left: clicks.left+1

    }

    setClicks(newClicks)

  }

  const handRightClick = () => {

    // REACT rule: never modify a state. copy the state into a new one

    const newClicks = {

      ...clicks,

      right: clicks.right+1

    }

    setClicks(newClicks)

  }

We can make the code shorter by making the arrow function return a single expression

  const handLeftClick = () => { setClicks({...clicks,left: clicks.left+1})}

And so this works and it is beautifully short

import { useState } from 'react'

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

 const App = () => {

  // clicks is an object with 2 fields clicks.left and click.right. setClicks is the updater method for teh clicks object

  const [clicks, setClicks ] = useState({left:0, right:0})

  const handLeftClick = () => { setClicks({...clicks,left: clicks.left+1})}

  const handRightClick = () => {setClicks({...clicks,right: clicks.right+1})}

  return (

    <div>

    <Display counter = {"Left value is "  + clicks.left}/>

    <Display counter = {"Right value is " + clicks.right}/>

    <button onClick = {handLeftClick}> LEFT </button>

    <button onClick = {handRightClick}> RIGHT</button>

    </div>

  )

}

export default App

But do not be fooled this

  const handRightClick = () => {setClicks({...clicks,right: clicks.right+1})}

Here the {...clicks,right: clicks.right+1})}

{…clicks} creates a copy of the clicks object where the right field is modified. That copy of the object is passed to the setClicks function, which updates the state and re-renders the page.

### D.3. Handling Arrays

We start from this version of the app that

* uses useState multiple times
* calls the event handlers as helpers

import { useState } from 'react'

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

 const App = () => {

  const [left, setLeft ] = useState(0)

  const [right,setRight] = useState(0)

  const handleLeftClick = () => {

    setLeft(left+1)

  }

  const handleRightClick = () => {

    setRight(right+1)

  }

  return (

    <div>

    <Display counter = {"Left value is "  + left}/>

    <Display counter = {"Right value is " + right}/>

    <button onClick = {handleLeftClick}> LEFT </button>

    <button onClick = {handleRightClick}> RIGHT</button>

    </div>

  )

}

export default App

Now each time click we want to display the series of strings L or R that are clicked like this

Text

Description automatically generated

To do this, we create a new state where the starting value is an empty list

const [allClicks, setAllClick] = useState([])

in the event handler helper function, we call the state method on a **copy**  of the state variable. In this copy we concatenate the state variable with the string corresponding to L or R.

Attention you should use concat, not push. Because push mutates the state variable, which is forbidden (because it creates problems that are hard to debug). On the contrary, concat creates a copy of the state variable, and adds the passed string to it.

Here is the helper function

  const handleLeftClick = () => {

    setLeft(left+1)

    setAllClick(allClicks.concat(['L']))

  }

Finally we display the list as a string where elements are separated with a space using the array.join method

 <Display print = {allClicks.join(' ')} />

All in all here is the code

import { useState } from 'react'

const Display = (props) => <div> {props.print}</div>

 const App = () => {

  const [left, setLeft ] = useState(0)

  const [right,setRight] = useState(0)

  const [allClicks, setAllClick] = useState([])

  const handleLeftClick = () => {

    setLeft(left+1)

    setAllClick(allClicks.concat(['L']))

  }

  const handleRightClick = () => {

    setRight(right+1)

    setAllClick(allClicks.concat(['R']))

  return (

    <div>

    <Display print = {"Left value is "  + left}/>

    <Display print = {"Right value is " + right}/>

    <button onClick = {handleLeftClick}> LEFT </button>

    <button onClick = {handleRightClick}> RIGHT</button>

    <Display print = {allClicks.join(' ')} />

    </div>

  )

}

export default App

### D.4. Conditional rendering

React documentation on conditional re-rendering

<https://reactjs.org/docs/conditional-rendering.html>

Definition : Conditional re-rendering is when a React component renders different React elements depending on the state (as defined i useState() ) of the application

Basically: This is about using if statements inside React components

Here we write a History component that handles the printing of the array. It must print a message when the array is empty, and the array itself if the array is not empty.

This is a component, not a helper to the App() component

Remember that when we define a component that is used later inside another component we use the props argument to get any attribute

Aside: to do a console.log, you must enclose it into brackets because it is Javascript code

    {console.log(allClicks)}

    {console.log(allClicks == [])}

Here is the history component

const History = (props) => {

  if (props.allClicks.length == 0) {

    return(

      <p> You are supposed to click the buttons to build a history</p>

    )

  }

  else

    return (

      <div> button press history: {props.allClicks.join(' ')}  </div>

    )

}

Here is the whole code. Note that we are using the Button React component

import { useState } from 'react'

const Display = (props) => <div> {props.print}</div>

const History = (props) => {

  if (props.allClicks.length == 0) {

    return(

      <p> You are supposed to click the buttons to build a history</p>

    )

  }

  else

    return (

      <div> button press history: {props.allClicks.join(' ')}  </div>

    )

}

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

 const App = () => {

  const [left, setLeft ] = useState(0)

  const [right,setRight] = useState(0)

  const [allClicks, setAllClick] = useState([])

  const handleLeftClick = () => {

    setLeft(left+1)

    setAllClick(allClicks.concat(['L']))

  }

  const handleRightClick = () => {

    setRight(right+1)

    setAllClick(allClicks.concat(['R']))

  }

  return (

    <div>

    <Display print = {"Left value is "  + left}/>

    <Display print = {"Right value is " + right}/>

    <Button onClick = {handleLeftClick}  text = "LEFT"/>

    <Button onClick = {handleRightClick}  text = "RIGHT"/>

    <History allClicks = {allClicks}/>

    {console.log(allClicks)}

    {console.log(allClicks == [])}

    </div>

  )

}

export default App

### D.5. Old React

React documentation state hook available from version 16.8.0 onwards

<https://reactjs.org/docs/hooks-state.html>

Findingthe 16.8.0 package

<https://www.npmjs.com/package/react/v/16.8.0>

Documentation REACT class component

<https://reactjs.org/docs/react-component.html>

before that, components that required state had to be defined as a class component

This course uses Hooks because it is the future of React. Later in the course we will learn about Reaxr class components because they still exist in legacy code, tutorials and examples.

### D.6. Debugging React applications

#### Rule 1 of web development

* As developers we spend most of the time not writing code, but figuring out how code works and debugging it
* Lucky for us, React is an extremely developer-friendly library when it comes to debugging.

#### The first rule of web development

**Keep the browser's developer console open at all times.**

Keep both your code and the web page open together **at the same time, all the time**.

don't write more code but rather find and fix the problem **immediately**.

#### Debugging method 1: Use console.log

For example to debug this

const Button = ({ onClick, text }) => (

<button onClick={onClick}>

{text}

</button>

)

Transform it to less compact form and do a consoe.log

const Button = (props) => {

console.log(props) const { onClick, text } = props

return (

<button onClick={onClick}>

{text}

</button>

)

}

* [Debugging React applications](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#debugging-react-applications)
* [Rules of Hooks](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#rules-of-hooks)

Use comma, not +, in console.log

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

NOT

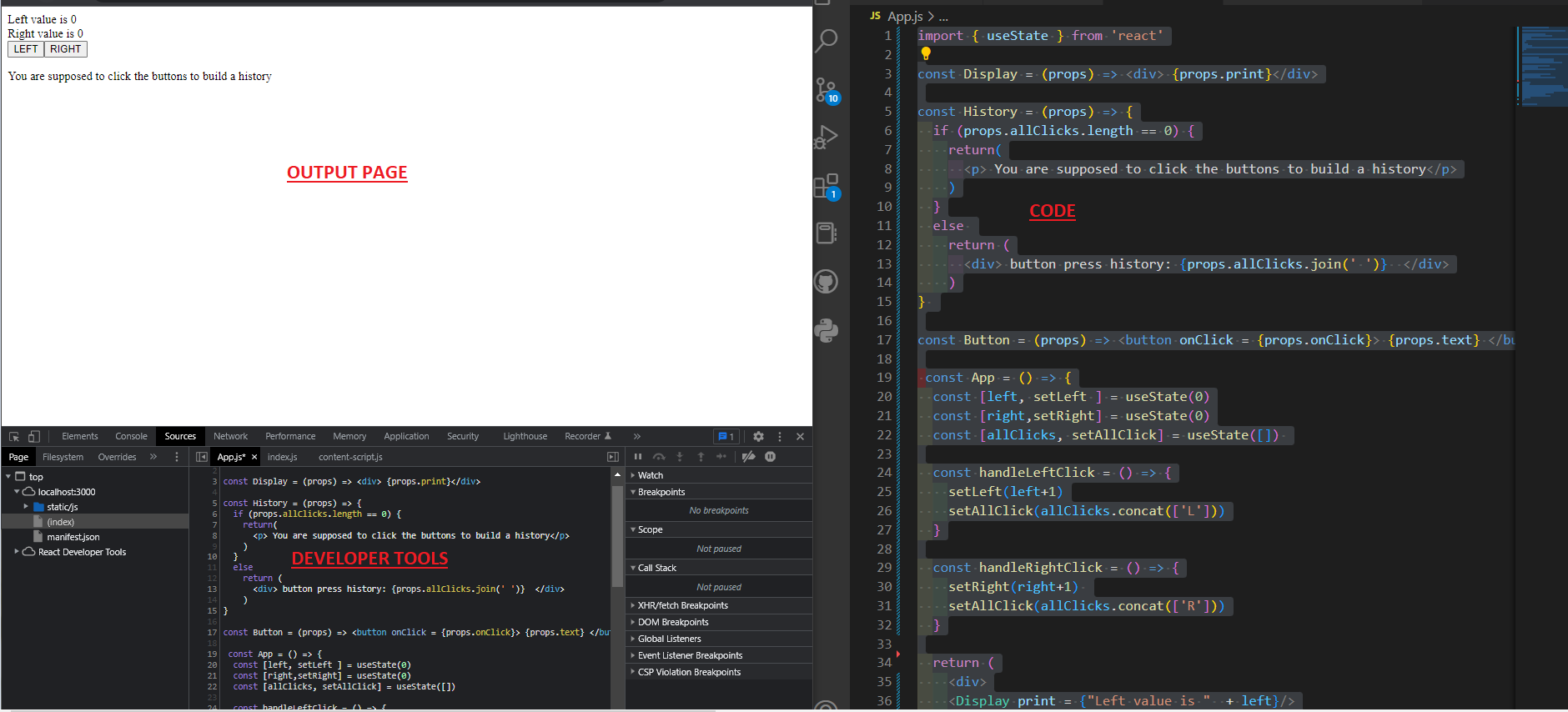
console.log('props value is ' + props)

By the way you can dock the developer tools down like this by clicking the three dots to the right and choose the dock side

Graphical user interface, text

Description automatically generated

So your work environment should look like this



It should display three things

* Your code
* Your output page
* The Developer tools

#### [IMPORTANT]Debugging method 2: Using the debugger

##### Method 1: Write “debugger” in the vscode code

* In the developer tab, you can inspect your code under the Sources tab
* First, on vscode (not in the developer source tab), anywhere in your code, write “debugger” like this

Text

Description automatically generated

* This will immediately display, on the webpage, “paused in debugger”, and will populate the right hand side of the developer tools with everything that you can access in the app: the variables, methods…

Graphical user interface, text

Description automatically generated

You can inspect the values of the variables that are listed in the scope above, in the console tab

Graphical user interface, website

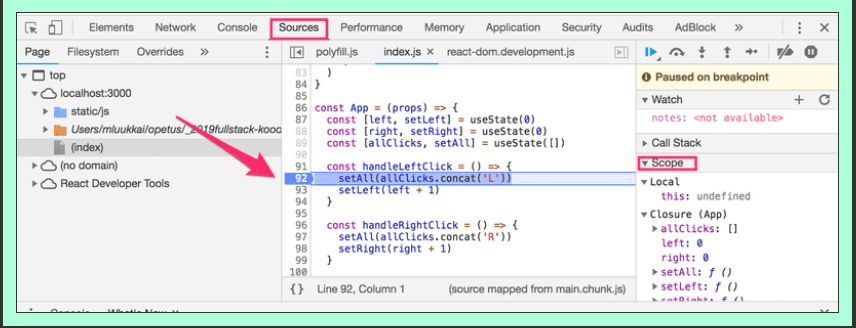
Description automatically generated

Once the cause of the bug is discovered you can remove the debugger command and refresh the page.

When debugging you can use the arrows



Importantly you can view the values in the scope section



I am here

##### Method 2: Add a breakpoint directly in the source tab of the developer tools

In source tab, click on any line. Or, right click and select add breakpoint

Then, refresh the page. It will become into “Pause in debugger” mode, with the scope populated with the values of the variables at the breakpoint

Text

Description automatically generated

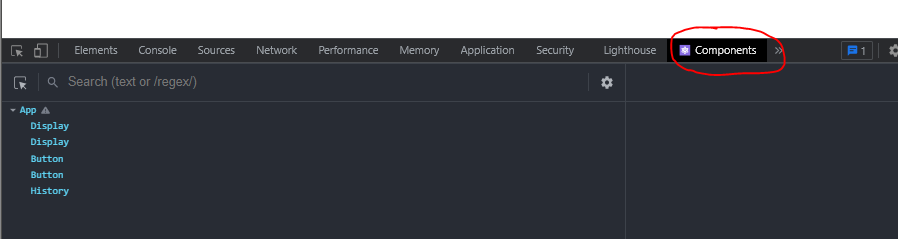
#### [IMPORTANT]Debugging method 3: Using the Components tab

Add the React Developer tools extension

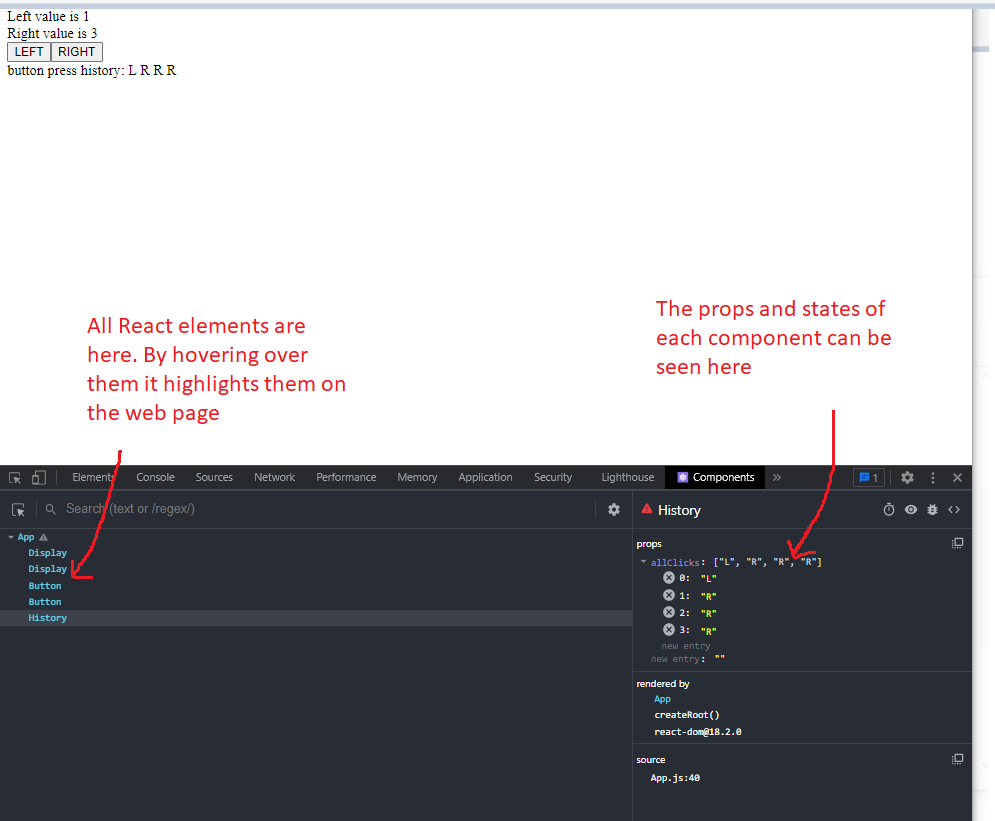
<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>

It will create a new tab called “components” in the developer tools

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>



This tab is amazing. It allows you to inspect the React Elements, along with their props and states



Importantly, click on the App component to the left. This will show you the state of hooks in the order of definition.

We defined states as

  const [left, setLeft ] = useState(0)

  const [right,setRight] = useState(0)

  const [allClicks, setAllClick] = useState([])

We can see the value of the state variable for each state hook

Graphical user interface, text, application

Description automatically generated

Last tip : the props pose problems. So

Graphical user interface, text

Description automatically generated

Important tip: console.log the props object in the component just before the return statement

const Content = (props) =>  {

  console.log(props)

  return(

    <>

    <Part parts = {props.parts[0]} exercise = {props.exercise[0]} />

    <Part parts = {props.parts[1]} exercise = {props.exercise[1]} />

    <Part parts = {props.parts[2]} exercise = {props.exercise[2]} />

    </>

  )

}

In summary the debugging methods are

1. Use console.log  
2. Use the developer tools debugger  
     a. Write debugger in vscode in the line you want to stop the code  
     b. Add a breakpoint directly in the source tab of the developer tools  
3. Use the components tab from the React Developer tools extension

* [Rules of Hooks](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#rules-of-hooks)
* [Event Handling Revisited](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#event-handling-revisited)
* [Function that returns a function](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#function-that-returns-a-function)
* [Passing Event Handlers to Child Components](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#passing-event-handlers-to-child-components)
* [Do Not Define Components Within Components](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#do-not-define-components-within-components)

### D.7. Rules of Hooks

Rule of Hooks :

* The useState function (as well as the useEffect function introduced later on in the course) *must not be called* from inside of a loop, a conditional expression, or any place that is not a function defining a component.
* hooks may only be called from the inside of a function body that defines a React component

The reason is

his must be done to ensure that the hooks are always called in the same order, and if this isn't the case the application will behave erratically.

This code shows examples of bad code

const App = () => {

  // these are ok

  const [age, setAge] = useState(0)

  const [name, setName] = useState('Juha Tauriainen')

  if ( age > 10 ) {

    // this does not work because useState is defined inside an if statement

    const [foobar, setFoobar] = useState(null)

  }

  for ( let i = 0; i < age; i++ ) {

    // this does not work because useState is defined inside aa for loop

    const [rightWay, setRightWay] = useState(false)

  }

  const notGood = () => {

    // // this does not work because useState is defined inside an if statement (event if it is wrapped inside a helper)

    const [x, setX] = useState(-1000)

  }

  return (

    //...

  )

}

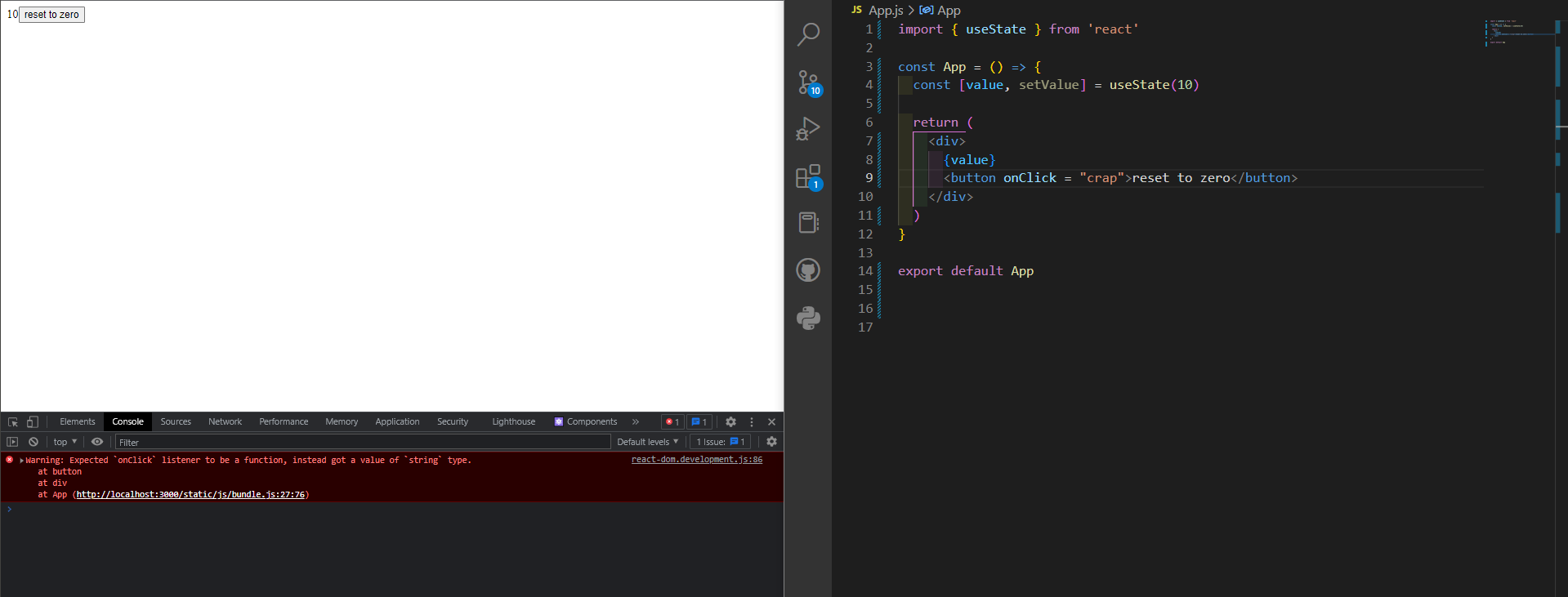
### D.8. Event Handling Revisited

This is just a review of event handler. It reinforces the concepts, but there is nothing new

Event handling has proven to be a difficult topic in previous iterations of this course.For this reason we will revisit the topic.

Event handlers must always be a function or a reference to a function

For example here It is set to a string which causes an error in the console



*Read the error:*

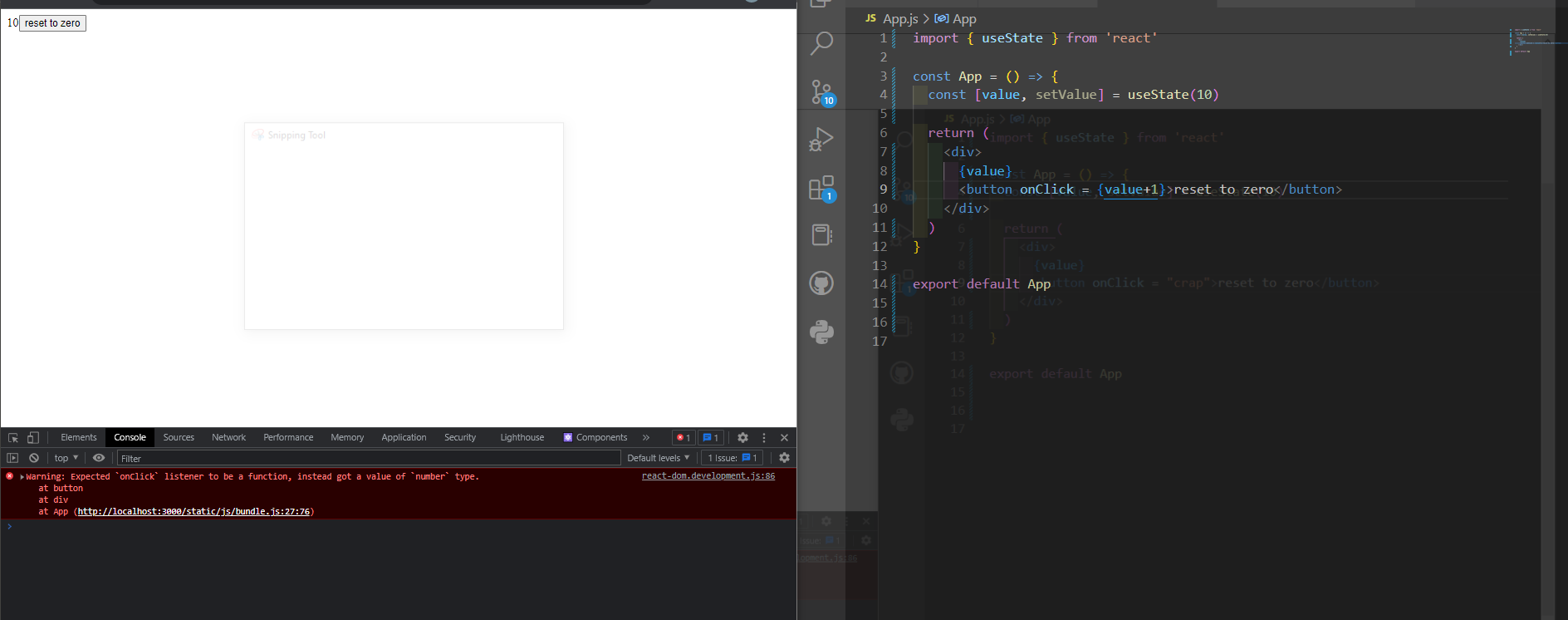
react-dom.development.js:86 Warning: Expected `onClick` listener to be a function, instead got a value of `string` type.

at button

at div

at App (http:/

This does not work either



Q. State two reasons why this attempt is flawed button onClick = {value=0}>

import { useState } from 'react'

const App = () => {

  const [value, setValue] = useState(10)

  return (

    <div>

      {value}

      <button onClick = {value=0}>reset to zero</button>

    </div>

  )

}

export default App

Reason 1: AN event handler accepts only a function or reference to a function. Not a variable or an assignment to a variable

Reason 2: We must never mutate a state directly in REACT

* [Event Handling Revisited](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#event-handling-revisited)
* [Function that returns a function](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#function-that-returns-a-function)
* [Passing Event Handlers to Child Components](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#passing-event-handlers-to-child-components)
* [Do Not Define Components Within Components](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#do-not-define-components-within-components)

Q. Why does this not work even when our event handler contains a function console.log?

<button onClick={console.log('clicked the button')}>

  button

</button>

The issue here is that our event handler is defined as a *function call* which means that the event handler is actually assigned the returned value from the function, which in the case of console.log is *undefined*.

Q. Why this does not work ? Give two reasons

<button onClick={setValue(0)}>button</button>

* First because we try to set a function call as event handler
* Second When the component is rendered the function setValue(0) gets executed which in turn causes the component to be re-rendered. Re-rendering in turn calls setValue(0) again, resulting in an infinite recursion.

Q. is this right ?

 <button onClick = {() => console.log('clicked the button')}>reset to zero</button>

Yes because it is a function definition

When the component gets rendered, no function gets called and only the reference to the arrow function is set to the event handler. Calling the function happens only once the button is clicked.

Q. Is this right ?

<button onClick={() => setValue(0)}>button</button>

Yes because function definition

Although Defining event handlers directly in the attribute of the button is not necessarily the best possible idea.

### D.9. Function that returns a function

Another way to define an event handler is to use function that returns a function.

You probably won't need to use functions that return functions in any of the exercises in this course. If the topic seems particularly confusing, you may skip over this section for now and return to it later.

The point is

Assume you have two buttons that get passed the same handler function. You want to parametrize that handler function so that each of the button can get its own individualized event response although both buttons relie on the same function

With the tools that we have now this is not possible. We would need to create two different event handler functions. This is because the event handler accepts only a function reference or definition, not a function call. Since we cannot call the function, we cannot pass a parameter to the function. Therefore we cannot personalize the event handler.

We want Both buttons get their own individualized event handlers.

The solution: There are two solutions that we will present here.Choosing between these two is a matter of taste. We will present the two solutions

* Solution 1: Passing to the handler a call to a function that returns a function
* Solution 2: Passing to the handler the definition of a function that calls a function (with some parameter)

#### Solution 1: Function that returns a function at the handler function level

Q. [Important that you code it yourself]Let’s create an app with three buttons. When the buttons are pressed, they print in the console Hello “something”. “Something” is passed as a parameter to the Hello function. Although the handler function is a function call, the function call must return a function definition, so the handler is effectively a function definition.

I start from this empty React template as I call it

import { useState } from 'react'

 const App = () => {

  return (

    <div>

    </div>

  )

}

export default App

I start with this simple code, where the three buttons are constrained to print the same thing

import { useState } from 'react'

 const App = () => {

    const hello = () => console.log("World")

  return (

    <div>

      <button onClick={hello}> World</button>

      <button onClick={hello}> React</button>

      <button onClick={hello}> Function</button>

    </div>

  )

}

export default App

Now to parametrize the hello functionI do this

import { useState } from 'react'

const App = () => {

    const hello = (word) => () => console.log(word)

  return (

    <div>

      <button onClick={hello("World")}> World</button>

      <button onClick={hello("React")}> React</button>

      <button onClick={hello("Function")}> Function</button>

    </div>

  )

}

export default App

Graphical user interface, text, application

Description automatically generated

Q. Now give the App() component a state. Create a handler function called setToValue that sets the state variable of a component to a given value that is passed as a parameter to the handler. Remember to make the handler a function that returns a function.Then create three buttons: One button that sets the value of the state to 1000. Another button that rests the value to 0. And finally a button that increments the state variable by one

Here is the code

import { useState } from 'react'

 const App = () => {

    const [value, setValue] = useState(10)

    const setToValue = (newValue) => () => setValue(newValue)

  return (

    <div>

      <button onClick={setToValue(1000)}> Thousand</button>

      <button onClick={setToValue(0)}> Reset</button>

      <button onClick={setToValue(value+1)}> Increment</button>

      {value}

    </div>

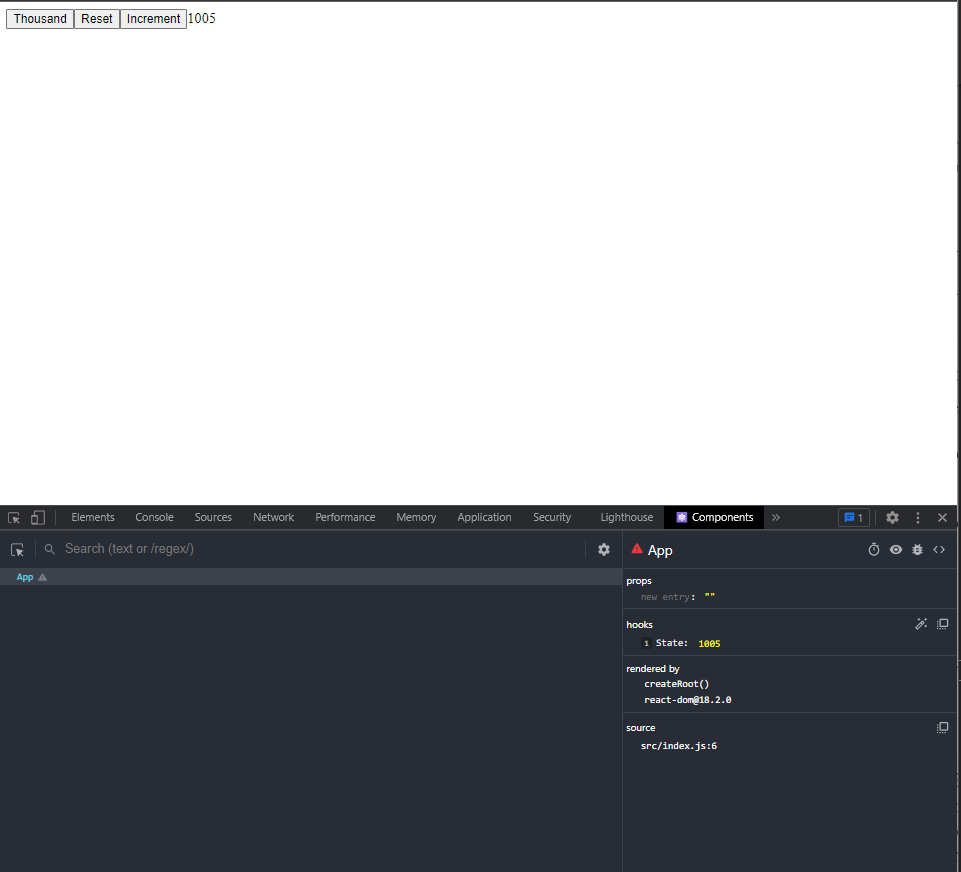
  )

}

export default App

Note that there are two ways to verify that It works

* Way 1: Display the {value} directly in the web page
* Way 2: Developer tools 🡪 components tab



#### Solution 2: Function that returns a function at the element level

We take the code above and change it in this way: We pass a function definition to the handler that returns the handler function

import { useState } from 'react'

 const App = () => {

    const [value, setValue] = useState(10)

    const setToValue = (newValue) =>  setValue(newValue)

  return (

    <div>

      <button onClick={() => setToValue(1000)}> Thousand</button>

      <button onClick={() => setToValue(0)}> Reset</button>

      <button onClick={() => setToValue(value+1)}> Increment</button>

      {value}

    </div>

  )

}

export default App

The handler is still a function that returns a function. The difference is that the “composition” of functions is done inside the element

Graphical user interface, application, Word

Description automatically generated

Choosing between the two is a matter of preference. I prefer method 1, because of the rule of handlers where it is preferred not to define the function inside the button element and use function reference instead

### D.10. Passing Event handlers to child components

Q. Now take the same code as above, this time using a Button component instead of the button element

Here is the code. It is important to code it myself, after having a look *once* at the solution code and understand it

import { useState } from 'react'

 const App = () => {

    const [value, setValue] = useState(10)

    const setToValue = (newValue) =>  setValue(newValue)

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

  return (

    <div>

      <Button handleClick = {() => setToValue(1000)} text = "Thousand"/>

      <Button handleClick = {() => setToValue(0)} text = "Reset"/>

      <Button handleClick = {() => setToValue(value+1)} text = "Increment"/>

      {value}

    </div>

  )

}

export default App

### D.11. Do not define components within components

Never define components inside of other components. The method provides no benefits and leads to many unpleasant problems. The biggest problems are due to the fact that React treats a component defined inside of another component as a new component in every render. This makes it impossible for React to optimize the component.

Q. What is wrong with this code ?

// This is the right place to define a component

const Button = (props) => (

  <button onClick={props.handleClick}>

    {props.text}

  </button>

)

const App = () => {

  const [value, setValue] = useState(10)

  const setToValue = newValue => {

    console.log('value now', newValue)

    setValue(newValue)

  }

  // Do not define components inside another component

  const Display = props => <div>{props.value}</div>

  return (

    <div>

      <Display value={value} />

      <Button handleClick={() => setToValue(1000)} text="thousand" />

      <Button handleClick={() => setToValue(0)} text="reset" />

      <Button handleClick={() => setToValue(value + 1)} text="increment" />

    </div>

  )

}

Answer: The display component is ill defined. It should be defined outsode the App component. It violates the rule of components

### D.12. Useful Reading

You may find the following links useful:

* The [official React documentation](https://reactjs.org/docs/hello-world.html) is worth checking out at some point, although most of it will become relevant only later on in the course. Also, everything related to class-based components is irrelevant to us;
* Some courses on [Egghead.io](https://egghead.io/) like [Start learning React](https://egghead.io/courses/start-learning-react) are of high quality, and recently updated [The Beginner's Guide to React](https://egghead.io/courses/the-beginner-s-guide-to-reactjs) is also relatively good; both courses introduce concepts that will also be introduced later on in this course. **NB** The first one uses class components but the latter uses the new functional ones.

### Exercices

There are 14 exercies to complete

This morning 14/4 = 3.5 exercices

#### Exercise 1.1

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

Use create-react-app to initialize a new application. Modify *index.js* to match the following

import React from 'react'

import ReactDOM from 'react-dom/client'

import App from './App'

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

and *App.js* to match the following

const App = () => {

const course = 'Half Stack application development'

const part1 = 'Fundamentals of React'

const exercises1 = 10

const part2 = 'Using props to pass data'

const exercises2 = 7

const part3 = 'State of a component'

const exercises3 = 14

return (

<div>

<h1>{course}</h1>

<p>

{part1} {exercises1}

</p>

<p>

{part2} {exercises2}

</p>

<p>

{part3} {exercises3}

</p>

<p>Number of exercises {exercises1 + exercises2 + exercises3}</p>

</div>

)

}

export default App

and remove extra files (App.css, App.test.js, index.css, logo.svg, setupTests.js, reportWebVitals.js)).

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

Define the new components in file *App.js*.

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

const App = () => {

// const-definitions

return (

<div>

<Header course={course} />

<Content ... />

<Total ... />

</div>

)

}

**WARNING** create-react-app automatically makes the project a git repository unless the application is created within an already existing repository. Most likely you **do not want** the project to become a repository, so run the command rm -rf .git in the root of the project.

* Steps to create a new app. In gitbash

npx create-react-app app-part1-1

npx kill-port 3000

cd app-part1-1

npm start

* This output must stay the same

Text

Description automatically generated

Here is the code

const Header = (props) => <h1>{props.course}</h1>

const Content = (props) => <p> {props.part} {props.exercise} </p>

const Total = (props) => <p>Number of exercises {props.elem}</p>

const App = () => {

  const course = 'Half Stack application development'

  const part1 = 'Fundamentals of React'

  const exercises1 = 10

  const part2 = 'Using props to pass data'

  const exercises2 = 7

  const part3 = 'State of a component'

  const exercises3 = 14

  return (

    <div>

      <Header course = {course}/>

      <Content part = {part1} exercise = {exercises1}/>

      <Content part = {part2} exercise = {exercises2}/>

      <Content part = {part3} exercise = {exercises3}/>

      <Total elem={exercises1+exercises2+exercises3}/>

    </div>

  )

}

export default App

I make a commit at this point

#### Exercise 1.2

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

const Content = ... {

return (

<div>

<Part .../>

<Part .../>

<Part .../>

</div>

)

}

Our application passes on information in quite a primitive way at the moment, since it is based on individual variables. This situation will improve soon.

Answer:

It was tricky and it seems weird. Here is my solution

Here is my solution

const Header = (props) => <h1>{props.course}</h1>

const Part = (props) => <p> {props.parts} {props.exercise} </p>

const Content = (props) =>  {

  return(

    <>

    <Part parts = {props.parts[0]} exercise = {props.exercise[0]} />

    <Part parts = {props.parts[1]} exercise = {props.exercise[1]} />

    <Part parts = {props.parts[2]} exercise = {props.exercise[2]} />

    </>

  )

}

const Total = (props) => <p>Number of exercises {props.elem}</p>

const App = () => {

  const course = 'Half Stack application development'

  const part1 = 'Fundamentals of React'

  const exercises1 = 10

  const part2 = 'Using props to pass data'

  const exercises2 = 7

  const part3 = 'State of a component'

  const exercises3 = 14

  return (

    <div>

      <Header course = {course}/>

      <Content parts = {[part1, part2, part3]} exercise = {[exercises1, exercises2, exercises3]}/>

      <Total elem={exercises1+exercises2+exercises3}/>

    </div>

  )

}

export default App

It could be better with objects but we are not asked about objects

Important tip: console.log the props object in the component just before the return statement

const Content = (props) =>  {

  console.log(props)

  return(

    <>

    <Part parts = {props.parts[0]} exercise = {props.exercise[0]} />

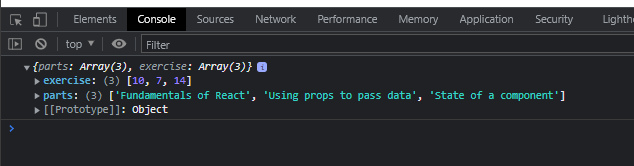
    <Part parts = {props.parts[1]} exercise = {props.exercise[1]} />

    <Part parts = {props.parts[2]} exercise = {props.exercise[2]} />

    </>

  )

}



#### Exercise 1.3

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

const App = () => {

const course = 'Half Stack application development'

const part1 = {

name: 'Fundamentals of React',

exercises: 10

}

const part2 = {

name: 'Using props to pass data',

exercises: 7

}

const part3 = {

name: 'State of a component',

exercises: 14

}

return (

<div>

...

</div>

)

}

Answer

This one is not easy. It requires a Test driven development mentality

My strategy is to advance step by step without error at each step. Use TDD method !! But without writing tests, just checking and correcting erros at this steps

A picture containing text

Description automatically generated

First step I keep just one part

const Header = (props) => <h1>{props.course}</h1>

const Part = (props) => <p> {props.parts} {props.exercise} </p>

const Content = (props) =>  {

  return(

    <>

    <Part parts = {props.parts[0]} exercise = {props.exercise[0]} />

    </>

  )

}

const Total = (props) => <p>Number of exercises {props.elem}</p>

const App = () => {

  const course = 'Half Stack application development'

  const part1 = 'Fundamentals of React'

  const exercises1 = 10

  return (

    <div>

      <Header course = {course}/>

      <Content parts = {[part1]} exercise = {[exercises1]}/>

      {/\* <Total elem={exercises1+exercises2+exercises3}/> \*/}

    </div>

  )

}

export default App

No error

Then I work out the props from down up

ç Text

Description automatically generated

No error

From here I add new objects. No errors. So here is the final version of this exercice

const Header = (props) => <h1>{props.course}</h1>

const Part = (props) => <p> {props.parts.name} {props.parts.exercises} </p>

const Content = (props) =>  {

  return(

    <>

    <Part parts = {props.parts[0]} />

    <Part parts = {props.parts[1]} />

    <Part parts = {props.parts[2]} />

    </>

  )

}

const Total = (props) => <p>Number of exercises {props.elem}</p>

const App = () => {

  const course = 'Half Stack application development'

  const part1 = {

    name: 'Fundamentals of React',

    exercises: 10

  }

  const part2 = {

    name: 'Using props to pass data',

    exercises: 7

  }

  const part3 = {

    name: 'State of a component',

    exercises: 14

  }

  return (

    <div>

      <Header course = {course}/>

      <Content parts = {[part1, part2, part3]}/>

      <Total elem={part1.exercises+part2.exercises+part3.exercises}/>

    </div>

  )

}

export default App

#### Exercise 1.4

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

const App = () => {

const course = 'Half Stack application development'

const parts = [

{

name: 'Fundamentals of React',

exercises: 10

},

{

name: 'Using props to pass data',

exercises: 7

},

{

name: 'State of a component',

exercises: 14

}

]

return (

<div>

...

</div>

)

}

**NB** at this point *you can assume that there are always three items*, so there is no need to go through the arrays using loops. We will come back to the topic of rendering components based on items in arrays with a more thorough exploration in the [next part of the course](https://fullstackopen.com/en/part2).

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

const App = () => {

// const definitions

return (

<div>

<Header course={course} />

<Content parts={parts} />

<Total parts={parts} />

</div>

)

}

The point of this exercice is to apply this component rule:

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

This one is easier. I proceed with precaution, adding the code first as comments, then uncommenting each step and ensuring that it works

we change the Total component as well

const Header = (props) => <h1>{props.course}</h1>

const Part = (props) => <p> {props.parts.name} {props.parts.exercises} </p>

const Content = (props) =>  {

  return(

    <>

    <Part parts = {props.parts[0]} />

    <Part parts = {props.parts[1]} />

    <Part parts = {props.parts[2]} />

    </>

  )

}

const Total = (props) => <p>Number of exercises {props.parts[0].exercises+props.parts[1].exercises+props.parts[2].exercises}</p>

const App = () => {

  const course = 'Half Stack application development'

  const parts = [

    {name: 'Fundamentals of React', exercises: 10},

    {name: 'Using props to pass data', exercises: 7},

    {name: 'State of a component', exercises: 14}

  ]

  return (

    <div>

      <Header course = {course}/>

      <Content parts = {parts}/>

      <Total parts={parts}/>

    </div>

  )

}

export default App

Notice at this point how beautifully the return statement is simplified from the initial

    <div>

      <Header course = {course}/>

      <Content part = {part1} exercise = {exercises1}/>

      <Content part = {part2} exercise = {exercises2}/>

      <Content part = {part3} exercise = {exercises3}/>

      <Total elem={exercises1+exercises2+exercises3}/>

    </div>

To the much better

  return (

    <div>

      <Header course = {course}/>

      <Content parts = {parts}/>

      <Total parts={parts}/>

    </div>

  )

#### Exercise 1.5

Q

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

const App = () => {

const course = {

name: 'Half Stack application development',

parts: [

{

name: 'Fundamentals of React',

exercises: 10

},

{

name: 'Using props to pass data',

exercises: 7

},

{

name: 'State of a component',

exercises: 14

}

]

}

return (

<div>

...

</div>

)

}

Here is the answer

const Header = (props) => <h1>{props.course.name}</h1>

const Part = (props) => <p> {props.parts.name} {props.parts.exercises} </p>

const Content = (props) =>  {

  return(

    <>

    <Part parts = {props.course.parts[0]} />

    <Part parts = {props.course.parts[1]} />

    <Part parts = {props.course.parts[2]} />

    </>

  )

}

const Total = (props) => <p>Number of exercises {props.course.parts[0].exercises+props.course.parts[1].exercises+props.course.parts[2].exercises}</p>

const App = () => {

  const course = {

    name: 'Half Stack application development',

    parts : [

      {name: 'Fundamentals of React', exercises: 10},

      {name: 'Using props to pass data', exercises: 7},

      {name: 'State of a component', exercises: 14}

    ]

  }

  return (

    <div>

      <Header course = {course}/>

      <Content course = {course}/>

      <Total course={course}/>

    </div>

  )

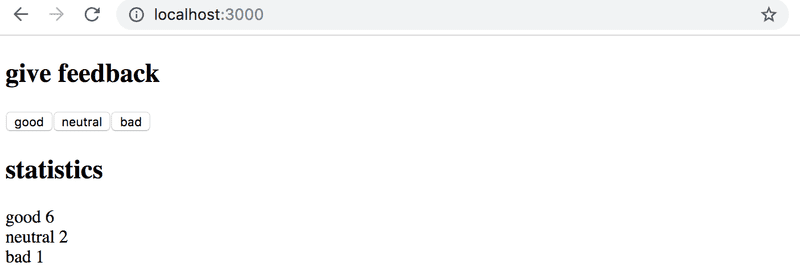
}

export default App

#### Exercise 1.6

Like most companies, [Unicafe](https://www.unicafe.fi/#/9/4) collects feedback from its customers. Your task is to implement a web application for collecting customer feedback. There are only three options for feedback: *good*, *neutral*, and *bad*.

The application must display the total number of collected feedback for each category. Your final application could look like this:



Note that your application needs to work only during a single browser session. Once you refresh the page, the collected feedback is allowed to disappear.

It is advisable to use the same structure that is used in material and previous exercise. File *index.js* is as follows:

import React from 'react'

import ReactDOM from 'react-dom/client'

import App from './App'

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

You can use the code below as a starting point for the *App.js* file:

import { useState } from 'react'

const App = () => {

// save clicks of each button to its own state

const [good, setGood] = useState(0)

const [neutral, setNeutral] = useState(0)

const [bad, setBad] = useState(0)

return (

<div>

code here

</div>

)

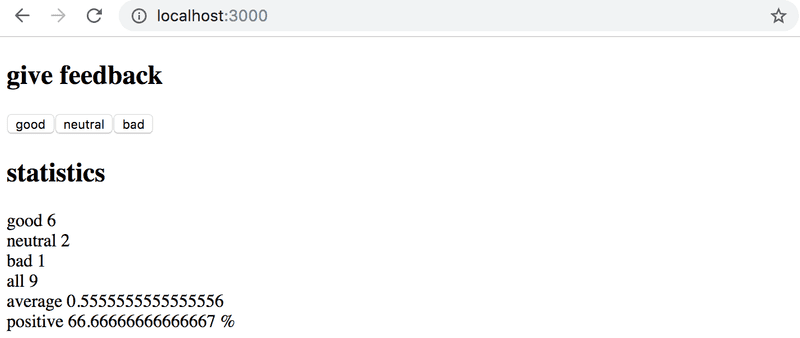
}

export default App

remove extra files (App.css, App.test.js, index.css, logo.svg, setupTests.js, reportWebVitals.js)).

#### Exercice 1.7

Expand your application so that it shows more statistics about the gathered feedback: the total number of collected feedback, the average score (good: 1, neutral: 0, bad: -1) and the percentage of positive feedback.



1.8: unicafe step3

import { useState } from 'react'

const App = () => {

  // save clicks of each button to its own state

  const [good, setGood] = useState(0)

  const [neutral, setNeutral] = useState(0)

  const [bad, setBad] = useState(0)

  const handleGood = () => setGood(good+1)

  const handleNeutral = () => setNeutral(neutral+1)

  const handleBad = () => setBad(bad+1)

  return (

    <div>

      <h1>give feedback</h1>

      <button onClick={handleGood}>good</button>

      <button onClick={handleNeutral}>neutral</button>

      <button onClick={handleBad}>bad</button>

      <h1>statistics</h1>

      <p> good {good}</p>

      <p> neutral {neutral}</p>

      <p> bad {bad}</p>

      <p> all {good+neutral+bad}</p>

      <p> average {(good+neutral+bad)/3}</p>

      <p> positive {(good/(good+neutral+bad)\*100 + "%")}</p>

    </div>

  )

}

export default App

#### Exercise 1.8

Refactor your application so that displaying the statistics is extracted into its own *Statistics* component. The state of the application should remain in the *App* root component.

Remember that components should not be defined inside other components:

// a proper place to define a component

const Statistics = (props) => {

// ...

}

const App = () => {

const [good, setGood] = useState(0)

const [neutral, setNeutral] = useState(0)

const [bad, setBad] = useState(0)

// do not define a component within another component

const Statistics = (props) => {

// ...

}

return (

// ...

)

}

Answer:

import { useState } from 'react'

const Statistics = (props) => <p> {props.text} {props.feedback}</p>

const App = () => {

  // save clicks of each button to its own state

  const [good, setGood] = useState(0)

  const [neutral, setNeutral] = useState(0)

  const [bad, setBad] = useState(0)

  const handleGood = () => setGood(good+1)

  const handleNeutral =   () => setNeutral(neutral+1)

  const handleBad = () => setBad(bad+1)

  return (

    <div>

      <h1>give feedback</h1>

      <button onClick={handleGood}>good</button>

      <button onClick={handleNeutral}>neutral</button>

      <button onClick={handleBad}>bad</button>

      <h1>statistics</h1>

      <Statistics text="good" feedback = {good}></Statistics>

      <Statistics text="neutral" feedback = {neutral}></Statistics>

      <Statistics text="bad" feedback = {bad}></Statistics>

      <Statistics text="all" feedback = {good+neutral+bad}></Statistics>

      <Statistics text="average" feedback = {(good+neutral+bad)/3}></Statistics>

      <Statistics text="positive" feedback = {(good/(good+neutral+bad)\*100 + "%")}></Statistics>

    </div>

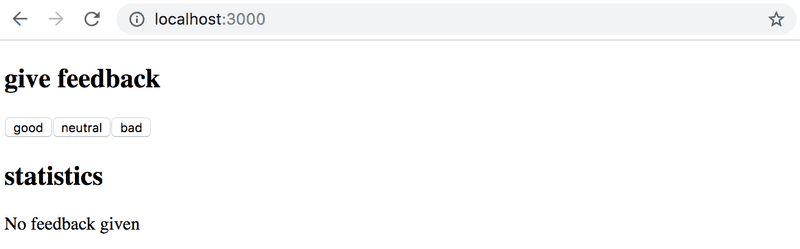
  )

}

export default App

#### Exercice 1.9

Change your application to display statistics only once feedback has been gathered.



Answer

I am here,the code is still incomplete, but it works

import { useState } from 'react'

const Statistics = (props) => {

  return(

    <>

      <p> {props.feedback.text[0]} {props.feedback.value[0]}</p>

      <p> {props.feedback.text[1]} {props.feedback.value[1]}</p>

      <p> {props.feedback.text[2]} {props.feedback.value[2]}</p>

      <p> {props.feedback.text[3]} {props.feedback.value[3]}</p>

      <p> {props.feedback.text[4]} {props.feedback.value[4]}</p>

      <p> {props.feedback.text[5]} {props.feedback.value[5]}</p>

    </>

  )

}

const App = () => {

  // save clicks of each button to its own state

  const [good, setGood] = useState(0)

  const [neutral, setNeutral] = useState(0)

  const [bad, setBad] = useState(0)

  const handleGood = () => setGood(good+1)

  const handleNeutral =   () => setNeutral(neutral+1)

  const handleBad = () => setBad(bad+1)

 const all = good+neutral+bad

 const average = all/3

 const positive = good/all

  const feedback = {

    text: ["good", "neutral", "bad", "all", "average", "positive"],

    value: [good, neutral, bad, all, average, positive]

  }

  return (

    <div>

      <h1>give feedback</h1>

      <button onClick={handleGood}>good</button>

      <button onClick={handleNeutral}>neutral</button>

      <button onClick={handleBad}>bad</button>

      <h1>statistics</h1>

      <Statistics feedback={feedback} ></Statistics>

      {/\* <Statistics text={feedback.text[1]} value={feedback.value[1]} ></Statistics>

      <Statistics text={feedback.text[2]} value={feedback.value[2]} ></Statistics>

      <Statistics text={feedback.text[3]} value={feedback.value[3]} ></Statistics>

      <Statistics text={feedback.text[4]} value={feedback.value[4]} ></Statistics>

      <Statistics text={feedback.text[5]} value={feedback.value[5]} ></Statistics> \*/}

    </div>

  )

}

export default App

#### Exercise 1.10

Let's continue refactoring the application. Extract the following two components:

* *Button* for defining the buttons used for submitting feedback
* *StatisticLine* for displaying a single statistic, e.g. the average score.

To be clear: the *StatisticLine* component always displays a single statistic, meaning that the application uses multiple components for rendering all of the statistics:

const Statistics = (props) => {

/// ...

return(

<div>

<StatisticLine text="good" value ={...} />

<StatisticLine text="neutral" value ={...} />

<StatisticLine text="bad" value ={...} />

// ...

</div>

)

}

The application's state should still be kept in the root *App* component.

Answer

import { useState } from 'react'

const StatisticLine = (props) => <p>{props.text} {props.value}</p>

const Statistics = (props) => {

  if (props.feedback.value[3] == 0) return (<p>No feedback given</p>)

  return(

    <>

      <StatisticLine text={props.feedback.text[0]} value={props.feedback.value[0]}/>

      <StatisticLine text={props.feedback.text[1]} value={props.feedback.value[1]}/>

      <StatisticLine text={props.feedback.text[2]} value={props.feedback.value[2]}/>

      <StatisticLine text={props.feedback.text[3]} value={props.feedback.value[3]}/>

      <StatisticLine text={props.feedback.text[4]} value={props.feedback.value[4]}/>

      <StatisticLine text={props.feedback.text[5]} value={props.feedback.value[5]}/>

    </>

  )

}

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

const App = () => {

  // save clicks of each button to its own state

  const [good, setGood] = useState(0)

  const [neutral, setNeutral] = useState(0)

  const [bad, setBad] = useState(0)

  const handleGood = () => setGood(good+1)

  const handleNeutral =   () => setNeutral(neutral+1)

  const handleBad = () => setBad(bad+1)

 const all = good+neutral+bad

 const average = all/3

 const positive = good/all

  const feedback = {

    text: ["good", "neutral", "bad", "all", "average", "positive"],

    value: [good, neutral, bad, all, average, positive]

  }

  return (

    <div>

      <h1>give feedback</h1>

      <Button text="Good" handleClick={handleGood}/>

      <Button text="Neutral" handleClick={handleNeutral}/>

      <Button text="Bad" handleClick={handleBad}/>

      <h1>statistics</h1>

      <Statistics feedback={feedback} ></Statistics>

    </div>

  )

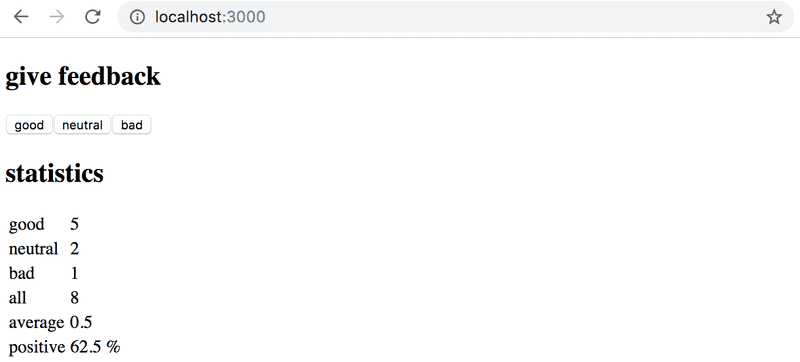
}

export default App

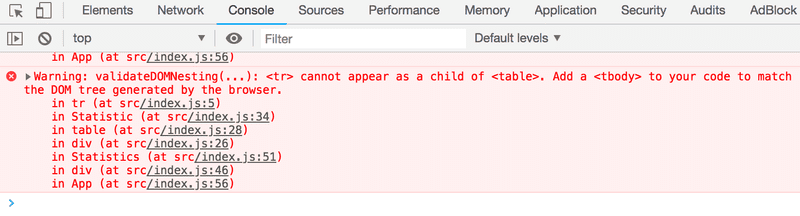
*Typical source of an error Unchecked runtime.lastError: Could not establish connection. Receiving end does nofreturnt exist.is Chrome extension. Try going to chrome://extensions/and try disabling them one by one and refreshing React app page; the error should eventually disappear.*

#### Exercise 1.11\*

Display the statistics in an HTML [table](https://developer.mozilla.org/en-US/docs/Learn/HTML/Tables/Basics), so that your application looks roughly like this:



Remember to keep your console open at all times. If you see this warning in your console:



Then perform the necessary actions to make the warning disappear. Try pasting the error message into a search engine if you get stuck.

*Typical source of an error Unchecked runtime.lastError: Could not establish connection. Receiving end does not exist.is Chrome extension. Try going to chrome://extensions/and try disabling them one by one and refreshing React app page; the error should eventually disappear.*

**Make sure that from now on you don't see any warnings in your console!**

Answer

This error

Warning: validateDOMNesting(...): <tr> cannot appear as a child of <table>. Add a <tbody>, <thead> or <tfoot> to your code to match the DOM tree generated by the browser.

Is solved by using tbody (see solution below)

The key here is to split the construction of the table between the components

import { useState } from 'react'

const StatisticLine = (props) => {

  return(

    <tr>

      <td> {props.text} </td>

      <td> {props.value} </td>

    </tr>

  )

}

const Statistics = (props) => {

  if (props.feedback.value[3] == 0) return (<p>No feedback given</p>)

  return(

    <>

      <table>

        <tbody>

           <StatisticLine text={props.feedback.text[0]} value={props.feedback.value[0]}/>

           <StatisticLine text={props.feedback.text[1]} value={props.feedback.value[1]}/>

           <StatisticLine text={props.feedback.text[2]} value={props.feedback.value[2]}/>

           <StatisticLine text={props.feedback.text[3]} value={props.feedback.value[3]}/>

           <StatisticLine text={props.feedback.text[4]} value={props.feedback.value[4]}/>

           <StatisticLine text={props.feedback.text[5]} value={props.feedback.value[5]}/>

        </tbody>

      </table>

    </>

  )

}

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

const App = () => {

  // save clicks of each button to its own state

  const [good, setGood] = useState(0)

  const [neutral, setNeutral] = useState(0)

  const [bad, setBad] = useState(0)

  const handleGood = () => setGood(good+1)

  const handleNeutral =   () => setNeutral(neutral+1)

  const handleBad = () => setBad(bad+1)

 const all = good+neutral+bad

 const average = Math.round(all\*100/3)/100

 const positive = Math.round((good/all)\*10000)/100 + "%"

  const feedback = {

    text: ["good", "neutral", "bad", "all", "average", "positive"],

    value: [good, neutral, bad, all, average, positive]

  }

  return (

    <div>

      <h1>give feedback</h1>

      <Button text="Good" handleClick={handleGood}/>

      <Button text="Neutral" handleClick={handleNeutral}/>

      <Button text="Bad" handleClick={handleBad}/>

      <h1>statistics</h1>

      <Statistics feedback={feedback} ></Statistics>

    </div>

  )

}

export default App

#### Exercise 1.12\*

The world of software engineering is filled with [anecdotes](http://www.comp.nus.edu.sg/~damithch/pages/SE-quotes.htm) that distill timeless truths from our field into short one-liners.

Expand the following application by adding a button that can be clicked to display a *random* anecdote from the field of software engineering:

import { useState } from 'react'

const App = () => {

const anecdotes = [

'If it hurts, do it more often.',

'Adding manpower to a late software project makes it later!',

'The first 90 percent of the code accounts for the first 10 percent of the development time...The remaining 10 percent of the code accounts for the other 90 percent of the development time.',

'Any fool can write code that a computer can understand. Good programmers write code that humans can understand.',

'Premature optimization is the root of all evil.',

'Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.',

'Programming without an extremely heavy use of console.log is same as if a doctor would refuse to use x-rays or blood tests when diagnosing patients.'

]

const [selected, setSelected] = useState(0)

return (

<div>

{anecdotes[selected]}

</div>

)

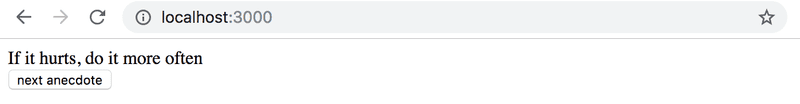
}

export default App

Content of the file *index.js* is same as in previous exercises.

Find out how to generate random numbers in JavaScript, eg. via search engine or on [Mozilla Developer Network](https://developer.mozilla.org/). Remember that you can test generating random numbers e.g. straight in the console of your browser.

Your finished application could look something like this:



**WARNING** create-react-app will automatically turn your project into a git-repository unless you create your application inside of an existing git repository. **Most likely you do not want each of your projects to be a separate repository**, so simply run the rm -rf .git command at the root of your application.

Answer

import { useState } from 'react'

const App = () => {

  const anecdotes = [

    'If it hurts, do it more often.',

    'Adding manpower to a late software project makes it later!',

    'The first 90 percent of the code accounts for the first 10 percent of the development time...The remaining 10 percent of the code accounts for the other 90 percent of the development time.',

    'Any fool can write code that a computer can understand. Good programmers write code that humans can understand.',

    'Premature optimization is the root of all evil.',

    'Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.',

    'Programming without an extremely heavy use of console.log is same as if a doctor would refuse to use x-rays or blood tests when diagnosing patients.'

  ]

  const [selected, setSelected] = useState(0)

  const handleCLick = () => setSelected(Math.floor(Math.random()\*anecdotes.length))

  return (

    <div>

      {anecdotes[selected]}

      <br></br>

      <button onClick={handleCLick} >next anecdote</button>

    </div>

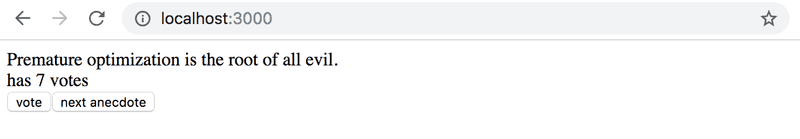
  )

}

export default App

#### Exercise 1.13

Expand your application so that you can vote for the displayed anecdote.



**NB** store the votes of each anecdote into an array or object in the component's state. Remember that the correct way of updating state stored in complex data structures like objects and arrays is to make a copy of the state.

You can create a copy of an object like this:

const points = { 0: 1, 1: 3, 2: 4, 3: 2 }

const copy = { ...points }

// increment the property 2 value by one

copy[2] += 1

OR a copy of an array like this:

const points = [1, 4, 6, 3]

const copy = [...points]

// increment the value in position 2 by one

copy[2] += 1

Using an array might be the simpler choice in this case. Searching the Internet will provide you with lots of hints on how to [create a zero-filled array of a desired length](https://stackoverflow.com/questions/20222501/how-to-create-a-zero-filled-javascript-array-of-arbitrary-length/22209781).

<https://stackoverflow.com/questions/20222501/how-to-create-a-zero-filled-javascript-array-of-arbitrary-length/22209781>

**NB** store the votes of each anecdote into an array or object in the component's state. Remember that the correct way of updating state stored in complex data structures like objects and arrays is to make a copy of the state.

You can create a copy of an object like this:

const points = { 0: 1, 1: 3, 2: 4, 3: 2 }

const copy = { ...points }

// increment the property 2 value by one

copy[2] += 1

OR a copy of an array like this:

const points = [1, 4, 6, 3]

const copy = [...points]

// increment the value in position 2 by one

copy[2] += 1

import { useState } from 'react'

const App = () => {

  const anecdotes = [

    'If it hurts, do it more often.',

    'Adding manpower to a late software project makes it later!',

    'The first 90 percent of the code accounts for the first 10 percent of the development time...The remaining 10 percent of the code accounts for the other 90 percent of the development time.',

    'Any fool can write code that a computer can understand. Good programmers write code that humans can understand.',

    'Premature optimization is the root of all evil.',

    'Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.',

    'Programming without an extremely heavy use of console.log is same as if a doctor would refuse to use x-rays or blood tests when diagnosing patients.'

  ]

  const [selected, setSelected] = useState(0)

  const [votes, setVotes] = useState(Array(anecdotes.length).fill(0))

  const handleNextAnecdote = () => {

    // Update first state

    const randomIndex = Math.floor(Math.random()\*anecdotes.length)

    setSelected(Math.floor(randomIndex))

  }

  const handleVote = () => {

    const newVotes = [...votes]

    newVotes[selected] += 1

    setVotes(newVotes)

  }

  return (

    <div>

      {anecdotes[selected]}

      <br></br>

      <p> has {votes[selected]} votes</p>

      <button onClick={handleVote}> vote</button>

      <button onClick={handleNextAnecdote} >next anecdote</button>

    </div>

  )

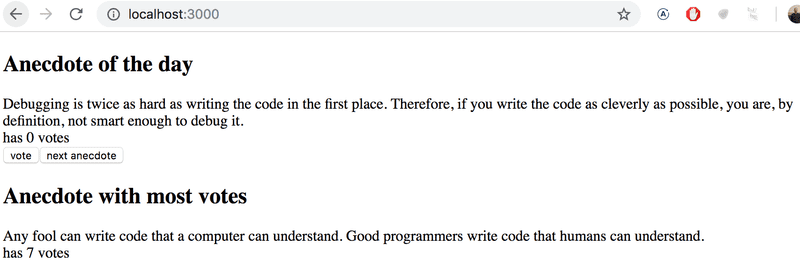
}

export default App

#### Exercise 1.14

.

Now implement the final version of the application that displays the anecdote with the largest number of votes:



If multiple anecdotes are tied for first place it is sufficient to just show one of them.

Answer:

Here is the code. I refactored the components

import { useState } from 'react'

const ShowAnecdote = (props) => {

  return (

    <>

      <h1> {props.title}</h1>

      <p>{props.anecdote}</p>

      <p>has {props.votes} votes </p>

    </>

  )

}

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

const App = () => {

  const anecdotes = [

    'If it hurts, do it more often.',

    'Adding manpower to a late software project makes it later!',

    'The first 90 percent of the code accounts for the first 10 percent of the development time...The remaining 10 percent of the code accounts for the other 90 percent of the development time.',

    'Any fool can write code that a computer can understand. Good programmers write code that humans can understand.',

    'Premature optimization is the root of all evil.',

    'Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.',

    'Programming without an extremely heavy use of console.log is same as if a doctor would refuse to use x-rays or blood tests when diagnosing patients.'

  ]

  const [selected, setSelected] = useState(0)

  const [votes, setVotes] = useState(Array(anecdotes.length).fill(0))

  const handleNextAnecdote = () => {

    // Update first state

    const randomIndex = Math.floor(Math.random()\*anecdotes.length)

    setSelected(Math.floor(randomIndex))

  }

  const handleVote = () => {

    const newVotes = [...votes]

    newVotes[selected] += 1

    setVotes(newVotes)

  }

  const maxAnecdoteIndex = votes.indexOf(Math.max(...votes))

  return (

    <div>

    <></>

      <ShowAnecdote title="Anecdote of the Day"  anecdote={anecdotes[selected]} votes={votes[selected]} />

      <Button text="Vote" onclick= {handleVote}></Button>

      <Button text="Next Anecdote" onclick= {handleNextAnecdote}></Button>

      <ShowAnecdote title="Anecdote with most votes"  anecdote={anecdotes[maxAnecdoteIndex]} votes={votes[maxAnecdoteIndex]} />

    </div>

  )

}

export default App

# Part 2 Communicating with server

## Work plan

* Number of sections : 25
* Number of exercices : 20
* Number of exercises I do in half a day: 5
* Number of days of work: 6
* Number of days spent on exercices (last 2 days): 20/(5\*2) = 2 days
* Number of days remaining for the sections: 4
* Number of sections per half day: 25/(4\*2)= 3.125 approx 4

[a Rendering a collection, modules](https://fullstackopen.com/en/part2/rendering_a_collection_modules)

* [console.log](https://fullstackopen.com/en/part2/rendering_a_collection_modules#console-log)
* [Protip: Visual Studio Code snippets](https://fullstackopen.com/en/part2/rendering_a_collection_modules#protip-visual-studio-code-snippets)
* [JavaScript Arrays](https://fullstackopen.com/en/part2/rendering_a_collection_modules#java-script-arrays)
* [Event Handlers Revisited](https://fullstackopen.com/en/part2/rendering_a_collection_modules#event-handlers-revisited)
* [Rendering Collections](https://fullstackopen.com/en/part2/rendering_a_collection_modules#rendering-collections)

* [Key-attribute](https://fullstackopen.com/en/part2/rendering_a_collection_modules" \l "key-attribute)
* [Map](https://fullstackopen.com/en/part2/rendering_a_collection_modules" \l "map)
* [Anti-pattern: Array Indexes as Keys](https://fullstackopen.com/en/part2/rendering_a_collection_modules#anti-pattern-array-indexes-as-keys)
* [Refactoring Modules](https://fullstackopen.com/en/part2/rendering_a_collection_modules#refactoring-modules)
* [When the Application Breaks](https://fullstackopen.com/en/part2/rendering_a_collection_modules#when-the-application-breaks)
* [Exercises 2.1.-2.5.](https://fullstackopen.com/en/part2/rendering_a_collection_modules#exercises-2-1-2-5)

[b Forms](https://fullstackopen.com/en/part2/forms)

* [Controlled component](https://fullstackopen.com/en/part2/forms" \l "controlled-component)
* [Filtering Displayed Elements](https://fullstackopen.com/en/part2/forms#filtering-displayed-elements)
* [Exercises 2.6.-2.10.](https://fullstackopen.com/en/part2/forms" \l "exercises-2-6-2-10)

[c Getting data from server](https://fullstackopen.com/en/part2/getting_data_from_server)

* [The browser as a runtime environment](https://fullstackopen.com/en/part2/getting_data_from_server" \l "the-browser-as-a-runtime-environment)
* [npm](https://fullstackopen.com/en/part2/getting_data_from_server#npm)
* [Axios and promises](https://fullstackopen.com/en/part2/getting_data_from_server#axios-and-promises)

* [Effect-hooks](https://fullstackopen.com/en/part2/getting_data_from_server" \l "effect-hooks)
* [The development runtime environment](https://fullstackopen.com/en/part2/getting_data_from_server#the-development-runtime-environment)
* [Exercises 2.11.-2.14.](https://fullstackopen.com/en/part2/getting_data_from_server" \l "exercises-2-11-2-14)

[d Altering data in server](https://fullstackopen.com/en/part2/altering_data_in_server)

* [REST](https://fullstackopen.com/en/part2/altering_data_in_server#rest)

* [Sending Data to the Server](https://fullstackopen.com/en/part2/altering_data_in_server" \l "sending-data-to-the-server)
* [Changing the Importance of Notes](https://fullstackopen.com/en/part2/altering_data_in_server#changing-the-importance-of-notes)

* [Extracting Communication with the Backend into a Separate Module](https://fullstackopen.com/en/part2/altering_data_in_server" \l "extracting-communication-with-the-backend-into-a-separate-module)
* [Cleaner Syntax for Defining Object Literals](https://fullstackopen.com/en/part2/altering_data_in_server#cleaner-syntax-for-defining-object-literals)
* [Promises and Errors](https://fullstackopen.com/en/part2/altering_data_in_server#promises-and-errors)
* [Exercises 2.15.-2.18.](https://fullstackopen.com/en/part2/altering_data_in_server" \l "exercises-2-15-2-18)

[e Adding styles to React app](https://fullstackopen.com/en/part2/adding_styles_to_react_app)

* [Improved error message](https://fullstackopen.com/en/part2/adding_styles_to_react_app" \l "improved-error-message)
* [Inline styles](https://fullstackopen.com/en/part2/adding_styles_to_react_app#inline-styles)
* [Exercises 2.19.-2.20.](https://fullstackopen.com/en/part2/adding_styles_to_react_app#exercises-2-19-2-20)

## Rendering a collection, modules

### A.1. console.log

Just a recap of the previous section

### A.2. [Important]Protip Visual Studio Code Snippets

Creating snippets with vscode

<https://code.visualstudio.com/docs/editor/userdefinedsnippets#_creating-your-own-snippets>

Useful, ready to use snippets can be found in vscode marketplace

<https://marketplace.visualstudio.com/items?itemName=dsznajder.es7-react-js-snippets>

Graphical user interface, text

Description automatically generated

You can download that extension above to get useful React snippets

Also in the marketplace you can find the most widely used snippets

<https://marketplace.visualstudio.com/search?term=console.log&target=VSCode&category=All%20categories&sortBy=Relevance>

in vscode, the console.log() snippet is built in ! just tap log and hit tab and you will get your concole.log. Otherwise you could build your own snippet

I follow the steps in the vscode website to create a snippet for console.log. You can build snippets for anything: for loops….

{

"console.log": {

"prefix": "clog",

"body": [

"console.log('$1')",

],

"description": "Log output to console"

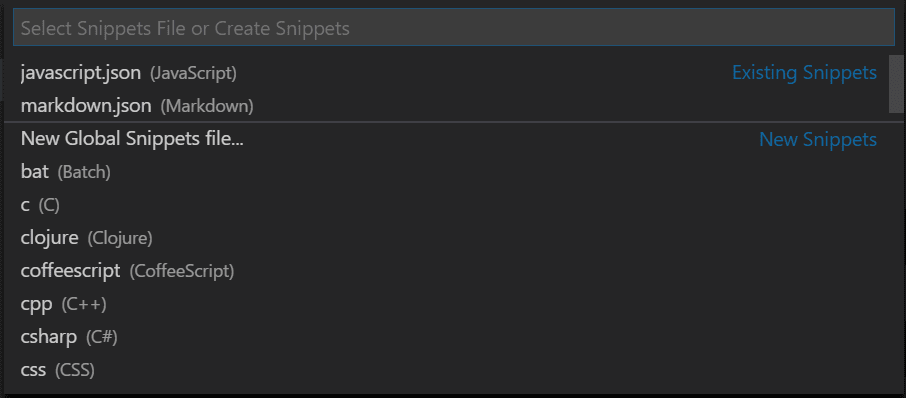
}

Here are the steps to build that snippet. They are explained here

<https://code.visualstudio.com/docs/editor/userdefinedsnippets#_creating-your-own-snippets>

[**Create your own snippets**](https://code.visualstudio.com/docs/editor/userdefinedsnippets#_create-your-own-snippets)

You can easily define your own snippets without any extension. To create or edit your own snippets, select **User Snippets** under **File** > **Preferences** (**Code** > **Preferences** on macOS), and then select the language (by [language identifier](https://code.visualstudio.com/docs/languages/identifiers)) for which the snippets should appear, or the **New Global Snippets file** option if they should appear for all languages. VS Code manages the creation and refreshing of the underlying snippets file(s) for you.



Snippets files are written in JSON, support C-style comments, and can define an unlimited number of snippets. Snippets support most TextMate syntax for dynamic behavior, intelligently format whitespace based on the insertion context, and allow easy multiline editing.

Below is an example of a for loop snippet for JavaScript:

// in file 'Code/User/snippets/javascript.json'

{

"For Loop": {

"prefix": ["for", "for-const"],

"body": ["for (const ${2:element} of ${1:array}) {", "\t$0", "}"],

"description": "A for loop."

}

}

In the example above:

* "For Loop" is the snippet name. It is displayed via IntelliSense if no description is provided.
* prefix defines one or more trigger words that display the snippet in IntelliSense. Substring matching is performed on prefixes, so in this case, "fc" could match "for-const".
* body is one or more lines of content, which will be joined as multiple lines upon insertion. Newlines and embedded tabs will be formatted according to the context in which the snippet is inserted.
* description is an optional description of the snippet displayed by IntelliSense.

Additionally, the body of the example above has three placeholders (listed in order of traversal): ${1:array}, ${2:element}, and $0. You can quickly jump to the next placeholder with Tab, at which point you may edit the placeholder or jump to the next one. The string after the colon : (if any) is the default text, for example element in ${2:element}. Placeholder traversal order is ascending by number, starting from one; zero is an optional special case that always comes last, and exits snippet mode with the cursor at the specified position.

For example next time in graphics programming I could create a snippet for empty example

Now I can type clog and I get console.log 😊

Also I create this snippet that displays a section comment

    "Comment section": {

        "prefix": "sec",

        "body": [

          "///////////////////////////////////////////////////////////",

          "// \*\*\*\* Section title here \*\*\*\* ",

          "///////////////////////////////////////////////////////////",

        ],

        "description": "Log output to console"

      }

### A.3. Javascript Arrays

MDN Javascript array

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>

You should watch a least the first three videos of this playlist

<https://www.youtube.com/playlist?list=PL0zVEGEvSaeEd9hlmCXrk5yUyqUag-n84>

Because we will do functional programming all the time

So let’s do it !

#### A.3.1. Higher order functions

##### Definition

Why use functional programming

* You make less bugs because code is clearer
* You can reuse your code more often.
* So you will be a much more efficient programmer

Functional programming is about using functions everywhere. For example, instead of using a for loop, you use a function for it as in the forEach function in ES6. Or instead of filtering an array using a for loop and an if statement, you use a filter function for it. Or instead of transforming elements of an array you use the map() or reduce() functions that we will see in this section.

Functional programming relies on the composition of functions, f o g.

In Javascript,

* function f is called the **higher order function**
* function g is called **the callback function**

##### In Javascript Functions are values

///////////////////////////////////////////////////////////

// \*\*\*\* Function are values \*\*\*\*

///////////////////////////////////////////////////////////

// In any program we define a function like this

function triple(x) {

    return x \* 3

}

//  In javavascript we can crate an anonymous function and

// assign it to a variable. because, in Javascript, functions

// are values

var triple = function(x) {

    return x \* 3

}

// Because it is a value we can pass it to other variables

var waffle = triple;

console.log(waffle(30));

Note: Structured JS file

///////////////////////////////////////////////////////////

// \*\*\*\* HELPER FUNCTIONS \*\*\*\*

// DO NOT WRITE BELOW THIS LINE

///////////////////////////////////////////////////////////

##### Therefore we can implement function composition: filter()

///////////////////////////////////////////////////////////

// \*\*\*\* Higher order functions \*\*\*\*

///////////////////////////////////////////////////////////

var animals = [

    {name: 'fdsfsdfsfd', species: 'rabbit'},

    {name: 'harold',     species: 'rabbit'},

    {name: 'Helena',     species: 'dog'},

    {name: 'Ursula',     species: 'fish'},

    {name: 'Leyy',       species: 'cat'},

    {name: 'kelly',      species: 'dog'},

]

// our objective is to filter the array to leave only the dogs

//  The old way with a foor loop

var dogs = []

for (var i=0; i<animals.length; i++) {

    if (animals[i].species == 'dog') {

        dogs.push(animals[i]);

    }

}

console.log("for loop dog list is", dogs);

// The functional programming way using filter function

// the filter() is the higher order function. (x) => x.species == 'dog' is the callback functio

var dog2 = animals.filter((x) => x.species == 'dog');

console.log("filtered dog list is", dog2);

// You can pass the callback function to a variable, and pass that variable to the higher order function

var isDog = (x) => x.species == 'dog'

var otherAnimals = animals.filter(isDog);

console.log('The other animals are', otherAnimals)

#### A.3.2. map()

Using the object “animal” defined above use map() to produce an array that contains the animal names . note that map can return anything

///////////////////////////////////////////////////////////

// \*\*\*\* Higher order functions: map() \*\*\*\*

///////////////////////////////////////////////////////////

// The objective is to get an array of strings containing the names of the animals

// The old way

var names = [];

for (var i=0; i<animals.length; i++) {

    names.push(animals[i].name);

}

console.log('The old names are', names);

// Using functional programming

var names2 = animals.map((x) => x.name);

console.log('The functional names are', names2);

#### A.3.3. reduce()

##### Recap: filter vs map

Higher order functions (map, filter, reduce) transform a list into something else

Map transforms the array into an array of the same length but with each element of the array transformed



Filter transforms an array into a smaller array

Diagram

Description automatically generated

Find does the same thing as filter but returns onlythe first item



##### Reduce

Reduce is a more general function than map, filter or find. In fact, it can implement reduce

You use reduce if you cannot find a pre-built higher order function that transforms your list

The higher order function, reduce(), takes two arguments

* a callback function
* an initial value to the result

The callback function takes two arguments

* the result variable
* The input variable that is transformed

The callback function defines a recursive relationship on which the function is based. Because remember any for loop can be expressed as a recursion. An callback functions are for loops. They iterate over a list elements

In the code below think of x as each element of the orders list

///////////////////////////////////////////////////////////

// \*\*\*\* Higher order functions: reduce() \*\*\*\*

///////////////////////////////////////////////////////////

var orders = [

    {amount: 250},

    {amount: 400},

    {amount: 850},

    {amount: 325}

]

// The objective is to compute the sum of orders

// The olde way

var sum = 0;

for (var i=0; i<orders.length; i++) {

    sum += orders[i].amount;

}

console.log('The sum is', sum);

// The functional way

var sum2 = orders.reduce((sum, x) => sum+x.amount ,0);

console.log('The functional sum is', sum2);

### A.4. Event handlers revisited

This is just telling you to review the section “Event handling revisited” from part 1

### A.5. Rendering collections

Starting from this

Index.js:

import React from 'react'

import ReactDOM from 'react-dom/client'

import App from './App'

const notes = [

  {

    id: 1,

    content: 'HTML is easy',

    date: '2019-05-30T17:30:31.098Z',

    important: true

  },

  {

    id: 2,

    content: 'Browser can execute only JavaScript',

    date: '2019-05-30T18:39:34.091Z',

    important: false

  },

  {

    id: 3,

    content: 'GET and POST are the most important methods of HTTP protocol',

    date: '2019-05-30T19:20:14.298Z',

    important: true

  }

]

ReactDOM.createRoot(document.getElementById('root')).render(

  <App notes={notes} />

)

Apps.js:

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

we use the map function to generate the list

Here you go

return (

    <div>

      <h1>Notes</h1>

      <ul>

        {notes.map(x => <li> {x.content} </li> )}

      </ul>

    </div>

  )

Note that you can make the code more readable like this

const App = (props) => {

  const { notes } = props

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {notes.map(x =>

          <li>

            {x.content}

          </li> )}

      </ul>

    </div>

  )

}

export default App

Note that x is an iterator an is an object literal, because the list’s elements are object literals

### A.6. Key-attribute

We have this error message

Graphical user interface

Description automatically generated

The documentation for it is here. Not clear to me

Graphical user interface

Description automatically generated

You just need to follow the error messahe. Each child (each li) should have a an attribute that identifies it uniquely. This is why we defined an id attribute in index.js

This drove me crazy But I solved it.

It is important to solve this error as it otherwise trickles down later in your code

The made mistake concerns this principle:

* Keys only make sense in the context of the surrounding array. The error I made is actually explained here
* A good rule of thumb is that elements inside the map() call need keys. Do not refactor keys !!

<https://reactjs.org/docs/lists-and-keys.html>

I made the exact error that is listed in the documentation

Graphical user interface, text

Description automatically generated

#### The wrong solution

This is wrong because we refactored the key outside of the map

const Notes = (props) => {

  // {console.log(props)}

  return (

    <>

      <li key={props.uniqueKey}> {props.content}</li>

    </>

  )

}

const App = (props) => {

  const { notes } = props

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {notes.map(x =>

          <Notes uniquekey={x.id} content={x.content}/>

          )

        }

      </ul>

    </div>

  )

}

export default App

#### The right solution

Key is kept inside map

const Notes = (props) => {

  // {console.log(props)}

  return (

    <>

      <li> {props.content}</li>

    </>

  )

}

const App = (props) => {

  const { notes } = props

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {notes.map(x =>

          <Notes key={x.id} content={x.content}/>

          )

        }

      </ul>

    </div>

  )

}

export default App

### A.7. Map

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map>

Q. What happens when you forget the curly braces in JSX ?

- it either prints the javascript formula (not the result) on the page

- Or it creates an explicit error

JSX value should be either an expression or a quoted JSX text.

### A.8. Anti pattern: Array indexes as keys

#### A.8.1 Using the index of an item as the key…

First let’s note that the map function can take up to 3 arguments:

// Arrow function

map((element) => { /\* … \*/ })

map((element, index) => { /\* … \*/ })

map((element, index, array) => { /\* … \*/ })

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map>

In particular the second argument allows you to use the index of the item

Therefore, one can use the index of an item as the key.

We will show how to do it. It works. However it is not recommended. We will see why

Note that the following method is not recommended: use the *index* of an item as its *key* when they render a list.

Another way is to pass as key value, the index of the iteration i. This index can be obtained as the second argument to map

const App = (props) => {

  const { notes } = props

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {**notes.map((x,i) =>**

**<li key={i}>**

            {x.content}

          </li> )}

      </ul>

    </div>

  )

}

export default App

#### A.8.2 … Is not a good idea

This article explains why it is not a good idea

<https://robinpokorny.medium.com/index-as-a-key-is-an-anti-pattern-e0349aece318>

The reason is this:

a *key* is the only thing React uses to identify DOM elements. What happens if you push an item to the list or remove something in the middle? If the *key* is same as before React assumes that the DOM element represents the same component as before. But that is no longer true.

Graphical user interface, text, application

Description automatically generated

The solution is

Each such item should have a *permanent* and *unique* property. Ideally, it should be assigned when the item is created. Of course, I am speaking about an *id*.

Condition under which you could use index as key

To help you decide, I put together three conditions which these examples have in common:

* the list and items are static–they are not computed and do not change;
* the items in the list have no ids;
* the list is *never* reordered or filtered.

When *all* of them are met, you **may safely use the index as a key**.

### A.9. [IMPORTANT]Refactoring Modules

#### A.9.1 .Refactoring and using a component inside map

Let’s create a Note component. It s very important that you try it yourself, not just copy the course.I present first how I do it myself, and thenimprovements from the course

const Note = (props) => {

  return (

    <>

      <h1>Notes</h1>

      <ul>

        {props.note.map(x =>

            <li key={x.id}>

              {x.content}

            </li> )}

      </ul>

    </>

  )

}

const App = (props) => {

  const { notes } = props

  return (

    <div>

      <Note note={notes} />

    </div>

  )

}

export default App

So: It does not make sense to use map inside the Note component.You want to control how many notes you write inside the app component. Hence keep the map inside the App component

So, here is a better solution

const Note = (props) => {

  // {console.log(props)}

  return (

    <>

      <li> {props.content}</li>

    </>

  )

}

const App = (props) => {

  const { notes } = props

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {notes.map(x =>

          <Note key={x.id} content={x.content}/>

          )

        }

      </ul>

    </div>

  )

}

export default App

Especially when props has one field, you can use destructuring as we saw in part 1

const Note = ({content}) => {

  // {console.log(props)}

  return (

    <>

      <li> {content}</li>

    </>

  )

}

const App = (props) => {

  const { notes } = props

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {notes.map(x =>

          <Note key={x.id} content={x.content}/>

          )

        }

      </ul>

    </div>

  )

}

export default App

#### A.9.2 .Putting all components inside a separate ditectory/module file

You create a directory ‘components’ inside src

You create a Note.js file

You put the component code declaration inside it.

Do not forget to add the export line

const Note = ({content}) => {

  // {console.log(props)}

  return (

    <>

      <li> {content}</li>

    </>

  )

}

export default Note

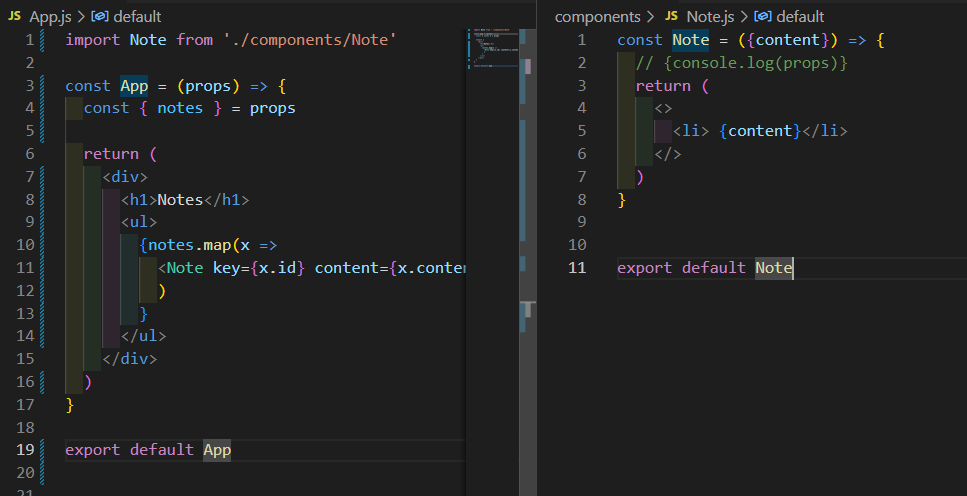
That Note name in export, you use in the app file in the import

import Note from './components/Note'

Shape, rectangle

Description automatically generated

You can split vscode to see all modules like this



### A.10. When the application breaks

When the application breaks you should do a console.log

* First do console.log in the App() component to see if it works.
* To do a console.log inside a component that is a one-liner, you must expand the component using the return statement

So in this code, you cannot add a console.log because there are no {} after the arrow.

const Course = ({ course }) => (

  <div>

    <Header course={course} />

  </div>

)

Must be turned into this

const Course = ({ course }) => {

  console.log(course)

  return (

    <div>

      <Header course={course} />

    </div>

  )

}

So remember

* To have multiple expressions inside a component, you need to follow the arow by { }
* To have multiple elements inside the return, you need to enclose the elements inside div tags

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 are, and destructuring fails as a result.

The problem often begins to solve itself when destructuring is removed and we see what the props actually contains.

This is even more 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.

const Course = (props) => {

  console.log(props)

  const { course } = props

  return (

    <div>

      <Header course={course} />

    </div>

  )

}

## Forms

React handling events documentation

<https://reactjs.org/docs/handling-events.html>

Default events that we prevent upon submitting the button . Among those events is that the page reloads

<https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/submit_event>

First let’s add a form with an handler that (prevents default behavior and) prints to the console the target DOM element that is clicked

Note that if you did not solve the unique keys problems beforehand correctly, this will not work 😊

import Note from './components/Note'

const App = (props) => {

  const { notes } = props

  const handleSubmit = (event) => {

    event.preventDefault()

    console.log(event.target)

  }

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {notes.map(x =>

          <Note key={x.id} content={x.content}/>

          )

        }

      </ul>

      <form onSubmit={handleSubmit}>

        <input type="text"/>

        <button type="submit">Submit</button>

      </form>

    </div>

  )

}

export default App

Graphical user interface, text, application, Word

Description automatically generated

Note

event.preventDefault() method, which prevents the default action of submitting a form. The default action would, [among other things](https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/submit_event), cause the page to reload.

### B.11. Controlled Components

React documentation for controlled components

<https://reactjs.org/docs/forms.html#controlled-components>

Now we want to create two states

A state notes that contains all the notes. Each time the form is submitted, we want to add whatever the user writes in the variable of that state

A state newNote that contains whatever the user types before it is submitted. The state variable newNote will be concatenated to the state variable note upon submission

Note that event handler for input does not require event.preventDefault()

The value attribute to the input will be the newNote variable. When the user writes something, he will see nothing. This is because the app component **controls** the behavior of the input element, since the value of the input element is set to the state variable newNote.

Therefore, for the user to see the text that he types, we must add an onChange handler to the input tag. That onChange updates the value of newNote to the target value.

So first, we create the two states and assign newNote as value to the input. We note that the user cannot type into the form. The error tells us what to do next

#### Step 1: Make the app control the value attribute of the input tag

import {useState} from 'react'

import Note from './components/Note'

const App = (props) => {

  const { notes } = props

  const [note, setNote] = useState(notes)

  const [newNote, setNewNote] = useState('...type here')

  const handleSubmit = (event) => {

    event.preventDefault()

    console.log(event.target)

  }

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {notes.map(x =>

          <Note key={x.id} content={x.content}/>

          )

        }

      </ul>

      <form onSubmit={handleSubmit}>

        <input value={newNote} type="text"/>

        <button type="submit">Submit</button>

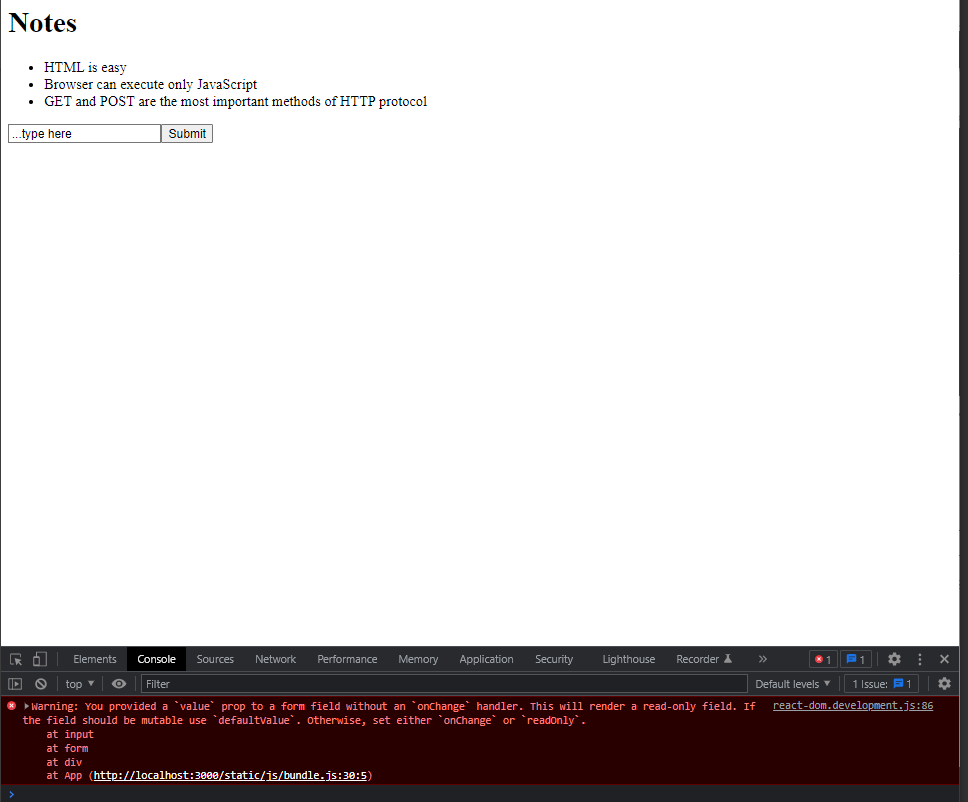
      </form>

    </div>

  )

}

export default App



#### Step 2: Update the state using an event handler in the input tag as the user types

So the error tells us to create an onChange handler in the input that will update the value of the input. Note that, unlike when sumbmitting a form, for an input we do not need to preventdefault

So we do this

import {useState} from 'react'

import Note from './components/Note'

const App = (props) => {

  const { notes } = props

  const [note, setNote] = useState(notes)

  const [newNote, setNewNote] = useState('...type here')

**const handleSubmit = (event) => {**

**event.preventDefault()**

**console.log(event.target)**

**}**

  const changeInput = (event) => {

    console.log(event.target.value)

    setNewNote(event.target.value)

  }

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {notes.map(x =>

          <Note key={x.id} content={x.content}/>

          )

        }

      </ul>

      <form onSubmit={handleSubmit}>

**<input value={newNote} onChange={changeInput} type="text"/>**

        <button type="submit">Submit</button>

      </form>

    </div>

  )

}

export default App

Now as we type inside the form, we can see the state changing live

Text

Description automatically generated

#### Step 3: When user input is submitted, store it to the note variable

We do this in handleSubmit

We create a variable newWrittenNote that follos the pattern of the notes. Remember in index.js the notes are like this

const notes = [

  {

    id: 1,

    content: 'HTML is easy',

    date: '2019-05-30T17:30:31.098Z',

    important: true

  },

  {

    id: 2,

    content: 'Browser can execute only JavaScript',

    date: '2019-05-30T18:39:34.091Z',

    important: false

  },

  {

    id: 3,

    content: 'GET and POST are the most important methods of HTTP protocol',

    date: '2019-05-30T19:20:14.298Z',

    important: true

  }

]

So we create that variable in the same pattern

We ensure that the id is unique using the length of the list . We add the current date. And we inputa random Boolean for important

Finally we concatenate the submitted note to the state note variable

And finally after submission we empty the form

const handleSubmit = (event) => {

    event.preventDefault()

    console.log(event.target)

    const newWrittenNote =  {

      id: note.length+1,

      content: newNote,

      date: new Date().toISOString(),

      important: Math.random()<0.5

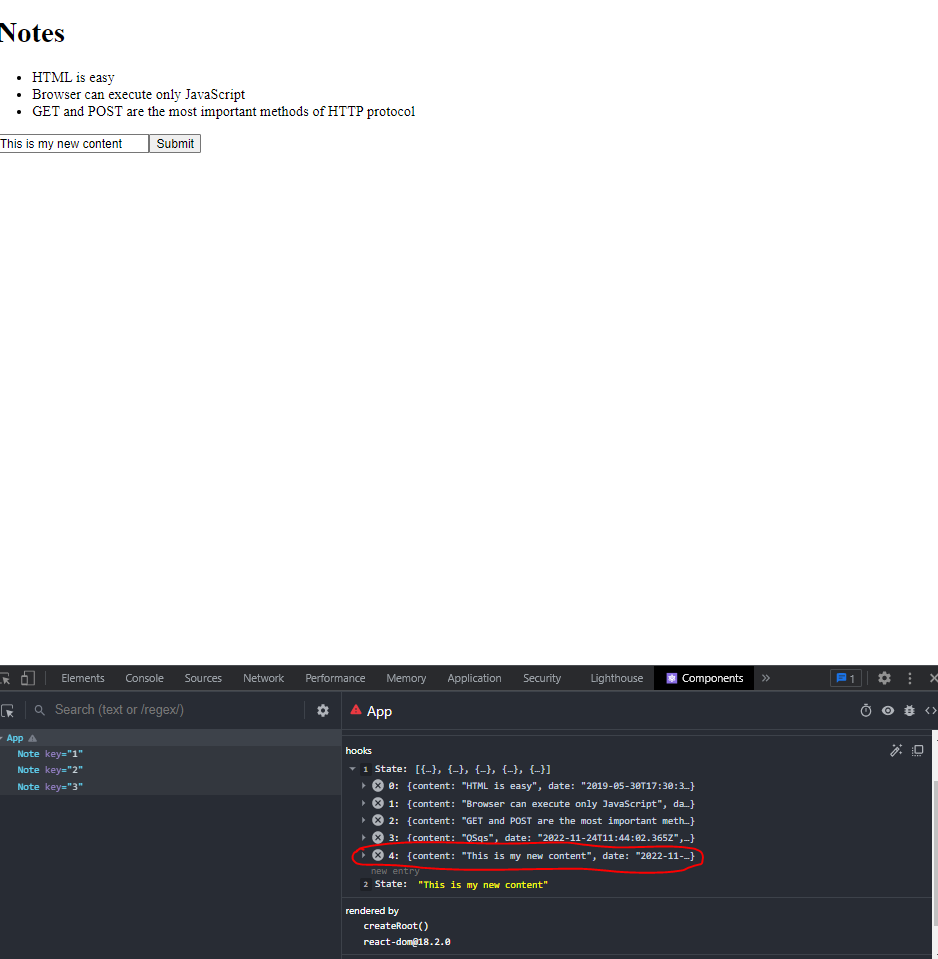
    }

    setNote(note.concat(newWrittenNote))

setNewNote('')

  }

Now when we submit the content we see the corresponding state update



### B.12. Filtering Displayed elements

#### Aside 1: Difference between == and ===

Equality comparison and sameness

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness>

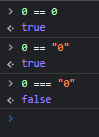
There is difference between == and ===

== is loose equality. It perfoams a type conversion when comparing two things

For example, 0 == “0” yiels true

=== is strict equality. It does not do type conversion. Therefore 0 ===”0” yield false.

Trying this on the console



#### Aside 2: Conditional ternary operator

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional\_Operator

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 ofval2.

Now for what we need to do in this section

So now we want to create a button. When you click the button

* The name of the button changes : it toggle between “show all” and “show important”
* The list of displayed note changes.
  + If the button is “show all” then it shows all notes
  + If the button is “show important” then it shows only the important notes

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

#### Step 1: Create a Boolean state and a variable whose value depends conditionally on that state

 const [showAll, setShowAll] = useState(true)

 const showNotes = showAll ? note : note.filter((x) => x.important)

check that this step works by

* Console.log(showNotes)
* Changing this line to false and checking that in the console.log the non important note is removed

const [showAll, setShowAll] = useState(false)

#### Step 2: Create a Toggle button that switches the showAll state variable and whose name is conditional

   <button onClick={() => setShowAll(!showAll)}> {showAll ? 'All Notes' : 'Important Notes'} </button>

So here is thew whole code

import {useState} from 'react'

import Note from './components/Note'

const App = (props) => {

  const { notes } = props

  const [note, setNote] = useState(notes)

  const [newNote, setNewNote] = useState('...type here')

  const [showAll, setShowAll] = useState(true)

  const showNotes = showAll ? note : note.filter((x) => x.important)

  const handleSubmit = (event) => {

    event.preventDefault()

    console.log(event.target)

    const newWrittenNote =  {

      id: note.length+1,

      content: newNote,

      date: new Date().toISOString(),

      important: Math.random()<0.5

    }

    setNote(note.concat(newWrittenNote))

    setNewNote('')

  }

  const changeInput = (event) => {

    console.log(event.target.value)

    setNewNote(event.target.value)

  }

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {showNotes.map(x =>

          <Note key={x.id} content={x.content}/>

          )

        }

      </ul>

      <form onSubmit={handleSubmit}>

        <input value={newNote} onChange={changeInput} type="text"/>

        <button type="submit">Submit</button>

      </form>

      <button onClick={() => setShowAll(!showAll)}> {showAll ? 'All Notes' : 'Important Notes'} </button>

    </div>

  )

}

export default App

## Getting Data from the server

* To run the JSON server: npm run server

Now we do back end work. Part 3 will cover back end in detail. This is an introduction

Here we will use JSON server. JSON server is a handy tool that enables us to use server side functionality is the development phase without the need to program it. We will se how to program it in detail in part 3 of the course

JSON server github repository is here

JSON server

<https://github.com/typicode/json-server>

* **First step: Install JSON server**

The first step is to install JSON server. Installation steps are easy but they are explained in the gitbub repo here JSON server

<https://github.com/typicode/json-server#getting-started>

There are two options:

* If you have all admin provileges: You can install JSON server globally on your machine using npm install -g json-server
* If you don’t, then in the root directry of your app do : npx json-server --port 3001 --watch db.json
* **Step 2: Create the JSON file that will sit on the JSON server**

Put the file in the root directory of the app. That is in the parent folder to src, here

Graphical user interface

Description automatically generated with medium confidence

The JSON file contains the notes objects that we had inside index.js

{

    "notes": [

      {

        "id": 1,

        "content": "HTML is easy",

        "date": "2022-1-17T17:30:31.098Z",

        "important": true

      },

      {

        "id": 2,

        "content": "Browser can execute only JavaScript",

        "date": "2022-1-17T18:39:34.091Z",

        "important": false

      },

      {

        "id": 3,

        "content": "GET and POST are the most important methods of HTTP protocol",

        "date": "2022-1-17T19:20:14.298Z",

        "important": true

      }

    ]

  }

* **Step 3: Run the server**

Now we run the server. We put the file db.json into the server. By default the JSON server runs on port 3000. But this port is already taken by REACT ! Therefore we run it on port 3001

 json-server --port 3001 --watch db.json

When you open the file in chrome in the location printed by the console you can see the JSON file in the browser, beautifully formatted using JSON Vue

<http://localhost:3001/notes>

Text

Description automatically generated

Text

Description automatically generated

Remaining sections

Getting data from server

Il reste 13 sections

* [The browser as a runtime environment](https://fullstackopen.com/en/part2/getting_data_from_server#the-browser-as-a-runtime-environment)
* [npm](https://fullstackopen.com/en/part2/getting_data_from_server#npm)
* [Axios and promises](https://fullstackopen.com/en/part2/getting_data_from_server#axios-and-promises)
* [Effect-hooks](https://fullstackopen.com/en/part2/getting_data_from_server#effect-hooks)

* [The development runtime environment](https://fullstackopen.com/en/part2/getting_data_from_server" \l "the-development-runtime-environment)
* [REST](https://fullstackopen.com/en/part2/altering_data_in_server#rest)
* [Sending Data to the Server](https://fullstackopen.com/en/part2/altering_data_in_server#sending-data-to-the-server)
* [Changing the Importance of Notes](https://fullstackopen.com/en/part2/altering_data_in_server#changing-the-importance-of-notes)

* [Extracting Communication with the Backend into a Separate Module](https://fullstackopen.com/en/part2/altering_data_in_server" \l "extracting-communication-with-the-backend-into-a-separate-module)
* [Cleaner Syntax for Defining Object Literals](https://fullstackopen.com/en/part2/altering_data_in_server#cleaner-syntax-for-defining-object-literals)
* [Promises and Errors](https://fullstackopen.com/en/part2/altering_data_in_server#promises-and-errors)
* [Improved error message](https://fullstackopen.com/en/part2/adding_styles_to_react_app#improved-error-message)
* [Inline styles](https://fullstackopen.com/en/part2/adding_styles_to_react_app#inline-styles)

### B.13. The browser as a runtime environment

Javascript asynchronous model

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop>

Javascript promises

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>

Javascript fetch

<https://developer.mozilla.org/en-US/docs/Web/API/fetch>

JavaScript engines, or runtime environments, follow the [**asynchronous model**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop). In principle, this requires all [IO-operations](https://en.wikipedia.org/wiki/Input/output) (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.

Text

Description automatically generated

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?](https://www.youtube.com/watch?v=8aGhZQkoFbQ)

In today's browsers, it is possible to run parallelized code with the help of so-called  [web workers](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers). The event loop of an individual browser window is, however, still only handled by a [single thread](https://medium.com/techtrument/multithreading-javascript-46156179cf9a)..

Here is the keynote talk

<https://www.youtube.com/watch?v=8aGhZQkoFbQ>

### B.14. npm

Npm documentation

<https://docs.npmjs.com/about-npm>

Axios

<https://github.com/axios/axios>

Instead of the fetch command, we use the axios command to manage the relationship between the browser and the server

#### Installing axios, a runtime dependency

We load axios using the npm.

In the package.json file, before installing axios, it does not show as a dependency

Text

Description automatically generated

Now let’s install the axios dependency using npm (nodes package manager)

In The command line

npm install axios

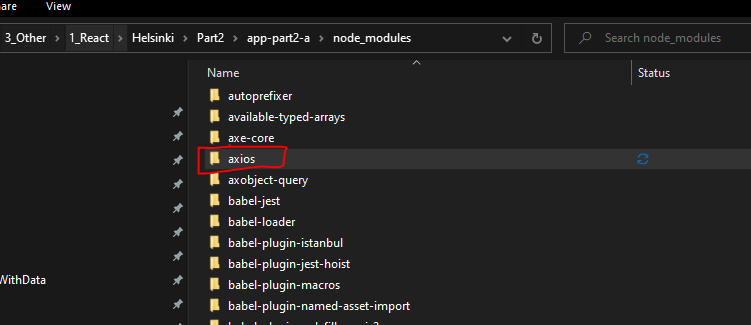
Then the package.json updates to show us this

Text

Description automatically generated

NB npm-commands should always be run in the project root directory, which is where the package.json file can be found.

Also, note that the comman *npm install* downloaded the code of the axios library in the folder nodes\_modules



Axios is a runtime dependency. This means that the execution of the program requires the existence of the library.

#### Installing json-server , as a development dependency

Previously we ran json-server as follows, using a long set of parameters

 json-server --port 3001 --watch db.json

We can, instead, import the json-server dependency as a development library. This means that the program itself doesn't require it. It is used for assistance during software development.

npm install json-server --save-dev

The –save-dev tag tells that this is a development dependency

Note that in json the development dependencies are shown in a specific place. This adds automatically after install

Text

Description automatically generated

Then in packages.json, we manually add a line that allows us to run the json-server without parameters, as follows

Graphical user interface, text

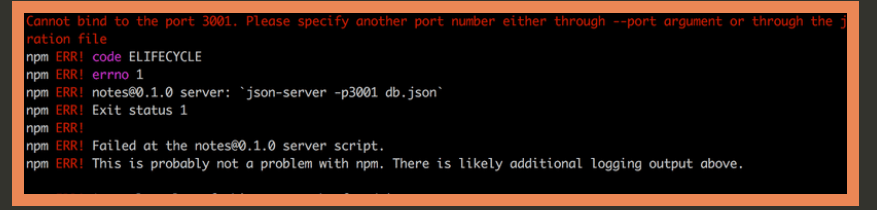
Description automatically generated with medium confidence

So the long code that we types in the command line before is now stored here

Next time when you want to run the server, simply write in the command line

* First if an instance of json-server is opened, close it because the port 3001 is taken. To do so go to the command line window and hit CTRL+C which means EXIT

If you don’t you will get this error



* Then run this command

**npm run server**

### B.15. Axios and promises

#### An axios get request returns a promise, ie an asynchronous reponse

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 react-app.

**Note:** when the content of the file *index.js* changes, React does not always notice that automatically, so you might need to refresh the browser to see your changes! A simple workaround to make React notice the change automatically, is to create a file named *.env* in the root directory of the project and add this line *FAST\_REFRESH=false*. Restart the app for the applied changes to take effect.

We add these lines to the index.js file of the app (after doing npm start on

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:

A promise can have three distinct states:

* 1. The promise is pending: It means that the final value (one of the following two) is not available yet.
  2. 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.
  3. The promise is rejected: It means that an error prevented the final value from being determined, which generally represents a failed operation.

In index.js we perform two axios get requests. That is we add the lines below

const promise = axios.get('http://localhost:3001/notes')

console.log(promise)

const promise2 = axios.get('http://localhost:3001/foobar')

console.log(promise2)

In chrome console we see this

Graphical user interface

Description automatically generated with low confidence

We note first the axio’s get method returns a promise

We note also that the first request is accepted, while the second request is rejected. The red error message tells us the reason. The status code is 404. A google search (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404?retiredLocale=de#:~:text=The%20HTTP%20404%20Not%20Found,absence%20is%20temporary%20or%20permanent>.)

Tells us that 404 is not found, which means that the server cannot find the requested resource

So we were trying to make an HTTP GET request to a non-existent address.

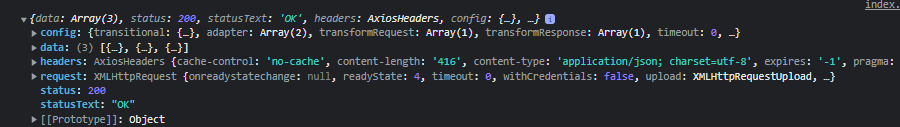
Assume you want to print the response. To do that you must register an event handler to the promise using the method then

promise.then(response => {

  console.log(response)

})

Here the event is that the browser receives a response to an HTTP request. Then the JS runtime environment calls the callback function registered by the then method. We print the response object that contains all the elements of the response.



#### Accessing the content of the response

Note that what we write in the JSON file is now contained in response.data

Note also that in practice we do not store the response in a variable. We chain the reponses like this

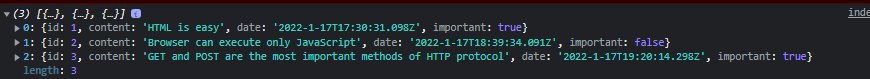
axios.get('http://localhost:3001/notes').then(response => {

  const notes = response.data

  console.log(notes)

})

And so here is your ever so familiar data



Practitionners make such chaining more readable using this syntax (going to the next line after each dot)

axios

.get('http://localhost:3001/notes')

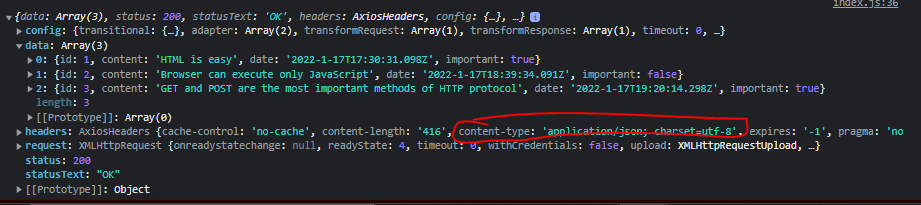
.then(response => {

  const notes = response.data

  console.log(notes)

})

If you remember from programming with data course, the data returned by the server is just one a long string. So the axios library parses that string into a Javascript array. To be able to do that axios must know the data type of the data in the server. It gets that answer from the content type as we see in here when printing the whole reponse



### B.16. Effect-hooks

#### The bad way of using the data fetched from the server into a component: rendering the component inside the fetch handler

The bad way is changing the index.js file like this

A picture containing chart

Description automatically generated

So index.js looks like this

import React from 'react'

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

  console.log(response)

  ReactDOM.createRoot(document.getElementById('root')).render(

    <App notes={notes} />

  )

})

When we do this, all seems to work perfectly. The app runs

The obvious problem with this approach is that we're rendering the entire App component only when we successfully retrieve a response. Because the rendering (.render method) is done inside an event handler. Read the axios got as it: get the response, then render

Thinking about this more broadly, we have two options

* Rendering the app component inside the fetching statement (not good)
* Fetching the data inside the app component

Let’s try the second option.

#### The good way of using the data fetched from the server into a component: Using state hooks

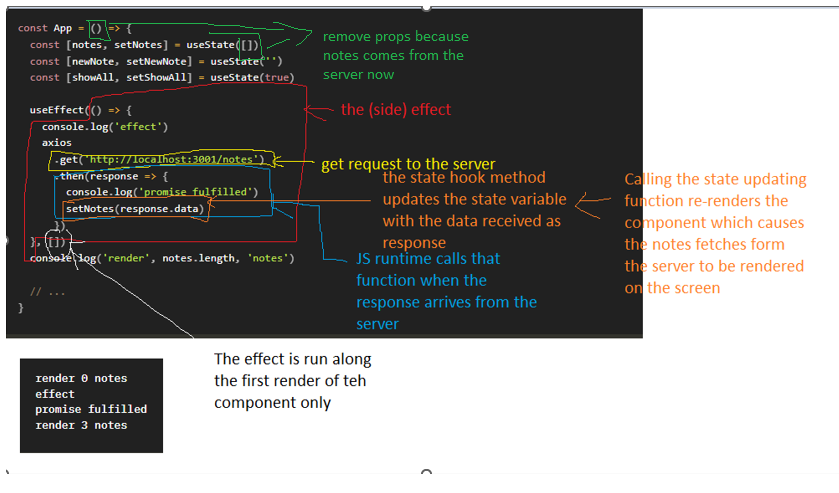
Effect hooks have been introduced in version 16.8.0

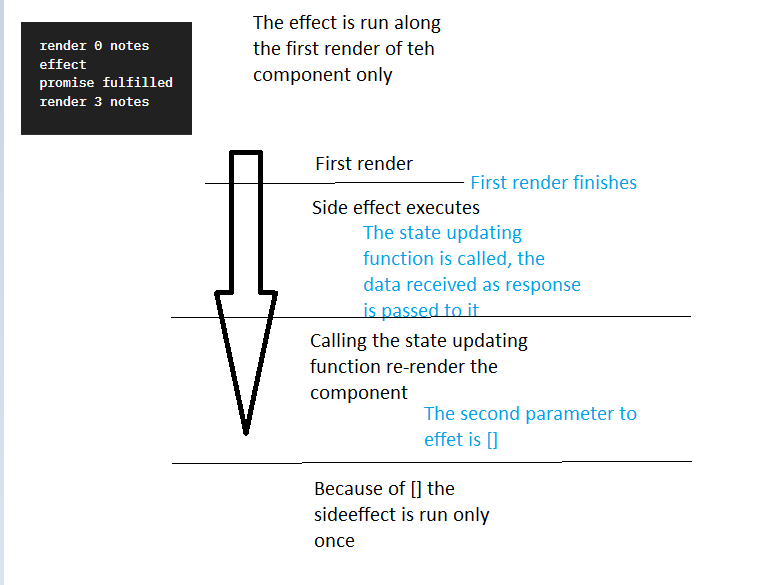
The Effect Hook lets you perform **side effects** on function components. Side effects are things that are not core to what the component does. An example of side effect is fetching data from a server

As such, effect hooks are precisely the right tool to use when fetching data from a server.

Text

Description automatically generated





This is called an effect

() => {

    console.log('effect')

    axios

      .get('http://localhost:3001/notes')

      .then(response => {

        console.log('promise fulfilled')

        setNotes(response.data)

      })

  }

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](https://reactjs.org/docs/hooks-reference.html#conditionally-firing-an-effect). If the second parameter is an empty array [], then the effect is only run along with the first render of the component.

The sequence of events is important to understand. It is illustrated by the console.log()

* 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.
* Then the side effect is automatically execute after that first rendering. This prints into the console: effect
* When The data arrives from the server, the handler in .then is called. This prints “promise is fulfilled”
* The response.data received from server is stored into the state updating function
* 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.

Now the problem we have is that when adding notes they are not stored in the server

Also do not forget these two import statements

import {useState, useEffect} from 'react'

import axios from 'axios'

I still have an error

A screenshot of a computer

Description automatically generated with medium confidence

The reason for this error is this line

       {showNotes.map

Bause showNotes does not contain anything, sionce it is defined as

const showNotes = showAll ? note : note.filter((x) => x.important)

and note contains nothing

The reason is that note contains nothings. The root cause of the problem is prop !

const App = (props) => {

  const [note, setNote] = useState([notes])

This does not work anymore because remember in index.js

ReactDOM.createRoot(document.getElementById('root')).render(

    <App />

We have removed the prop note=notes. Notes comes from the server now

The solution is to removes the props

const App = () => {

  const [note, setNote] = useState([])

All in all, our application looks like this

Index.js

import React from 'react'

import ReactDOM from 'react-dom/client'

import axios from 'axios'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(

  <App />

)

App.js

import {useState, useEffect} from 'react'

import axios from 'axios'

import Note from './components/Note'

const App = () => {

  const [note, setNote] = useState([])

  const [newNote, setNewNote] = useState('...type here')

  const [showAll, setShowAll] = useState(true)

  const showNotes = showAll ? note : note.filter((x) => x.important)

  useEffect(() => {

  axios

  .get('http://localhost:3001/notes')

  .then(response => {

    setNote(response.data)

    })

  }, [] )

  console.log('render', note.length, 'notes')

  const handleSubmit = (event) => {

    event.preventDefault()

    console.log(event.target)

    const newWrittenNote =  {

      id: note.length+1,

      content: newNote,

      date: new Date().toISOString(),

      important: Math.random()<0.5

    }

    setNote(note.concat(newWrittenNote))

    setNewNote('')

  }

  const changeInput = (event) => {

    console.log(event.target.value)

    setNewNote(event.target.value)

  }

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {showNotes.map(x =>

          <Note key={x.id} content={x.content}/>

          )

        }

      </ul>

      <form onSubmit={handleSubmit}>

        <input value={newNote} onChange={changeInput} type="text"/>

        <button type="submit">Submit</button>

      </form>

      <button onClick={() => setShowAll(!showAll)}> {showAll ? 'All Notes' : 'Important Notes'} </button>

    </div>

  )

}

export default App

Remember that for the app to be running you need to have to separate command line instances open

* One for running the React app: npm start app-part2-a
* One for running the JSON server: npm run server

### B.17. The development runtime environment

Here is the structure of the app.the Ract dev server will be explained in more detail in part 7 of the course

Diagram

Description automatically generated

## Altering data in server

REST API Wikipedia definition

<https://en.wikipedia.org/wiki/Representational_state_transfer>

Routes

<https://github.com/typicode/json-server#routes>

Conventions

<https://en.wikipedia.org/wiki/Representational_state_transfer#Applied_to_web_services>

### B.18 REST

We will look at REST APIs in more detail in oart 3 of course. Here we familiarize ourselves with some convenions

As a reminder the JSON that sits on our JSON server looks like this

[

* {
  + **id**: 1,
  + **content**: "HTML is easy",
  + **date**: "2022-1-17T17:30:31.098Z",
  + **important**: true

},

* {
  + **id**: 2,
  + **content**: "Browser can execute only JavaScript",
  + **date**: "2022-1-17T18:39:34.091Z",
  + **important**: false

},

* {
  + **id**: 3,
  + **content**: "GET and POST are the most important methods of HTTP protocol",
  + **date**: "2022-1-17T19:20:14.298Z",
  + **important**: true

}

]

* IN REST terminology every data object contained in the server called a **resource**
  + In the JSON server, we have stored 3 resources because we have 3 individual objects
* Each **resource** has a **unique** address in the server, called **URL**
  + The URL for the server resource is <http://localhost:3001/notes>. It gives us a resource collection (a collection is an array for APIs) containing all the individual resources

Text

Description automatically generated

* The URL for each resource is URL\_of\_server/id\_of\_resource.
  + For example the URL of the 3rd resource is <http://localhost:3001/notes/3>
  + If you put that URL in your browser you get the object corresponding to that resource
    - Text

      Description automatically generated
* We use Get requests to fetch resources from the server
  + To fetch the **resource collection** notes, we do a get request to the URL <http://localhost:3001/notes>. Because this is a **collection,**  the request will return a **list**  of all notes
  + To fetch an individual **resource**, such as the one with id 3, we do a get request to the URL <http://localhost:3001/notes>/3. This will return an the object literal whose id is 3
* json-server requires all data to be sent in JSON format. This means that
  + The data must be a string formatted as a Javascript object literal ie as JSON. Remember that we created a db.json file and the data inside it has the form of an object literal in JS
  + the request must contain the Content-Type request header with the value application/json.

Remember from part 0 that the content-type refers to what you see in the network tab here

Text

Description automatically generated

And remember we visualized the content-type of the response by doing a console.log(response.data)

Text

Description automatically generated

* 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.

### B.19 Sending data to the server: axios.post

We want the user to add a new note and we want to post that data to the server

So we will do an axos.post(URL, JSON\_object\_to\_send)

First question is: where do we put that request in the code ? Here is the App code before doing anything

const App = () => {

  const [note, setNote] = useState([])

  const [newNote, setNewNote] = useState('...type here')

  const [showAll, setShowAll] = useState(true)

  const showNotes = showAll ? note : note.filter((x) => x.important)

  useEffect(() => {

  axios

  .get('http://localhost:3001/notes')

  .then(response => {

    setNote(response.data)

    console.log(response)

    })

  }, [] )

  console.log('render', note.length, 'notes')

  const handleSubmit = event => {

    event.preventDefault()

    console.log(event.target)

    const newWrittenNote =  {

      id: note.length+1,

      content: newNote,

      date: new Date().toISOString(),

      important: Math.random()<0.5

    }

    setNote(note.concat(newWrittenNote))

    setNewNote('')

  }

  const changeInput = (event) => {

    console.log(event.target.value)

    setNewNote(event.target.value)

  }

  return (

// …

Do we put this in useEffect() ? No because the callback function (first argument) of useEffect() executes when after the first render of the page. We do not want to send a new resource to the server when the page renders.

We want to send the new resource when the button submit is clicked. So the axios.get should be inside the handleSubmit event handler 😊

Note that the server will generate its own id. Therefore we must remove this line

      id: note.length+1,

Graphical user interface, text, application

Description automatically generated

So what we do is this

const handleSubmit = event => {

    event.preventDefault()

    console.log(event.target)

**const newWrittenNote =  {**

**content: newNote,**

**date: new Date().toISOString(),**

**important: Math.random()<0.5**

**}**

**axios**

**.post('http://localhost:3001/notes', newWrittenNote)**

**.then(response => {**

**console.log('axios.post response:', response)**

**})**

    setNote(note.concat(newWrittenNote))

    setNewNote('')

  }

We hit the submit button to test if this works. We printed the response in the console. It looks like this

Text

Description automatically generated

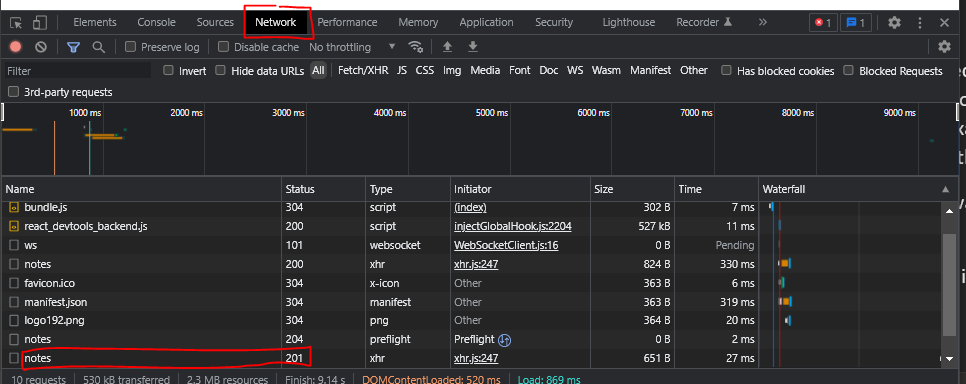
The response tells us that

* (1) The HTTP 201 Created success status response code indicates that the request has succeeded and has led to the creation of a resource.
* (2) The data shows us exactly the content of the resource that was created
* (3) The headers tells us that the content-type is JSON as expected

When inspecting a request we want to check that

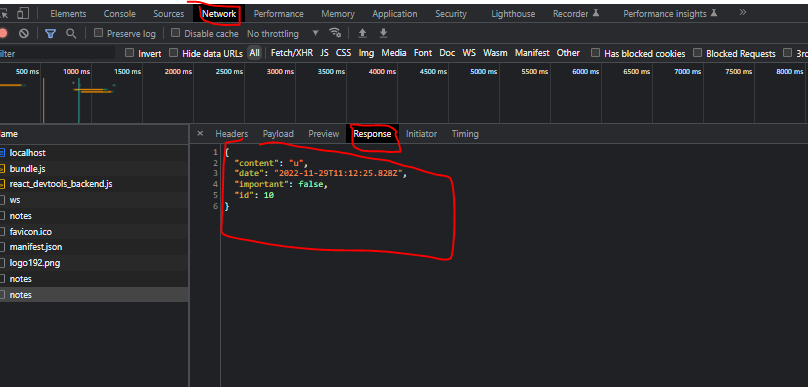
* The headers are correct
* The content of the data is correct

In the *Networks* tab, we can see the response

* 

Let’s click on that response. In network tab 🡪 header, you can see that status and the content-type

And in Network 🡪 Response we can see the content of the resource that has been added to the server



Text

Description automatically generated

To do: certificate de propriete in folder

By the way, note that axios is able to recognize by itself that the content-type is JSON,

Also, when we receive the response from the server, we update that note state variable with the response.data as follow

axios

    .post('http://localhost:3001/notes', newWrittenNote)

    .then(response => {

      console.log('axios.post response:', response)

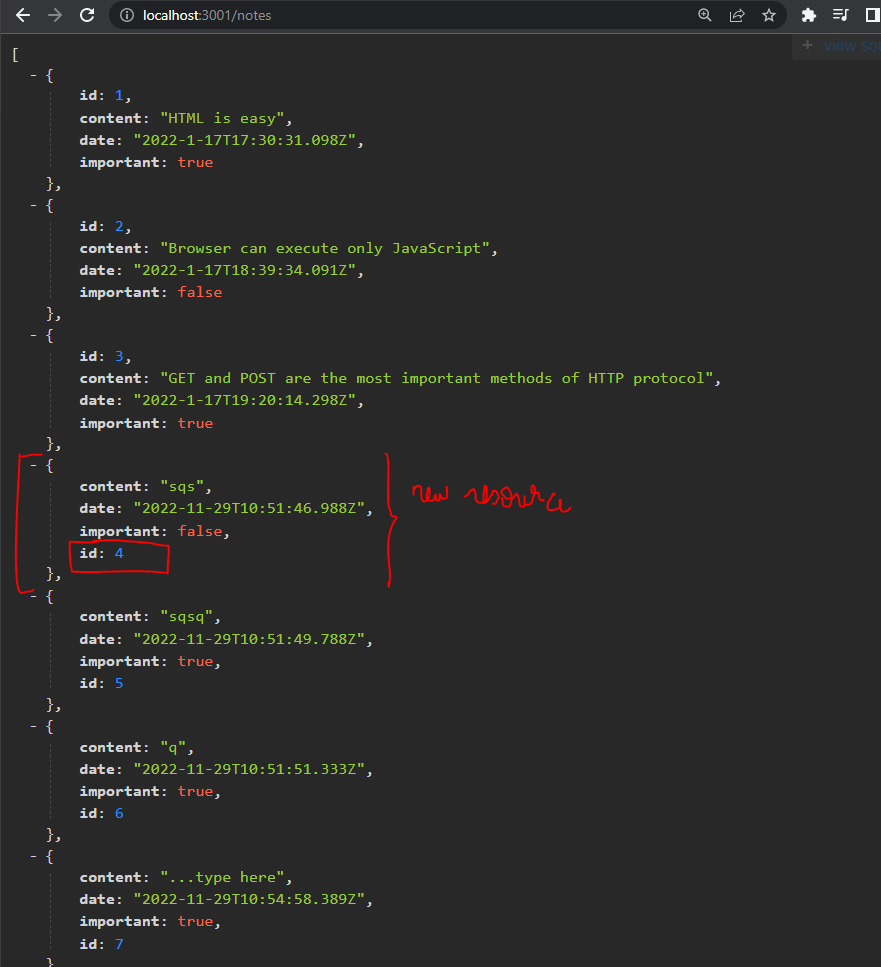
**setNote(note.concat(response.data))**

      setNewNote('')

    })

The new note returned by the backend server is added to the list of notes in our application's state

Finally, in the port 3001 you can check that the new notes have been added to the server. You can see that the server added new ids automatically for each resource



Note that

* + In the current version of our application, the browser adds the creation date property to the note. Since the clock of the machine running the browser can be wrongly configured, it's much wiser to let the backend server generate this timestamp for us. This is in fact what we will do in the next part of the course.
  + Later in this course we will use this to help us debug our server applications: <https://www.postman.com/downloads/>

### B.20 Changing the importance of notes: axios.put

#### Aside: template literals

console.log(`importance of ' + id + ' needs to be toggled`)

The template string syntax added in ES6 can be used to write similar strings in a much nicer way:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals>

Attention you must use the oblique bracket ` (keyboard of number 7) or it will note work

console.log(`importance of ${id} needs to be toggled`)

We can now use the "dollar-bracket"-syntax to add parts to the string that will evaluate JavaScript expressions, e.g. the value of a variable. Note that we use backticks in template strings instead of quotation marks used in regular JavaScript strings.

Using template literals will be important as we will manipulate string URLs of individual resources, each having a specific value for the id variable

I added a code snippet in vscode “dol” that adds ${id}

I amended the code snippet “clog” to ùse the ` bracket systematically

#### Aside the array find() method

This is another higher order function

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find>

The find() method returns the first element in the provided array that satisfies the provided testing function. If no values satisfy the testing function, [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) is returned.

const array1 = [5, 12, 8, 130, 44];

const found = array1.find(element => element > 10);

console.log(found);

#### Aside: Deep vs shallow copy

This is explained here. Assume B is a copy of A. And A contains some objects.

<https://en.wikipedia.org/wiki/Object_copying#Shallow_copy>

* A shallow copy : B will copy the references to the objects that A contains. The referenced objects are therefore *shared* between A and B. Therefore is one of these objects is modified (from A or B) then the change will be visible in the other . Shallow copies are cheap
* A deep copy: Here we create new memory addresses that contain copies of objects in A, and these objects will be put in B. Think of it as *dereferencing then copying* . This is more expensive in terms of memory. But, when one object in memory is changed in one, it is not changed in the other

The object spread […] in Javascript does a shallow copy

#### Step 1

*Prompt*

It is very important to do this step as an exercise. You learn a lot

We want to create a button to every note used for toggling its importance

So first step is to think: Where do we put that code ? Since *every note* will have its own button, we need to create the button inside the Note component . The actual HTML button must have a label and a handler attribute with argument id. The job of the handler will be to switch the Boolean “important” field when the button is clicked

In the first step we just create the button for each note. If, for example, the button associated with note 3 is clicked, we should print the following message to the console

“Importance of note 3 needs to be toggled”

Answer

Many things to note

* It is better to redefine the props as being a single note, instead of a single note’s content. This allows to access the important field of the note as well, and not only its content
* In the button, the handler must be a reference. If you add () to the function the handler will be called everytime a button is rendered on the page and will not wait for the button to be clicked
* Note that we “parametrized” the handler function using the method that we saw in part 1, by making each note having its own handler function
* handleImportance = {() => console.log(`Importance of note of ${x.id} must be changed` )}  />

Here is the note component

const Note = ({myNote, handleImportance}) => {

  const label = myNote.important ?  'note is important' : 'note is not important'

  return (

    <>

      <li> {myNote.content}</li>

      <button onClick = {handleImportance} > {label}</button>

    </>

  )

}

export default Note

Here is the App component

import {useState, useEffect} from 'react'

import axios from 'axios'

import Note from './components/Note'

const App = () => {

  const [notes, setNote] = useState([])

  const [newNote, setNewNote] = useState('...type here')

  const [showAll, setShowAll] = useState(true)

  const showNotes = showAll ? notes : notes.filter((x) => x.important)

  useEffect(() => {

  axios

  .get('http://localhost:3001/notes')

  .then(response => {

    setNote(response.data)

    console.log('axios.get response:', response)

    })

  }, [] )

  console.log('render', notes.length, 'notes')

  const handleSubmit = event => {

    event.preventDefault()

    console.log(event.target)

    const newWrittenNote =  {

      content: newNote,

      date: new Date().toISOString(),

      important: Math.random()<0.5

    }

    axios

    .post('http://localhost:3001/notes', newWrittenNote)

    .then(response => {

      console.log('axios.post response:', response)

      setNote(notes.concat(response.data))

      setNewNote('')

    })

  }

  const changeInput = (event) => {

    console.log(event.target.value)

    setNewNote(event.target.value)

  }

// expected output: 12

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {showNotes.map(x =>

          <Note key={x.id}

                myNote={x}

                handleImportance = {() => console.log(`Importance of note of ${x.id} must be changed` )}  />

          )

        }

      </ul>

      <form onSubmit={handleSubmit}>

        <input value={newNote} onChange={changeInput} type="text"/>

        <button type="submit">Submit</button>

      </form>

      <button onClick={() => setShowAll(!showAll)}> {showAll ? 'All Notes' : 'Important Notes'} </button>

    </div>

  )

}

export default App

#### Step 2

*Prompt*

Now we define the body of the handler function of the button. Follow these steps

* Define a url variable that store the URL specific to the resource with the id, *id*
* Define a “note” variable that stores the nore with that id. Use the find higher order function
* Create a variable changeNote, which is a copy of the note variable, where the important attribute is changed to its opposite value. Use the [object spread](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax)  ES6 technique. Remember that
* Finally use axios.put (is uses the same arguments as axios.post . When you get a response, use the state method to update the notes variable with the notes received as a response from the server.To do the update, use the map function with a one line conditional statement. In the conditional statement, for aeach note, if the id is not the one, then simply take the old note, otherweise take the note received in response

First I extract the handler into its own function. Now the the handler cannot be a function call, so I parametrize the handler using a function that returns a function as we saw in part one

   const handleImportance = (id) => {

    console.log(`Importance of note of ${id} must be changed` )

   }

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {showNotes.map(x =>

          <Note key={x.id}

                myNote={x}

                handleImportance = {() => handleImportance(x.id)}  />

          )

        }

      </ul>

Then, have the console open and each time you write a line verify that it works using console.log. For example

   const handleImportance = (id) => {

    const url = `http://localhost:3001/notes/${id}`

    console.log('The url is', url)

    console.log(`Importance of note of ${id} must be changed` )

   }

Now I arrive here and I get the mistake (the console.logs are a form of test driven development )

const handleImportance = (id) => {

    console.log(`Importance of note of ${id} must be changed` )

    const url = `http://localhost:3001/notes/${id}`

    console.log('The url is', url)

    const note = notes.find((x) => x.id == id )

    console.log(`the note that we found is ${note.id}`)

    console.log(note)

    const changedNote = {...note, important: !note.important}

    console.log(`The importance of the original is ${note.important} while the importance of the copy is ${changedNote.important}`)

    axios

    .post('http://localhost:3001/notes', changedNote)

    .then (response => {

      console.log(`axios post response is ${response}` )

    })

I get this error

A screenshot of a computer

Description automatically generated

So basically the put request failed. To get more information, open the shell that was used to run the server

Text

Description automatically generated

It is trying to insert a new note. This is not what we wanted initially. The reason is that I used axios.post (which creates a new resource) instead of axios.put() which modifies an existing resource

So I change to this

    axios

    .put('http://localhost:3001/notes', changedNote)

    .then (response => {

      console.log(`axios post response is ${response}` )

    })

Now I get this error

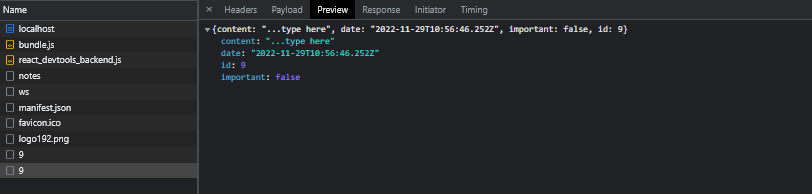


404 means that The HTTP 404 Not Found response status code indicates that **the server cannot find the requested resource**.

The server cannot find the resource. Why ? Because we are passing it the whole *resource collection* note and note a specifi resource. Also changeNote must not be a copy of the whole notes, but only of the specific note object with that id

So we do this and it works.

We can check in the network



So here is my un-neated code with lots of console.logs alog the way

   const handleImportance = (id) => {

    console.log(`Importance of note of ${id} must be changed` )

    const url = `http://localhost:3001/notes/${id}`

    console.log('The url is', url)

    const originalnote = notes.find((x) => x.id == id )

    console.log(`the note that we found is ${originalnote.id}`)

    console.log('the original note is', originalnote)

    const changedNote = {...originalnote, important: !originalnote.important}

    console.log(`The importance of the original is ${originalnote.important} while the importance of the copy is ${changedNote.important}`)

    console.log('changed note is', changedNote)

    axios

    .put(url, changedNote)

    .then (response => {

      console.log('response data is ', response.data)

      const notesCopy = [...notes]

      notesCopy.map((x) => x.id==id ? response.data: x)

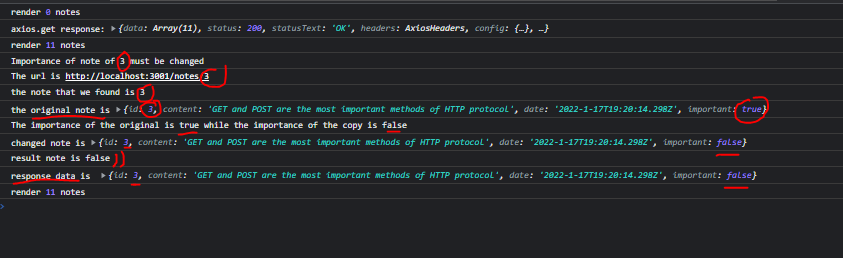
      setNote(notesCopy)

    })

    console.log(`result note is ${notes[id].important}`)

   }

The console.logs allow me to check that the code works at the end. For example



Now let’s neaten the code. We keep the console.logs for now.

First note that maps is not a mutator function. Therefore you do not need to make a copy of the state note notes before maping it

Therefore we do this

    axios

    .put(url, changedNote)

    .then (response => {

      setNote(notes.map((x) => x.id==id ? response.data: x))

    })

You should understand what it does:

* notes.map((x) => x.id==id ? response.data: x) **Returns a shallow copy** of the notes array with the change defined in the callback function

So the final code with the clogs looks like this

/\*

Now we define the body of the handler function of the button. Follow these steps

- Define a url variable that store the URL specific to the resource with the id, id

- Define a “note” variable that stores the note with that id. Use the find higher order function

- Create a variable changeNote, which is a copy of the note variable, where the important attribute is changed to its opposite value. Use the object spread  ES6 technique. Remember that

- Finally use axios.put (is uses the same arguments as axios.post .

  When you get a response, use the state method to update the notes

   variable with the notes received as a response from the server.To do

  the update, use the map function with a one line conditional statement.

  In the conditional statement, for aeach note, if the id is not the one,

  then simply take the old note, otherweise take the note received in

  response

\*/

const handleImportance = (id) => {

    console.log(`Importance of note of ${id} must be changed` )

    const url = `http://localhost:3001/notes/${id}`

    console.log('The url is', url)

    const originalnote = notes.find((x) => x.id == id )

    console.log(`the note that we found is ${originalnote.id}`)

    console.log('the original note is', originalnote)

    const changedNote = {...originalnote, important: !originalnote.important}

    console.log(`The importance of the original is ${originalnote.important} while the importance of the copy is ${changedNote.important}`)

    console.log('changed note is', changedNote)

    axios

    .put(url, changedNote)

    .then (response => {

      console.log('response data is ', response.data)

      setNote(notes.map((x) => x.id==id ? response.data: x))

    })

    console.log(`result note is ${notes[id].important}`)

   }

Let’s say goodbye to the clogs

Here is the version of the code at this point

Index.js

import React from 'react'

import ReactDOM from 'react-dom/client'

import axios from 'axios'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(

  <App />

)

App.js

import {useState, useEffect} from 'react'

import axios from 'axios'

import Note from './components/Note'

const App = () => {

  const [notes, setNote] = useState([])

  const [newNote, setNewNote] = useState('...type here')

  const [showAll, setShowAll] = useState(true)

  const showNotes = showAll ? notes : notes.filter((x) => x.important)

  useEffect(() => {

  axios

  .get('http://localhost:3001/notes')

  .then(response => {

    setNote(response.data)

    console.log('axios.get response:', response)

    })

  }, [] )

  console.log('render', notes.length, 'notes')

  const handleSubmit = event => {

    event.preventDefault()

    console.log(event.target)

    const newWrittenNote =  {

      content: newNote,

      date: new Date().toISOString(),

      important: Math.random()<0.5

    }

    axios

    .post('http://localhost:3001/notes', newWrittenNote)

    .then(response => {

      console.log('axios.post response:', response)

      setNote(notes.concat(response.data))

      setNewNote('')

    })

  }

  const changeInput = (event) => {

    console.log(event.target.value)

    setNewNote(event.target.value)

  }

   const handleImportance = (id) => {

    const url = `http://localhost:3001/notes/${id}`

    const originalnote = notes.find((x) => x.id == id )

    const changedNote = {...originalnote, important: !originalnote.important}

    axios

    .put(url, changedNote)

    .then (response => {

      setNote(notes.map((x) => x.id==id ? response.data: x))

    })

   }

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {showNotes.map(x =>

          <Note key={x.id}

                myNote={x}

                handleImportance = {() => handleImportance(x.id)}  />

          )

        }

      </ul>

      <form onSubmit={handleSubmit}>

        <input value={newNote} onChange={changeInput} type="text"/>

        <button type="submit">Submit</button>

      </form>

      <button onClick={() => setShowAll(!showAll)}> {showAll ? 'All Notes' : 'Important Notes'} </button>

    </div>

  )

}

export default App

### B.21. Extracting Communication with the Backend into a Separate Module

#### Export ES6 syntax

[Export](https://developer.mozilla.org/en/docs/web/javascript/reference/statements/export) like export default HelloWorld; and [import](https://developer.mozilla.org/en/docs/web/javascript/reference/statements/import), such as import React from 'react' are part of the [ES6 modules system](http://www.2ality.com/2014/09/es6-modules-final.html).

A module is a self contained unit that can expose assets to other modules using export, and acquire assets from other modules using import.

In ES6 there are two kinds of exports:

**Named exports** - for example export function func() {} is a named export with the name of func. Named modules can be imported using import { exportName } from 'module';. **In this case, the name of the import should be the same as the name of the export.** To import the func in the example, you'll have to use import { func } from 'module';. There can be multiple named exports in one module.

**Default export** - is the value that will be imported from the module, if you use the simple import statement import X from 'module'. **X is the name that will be given locally to the variable assigned to contain the value, and it doesn't have to be named like the origin export.** There can be only one default export.

A module can contain both named exports and a default export, and they can be imported together using import defaultExport, { namedExport1, namedExport3, etc... } from 'module';.

Here we use this the default export syntax

Export X : Y X is the new name of what is exported, Y is what is exported

#### Useful readings on chaining promises

<https://javascript.info/promise-chaining>

<https://github.com/getify/You-Dont-Know-JS/tree/1st-ed>

<https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/async%20%26%20performance/ch3.md>

Note: the then method returns a promise

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then>

#### Extracting communication with server into a module

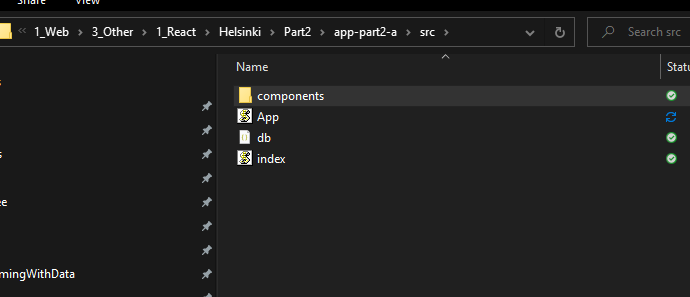
Single Responsibility Principle

<https://en.wikipedia.org/wiki/Single-responsibility_principle>

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](https://en.wikipedia.org/wiki/Single_responsibility_principle), we deem it wise to extract this communication into its own [module](https://fullstackopen.com/en/part2/rendering_a_collection_modules#refactoring-modules).

This is a good example of module cohesion (software design and development course)

First let’s remember how we structured our folder. Remember that we created a folder Components that contains the Note component.



The module on server is not a component. So we create another directory called services. Then we create a new file called notes.js. I use the command line :

Mkdir services

cd services

type nul > services.js

So we want to extract the promises. Note that they are chained. We always have a get (or put or post) followed by a then. At each level, a promise is returned axios.get returns a promise, then .get works on that response and also returns a promise

 useEffect(() => {

  axios

  .get('http://localhost:3001/notes')

  .then(response => {

    setNote(response.data)

    })

  }, [] )

Diagram

Description automatically generated

##### Extracting the first level promises

Important that you code it yourself ! You learn so much !

Here we move all the code in App.js that pertains to axios.get, axios.put and axios. We use import and export

###### Setting up the modular infrastructure

First I identify all the bits of App.js that want to put in the module

First in services.js I write this code. On the export: Remember that the export we did for the Note.js component is

export default Note

Here we export many functions so we use an object literal. The field is the export name, and the value is the actual name of the function being exported

import axios from 'axios'

const baseUrl = 'http://localhost:3001/notes'

const getAll = () => {

    return axios.get(baseUrl)

}

const insert = (resource) => {

    return axios.post(baseUrl, resource)

}

const update = (url, newNote) => {

    return axios.put(url, newNote)

}

const printMe = () => console.log(`Hello from services`)

export default {

    getAll: getAll,

    insert: insert,

    update: update,

    printMe: printMe

}

Note that I add the printMe function for testing purposes. In app.js I call the printMe function and it indeed prints the message

const App = () => {

///…

  services.printMe()

///…

Note that (I noticed this after a number off errors

* When declaring the functions, use the rule of return. Brackets are married to return !
* The update function requires a url argument. That is the url to a specific resource, not the url to the resource collection

###### Using imported module functions

Next in app.js we use services. For example

  useEffect(() => {

    services

    .getAll()

    .then(response => {

      console.log(response)

      setNote(response.data)

      })

    }, [] )

Here is the whole app.js at this point

import {useState, useEffect} from 'react'

import axios from 'axios'

import Note from './components/Note'

import services from './services/services'

const App = () => {

  const [notes, setNote] = useState([])

  const [newNote, setNewNote] = useState('...type here')

  const [showAll, setShowAll] = useState(true)

  const showNotes = showAll ? notes : notes.filter((x) => x.important)

  useEffect(() => {

    services

    .getAll()

    .then(response => {

      console.log(response)

      setNote(response.data)

      })

    }, [] )

  const handleSubmit = event => {

    event.preventDefault()

    const newWrittenNote =  {

      content: newNote,

      date: new Date().toISOString(),

      important: Math.random()<0.5

    }

    services

    .insert(newWrittenNote)

    .then(response => {

      setNote(notes.concat(response.data))

      setNewNote('')

    })

  }

   const handleImportance = (id) => {

    const url = `http://localhost:3001/notes/${id}`

    const originalnote = notes.find((x) => x.id == id )

    const changedNote = {...originalnote, important: !originalnote.important}

    services

    .update(url, changedNote)

    .then (response => {

      setNote(notes.map((x) => x.id==id ? response.data: x))

    })

   }

   const changeInput = (event) => {

    setNewNote(event.target.value)

  }

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {showNotes.map(x =>

          <Note key={x.id}

                myNote={x}

                handleImportance = {() => handleImportance(x.id)}  />

          )

        }

      </ul>

      <form onSubmit={handleSubmit}>

        <input value={newNote} onChange={changeInput} type="text"/>

        <button type="submit">Submit</button>

      </form>

      <button onClick={() => setShowAll(!showAll)}> {showAll ? 'All Notes' : 'Important Notes'} </button>

    </div>

  )

}

export default App

##### Refactoring the chained promises

This section illustrates the concept of playing with chained promises ie .then

Here

services

    .getAll()

    .then(response => {

      setNote(response.data)

      })

The .then receives the whole HTML response. However, it uses only the response.data.

We change the code so that the .then receives the response.data only

const getAll = () => {

    const request = axios.get(baseUrl)

    return request.then(response => response.data)

}

We use chaining (ie a then) inside the getAll function: request is the whole HTML response. Using chained promises (then) we take that response and reduce it to response.data.

Back to app.js this allows us to write

  useEffect(() => {

    services

    .getAll()

    .then(initialNotes => {

      setNote(initialNotes)

      })

    }, [] )

Hence the .then gets a response.data (we called the argument initialNotes, we could have called it x)

And returns a response.data

So, the whole code at this point looks like this

Services.js

import axios from 'axios'

const baseUrl = 'http://localhost:3001/notes'

const getAll = () => {

    const request = axios.get(baseUrl)

    return request.then(response => response.data)

}

const insert = (resource) => {

    const request = axios.post(baseUrl, resource)

    return request.then(response => response.data)

}

const update = (url, newNote) => {

    const request = axios.put(url, newNote)

    return request.then(response => response.data)

}

const printMe = () => console.log(`Hello from services`)

export default {

    getAll: getAll,

    insert: insert,

    update: update,

    printMe: printMe

}

App.js

import {useState, useEffect} from 'react'

import axios from 'axios'

import Note from './components/Note'

import services from './services/services'

const App = () => {

  const [notes, setNote] = useState([])

  const [newNote, setNewNote] = useState('...type here')

  const [showAll, setShowAll] = useState(true)

  const showNotes = showAll ? notes : notes.filter((x) => x.important)

  useEffect(() => {

    services

    .getAll()

    .then(initialNotes => {

      setNote(initialNotes)

      })

    }, [] )

  const handleSubmit = event => {

    event.preventDefault()

    const newWrittenNote =  {

      content: newNote,

      date: new Date().toISOString(),

      important: Math.random()<0.5

    }

    services

    .insert(newWrittenNote)

    .then(noteToInsert => {

      setNote(notes.concat(noteToInsert))

      setNewNote('')

    })

  }

   const handleImportance = (id) => {

    const url = `http://localhost:3001/notes/${id}`

    const originalnote = notes.find((x) => x.id == id )

    const changedNote = {...originalnote, important: !originalnote.important}

    services

    .update(url, changedNote)

    .then (noteChanged => {

      setNote(notes.map((x) => x.id==id ? noteChanged: x))

    })

   }

   const changeInput = (event) => {

    setNewNote(event.target.value)

  }

  return (

    <div>

      <h1>Notes</h1>

      <ul>

        {showNotes.map(x =>

          <Note key={x.id}

                myNote={x}

                handleImportance = {() => handleImportance(x.id)}  />

          )

        }

      </ul>

      <form onSubmit={handleSubmit}>

        <input value={newNote} onChange={changeInput} type="text"/>

        <button type="submit">Submit</button>

      </form>

      <button onClick={() => setShowAll(!showAll)}> {showAll ? 'All Notes' : 'Important Notes'} </button>

    </div>

  )

}

export default App

### B.22. Cleaner Syntax for defining object literals

In this code

export default {

  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 of the module.

We can reduce the above syntax to

export default {

  getAll,

  create,

  update

}

Here we are using this new feature introduced in ES6

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#property_definitions>

const object1 = { a: 'foo', b: 42, c: {} };

console.log(object1.a);

// expected output: "foo"

const a = 'foo';

const b = 42;

const c = {};

const object2 = { a: a, b: b, c: c };

console.log(object2.b);

// expected output: 42

const object3 = { a, b, c };

console.log(object3.a);

// expected output: "foo"

To demonstrate this feature, let's consider a situation where we have the following values assigned to variables:

const name = 'Leevi'

const age = 0

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

const person = {

name: name,

age: age

}

However, since both the property fields and the variable names in the object are the same, it's enough to simply write the following in ES6 JavaScript:

const person = { name, age }

### B.22. Promises and Errors

We saw earlier that

A promise can have three distinct states:

* 1. The promise is pending: It means that the final value (one of the following two) is not available yet.
  2. 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.
  3. The promise is rejected: It means that an error prevented the final value from being determined, which generally represents a failed operation.

#### Method : We handle rejection of a promise with a .catch() that follows a .then()

This section is about how to handle how a promise is rejected

I structure this section in two steps

* First I expose the new concepts, with references and examples of how to code them
* Then I write prompts
* Then I code *myself -*This is super important for learning

MDN how to handle the rejection of a promise

The rejection of a promise is [handled](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises) 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch) method.

MDN catch method

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch>

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.

Again the Javascript page on promise chaining

<https://javascript.info/promise-chaining>

The catch method is often utilized by placing it at the end of a promise chain, which is called once any promise in the chain throws an error and the promise becomes rejected.

We can make the catch throw an alert

<https://developer.mozilla.org/en-US/docs/Web/API/Window/alert>

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.

#### Try it yourself

The version of the code up to now is saved in git as “B.22 Cleaner Syntax for defining object literals” with SHA 318380e in this folder

C:\Users\yassi\OneDrive\1\_Work\2\_Innovation\2\_WebMobile\1\_Web\3\_Other\1\_React\Helsinki\Part2\app-part2-a

##### Create a 404 not found server error

Q. In the getAll method, return an array that contains the notes received from the server, concatenated with some note that you paste into it directly. Paste a picture of the error that shows rejection of the promise

We do this

const getAll = () => {

    const request = axios.get(baseUrl)

    const  nonExistingNote = {

          id: 10000,

          content: 'This note does not exist on the server',

          date: '2019-05-30T17:30:31.098Z',

          important: true

        }

    return request.then(response => response.data.concat(nonExistingNote))

}

Then in the note on the app we click

Graphical user interface, application

Description automatically generated

When you click, the following handler fires. Services.update is a put request. So, a PUT request is sent. A out request is an update to an already existing resource

Text

Description automatically generated

But: The resource with that id (id=10000 ) is not found. So, you receive a not found error

Text

Description automatically generated

##### Catch the promise rejection and create an alert

Q. Use the catch syntw shown above. Go to the App component. In the promises chain that contains the update, add a catch that alerts `the note XXX was already deleted from server` where X is the content of the note

Catch receives as argument a callback function whose argument is error

   const handleImportance = (id) => {

    const url = `http://localhost:3001/notes/${id}`

    const originalnote = notes.find((x) => x.id == id )

    const changedNote = {...originalnote, important: !originalnote.important}

    services

    .update(url, changedNote)

    .then (noteChanged => {

      setNote(notes.map((x) => x.id==id ? noteChanged: x))

    })

**.catch(error => {**

**alert(`the note with content "${originalnote.content}" is not on the server`)**

**})**

**}**

##### Delete the note that exists on the page but not on the server

Q. Now inside the catch, solve the problem by deleting the note that is not part of the server using the filter method

Let’s recap what we are doing here

* You click the “Note is important” button for the note whose content is “This note does not exist on the server”.
* The handler above fires with id = 1000
* We receive A 404 not found server error
* The callback function inside the catch fires
* Inside the callback we delete the note id 1000. We apply a filter on the notes array where only the notes whose id IS NOT 1000 are kept

   const handleImportance = (id) => {

    const url = `http://localhost:3001/notes/${id}`

    const originalnote = notes.find((x) => x.id == id )

    const changedNote = {...originalnote, important: !originalnote.important}

    services

    .update(url, changedNote)

    .then (noteChanged => {

      setNote(notes.map((x) => x.id==id ? noteChanged: x))

    })

    .catch(error => {

      alert(`the note with content "${originalnote.content}" is not on the server`)

**notes.filter( note => note.id !==id)**

    })

   }

## Adding styles to REACT apps

CSS preprocessors

<https://developer.mozilla.org/en-US/docs/Glossary/CSS_preprocessor>

CSS selectors

<https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors>

In React you must use the className attriute

### B.24 Using CSS in REACT

Q. Create an index.css file and import it. Where should you import it ?

You should import it in the index.js file, not in the App.js file

import './index.css'

Q. make all h1 elements green with an italic font

In index.css

h1 {

    color: green;

    font-style: italic

}

Q. Now target the li tags that contain notes. To do that give these <li> a class “note”. Attention in React the attribute to use is className. Then in CSS use these values for notes

color: grey;

padding-top: 5px;

font-size: 15px;

Answer: In notes.js where the li component is defined add a className attribute

const Note = ({myNote, handleImportance}) => {

  const label = myNote.important ?  'note is important' : 'note is not important'

  return (

    <>

      <li **className = "note">** {myNote.content}</li>

      <button onClick = {handleImportance} > {label}</button>

    </>

  )

}

Then in CSS

.note {

color: grey;

padding-top: 5px;

font-size: 15px;

}

### B.25. Improved error message

#### Create a component for the error message

Q. Create a component called Notification. That component has one prop “message”. If message is num, return null. Otherwise. Return a div whose content is the message in the props. Give that div the className “error”. Define the component in the module for components. Add the component in the APP

Answer:

Create a components folder. Put each component inside of modules

Text

Description automatically generated

const Notification = ({message}) => {

    if (message ==null) {

      return (null)

    }

    else {

      return(

        <div className = "error">

            {message}

        </div>

      )

    }

  }

  export default Notification

Then in App.js

The import Notification from './components/Notification'

And finally check that it works by using the component to print a message on the screen

<Notification message="Hi"/>

#### Add the logic for displaying the error component

Q. First create a state variable for the error

  const [errorMessage, setErrorMessage] = useState(null)

Q. Second, when an error is catched:

- i. Remove the alert

- ii. Update the error state variable to contain the same string that of the alert

- iii. In the catch add the timer that removes the component after 5 seconds

  .catch(error => {

      setErrorMessage(`the note with content "${originalnote.content}" is not on the server`)

      setTimeout(() => setErrorMessage(null)  , "5000")

      notes.filter( note => note.id !==id)

    }

#### Style the error component

Q. Style the div inside the notification component like this

Color : red;

background: lightgrey;

font-size: 20px;

border-style: solid;

border-radius: 5px;

padding: 10px;

margin-bottom: 10px;

So: When the user toggles the importance of a note that is deleted on the server (the last note here) this message appears for 5 seconds then disappears



The code is saved in git with sha **4f16572**

### B.26. Inline styles

REACT prefers to use inline styles instead of grouping all styles in a single CSS file.

The styles are defined as an object, and the object is passed as a “style” attribute of that component.

REACT basically groups the all three sthings : HTML, Jacascript and CSS specific to a component into one component.

#### The styling philosophy of REACT

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.**

#### Writing CSS code inside REACT component

There are slight difference in terms how we write CSS in REACT

Text

Description automatically generated

The 3 differences are

* Instead of – we use camelCase. So fontStyle becomes font-style
* Strings like ‘green’ are enclosed in brackets
* Numeric values for pixels can be simply defined as integers. So we write 16 instead of 16px

One limit of inline CSS is that you cannot use pseudoclasses

The REACT way of writing CSS is better, because it is the same as writing any Javascript object

<https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes>

/\* Any button over which the user's pointer is hovering \*/

button:hover {

  color: blue;

}

Q. Add a footer component. The footer should be have green color text, italic fontstyle and 16 fontsize. Give it any text you want

First: Create a Footer.js module and write the following code inside it

const Footer = () => {

     const footer = {

        color : 'green',

        fontStyle: 'Italic',

        fontSize: 16

     }

    return(

        <div style = {footer}>

            <em>Hi I am a footer</em>

        </div>

    )

}

export default Footer

Import that module in App.js

import Footer from './components/Footer'

And finally add the footer and the end of the App component

  <Footer/>