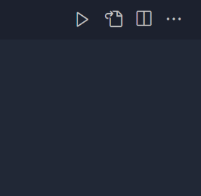
React

* React can be used for native applications for ios and android.
* For the web we use react-dom.
* Install nodeJs, because we need the npm package manager.

Reactjs CDN

* Load time, better support for updates
* Reactjs.org Docs > cdn links

Vscode extensions and tips:

* ES7 React/Redux/GraphQL/React-Native snippets
* **!** //In vscode auto-generate HTML, emmet
* rcc + tab // creates a **default react class component** model
* rfc + tab // creates a **default react function component** model
* ext: Live Server
* ext: **React Developer Tools** browser extension, search it for chrome
* <https://babeljs.io/> // shows how js codes are compiled
* ext: **Prettier** - Code formatted // it automatically imports the component when you write <component />
* In the Status bar -> click on javascript -> type react // You can use emmets now for each file
* In settings -> edit json //

"files.associations": {

"\*.js": "javascriptreact"

},

* Indentation
  + Prettier

"files.associations": {

"\*.js": "javascriptreact"

},

"editor.defaultFormatter": "esbenp.prettier-vscode",

"[javascript]": {

"editor.defaultFormatter": "esbenp.prettier-vscode"

},

"editor.formatOnSave": true,

"prettier.semi": false,

"prettier.printWidth": 9999,

"editor.tabSize": 2,

How to create manual snippets to create components

* ctr+shift+p => user snippet => javascript react => javascriptreact.json

**"React Component": {**

**"prefix": "rc",**

**"body": [**

**"import React, { useEffect } from \"react\"",**

**"",**

**"function ${1:ComponentName}() {",**

**" return (",**

**" <>",**

**" $2",**

**" </>",**

**" )",**

**"}",**

**"",**

**"export default ${1:ComponentName}"**

**],**

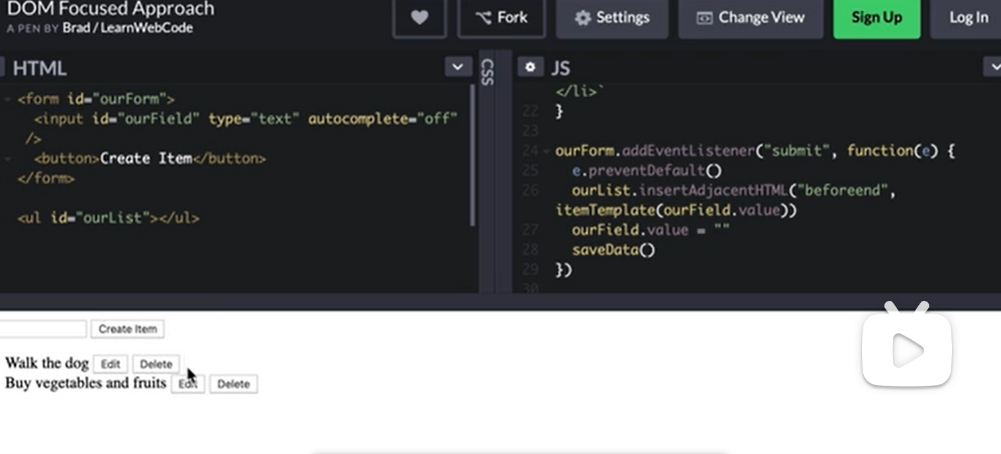
**"description": "React Component"**

**}**

Why do we use react?

DOM focused approach

* While adding and removing something on the page, we don't use the javascript arrays of data, we literally remove and add the elements to ul.



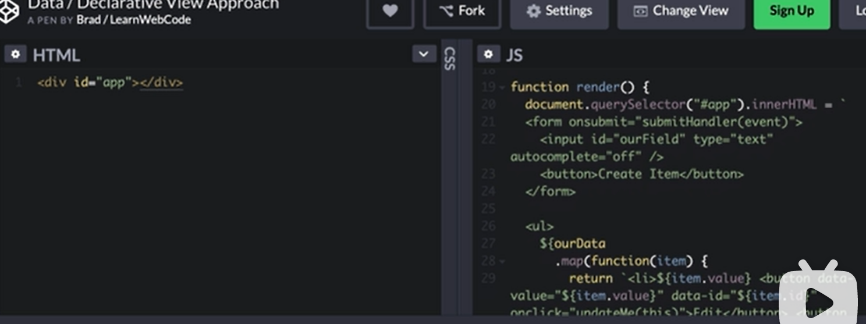
* If the browser is reloaded we can use local storage to keep values. It should be in array format.

localStorage.setItem("downWayData", JSON.stringify(allItems))

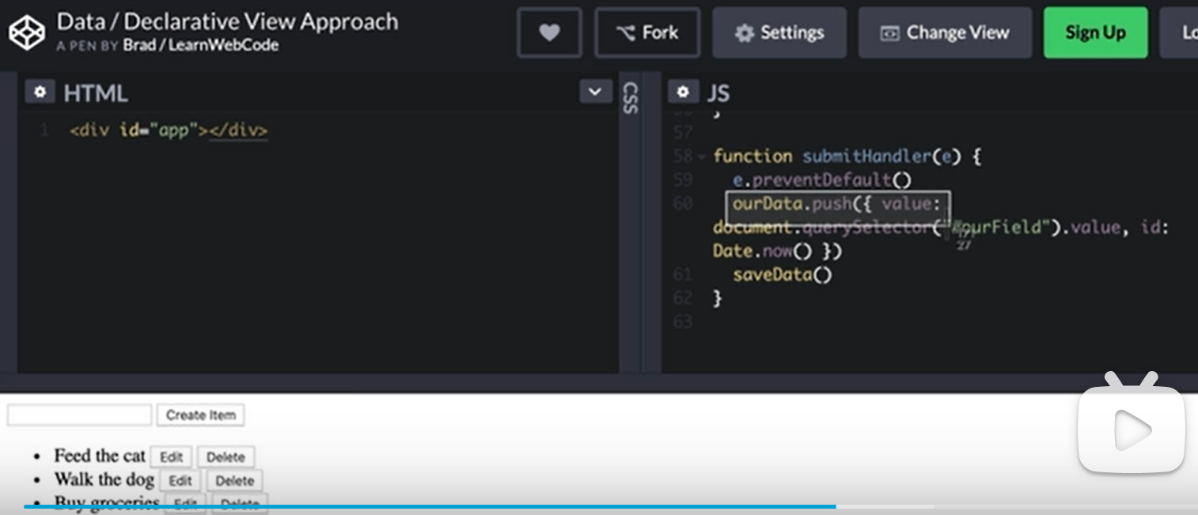
* **Problems**
  + Working with the dom is slow. We don't want the dom to be our source of truth.
  + Spaghetti problem. Our codes are fragmented // Some are in HTML, some in javascript. Babysit tons of elements manually instead of centrally located places for our interface.

Data / Declarative View

* **Render function** // all the interface is located here



* The submit does not modify the dom but it uses a raw javascript array of data. SaveData calls the render function and saves locally.



* **Problems**
  + Speed nightmare // If we have any changes in our app we are rendering the entire application! The form, all the items. Adding and removing a big section of the dom!

React, closer to data/declarative view

* One centralized place for the interface
* Not the entire interface gets rendered only the part that needs to be changed.
* Angular and vue.js do similarly. React it's a simple solution. Our user interface will magically react to data we change!!
* Keep data outside of the interface.

Webpack

* To Bundle up node module packages and react packages with our code
* Automating tool that can both bundle up our code with our dependency and transpile the jsx to normal js
* If we don't use npm install react-app, we have to install:
  + **npm install webpack webpack-cli webpack-dev-server**
    - inside the overall project folder
      * webpack.config.js
      * It will package our code and dependencies into a filename: "bundled.js". By default, this file is not saved on hard but in memory.
  + **npm install @babel/core @babel/preset-env @babel/preset-react babel-loader**
    - so our preset in webpack.config.js would work. ]

<body>

<div id="app"></div>

<script src="/bundled.js"></script>

</body>

* + Watch and bundle up
    - package.json

"scripts": {

"dev": "webpack serve",

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

},

* **npm run dev**
* **http://localhost:3000/**
* When automating prevents the browser from refreshing at all! It will load the new JavaScript on the fly asynchronously. In main.js where root.render() is:

if(module.hot){

module.hot.accept()

}

* installing npm ERR! code EPERM // use

npm cache clean --force

Installation

* Install node.js // to combine our code with react code, and transpile jsx syntax to regular javascript that browsers can understand. It can be used as a robot assistant that performs tasks for us automatically.
* npx create-react-app .

First method

* npm, init -y // In any project you should have a package.json file. It includes a list of packages our project needs.
* npm install react react-dom // it does not create the app folder just node\_modules
* npm install npm@latest -g

Second Method

* npm **install create-react-app** folder-name (.) // create-react-app is a standard of all of the tools we need. For every project, we have to download these tools (node modules). But for the test, we only use one. If the installation completes successfully, "Happy hacking! " will be printed. Dot is not to make a new folder but to create it in the current folder.
* It is recommended not to create the react locally as each project can have a version dependency
* npx -g create-react-app folder-name // Does the same thing as the above, -g activates global
* npm start // In that folder that we created we run this command, it'll open a page in the browser on the localhost with port 3000
* npm uninstall -g create-react-app folderName
* if npm install create-react-app folder-name does not work use:

npm rm -g create-react-app

npm install -g create-react-app

npx create-react-app my-app

React Packages

* **npm install react-router-dom**
* **npm install bootstrap** // to use it in react, import this directory:

import 'bootstrap/dist/css/bootstrap.css'

* **Axios**
* **Immer**
* **npm install react-markdown** // markdown syntax parser
* **npm, install react-tooltip** // to help us with hover functionalities and …
* **npm, install --save-dev style-loader css-loader** // in webpack.config.js add a new module rule ( you can do it for scss too. This will allow us to import css files instead of using a link and rel = " stylesheet " in the head tag.

{

test: /\.css$/,

use: ["style-loader", "css-loader"],

},

* **React icon**
* **npm install react-transition-group**
* **npm install socket.io-client**
* **npm install node-sass –save** // to load sass without the webpack.config.js
* **npm i --save @fortawesome/fontawesome-svg-core font awesome**

npm i --save @fortawesome/fontawesome-svg-core

# Free icons styles

npm i --save @fortawesome/free-solid-svg-icons

npm i --save @fortawesome/free-regular-svg-icons

npm i --save @fortawesome/react-fontawesome@latest

// add to package.json:

"babelMacros": {

"fontawesome-svg-core": {

"license": "free"

}

}

import { FontAwesomeIcon } from "@fortawesome/react-fontawesome"

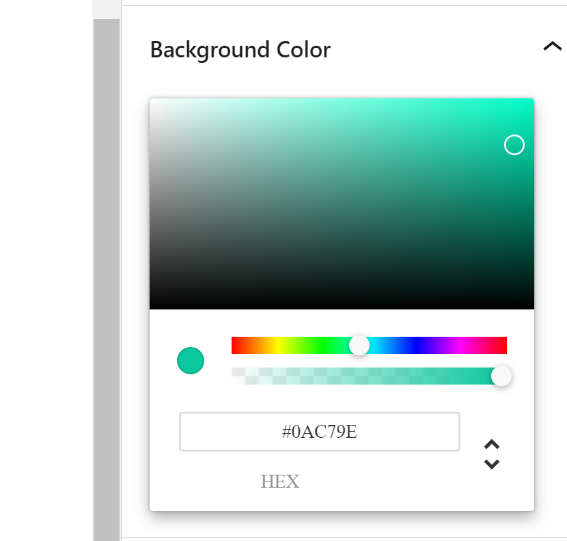
import { icon } from "@fortawesome/fontawesome-svg-core**/import.macro**"

### …

<FontAwesomeIcon icon={icon({ name: "phone" })} style={{ marginRight: "5px" }} />

* **React Color** // it has different kind of color pickers
  + npm install react-color
  + disableAlpha = {true} // an attribute

import {ChromePicker} from "react-color"



* **npm i react-alice-carousel**
* **npm install react-bootstrap bootstrap**

<Container fluid={true}></Container>

* **npm i styled-components** // it helps with the automation of styling
* **npm i react-hook-form** // form validation

const {

register,

handleSubmit,

formState: { errors },

} = useForm({ mode: "onBlur", reValidateMode: false })

const userName = register("userName", {

required: "Username is required",

minLength: { value: 6, message: "The username should be at least 6 characters" },

maxLength: { value: 20,message: "The username should be at most 20 characters"},

pattern: {

value: /^(?![\_.])(?!.\*[\_.]{2})[a-zA-Z0-9.\_]+(?<![\_.])$/,

message: `The username must follow these rules:

\n 1. only alphabets and numbers

\n 2. can not contain \_., .\_, \_\_, ..

\n 3. no \_ at the beginning or the end`,

},

})

### …

<input {...userName} onBlur={(e) => userName.onBlur(e)} placeholder="User Name" />

<CSSTransition in={Boolean(errors.userName)} nodeRef={usernameErrorRef} classNames="formErrorMessage" timeout={330} unmountOnExit>

<div className="alert alert-danger small formErrorMessage"> {errors.userName?.message} </div>

</CSSTransition>

Imports

* import React from "react"
* import ReactDOM from "react-dom"
* import ReactDOM from "react-dom/client"
* import **React**, { **useState**, **useReducer**, **useEffect**, **Suspense, useRef** } from "**react**"
* import { **useImmerReducer** } from "**use-immer**"
* import { **BrowserRouter**, **Routes**, **Route**, **useNavigate**, **withRouter**, **NavLink**, **Link**, **useParams** } from "**react-router-dom**"
* import **Axios** from "**axios**"
* import **ReactMarkdown** from "**react-markdown**"
* import **ReactTooltip** from "**react-tooltip**"
* import io from **'socket.io-client'**
* import { Container, Row, Col } from "react-bootstrap"
* import { useForm } from "react-hook-form"

File structure

* Index.html //Inside the **public** folder, we have an HTML file that contains the mother tag. With id="root"
* Index.js // in the src folder. It is our main js file that renders the mother component and connects it to the mother tag. All other components are related to it, with siblings or parent-child relations.

import React from 'react';

import ReactDOM from 'react-dom';

import App from './components/App.js';

ReactDOM.render(

<App />

,

document.getElementById('root'));

* App.js // Is the file that holds our mother component and can have any name. It is a class that extends the main component class.

import React, { Component } from 'react'

class App extends **Component** {

**render()** {**return** (<div><h1>Hi Derry</h1></div> ) }}

export default App

* Create a folder app and put index.html and Main.js inside it.

import React from "react"

import ReactDOM from "react-dom/client"

function ExampleComponent() {

return (

<div>

<h1>This is our app!</h1>

<p>The sky is blue.</p>

</div> )}

// which element to render into

const root = ReactDOM.createRoot(document.querySelector("#app"))

// what we want to render, JSX

root.render(<ExampleComponent />)

React without JSX

* When creating an element with react we can only have one top-level element.
* **React.getElementById()**
* **React.createElement ('tag',}className{: 'box'}, innerText)**
* **ReactDOM.render(addTag, InWhatTag)**
* root = **ReactDOM.createRoot(document.querySelector('#app'))**
* **root.render(React.createElement(OurApp))**

const root = ReactDOM.createRoot(document.querySelector('#app'))

root.render(React.createElement("h1", null, "Our Amazing App Header"))

* only the p will get rendered each 1s

function OurApp(){

return React.createElement("h1", null, "Our Amazing App Header")

}

const root = ReactDOM.createRoot(document.querySelector('#app'))

root.render(React.createElement(OurApp))

function OurApp(){

return React.createElement("div", null, [

React.createElement("h1", null, "Our Amazing App Header"),

React.createElement("p", null, `The current time is ${new Date().toLocaleString()}`),

React.createElement("small", null, "Copyright Footer Text") ])}

const root = ReactDOM.createRoot(document.querySelector('#app'))

setInterval(()=>{

root.render(React.createElement(OurApp))

}, 1000)

* **<script type="text/babel">**

<div id="root"></div>

<script **type="text/babel"**>

const root = document.getElementById('root')

const element = React.createElement('h1', {className:'flexBox'}, React.createElement('span', null ,'span'))

ReactDOM.**render**(element, root)

</script>

<script crossorigin src="https://unpkg.com/react@17/umd/react.production.min.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.production.min.js"></script>

<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

* If the script has type="text/babel" you can use something like this:

<div id="root"></div>

<script type="text/babel">

const root = document.getElementById('root')

const element **= <h1 className="box"><span></span></h1>**

ReactDOM.render(element, root)

</script>

JSX syntax

* JSX stands for JavaScript XML. It is simply a syntax extension of JavaScript. It allows us to directly write HTML in React (within JavaScript code). It is easy to create a template using JSX in React, but it is not a simple template language; instead, it comes with the full power of JavaScript.
* It's a syntax extension to js
* You need the **Babel** interpreter
* Return () // () is not needed if we have only one line

Function OurApp(){

Return(

<div>

<h1>Our Amazing App Header</h1>

<p>The current time is {new Date().tolocalString()}}.</p>

<small>Copyright Footer Text</small>

</div>

)

}

* **{" "}** // use as white space. In JSX white spaces between tags are not considered like html.
* **<Header /> or <Header>hi</Header>** // We can call components in these two formats. In the second one, it prints hi before the Header component content. When we want to render them.

const root = ReactDOM.createRoot(document.querySelector('#app'))

setInterval(()=>{

root.render(<OurApp />)

}, 1000)

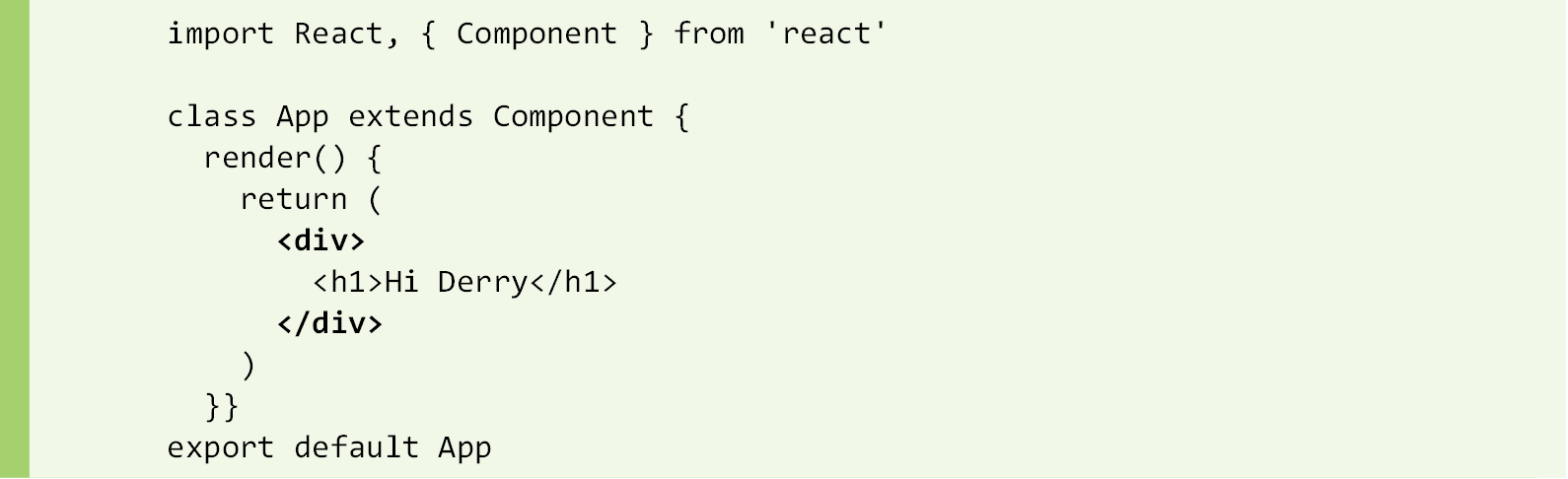
* In json setting file of vs code add: // To use html emmet (html tab trigger) in jsx

"files.associations": {

"\*.js": "javascriptreact"

},

* **<> </>** // **fragment tag**, otherwise, only one top-level tag is allowed. When we don't want a wrapper div. To create a var with html. In recent versions, we can use it as the main returned component tag, before that we had to use fragment tag, then div/span/….



const useme =(

<>

<h1 className="headline headline--large">Welcome!!!</h1>

<h2 className="headline headline--medium">We think you&rsquo;ll like it here.</h2>

<h3 className="headline headline--small">Why don&rsquo;t you check out the <strong>major</strong> you&rsquo;re interested in?</h3>

<a href="#" className="btn btn--large btn--blue">Find Your Major</a>

</>

)

* **/>** //is needed if we don't write the closing tag
* **{}** // make it dynamic, write code
* **expression** // in jsx we can't have if statement only expressions, something that boils down to a value if just controls the flow of the code, use the **ternary** operator
* **{ Condition && ( code ) }** // multiple line of content. Its better to use ternary operator if it is possible, because this may output 0 as the evaluated condition.
  + !props.noAuthor && **<>**by {post.author.username}**</>**

{isCorrectDelayed === true && index == props.correctAnswer && **(**

**<svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" stroke="black" stroke-width="1" className="bi bi-check paying-attention-frontend--icon-correct" viewBox="0 0 16 16" >**

**<path d="M10.97 4.97a.75.75 0 0 1 1.07 1.05l-3.99 4.99a.75.75 0 0 1-1.08.02L4.324 8.384a.75.75 0 1 1 1.06-1.06l2.094 2.093 3.473-4.425a.267.267 0 0 1 .02-.022z"/>**

**</svg>)}**

* **<Header textHead= "Hi" />** // sending a prop to Header.js component. There it can be accessed by this.prop.textHead. It can be functions.

Imports …

export default function UserSingle() {

// To use **useSearchParams** we should use function component. The useSearchParams hook is used to read and modify the query string in the URL for the current location.

const [searchParams, setSearchParams] = useSearchParams();

const userid = searchParams.get('userid')

let user = sampleUsers.find(item => item.id == userid)

const [users, setUsers] = useState(0);

const [bodyContent,setBodyContent] = useState(0);

let updateBody = (content) => {setBodyContent(content)}

const othercontent = <div className='single-user'><div><img src={user.image} /></div></div>

return (

<div>

<Header updateBody = {updateBody} users={sampleUsers} />

<Body content={bodyContent} otherContent={othercontent}>

</Body><Footer /></div> )}

Predefined Properties (Attributes like ClassName)

* for the jsx elements
  + **className = {}** // Since the class is a reserved word use this to give tags classes
  + **style ={{backgroundColor: 'red', color: 'white'}}** // the first curly bracket is doing something dynamic, and the second is the object.
  + **onClick={}**
  + **onSubmit={}**
  + **autoComplete** // in html it is autocomplete
  + **htmlFor** // instead of for in the label tag
  + **autofocus**
  + **onBlur**
  + **defaultValue=""** // Input value => and not value=""

Basic parts (Component, State, Prop)

* We don't use a different HTML and manipulate that HTML with react, the basic concept of react is to be component-based, so you can have the ability to use the same component in another place.

COMPONENT

* We can define a component once and use it again. They are different parts of the HTML tag with a specific purpose and functionality, like the menu. The first letter of components is uppercase. We have only **Static** or only **dynamic** components. Components are either **class-based** or **function-based**. Function-based does not have a **state** concept but has a higher speed. The class has states.
* Single Responsibility
* Components are reusable. Can be rendered multiple times in other places.

function OurApp(){

return(

<>

<OurHeader />

<The current time is {new Date().toLocalString()}}.<p>

<small>Copyright Footer Text </small> </>)}

function OurHeader(){

return <h1>Our Amazing App Header </h1>}

STATE

* The states a component has, like exists or not, is available or not.

PROPS

* Component's characteristics, attributes, and properties. // like how we give normal html attributes.

<Pet name="" species="" age="" />

function Pet(props){

return <li>{props.name} is a {props.species} and is {props.age} years old.</li>}

General Concepts

Redirecting

import { useNavigate } from "react-router-dom"

….

const navigate = useNavigate()

### ….

navigate(`/post/${response.data}`)

Flash Messaging

* We want the success message to be seen no matter where we are directed or click. We put it in the Main component with Router in it.

Images

* Images are modules in React that should be imported.

import myImg from ./../images/img.jpg

…

//in component

<img src={ myImg } />

* You also can put a folder named images in the public folder of react and use the address:

images/img.jpg

Loading Component

* By using …

const [isLoading, setIsLoading] = useState(true)

if(isLoading) return <Page title="...">Loading...</Page>

* Simulate slow connection -> devtools-> network -> no throttling -> slow 3g
* We may see a very slow connection and a blank page. It is because of the automated webpack that injects updated js into the browser.

import React, { useEffect } from "react"

function LoadingDotsIcon(props) {

return (

<div className="dots-loading">

<div></div>

</div>

)

}

export default LoadingDotsIcon

Markdown

* A popular formatting syntax. GitHub uses it. It will convert the syntax to html tags. Check this [link](https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet).
* To style a text //
* \*\* blue\*\* // bold
* \*green\* // italic
* ### Additional Details // header level 3
* - red // bulleted list

Markdown parsing

* Client-side routing we need to do it on the client side.
* **npm install react-markdown**

import ReactMarkdown from "react-markdown" // this may not work

* **Markdown component**
  + **Children**

<ReactMarkdown **children** = {post.body} />

* **allowedTypes = {[]}** // allowed elements

<ReactMarkdown children={post.body} allowedTypes={["paragraph", "strong", "emphasis", "text", "heading", "list", "listItem"]} />

Tooltip package

* Tooltips display informative text when users hover over, focus on, or tap an element. // Check the [link](file:///C:\Users\Alma\AppData\Roaming\Microsoft\Word\import%20%7b%20Tooltip%20as%20ReactTooltip%20%7d%20from%20%22react-tooltip%22)
* npm install --save-dev style-loader css-loader

import "react-tooltip/dist/react-tooltip.css"

import { Tooltip as ReactTooltip } from "react-tooltip"

* <ReactTooltip id="edit" className="custom-tooltip" />
* **data-tooltip-content="Edit"** // the content of hover
* **data-tooltip-id="edit"** // to reference it

<a data-tooltip-content="Edit" data-tooltip-id="edit" href="#" className = "text-primary mr-2" title="Edit">

<i className="fas fa-edit"></i>

</a>

{/\* id matches the data-for \*/}

<ReactTooltip id="edit" className="custom-tooltip" />

* **place** = "bottom"

Url sharable tab

* Use **NavLink** for the tabs, and give it the current URL address.
* **Routes** (Switch) to change the content of the tab, based on the URL
* **When we are in a different component and we use Routes you just use /restofaddress to expand it.**

**<NavLink** **exact=**{true} end **to=**{`**/**profile/${state.profileData.profileUsername}`} className="nav-item nav-link">

Posts: {state.profileData.counts.postCount}

</NavLink>

<NavLink to={`/profile/${state.profileData.profileUsername}**/followers**`} className="nav-item nav-link">

Followers: {state.profileData.counts.followerCount}

</NavLink>

<NavLink to={`/profile/${state.profileData.profileUsername}**/following**`} className="nav-item nav-link">

Following: {state.profileData.counts.followingCount}

</NavLink>

</div>

**<Routes>**

<Route exact **path="/"** **element={<ProfilePosts />**}></Route>

<Route **path="/followers"** element={<ProfileFollowers />}></Route>

<Route **path="/following"** element={<ProfileFollowing />}></Route>

**</Routes>**

Router

* To help with spa // it will update the URL as well
* We don't need to rely on server-side routing. We can use client-side routing.
* Client-side rendering
  + Speed of javascript rendering with the functionality of sharable and bookmarkable URLs (about-us for example)
  + Browser back-and-forth buttons
* Main and side pages // to manage errors inside our site
* **npm install react-router-dom** //
* **switch** command, in router-dom not working // if we use CDN we may not have this problem.

How to use it

**Way 1**

* with react-router, we need to wrap our application within react-router
  + This is the main component that **root.render()** gets

function Main() {

return (

// with react router we need to wrap our application within reactRouter

<BrowserRouter>

{/\* the header and footer does not change keep them out of Routes \*/}

**<Header />**

<Routes>

<Route path="/" element={<HomeGues />} />

<Route path="/about-us" element={<About />} />

<Route path="/terms" element={<Terms />} />

</Routes>

**<Footer />**

</BrowserRouter>

)

}

**Way2**

* Index.js renders <Router /> into root div -> Router has routes set for home on <App element /> -> <App /> and other pages all have the header and footer

<div id="root">World</div>

* Make a component named router
* In index.js what is passed as the main component to the render is the router
* We can make a separate folder for the components
* By using a router we can say which page is the main, which is the error page, and which are side pages.
* By routing, we assign different components to different pages.

In the router, we can have

* Main page
* Manage errors: Not found: 404-not found (error set 4), error set 5 (very bad) // [ref\_httpmessages.asp](https://www.w3schools.com/tags/ref_httpmessages.asp) (there are more than what is shown in this link). 403 is for when pages exist, but we are not authorized to access it.
* About us, gallery, product // each is a different page
* If a certain URL is called, you can define to execute what component for that URL. You can have different pages and connect them to a component.

Router.js component

Imports

* **import react {component} from 'react'**
* **import {BrowserRouter, Switch, Route} from 'react-router-dom'** // to use Router
* **import {BrowserRouter, Routes, Route} from 'react-router-dom'**
* You have to **import the components** that you want to route to.
  + In **V6 t**he **switch is replaced** with **Routes**
  + **<BrowserRouter>** // creates a router on the browser

Route Component

* **<BrowserRouter> <Switch> <Route /><Route></Switch></BrowserRouter>** // return this in the render method.
  + We can have more than one switch but it gets complicated
* **<Route** **exact** path="/ " **Component**={test} /> // exact means main, only one in one page

Route Props

* **exact** // use this keyword
* **path="/"** // main page

<Route path="/" element={<Home />}></Route>

* **path="\*"** // for routing the rest of URLs as a whole

<Route **path="\***" element={<NotFound />} />

* **"/post/:id"** // id is like variable

<Route path="/shop/:shopId" element={} /> // If this is the address and whatever comes after :, direct to the defined component

* **element** // In **V6**, you **can't use** the **component prop anymore**. It was replaced in favor of the **element**
  + **Component={test}** // in older versions. call this component, which can be different sections or different pages

Navigate to a new URL automatically

* **import {useNavigate, withRouter} from 'react-router-dom'**
* **useNavigate**

navigate(`/post/${response.data}`)

const navigate = useNavigate()

* **withRouter** // needs the props since withRouter gives the state. It is **removed** in v6. use [history](https://stackoverflow.com/questions/42701129/how-to-push-to-history-in-react-router-v4) or hooks: useLocation(); useNavigate(); useParams();

**export default withRouter(EditPost)** // EditPost the name of the component

**Props.history.push("/")**

Route hooks and Components

Components

* **BrowserRoute**
* **Routes**
* **Route**
* **NavLink** // Similar to the link but it adds an active or selected class to the matching link. (Visually highlighted or selected tab). Based on the URL it handles which NavLink is active, and adds an active class to it. It also helps with urls, so urls have a sharable address for each tab.
  + **end** // instead of exact in new versions. It helps with activating the NavLink correctly in similar routes

<div className="profile-nav nav nav-tabs pt-2 mb-4">

<NavLink exact to={`/profile/${state.profileData.profileUsername}`} className="nav-item nav-link">

Posts: {state.profileData.counts.postCount}

</NavLink>

<NavLink to={`/profile/${state.profileData.profileUsername}/followers`} className="nav-item nav-link">

Followers: {state.profileData.counts.followerCount}

</NavLink>

<NavLink to={`/profile/${state.profileData.profileUsername}/following`} className="nav-item nav-link">

Following: {state.profileData.counts.followingCount}

</NavLink>

</div>

* **Link**

hooks

* **useNavigate()** // instead of useHistory in new versions. To redirect
* **withRouter()**
* **useParams()**
* **useLocation()** // get the current location that is set by Route
  + **location.pathname**

Parent and descendent Routes

* **Relative addressing children** // If you want to define new routes in a route, the paths should be relative
* **/\*** // The parent route should have /\* at the end of it path.

<Route path="/profile/:username/\*" element={<Profile />}>

* In a child component of that route. The addresses are relative

<Routes>

<Route path="/" exact element={<ProfilePosts />}></Route>

<Route path="/followers" element={<ProfilePosts />}></Route>

<Route path="/following">{/\* <ProfileFollow action="following" /> \*/}</Route>

</Routes>

Link component

* **import { Link } from "react-router-dom"** // <a> tag is going to ask for an entirely new html document from the server. This is for server-side routing. Instead, we use Link.
  + To // instead of Href

<BrowserRouter>

{/\* the header and footer does not change keep them out of Routes \*/}

<Header />

<Routes>

<Route path="/" element={<HomeGues />} />

<Route path="/about-us" element={<About />} />

<Route path="/terms" element={<Terms />} />

</Routes>

<Footer />

</BrowserRouter>

)

import { Link } from "react-router-dom"

…

<Link to="/" className="mx-1">

Home

</Link>

* The link and router will create an illusion for URLs, if we actually enter the URL on the browser address with about us, it will cause an error
  + **historyApiFallback: { index: "index.html" }** // No matter what URL is given send the index.js so our route can handle the relevant content for the URL. In **webpack.config.js:**

devServer: {

port: 3000,

static: {

directory: path.join(\_\_dirname, "app"),

},

hot: true,

liveReload: false,

**historyApiFallback: { index: "index.html" },**

},

Composition

* **composition** // **Components using other components** and passing props to them. For example, wide here does not come from the page, but from the component that calls it. A specific component uses a general Component like Page. To avoid repetition, instead of inheritance, the creator of react suggests a composition.
* The page title on the browser tab changes when we have a response from the server.
  + **document. title** // js

import React, { useEffect } from "react"

import Container from "./Container.js"

function Page(props) {

useEffect(() => {

**document.title** = `${props.title} | ComplexApp`

window.scrollTo(0, 0)

}, [])

return <Container wide={props.wide}>{props.children}</Container>

}

export default Page

Props

* + When you want to call it

<User name="" />

<User name={javascript code} />

* In the component pass it as an argument in the function component or constructor of the class component, then you can use it

// in function component

porps.name

// in class component

this.porps.name

* **props.children** (or read all props as an object and mention children too {x,y,z,children}) //give all the nests jsx content. It gives everything inside the component wherever it is used.

import React from "react"

function Container(props) {

return(

<div className="container container--narrow py-md-5">

{**props.children**} // it will give all the h2 p and …

</div>

)

}

export default Container

import React, { useEffect } from "react"

import Page from "./Page"

function About() {

return (

<Container>

**<h2>About Us</h2>**

**<p className="lead text-muted">Lorem que?</p>**

**<p> sed veniam?</p>**

**<p>Lorem, te!</p>**

</Container >

)}

export default About

State Change

* Events that can change the state
  + Time
  + Click on button
  + Type letters
  + Scroll
* We **don't want to modify the state directly**. You give the react the desired value, and react will handle it.
  + **Push** for example in an array will modify it directly
  + **Concat** // will return a new array

Function components

* Every code in the component function will run again every time a component is rerendered
  + If you need to run something at specific times use useEffect
* **const useState = React.useState** // to don't write React.useState. In codepen for example.
* **React.useState(default value)** will return an array, the first easy access to state, second a method to change it. It will store the data in a memory cell when the page is loaded. When the state changes react is going to run the component(function) again, It won't run the useState declaration again, just once.
* **It returns two values:** Destructure it
  + A var to access the value of the state
  + A function that we can update the value

import useState from React

function Quiz(props){

const [time, setTime] = **useState(new Date().toLocalString())**

**setTimeout(**

**()=> {**

setTime (**new Date().toLocalString())}**

**,1000)**

**return <p>The current time is {** **time }** </p>}

* setLikeCount ((prev)=>{return prev + 1}) // prev is given

import useState from React

function Quiz(props){

const [isCorrect, setIsCorrect] = **useState(undefined)**

**…**

Class components

* The **constructor** can be used to change the state and props
* this.set**State**(()) // it's a react setter that sets a property named **state** (**user defined**)

state = {

toturial:{},

order:{}

}

loadSample =()=>{

this.setState(()=>{

return{toturial:sample}})}

* The super (and override) is usually common in all programming languages.
* Using the constructor function and **prevState** keyword

import React, { Component } from 'react'

class App extends Component {

**constructor**(**props**){//it says don't change anything in the props and inherit from parent

**super**(**props**);

this.state = {

counter : 0 }}

plus = () => {

this.**setState**((**prevState**) => {

**return** {

counter : prevState.counter + 1 }});}

minus = () =>{

this.setState((prevState) => {

return {

counter : prevState.counter – 1 }})}

reset = () =>{

this.setState(() => {

return {

counter : 0 }})}

render() {

return (

<div>

<button onClick={this.plus}>plus</button>

<h2>counter : {this.state.counter}</h2>

<button onClick={this.minus}>minus</button>

<hr />

<button onClick={this.reset}>Reset</button>

</div>)}}

export default App;

Collection State

* **Concatenate** // A state that is a collection or an array cannot be directly set.
* **Props** // to send a set method of a collection state as a prop to another component we need to write a function that uses concatenation and pass that new function.

const [flashMessages, setFlashMessages] = useState([])

function addFlashMessage(msg) {

setFlashMessages((prev) => prev.concat(msg))

}

Closure problem

* Adding two states after each other will not work. The value of is correct is the same for both lines (false). SetState is **asynchronous** by design.
* In Class components, we can use the callback, as a second argument to setState to call another setState after the first one finishes.

import React, { Component } from 'react';

class App extends Component {

constructor(props) {

super(props);

this.state = {

age: 0,

};

}

// this.checkAge is passed as the callback to setState

updateAge = (value) => {

this.setState**({ age: value}, this.checkAge);**

};

…

* Unlike class, the component's this.setState, function component's useState state updater function doesn't take a second callback argument to be called after any state update.
  + **useEffect hook** // still you can't setState another time with animation, because the change will be very quickly

function handleAnswer(index){

if(props.correctAnswer == index){

setIsCorrect(true)

}else{

if(isCorrect == false) { setIsCorrect((prevState) => {isCorrect: undefined}) }

setIsCorrect(false)

message').classList.add('incorrect-message\_\_visible')

}

}

React.useEffect(() => {

alert(isCorrect) **// order of alert: undefined - true/false - undefined**

if(isCorrect === true || isCorrect === false){

**setIsCorrect(undefined)**

}

}, [isCorrect]);

* with animations use setTimeout

React.useEffect(() => {

// alert(isCorrect) // order of alert: undefined - true/false - undefined

if(isCorrect === false){

setTimeout(() => {

setIsCorrect(undefined)

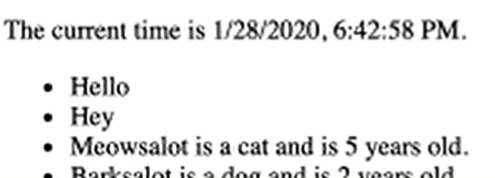
}, 2600);

}

}, [isCorrect]);

Loop Through Collections in JSX

* **{[<li>Hello</li>, <li>Hey</li>]}** // This is already javascript {}, we don't need to convert it to string. This is why jsx is good. We don't need .join to convert it to a string. We can use javascript {} and inside it an array of elements or components. The example just like that in the return () will be shown like:



* **map** // every array in js has access to this. It will return a new array. We use it to return an array of elements and components. And it will be shown like the above. We can use it to read an array and assign the html elements we want to each and we will have an array of the html elements with their content and values.
* **Key** // for internal react performance reasons. To track each instance of a component.

function OurApp(){

return (

<>

<OurHedaer />

<ul>

{Pets.map(pet=> <Pet name={pet.name} specieies={pet.species} key={pet.id} /> )}

</ul>

</>

)}

const pets = [

{ name: "Meowsalot", species: "cat", age: "5", id: 456486953},

{ name: "Barksalot", species: "dog", age: "3", id: 456490456 },

{name: "Fluffy", species: "rabbit", age: "2", id: 456856456},

{name: "Purrsloud", species: "cat", age: "1", id: 456456456},

{name: "Paws", species: "dog", age: "6", id: 789789789 }]

Forms and Input Values

* In react the dom is never the source of truth, we don't manually read the values from inputs. **We listen to the event that the value of the input changes!** **Every time it changes we store the new value in a state. So after submission, we don't retrieve the value from the dom we use the state.**
  + **onChange = {handleSubmit} // it will pass the e to the function handleSubmit**
  + **e.preventDefault()**

const [name, setName] = useState()

const [species, setSpecies] = useState()

const [age, setAge] = useState()

* **Clear the text inputs after custom logic in onChange callback** // for user convenience to add new values

setName("")

setSpecies("")

setAge("")

* + **Then you need to update the values**

<input **value = {name}** onChange ={e => setName(e.target.value)} placeholder="Name" />

* **Controlled input** // By giving a value to the input we can't change the value anymore we need the onChange prop.

Example

* this example needs some modification to work -> adding components like Pet

const useState = React.useState

function AddPetForm(props){

**const [name, setName] = useState()**

**const [species, setSpecies] = useState()**

**const [age, setAge] = useState()**

function handleSubmit(e){

//by deafult the browser is going to referesh the page. It's a javascript method

**e.preventDefault();**

// since the name of the value and key are the same just write one

**props.setPets(prev => prev.concat({name, species, age, id: Date.now()}))**

**setName("")**

**setSpecies("")**

**setAge("")}**

return (

<form onSubmit={handleSubmit}>

<fieldset>

<legend>Add New Pet</legend>

<input **value = {name} onChange ={e => setName(e.target.value)}** placeholder="Name" />

<input value = {species} onChange ={e => setSpecies(e.target.value)} placeholder="Species" />

<input value = {age} onChange ={e => setAge(e.target.value)} placeholder="Age In Years" />

<button>Add Pet</button>

</fieldset></form>)}

function OurApp(){

const [pets,setPets] = useState([

{name: "Meowsalot", species: "cat", age: "5", id: 456486953},

{name: "Barksalot", species: "dog", age: "3", id: 456490456 },

{name: "Fluffy", species: "rabbit", age: "2", id: 456856456},

{name: "Purrsloud", species: "cat", age: "1", id: 456456456},

{name: "Paws", species: "dog", age: "6", id: 789789789 }])

return (

<>

<OurHeader />

<AddPetForm **setPets={setPets}** />

<ul>

{pets.map(pet=> <Pet **name={pet.name}** **species={pet.species}** **age={pet.age}** key={pet.id} /> )}

</ul></>)}

An example

import { useState } from 'react';

export default function App() {

const [message, setMessage] = useState('');

const handleChange = (event) => {

// 👇 Get input value from "event"

setMessage(event.target.value);

};

return (

<div>

<input type="text" id="message" name="message" onChange={handleChange} />

<h2>Message: {message}</h2>

</div>

);}

import React, { Component } from 'react'

class Search extends Component {

state = { foundItems:[] }

loadFounds=(**e**)=>{

const userName = **e.target.elements.userName.value**

e.preventDefault()

this.searchResult(userName); }

searchResult = (userName) => {

let {updateBody, users} = this.props

const foundItes = users.filter((item)=>{

return item.name.includes(userName) })

updateBody(foundItes) }

render() {

return (

<div className='search'>

<div style={{width: "30%"}} >

<form className='search-form' **onSubmit={this.loadFounds}** action=''>

<input name="userName" id="user-name" type="text" placeholder="Please write the item's name"/>

<input type="submit" value="search"/>

</form>

</div>

</div>

) } }

export default Search;

useRef hook

* a box that holds the value in it and we can mutate it directly. It is used when we don't have an option but access the dom directly. We still don't use documents.getElemenet but we use this hook that react offers us. For example, we want to focus on input inside a component that does not add or remove to the dom, so we cant use html prop focus. We need to tell exactly what to do and it's not like onClick or onChange, to get the value. How can we access it? By userRef.
* **useRef()**

import {useRef} from "react"

useEffect(() => {

if (appState.isChatOpen) {

// current is the actual value of the ref. treat chatField.current like any other dom element

chatField.current.focus()

}

}, [appState.isChatOpen])

const chatField = useRef(null)

<input onChange= { handleFieldChange} ref={chatField} type="text" className="chat-field" id="chatField" placeholder="Type a message…" autoComplete="off" />

* ref = " " // a command in reactJs, that helps to (input, button) to get its value and data. To reference somewhere.
* **React.createRef()** // it creates an address for form input. So we can send and access it in other components. This method is used to access any DOM element in a component and it returns a mutable ref object which will be persisted as long as the component is placed in the DOM. If we pass a ref object to any DOM element, then the current property to the corresponding DOM node elements will be added whenever the node changes.

import React, { Component } from 'react'

class Shop2 extends Component {

**myInput** = **React.createRef();**

goToShop=(e)=>{

console.log(this.props);

e.preventDefult();

const urlPath=**this.myInput.current.value**;

**this.props.history.push**(`/shop/${urlPath}`)}

render() {

return (

<form className='shop-form'onSubmit={this.goToShop} >

<h2>enter shop name</h2>

<input required **ref={this.myInput}** type="text" placeholder='shop name'/>

<button type='submit'>visit shop</button>

</form>

)

}

}

export default Shop2;

Validation

* **Reduce** // when we want to work with complex logic
* Include the logic of the form in the reducer.
* Don't put the Axios request in the reducer. It is not a react functionality.
* Put these in the reducer
  + A dispatch type that sets a state for disabling submits button after click and another one when the request axios.post is done to enable it again. By the state=ture/false we can disable the button.
    - **disabled={state.isSaving}**
  + A dispatch type for validating inputs
    - **onBlur** // you can use onblur to show the validation errors. You give it a new dispatch type for validation rules. It is activated when you click out of the input.

<input onBlur={e=>dispatch({type: "titleRules", value: e.target.value})}

* Do not forget the returns in the reducer switch/case

const originalState = {

title: {

value: "",

hasErrors: false,

message: "",

},

body: {

value: "",

hasErrors: false,

message: "",

},

isFetching: true,

// disable save button so a second request is not sent

isSaving: false,

id: useParams().id,

// how many time we sent axios request. the useEffect axios can watch this for change.

// with this we don't have to include axios n=in the reducer

// sending a request like that is a side effect that is not related to react

sendCount: 0,

}

// action is what we send in dispatch({}). draft a copy of state

// don not forget the returns!!!

function ourReducer(draft, action) {

switch (action.type) {

case "fetchComplete":

draft.title.value = action.value.title

draft.body.value = action.value.body

draft.isFetching = false

return

case "titleChange":

// when the title changes set errors to false

draft.title.hasErrors = false

draft.title.value = action.value

return

case "bodyChange":

// when the title changes set errors to false

draft.body.hasErrors = false

draft.body.value = action.value

return

case "submitRequest":

if (!draft.title.hasErrors && !draft.body.hasErrors) {

draft.sendCount++

}

return

case "saveRequestStarted":

draft.isSaving = true

return

case "saveRequestFinished":

draft.isSaving = false

return

case "titleRules":

if (!action.value.trim()) {

draft.title.hasErrors = true

draft.title.message = "You must provide a title"

}

return

case "bodyRules":

if (!action.value.trim()) {

draft.body.hasErrors = true

draft.body.message = "You must provide body content."

}

}

}

Another form validation example

* Put your validation logic inside the Reducer Function

function ourReducer(draft, action) {

switch (action.type) {

case "usernameImmediately":

**// at first we assume there is no error so we can store it**

draft.username.hasErrors = false

draft.username.value = action.value

if (draft.username.value.length > 30) {

draft.username.hasErrors = true

draft.username.message = "Username cannot exceed 30 characters."

}

**// include non-alphameric characters. test method available for the regular expression**

if (draft.username.value && !/^([a-zA-Z0-9]+)$/.test(draft.username.value)) {

draft.username.hasErrors = true

draft.username.message = "Username can only contain letters and numbers." }

return

case "usernameAfterDelay":

if (draft.username.value.length < 3) {

draft.username.hasErrors = true

draft.username.message = "Username must be at least 3 characters."

}

// **increase the count to send the request in another useEffect**

// action.noRequest: we don't check isunique if we just submit instead of typing in the field.!action.noRequest is true when action.noRequest is false, and it is false when that value does not exist which means when we wait after the first key strokes(write something in input) and ask for a dipatch to check for initial errors in a useEffect, then we add the checkCount to check the uniquness after a while action.noRequest is not set in the dispatch after typing so it is false

if (!draft.username.hasErrors && !action.noRequest) {

// check for uniqueness

draft.username.checkCount++

return

case "usernameUniqueResults":

if (action.value) {

draft.username.hasErrors = true

draft.username.isUnique = false

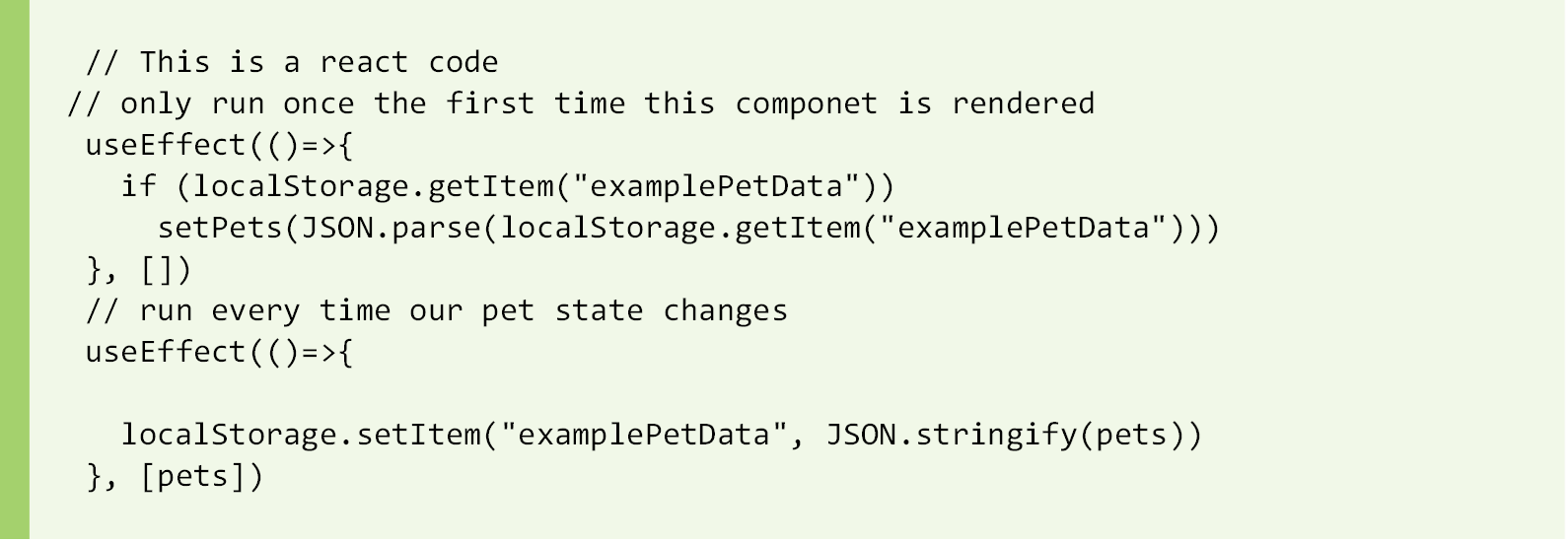
draft.username.message = "That username is already taken."

} else {

draft.username.isUnique = true}

return …

useEffect

* When we want to run something at **specific times**, not every time the function is rerendered. For example, sending a request to the server, but we don't want that to happen every time the state changes and components are rendered again. Also when we want to work with something that is not react-related, such as eventListener, Axios, setTimeout, ….
* **import React, {useEffect} from "react"**
* **Closure problem**
* **setTimeout and setInterval** // we don't want an ongoing number of intervals going since if the state changes the. Only call setInterval once when the component first time that is rendered.

function TimeArea(){

const [theTime, setTheTime] = useState(new Date().toLocaleString())

// we don't want ongoing number of intervals going since if the state changes the

useEffect(()=>{

const inteval = setInterval(()=> setTheTime(new Date().toLocaleString()), 1000)

return () => clearInterval(interval)

},[])

return <p>The current Time is {theTime}. </p>}

* **useEffect clean up function** // Clear the effects. Whatever we give to the return of useEffect will be used as the cleanup function. When we do something in the future (intervals) we want to cancel or negate it if the component is removed or unmounted from the page before the action is complete. For example, we navigate to a new screen that doesn't need to show the time.
  + Clear setInterval

Changing dom directly or working with a browser ( like Local Storage)

* When we want to do something with a side-effect of not using react, and work with dom or browser local storage(), we can use useEffect()
* Keep updating, and deleting data for local Storage in one place. In Main.j
* No component needs to access the data from the localStorage anymore, can use a state.

const initialState = {

loggedIn: Boolean(localStorage.getItem("complexappToken")),

flashMessages: [],

// No component need to access the data from the localStorage, can use this.

user: {

token: localStorage.getItem("complexappToken"),

username: localStorage.getItem("complexappUsername"),

avatar: localStorage.getItem("complexappAvatar"),

},

}

* On successful login

appDispatch({ type: "login", data: response.data })

* in Main.js

useEffect(() => {

if (state.loggedIn) {

localStorage.setItem("complexappToken", state.user.token)

localStorage.setItem("complexappUsername", state.user.username)

localStorage.setItem("complexappAvatar", state.user.avatar)

} else {

localStorage.removeItem("complexappToken")

localStorage.removeItem("complexappUsername")

localStorage.removeItem("complexappAvatar")

}

}, [state.loggedIn])

Cleanup function

* Updating States when the component is unmounted. For example, while executing Axios we go to another page. But cleanup is not limited to async functions, but when the component is unmounted. For example, full-screen overlay when the user presses a key to close it, but when it is no longer available we want to remove the key action.
* It also will run whenever the useEffect is executed again. Before that useEffect is called again.
* When using await in an useEffect we have to clean up after ourselves.
* A function that useEffect returns are the cleanup function that runs when the component is unmounted.
* **const ourRequest = Axios.CancelToken.source() // Generates a token that can be used. Cancel token**. An identifier for the specific Axios.
* **Axios.get(`/post/${id}`, {cancelToken: ourRequest.token**}) // in the post request it would be the third argument.
* **ourRequest.cancel()**
* **The catch block will be triggered nonetheless.**

useEffect(() => {

**const ourRequest = Axios.CancelToken.source()**

console.log("ok")

async function fetchPost() {

try {

console.log("ok")

const response = await Axios.get(`/post/${id}`, **{ cancelToken: ourRequest.token })**

console.log(response.data)

setPost(response.data)

setIsLoading(false)

} catch (e) {

console.log("There was a problem or the request was **cancelled**.")

}

}

fetchPost()

**return () => {**

**ourRequest.cancel()**

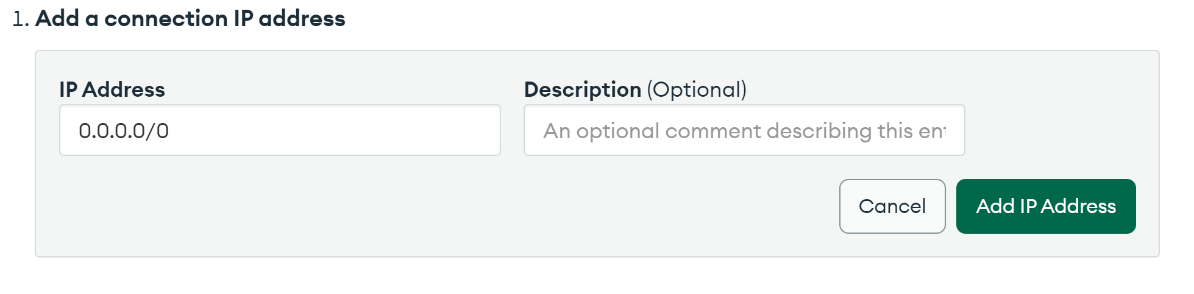
**}**

}, [])

Connecting to server

MongoDB

* Search for mangoDB -> create account -> select database -> collections -> add my own data -> add db and collection (like users table)
* Connect -> database tab -> connect -> add a different Ip address (since we want to move our app to the server)
* Address
  + 0.0.0.0/0 // let any ip address connect to db as long it knows the secure password and username.



* MongoDB user for individual dbs.
* Choose a connection method -> connect your application
* **Select your driver and version // the backend language**
* **connection string // Connect to our db** 
  + Add the password and db name to the string in the exact places

+srv://mainUser:**J@6jrdB\_8VBuMxP**@cluster0.0yxazuk.mongodb.net/**ReactCourse**?retryWrites=true&w=majority

* Indexes // When we want to execute a search on MongoDB we need to do indexing. Got to the collection you want -> indexes tab

{

"title": "<text>",

"body": "<text>"

}

Backend with node.js

* In the root of the backe-end folder in the example: a file named .env // no space between the equals sign
  + JWTSECRET // A secret phrase
  + / ? # [ ] @ // You may need to change these characters to their corresponding **encoded format**. Check this [Link](https://www.mongodb.com/docs/atlas/troubleshoot-connection/#special-characters-in-connection-string-password).
  + **UnhandledPromiseRejectionWarning:** MongoServerSelectionError: getaddrinfo ENOTFOUND cluster0.0yxazuk.mongodb.net // if this error shows try with another ip, turn off your VPN

# enviornment variables

CONNECTIONSTRING=mongodb+srv://mainUser:J@6jrdB\_8VBuMxP@cluster0.0yxazuk.mongodb.net/ReactCourse?retryWrites=true&w=majority

PORT=8080

JWTSECRET= mup$ersecretL22/0phras6546e

* **npm install** // In the command line. To install the packages needed from package.json
* **npm run start**
  + **localhost: 8080** // see if it's working

An alternative way of setting the backend to only accessible in the local

* Docker desktop // install
* Use the file in the resources of the course called docker-compose.yml.
* Create a folder named something, and put the file inside it. Open in vs code:
* **docker compose up -d** // run in a detached mode so that we won't see the output of all actions in the command line.
  + docker compose stop
  + docker compose start
  + docker GUI app // if it runs on start up, you don't need to start it manually
* db folder has the mongo db data locally
* **MongoDB Compass** // To visualize the data
  + **Msi version windows** // easier
  + In the app connection string: mongodb://root:root@localhost/ourApp

Connect react to backend

* Run both in command
* When the form is submitted we send the data to the backend server
  + Store in MongoDB
* In HomeGuest Component

<form onSubmit={}>

* Fetch // Modern web browser native tool for sending an asynchronous request
* Axios // A library

**Fetch**

* APIs
* If we put the API data in a variable is harder than setting states for it.
* Fetch(object)

State={User: []}

componentDidMount(){

fetch('https://randomuser.me/api').then(response => {

response.json().then(data => {

this.setState({users:data.results})

});})} // api url}

render() {

console.log(this.state.user.name)

return

{this.states.users.**map**((user,index)=>

<li key={index}>

<h2>

{User.name.first}{User.name.last}

</h2>

<img src={user.picture.large} alt={user.name.first}/>

</li>)}}

//app

State={

number=0

newNumber = }

inputChange=(e) =>{

this.setState({number: Number(this.state.number)})}

Show=(e)=>{

e.preventDefault(); // The preventDefault() method of the Event interface tells the user agent that if the event does not get explicitly handled, its default action should not be taken as it normally would be.

This.setState({newNumber:Number(this.state.number)})}

render(){const users = []

for(i=0; i<this.state.newNumber; i++){

users.push(<User/>);}

return

<div> <form>

<input type ="text" onchange={this.inputChange}>

<button onClick={this.show}>click me </button>

</form>

<div>{users}</div></div>}

**Axios**

* **npm install axios**
* To avoid url repetition // In Main.js with the router
  + **import Axios from "axios"**
  + **Axios.defaults.baseURL** = "http://localhost:8080"
* Axios.post(url, {})
* Security // the cryptographically secure token that we saved, we can send it as part of the post object of Axios
  + await Axios.post("/create-post", { title: "test", body: "test content", **token: localStorage.getItem("complexappToken") })**

import Axios from 'axios' // 'axios' lower case beceaus it is the npm package

function HomeGuest() {

**async** function handleSubmit(e) {

// asynchronously

e.preventDefault()

// http://localhost:8080/: where backend lives

// register the action or url we are interested in

**try {**

// when using await we need to be in an async function

**await Axios.post("http://localhost:8080/register", { username: "test", email: "test@test.com", password: "qwerty123456" })**

console.log("success")

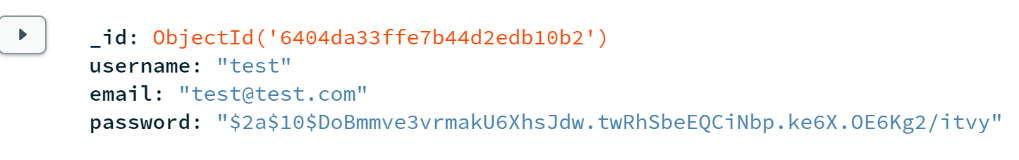
} catch (e) {

// () is needed for microsoft edge

console.log("error")

}} …

* Check the db
  + Users should be unique



inputs

* Create states

<input onChange={(e) => setPassword(e.target.value)} id="password-register" name="password" className="form-control" type="password" placeholder="Create a password" />

const { username, setUsername } = useState()

const { email, setEmail } = useState()

const { password, setPassword } = useState()

Menu login change

* Different logged-in and logged-out components and a state which is set to false or true
* localStorage.setItem("complexappUsername", response.data.username)

{loggedIn ? <HeaderLoggedIn setLoggedIn={setLoggedIn} /> : <HeaderLoggedOut setLoggedIn={setLoggedIn} />}

* Based on the response in try-catch

async function handleSubmit(e) {

e.preventDefault()

try {

const response = await Axios.post("http://localhost:8080/login", { username, password })

if (response.data) {

// save the properties in the browser local storage

localStorage.setItem("complexappToken", response.data.token)

localStorage.setItem("complexappUsername", response.data.username)

localStorage.setItem("complexappAvatar", response.data.avatar)

props.setLoggedIn(true)

} else {

console.log("Incorrect use pass")

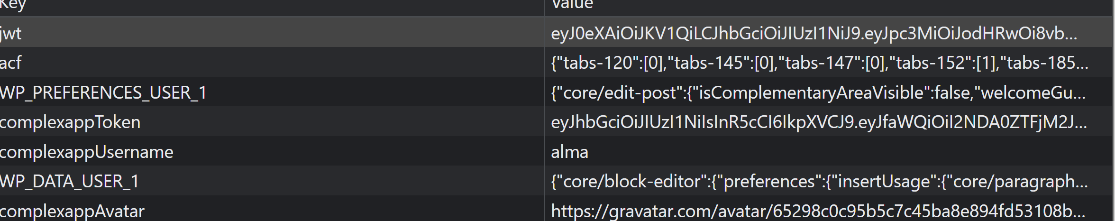
}

} catch (e) {

console.log("There was a problem.")

}}

* localStorage.removeItem("complexappToken") // Onlogout



Redux

* Redux is an open-source JavaScript library for managing and centralizing the application state. It is most commonly used with libraries such as React or Angular for building user interfaces. Similar to Facebook's Flux architecture, it was created by Dan Abramov and Andrew Clark.
* A framework of Javascript (2015). // for the web. On state manages the program.
* It's a pattern for better performance of the project.
* It's a complementary tool to use with react and angular.
* Best project managers: git, git lab, Msp, s6,
* Flusk => action, store (multiple stores), the dispatcher (redux doesn't have this).
* Store, middleware, reduce // libraries
* **Action**: a signal that transfers data to db.
* **Reduce** (dispatcher): States of the program what to do on data received. Store and action relation. Reduce is the connection between action and store. It filters.
* Store: Object.
* Debugging is easier in redux because of its structure.
* It works on enterprise projects.
* <https://www.w3schools.blog/redux-reactjs>
* <https://redux.js.org/introduction/getting-started>

install

* npm install redux react-redux -save // from w3schools
* If you want to install from the redux site: you need three steps.

npm install redux

npm install @reduxjs/toolkit

npx create-react-app folderName --template redux // This create a react app with a redux emplate

* Npm start // like running react app

Lifting state up - Move state up

* In simple applications, this method works
* move your state up into the component tree, and then if in any subcomponent pass it as necessary
* pass the state manually as props to another component// the traditional way of doing it
* Move the logged-in state to the Main component that has the router

const [loggedIn, setLoggedIn] = useState(Boolean(localStorage.getItem("complexappToken")))

<Route path="/" element={loggedIn ? <Home /> : <HomeGuest />} />

More complex applications

* Lifting the state up is not the best method.
  + **Redux** // in the most app we don't need these third-party state manager libraries
  + **Native state management react tools**
  + **Context**
  + **Use Reducer**

Context

* Make it **easy to share some data through our application**. In some cases, we have to pass a state to different components and or **in many levels** nested for example 4 components to eventually be used.
* A container – Wrapper component that all their nested children can use its states.
* A file named ExampleContext.js is in the root of the app folder. It is **not a traditional react Component**. We create a different file so we can access it everywhere.
* **createContext:**

import { createContext } from "react"

const ExampleContext = createContext()

export default ExampleContext

* in Main.js with the route add

import ExampleContext from "./ExampleContext"

* **ExampleContext.Provider** // add it as a wrapper component
  + Anything inside the provider can consume the values
  + **value** // A prop with exactly this name and the value of the state or setState that we want to pass. **the value can be: Object, string, function, …**

**<ExampleContext.Provider value={addFlashMessage}>**

<BrowserRouter>

{/\* the header and footer does not change keep them out of Routes \*/}

<FlashMessages messages={flashMessages} />

<Header loggedIn={loggedIn} setLoggedIn={setLoggedIn} />

<Routes>

<Route path="/" element={loggedIn ? <Home /> : <HomeGuest />} />

<Route path="/post/:id" element={<ViewSinglePost />} />

{/\* without Context \*/}

{/\* <Route path="/create-post" element={<CreatePost addFlashMessage={addFlashMessage} />} /> \*/}

{/\* with Context \*/}

<Route path="/create-post" **element={<CreatePost />}** />

<Route path="/about-us" element={<About />} />

<Route path="/terms" element={<Terms />} />

</Routes> <Footer /> </BrowserRouter>

**</ExampleContext.Provider>**

* **useContext** // Where we want to use that state. It will look up the component tree, and finds the closest ancestor that uses the given context provider no matter how component-nested deep we are!

import React, { useEffect, useState, useContext } from "react"

* inside the component

const addFlashMessage = useContext(ExampleContext)

* **multiple values**

<ExampleContext.Provider value={{addFlashMessage, setLoggedIn}}>

…

* + **Destructure {}** // the value, use the {}

const {addFlashMessage} = useContext(ExampleContext)

useReducer

* Check useImmerReducer as an alternative that creates a draft of the state for easy modifications.
* Keep all of our logic in one centrally located place
* useReducer is like a sibling to useState // When we have **complex state logic** that involves multiple sub-values.

import React, { useState, useReducer } from "react"

* **const [state, dispatch] = useReducer(a function, default state value)**
* **dispatch** // dispatch is not as simple as setState but similar
  + Perform different actions
  + **It tells us just what to do**, but how is done, if useState is used, all happens in ourReducer function.
  + Instead of using different useStates, combine them and use useReducer
  + New state value // Any other property can be added to dispatch

const [state, dispatch] = useReducer(ourReducer, initialState)

const initialState = {}

function ourReducer() {}

// dispatch is not a simple as setState but similar

const [**state, dispatch**] = useReducer(ourReducer, initialState)

dispatch({ type: "login" } // don’t usually use it here, we just pass the dispatch to other components, by context for example

* **dispatch({ type: "login" }** // dispatch a command, it is used as action in ourReducr.
* **Switch** // standard way of handling actions.
* We have our states in one central place and we only pass the state and dispatch variables to context value and we can use them in any Component.
* To **change the state values** in useReducer we need to set all of our states based on their new or prev values as needed. We return the new value as a whole. In immer we can directly change the draft that is provided.

const initialState = {

loggedIn: Boolean(localStorage.getItem("complexappToken")),

flashMessages: [],

}

function ourReducer(state, action) {

//state here is the current or prev state

//cant directly change the state

switch (action.type) {

case "login":

**return { loggedIn: true, flashMessages: state.flashMessages }**

case "logout":

return { loggedIn: false, flashMessages: state.flashMessages }

case "flashMessage":

return { loggedIn: state.loggedIn, flashMessages: **state.flashMessages.concat(action.value**) }

}

}

const [state, dispatch] = useReducer(ourReducer, initialState)

* **Not an optimal way to use the useReduce in context** // Every time the state changes, each component inside the context will be rendered to make sure they use the current state. Not all components need access to the state.

<ExampleContext.Provider value={{ state, dispatch }}>

* **Solution? // react suggests using two Context.Provider, one for the state, one for dispatch**

<StateContext.Provider value={state}>

<DispatchContext.Provider value={dispatch}>

<BrowserRouter>

…

To use dispatch and state

* In the same file as we define useReducer

<FlashMessages messages={**state.flashMessages**} />

* dispatch // Import the required Context

import DispatchContext from "../DispatchContext"

function HeaderLoggedIn(props) {

**const appDispatch = useContext(DispatchContext)**

function handleLogout() {

**appDispatch({type: "logout"})**

### ….

* State

**import StateContext from "../StateContext"**

function Header(props) {

// the global state coming from main useReducer

**const appState = useContext(StateContext)**

### …

**{appState.loggedIn** ? <HeaderLoggedIn /> : <HeaderLoggedOut />}

* Setting new **values** for the state

**appDispatch**({ type: "flashMessage", **value: "Congrats, You successfully created a post"** })

Immer (a reducer package)

* Make it easy to work with complex objects in an immutable fashion. Changing many different states or properties in the ourReducer() function
* We don't want to spell all other properties when we only want to change one. But we can't directly mutate state values.

return { loggedIn: true, flashMessages: state.flashMessages }

* Immer gives us a carbon copy, a draft of the state, which we directly can modify
* **npm install immer use-immer** // use-immer is specifically to use it in react
* can be used for both useState and useReducer

import {useImmerReducer} from 'use-immer'

* it passes a draft of the state that we can do whatever we want with it.
* We need to return or break so our switch works

function ourReducer(draft, action) {

switch (action.type) {

case "login":

draft.loggedIn = true

return

case "logout":

draft.loggedIn = false

return

case "flashMessage":

draft.flashMessages.push(action.value)

return

}

}

const [state, dispatch] = useImmerReducer(ourReducer, initialState)

* Inside the ImmerReducer we don't have access to the state but draft
* import {useParams} from 'react-router-dom'

// it returns an object we want the username. for axios use.

const {username} = useParams()

* **useImmer()** // similar to useState. Accepts an object as an initial value instead of setting multiple useState for multiple states, and it's easy to update each later in an immutable fashion. Something between useState and useImmerReduc.er

const [state, **setState**] = useImmer({

searchTerm: "",

results: [],

// can have a value of loading Icon or results. neither means we haven't typed anything yet

show: "neither",

requestCount: 0, })

import {useImmer} from 'use-immer'

* + we don't want **to return** so use **{}** to **mutate the draft directly.** request count is changed in another useEffect after a short time, to reduce the traffic usage

useEffect(() => {

if (state.requestCount) {

console.log(state.requestCount)

//send axios request here

// cancel if unmounted. ourRequest.token

const ourRequest = Axios.CancelToken.source()

async function fetchResults() {

try {

const response = await Axios.post("/search", { searchTerm: state.searchTerm }, { cancelToken: ourRequest.token })

**setState((draft) => {**

**draft.results = response.data**

**draft.show = "results"**

**})**

} catch (e) {

console.log("There was a problem or the request was cancelled.")

} }

fetchResults()

return () => ourRequest.cancel()

}

}, [state.requestCount])

Transition Group

* Using css transition may be tricky as react wants to add the component to the dom.
* When we use the ternary operator to include a component or not, the component is not in dom, using transition won't work!

npm install react-transition-group

* in Main.js

import { CSSTransition } from "react-transition-group"

* we write the css and we give the component these props:
  + timeout ={} // duration of transition
  + in = {} // it accepts true or false when to affect the transition
  + unmountOnExit // when invisible remove it from dom
  + className**s** = {"search-overlay"} // plural
* we give it only one class but it is smart that adds the below, and knows when to use them.
  + -enter // at the beginning
  + -enter-active // then when we open the overlay
  + -exit // the initial state for exit. Be careful of the **names of classes**.
  + -exit-active // the exit state we want to go to.
  + -exit-active

.search-overlay-enter {

opacity: 0;

transform: scale(1.3);}

.search-overlay-enter-active {

opacity: 1;

transform: scale(1);

transition: 0.33s visibility ease-in-out, 0.33s opacity ease-in-out, 0.33s transform ease-in-out;}

.search-overlay-exit {

opacity: 1;

transform: scale(1);}

.search-overlay-exit-active {

opacity: 0;

transform: scale(1.3);

transition: 0.33s visibility ease-in-out, 0.33s opacity ease-in-out, 0.33s transform ease-in-out;}

Socket, Live chat

* isChatOpen // Add a new state in main. Toggle to open or close it

case "toggleChat":

draft.isChatOpen = !draft.isChatOpen

return

case "CloseChat":

draft.isChatOpen = false

return

* We don't want to unmount the chat so we can receive info about unread chat from the server but we could divide that functionality to live in the header.
* Autofocus doesn't work // use a useEffect
* useRef in odd situations, we can use the ref to access the dom. Still, we don't use the document.querySelector().
* Create a state with useImmer that has the field value, all the messages that we can loop through, and the change it will automatically rerender the chat history box.

function handleSubmit(e) {

e.preventDefault()

//send message to chat server

setState((draft) => {

// add Message to state collection of messages

draft.chatMessages.push({ message: draft.fieldValue, username: appState.user.username, avatar: appState.user.avatar })

draft.fieldValue = ""

})

}

* Show the messages

{state.chatMessages.map((message, index) => {

// within a function and not jsx we can use jsx

if (message.username == appState.user.username) {

return (

<div className="chat-self">

<div className="chat-message">

<div className="chat-message-inner">{message.message}</div>

</div>

<img className="chat-avatar avatar-tiny" src={message.avatar} />

</div>

) }

return (

<div className="chat-other">

<a href="#">

<img className="avatar-tiny" src="https://gravatar.com/avatar/b9216295c1e3931655bae6574ac0e4c2?s=128" />

</a>

<div className="chat-message">

<div className="chat-message-inner">

<a href="#">

<strong>barksalot:</strong>

</a>

Hey, I am good, how about you?

</div>

</div>

</div>

)})}

Connect to the chat server

Socket

* Two-way communication. You want the server to push data to you proactively.
* We can't use Axios, we can send a request to the server but we can not receive the messages. Traditionally web browser sends a request to the server, not the reverse We need to open a socket between the server and browser.
* **Socket.io** // implement it on the backend and on the front end both.
* **npm install socket.io-client**

import io from 'socket.io-client'

// establish an ongoing connection between browser and server

const socket = io("http://localhost:8080")

* **socket.emit('backend event', {message:"", token:""})** // start the socket. The first argument is the type of event that the backend will look for that event, the name should be exactly the same as that we see on the backend. The token is for security to trust us. By this when we send the message, the server will broadcast the message to all connected users on the website.

socket.emit("chatFromBrowser", { message: state.fieldValue, token: appState.user.token })

Listen to the event on the client

* useEffect
* A new event is set on the backend.
* **socket.on**

socket.on("chatFromServer", "function when the event happens")

* close the socket

// if loggedout close the socket in useeffect cleanup function

return() => socket.disconnect()

scroll the chat automatically

* useEffect

// when new messages are pushed we want to scroll to the bottom

useEffect(() => {

// chatLog.current is a dom element

chatLog.current.scrollTop = chatLog.current.scrollHeight

}, [state.chatMessages])

* an alternative way to keep the socket after the user logouts and logins again, since the useEfect only runs once on load. We put the socket inside the component instead of in the import section because if we close it - by Logging out we can't access it anymore.

const socket = useRef(null)

### ….

useEffect(() => {

socket.current = io("http://localhost:8080")

socket.current.on("chatFromServer", (message) => {

setState((draft) => {

draft.chatMessages.push(message)

})

})

// if loggedout close the socket

return () => socket.current.disconnect()

}, [])

Validate the stored token expiry

* Send a request to check the token is still valid

useEffect(() => {

// if they are loggedin and have a token

if (state.loggedIn) {

//send axios request here

// cancel if unmounted. ourRequest.token

const ourRequest = Axios.CancelToken.source()

async function fetchResults() {

try {

const response = await Axios.post("/checkToken", { token: state.user.token }, { cancelToken: ourRequest.token })

// returns true : valid token or false

// when not valid logout

if (!response.data) {

dispatch({ type: "logout" })

// we are not using sessions in the backend in this app but just use the word

dispatch({ type: "flashMessage", value: "Your session has expired, Please login again." })

}

} catch (e) {

console.log("There was a problem or the request was cancelled.")

}

}

fetchResults()

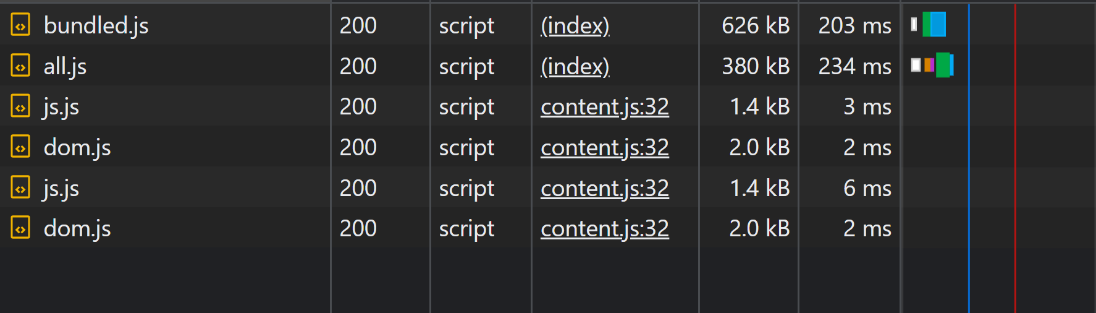
return () => ourRequest.cancel()

}

}, [])

Suspense, Lazy Loading Components

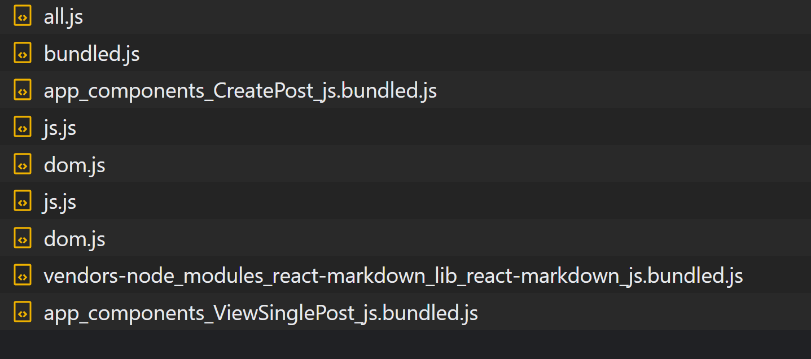
* Not load until it is needed.
* In the network tab of dev tools // click disable cache. Very large. This is for **development mode.**



* Avoid users to load one big js file: bundled.js. They don't need everything for each page.
* **React.lazy(() => import("./components/CreatePost"))** // it returns a promise. CratePost var does not have the content of that component.

const CreatePost = React.lazy(() => import("./components/CreatePost"))

* **import {Suspense} from "react**
* **<Suspense />** //will babysit things, and see if you use a component that hasn't been loaded yet, it shows fallback until the request or promise is busy completing, and when loaded the actual component is shown.
  + **If we depend on another package in one of the lazy loading components, the suspense is smart enough to check if it makes the component very big, or if other components use it, it will create a separate file for it to be loaded.**



* wrap suspense around the component that you want to lazy load // here we wrap it around Routes because the components rely on the URL and are good candidates for lazy loading.
* **fallback**={<LoadingDotIcon />} // the html that is shown while the lazy loading component is busy loading

**<Suspense fallback={<LoadingDotsIcon />}>**

<Routes>

<Route path="/profile/:username/\*" element={<Profile />}>

### ….

<Route exact element={<NotFound />} />

</Routes>

**</Suspense>**

* **We don't want to lazyload everything** // React is supposed to be a single-page application, it is good because of instant rendering.
  + **Size of Component** // About us page less than 1kb // Do we need lazy loading? No, we can instantly render it and no additional network request is done.
  + **How often it is used** // But how many will read the About us? So we can lazyloading

Lazy loading Search

* It is not in the Routes component
* CSSTransition // it will the css to its child element Search, if Search does not exist what should we do?

<CSSTransition classNames={"search-overlay"} unmountOnExit timeout={330} in={state.isSearchOpen}>

<Search />

</CSSTransition>

* Add another div // add a div as the wrapper and put the suspense inside it.

<CSSTransition classNames={"search-overlay"} unmountOnExit timeout={330} in={state.isSearchOpen}>

<div className="search-overlay">

<**Suspense** fallback="">

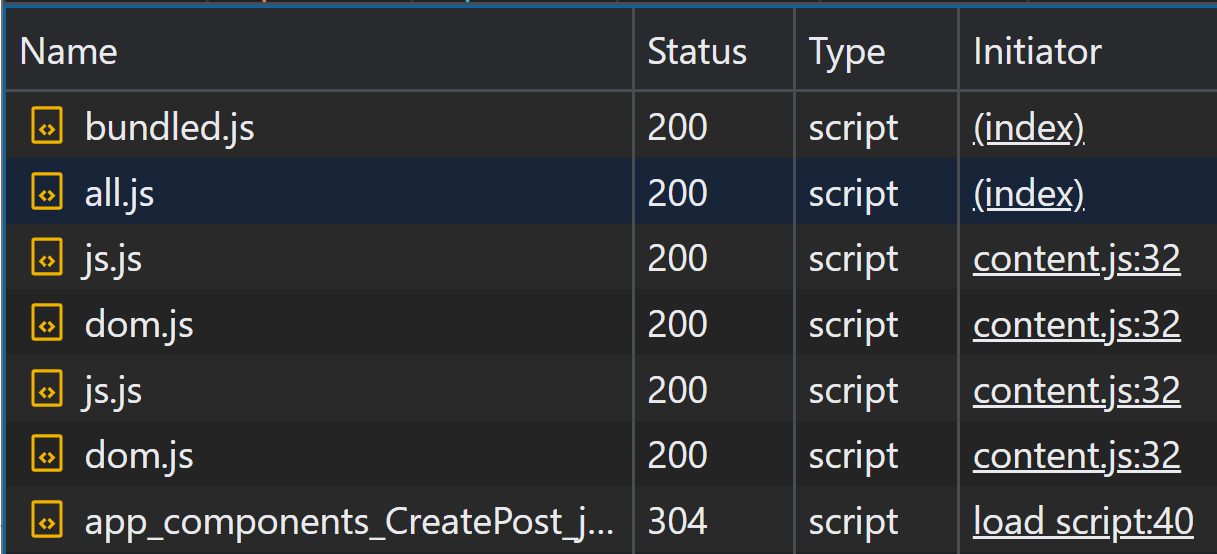
<Search />

</**Suspense**>

</div>

</CSSTransition>

* added when needed



\_dist\_ distributable folder

* **Development workflow** // the bundle.js and other related files are stored in memory, not hard drive. They include comments, sourcmaps, ….
* **De npm install dotenv-webpack**
* **We need to change the webpack.confi.js, and install a few packages**
  + **npm install** 
    - **dotenv-webpack**
    - **clean-webpack-plugin**
    - **html-webpack-harddisk-plugin**
    - **-webpack-plugin**
    - **fs-extra**
    - **html-webpack-plugin**
* remove the script because the webpack setup will add the references for us. After optimizing the files, It also will add a unique cache-busting chunk hash string to the end of the file name that we cannot predict.

<script src="/bundled.js"></script>

* Rename the index.html file to

index-tempalte.html

* Add a new file .env

BACKENDURL = http://localhost:8080

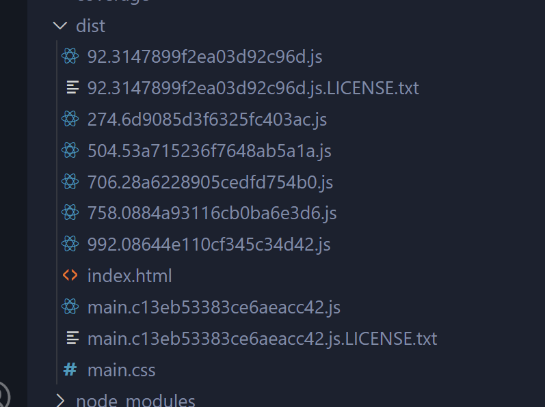
* In Main.js. When pushing in the web we don't include the .env so it can use the second option.

Axios.defaults.baseURL = process.env.BACKENDURL || ""

* In package.json // **Adding the dist folder**, webpackBuild name is what we made in webpack.config.js

"webpackBuild": "webpack",

Npm run webpackBuild



To run the index.html in the dist folder

* the root of the domain is not inside index.html, we can't just run that, and also not all of our routes use the index.html.
* Create a file named previewDist.js

**npm install express**

const express = require("express")

const path = require("path")

const app = new express()

app.use(express.static(path.join(\_\_dirname, "dist")))

app.get("\*", (req, res) => res.sendFile(\_\_dirname + "/dist/index.html"))

app.listen("4000")

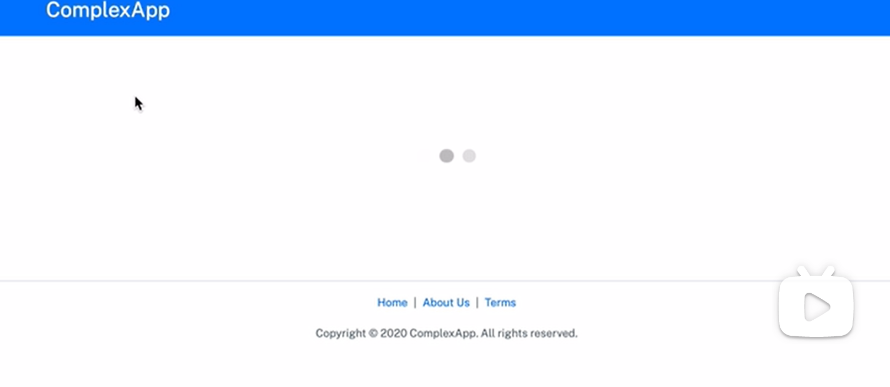
* in package.json

"previewDist": "node previewDist",

* The new index.html created in the app folder is what webpack creates with added scripts out of index-template.html

**Reactdom outside of the web browser**

* Loading our components is very slow.
* Display something // static HTML



* Automated process // based on new and edited layouts.
* import ReactDOMServer from "react-dom/server" // react-dom/server allows us to create html outside of the web browser
* generateHtml.js
* renderToPipeableStream

const myStream = ReactDOMServer.renderToPipeableStream(<Shell />, {

onAllReady() {

myStream.pipe(writeStream)

// End the stream with the final bit of our HTML

writeStream.end(endOfHTML)

}

})

* add the npm script
* npm install @babel/node
* npm install @babel/core // you may face some errors, module-not-found, install the packages again

"generate": "babel-node --presets=@babel/preset-react, @babel/preset-env generateHtml.js",

* To make the generation automatically
  + -s means sequential and not parallel wait until finishes
  + npm istall npm-run-all

"scripts": {

**"dev": "npm-run-all -s generate webpackDev",**

**"webpackDev": "webpack serve",**

**"build": "npm-run-all -s generate webpackBuild",**

**"webpackBuild": "webpack",**

"previewDist": "node previewDist",

"generate": "babel-node --presets=@babel/preset-react,@babel/preset-env generateHtml.js",

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

},

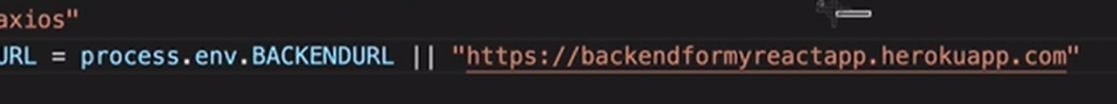
* **Next.js and Gatsby.js** // Combining client-side react with server-side react (primitively statically generated react)

Pushing to web

* Heroku // backend. Push to github.
* Ignore the .env // it Is credentials, like db password, we set them as vars in the web services project setting
* Ignore node\_modules

In front end

* Copy the backend URL// delete the forward slash at the end. Set it for Axios URL



* Remove ("http://localhost:8080")

socket.current = io(process.env.BACKENDURL || "WEB ADDRESS ")

* **Netlify** account
* Create the gitignore,
* **and netlify.toml** // A file in the root. Tell it to use index.html for all URLs because we use client-side routing.

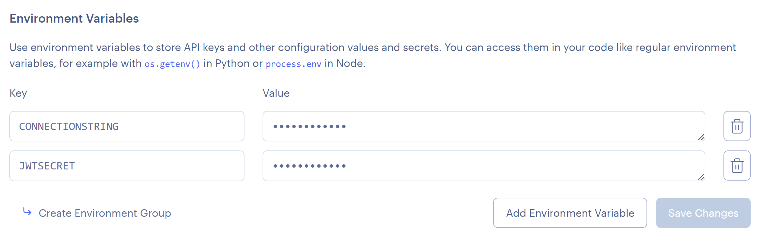
[[redirects]]

from = "/\*"

to ="/index.html"

status = 200

* We don't need a dist folder for netlify because it can perform our build task for us
* Add the npm run build
* don't forget the environment vars in the backend setting.



Common mistakes

* Check for console // not console, console.log
* Check for the useEffect async function call, you should call it after it is set. Otherwise, it will not work properly. And check if it is not called inside the definition! That happened before!
* Check for variables when you copy a code for a new variable name.
* Check for the returns in which case
* Check for varname++. You sometimes forget the ++