React Course Notes

Table of Contents

[What is React? 5](#_Toc30748479)

[Next Generation JS 5](#_Toc30748480)

[Let & Const 5](#_Toc30748481)

[Let 5](#_Toc30748482)

[Const 5](#_Toc30748483)

[Arrow Functions 5](#_Toc30748484)

[Exports & Imports (Modules) 5](#_Toc30748485)

[Spread & Rest Operators 6](#_Toc30748486)

[Destructuring 7](#_Toc30748487)

[Array Functions 7](#_Toc30748488)

[JSX 8](#_Toc30748489)

[State and Props 8](#_Toc30748490)

[Props 8](#_Toc30748491)

[State 8](#_Toc30748492)

[setState 8](#_Toc30748493)

[Stateless vs Stateful 9](#_Toc30748494)

[Stateless 9](#_Toc30748495)

[Stateful 9](#_Toc30748496)

[Event Listening 9](#_Toc30748497)

[Lists and Conditionals 9](#_Toc30748498)

[Lists 9](#_Toc30748499)

[Conditionals 9](#_Toc30748500)

[Styling Components 10](#_Toc30748501)

[Dynamic styles 10](#_Toc30748502)

[Dynamic classes 11](#_Toc30748503)

[Radium for Sudo selectors and Media Queries 11](#_Toc30748504)

[Sudo selectors inline 11](#_Toc30748505)

[Advanced features (Media-Queries, Keyframes) 11](#_Toc30748506)

[CSS Modules 12](#_Toc30748507)

[Error-handling 12](#_Toc30748508)

[Components & React internals Deep dive 14](#_Toc30748509)

[Containers 14](#_Toc30748510)

[Class-Based vs Functional Components 14](#_Toc30748511)

[Component Lifecycle 14](#_Toc30748512)

[Functional-Component Lifecycle 15](#_Toc30748513)

[React.memo() 16](#_Toc30748514)

[PureComponent 16](#_Toc30748515)

[How does React update the DOM 16](#_Toc30748516)

[Higher Order Component (HOC) 17](#_Toc30748517)

[Rendering Adjacent JSX elements 17](#_Toc30748518)

[PropTypes 17](#_Toc30748519)

[Refs 18](#_Toc30748520)

[Context API 18](#_Toc30748521)

[Reaching out to the Web (Http / Ajax) 20](#_Toc30748522)

[Axios 20](#_Toc30748523)

[Redux 22](#_Toc30748524)

[The Redux Store 22](#_Toc30748525)

[Actions & Connection component to reducer and store 23](#_Toc30748526)

[Reducers 24](#_Toc30748527)

[Immutably updating state 24](#_Toc30748528)

[Outsourcing Action Types 24](#_Toc30748529)

[Multiple Reducer 25](#_Toc30748530)

[Advanced Redux 26](#_Toc30748531)

[Middleware 26](#_Toc30748532)

[Action Creators 26](#_Toc30748533)

[Action Creators for Asynchronous code 27](#_Toc30748534)

[Restructuring Actions 28](#_Toc30748535)

[Where should put data transforming logic? 28](#_Toc30748536)

[Accessing the state from action creators 28](#_Toc30748537)

[Utility Function 28](#_Toc30748538)

[Testing 30](#_Toc30748539)

[Testing Tools 30](#_Toc30748540)

[Writing Tests 30](#_Toc30748541)

[Test Redux 31](#_Toc30748542)

[Working with Webpack 32](#_Toc30748543)

[What is Webpack? 32](#_Toc30748544)

[How does Webpack work? 32](#_Toc30748545)

[Webpack Project Initialization and configuration 32](#_Toc30748546)

[Next.js 36](#_Toc30748547)

[Configuring Next.js Project 36](#_Toc30748548)

[Links 36](#_Toc30748549)

[Components 36](#_Toc30748550)

[Styling 37](#_Toc30748551)

[Handling (404) Errors 37](#_Toc30748552)

[Lifecycle Hook 37](#_Toc30748553)

[React Hooks 38](#_Toc30748554)

# What is React?

React is a JavaScript library for building user interfaces. The user interfaces are built by splitting them in Components.

# Next Generation JS

## Let & Const

Different way of creating variables.

## Let

Used for variable values. Values that will be change by the logic processing.

## Const

Const is used for values that won’t change.

## Arrow Functions

Solves problem with the “this” keyword.

const functionName = (arguments) => {

//Logic

}

Automatic return for one-liners.

const functionName = onlyOneArgument => “hey”;

## Exports & Imports (Modules)

Inside of a JS-File you can import content of another file.

#### Default Export

Always exports what’s named on the export per default. Therefor when you import the you can use whatever name you like. See picture below(class person.js).

#### Named Export and Import

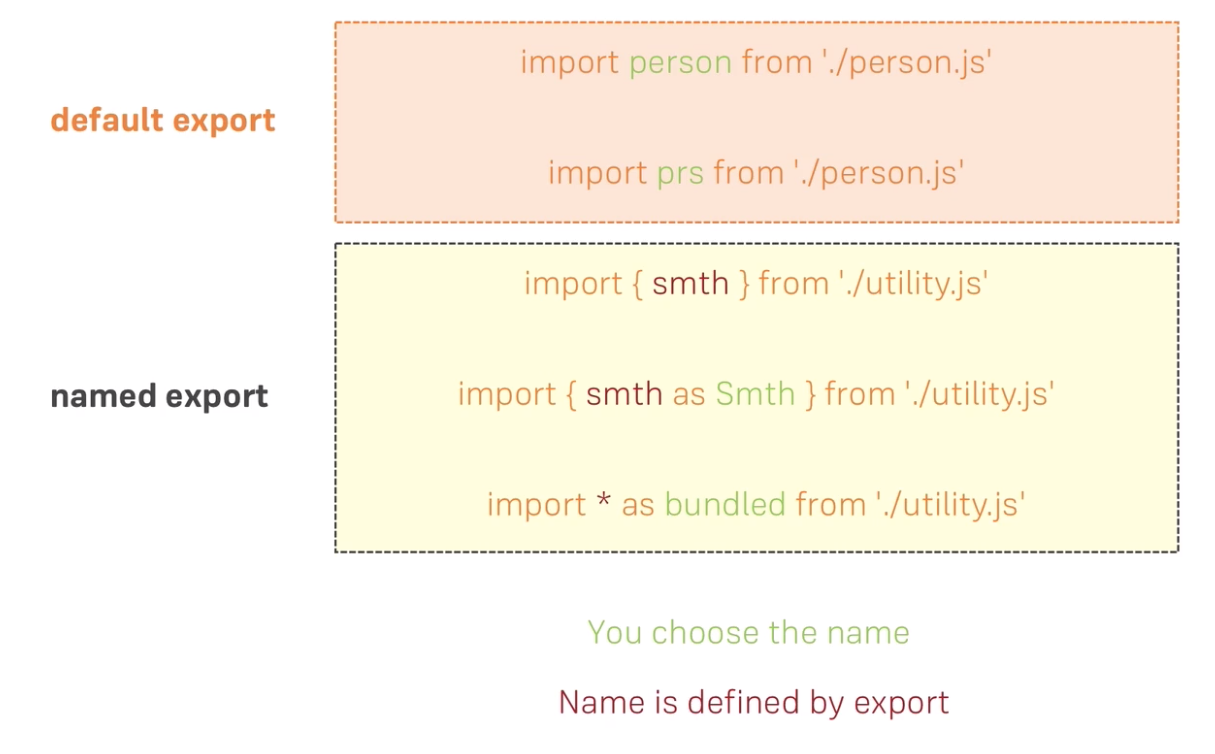
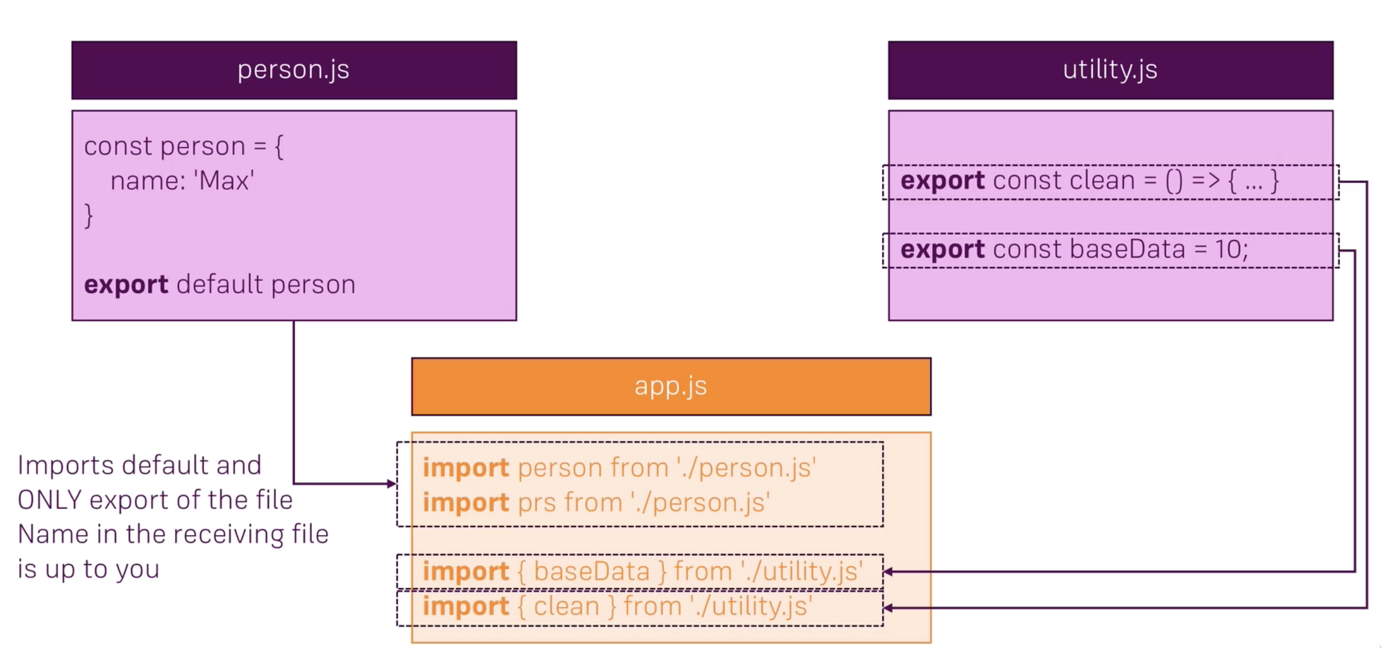
With named exports you can have multiple exports. See picture below(class utility.js). The named imports are called between curly braces. Further export and Import details on Figure 2

Figure 1 Exports and Imports Example

Figure 2 Default and Named Export

## Spread & Rest Operators

#### Spread

Used to split up array element OR object properties.

Examples:

To save values oldArray in newArray and add new values:

const newArray = […oldArray,1,2];

To save properties of oldObject in newObject as key-value-pairs and add a new property:

const newObject = {…oldObject, newProp:5}

#### Rest

Used to merge a list of function arguments into an array.

Example:

Merges all arguments passed to the function into an array.

function sortArgs(…args){

return args.sort();

}

## Destructuring

Easily extract array elements or objects properties and store them in variables. Not same as spread because destructuring allows you to pull single values or properties and store them in oppose to spread saving the whole array/object.

#### Array destructuring

[a,b] = [‘Hello’, ‘Mr.’, ‘Max’]

console.log(a); //Hello

console.log(b); //Mr

[a, ,b] = [‘Hello’, ‘Mr.’, ‘Max’]

console.log(a); //Hello

console.log(b); //Max

#### Object destructuring

{name} = {name: ‘Max’, age: 20};

console.log(name);//Max

console.log(age);//undefined

## Array Functions

Array functions take function as an argument which is executed on each element in the array.

# JSX

Allows to write html-similar code. It’s not the real html-text, React translates it in the background to render the proper html. Components return one element which wraps all component-elements -> ex. div wraps p,h1etc.

# State and Props

React watches if state or props change and analyzes the code already rendered to the DOM and the code it would render after the change. Next it updates the existing DOM everywhere it needs to be updated so the new state/props are properly displayed.

## Props

To get something from outside the component by passing it as props.

Props are passed from the outside.

## State

To change some of the components values from inside the component. State is managed from the inside. State should be used when it makes sense, as too much state can make the application difficult to maintain.

## setState

There’s two ways of setting the state. One is if your new state is not dependent on your old state and the other if your new state is dependent on your old state.

1. The first way is simple and self explainatory:

this.setState({  
 value: newValue  
})

1. However, the second way is a bit more complicated. Even though you call *setState* synchronously it’s not guaranteed to be executed immediately. It’s executed when React find the best time to execute it. Because of that we need to use to optional syntax of the *setState* method that receives to arguments, namely *prevState* and *props*. The code for that:

this.setState((prevState, props) => {  
 return {  
 counter: prevState.counter+ 1  
 };  
});

# Stateless vs Stateful

## Stateless

A component without state. Best to have more stateless components, than stateful because it makes the app easier to maintain.

## Stateful

Stateful is a component that manages state. Functional component that manages its state with *useState* is also a Stateful component.

# Event Listening

You are able to listen to a lot of events all of them listed in the link below.

<https://reactjs.org/docs/events.html#supported-events>

# Lists and Conditionals

## Lists

To repeat an HTML element or group of elements the number of items you have in a list you use the “map” function. The “map” function is an array function which executes logic for each element in the list. See the example below.

<div>  
 {*this*.state.people.map((person, index) => {  
 *return* (  
 <Person *name*={person.name}  
 *age*={person.age}  
 *click*={() => *this*.*deletePerson*(index)}  
 *key*={person.id}  
 *changed*={(event) => *this*.*nameChangedHandler*(event, person.id)}  
 >Hobby</Person>  
 );  
 }}  
 </div>

The person component is repeated the with different values for each array element.

## Conditionals

There are two possibilities on the conditionals.

1. One is directly in the render-return where the whole JSX of the component is returned and it works as follows:

{1>2 ? <p>Hey Hey</p> : <p>Bye Bye</p>}

In this specific case “Bye Bye” is displayed on the website because one isn’t larger than 2. The part after the colon is the “else” part.

1. The second possibility is still in the render method but before the JSX-return.

*let* message = *null*;  
  
*if* (1>2) {  
 message = (  
 <p>Hey Hey</p>  
 );  
} else {

message = (  
 <p>Bye Bye</p>  
 );

}

This saves the wished JSX in the variable “message” according to the condition being true or false. This variable then can be inserted in the normal JSX code. Example:

*return* (  
 <div>  
 {message}  
 <p>Mr/Ms</p>  
</div>  
);

# Styling Components

## Dynamic styles

You can assign CSS properties and values to JavaScript objects. This way you can anytime change the CSS. Example:

*const* style = {  
 backgroundColor: 'lime',  
 font: 'inherit',  
 border: '1px solid transparent',  
 padding: '8px',  
 marginTop: '20px',  
 cursor: 'pointer',  
 color: '#3e3e3e'  
};

if (true) {

style.backgroundColor: ‘tomato’

}

This style is then assigned to the element you want the following way:

<button style={style}> Click Me </button>

## Dynamic classes

You can preprogram the style for a certain class in the CSS file then later set this class to an element through the React code. You create an empty array which will later hold the classes for the element. Then on you can add the classes you’d like to the array.

*let* classes = [];  
*if* (*1* <= 2){  
 classes.push(class-one);  
}  
*if* (*0* <= 1){  
 classes.push(class-two');  
}

To be able to set this array to an HTML element you need to join it on an empty space like this:

<h1 *className*={classes.join(' ')}>Hello World</h1>

## Radium for Sudo selectors and Media Queries

*Alternative to Radium is styled-components.*

Radium is a library you need to install via npm/yarn etc., which allows you to use sudo selectors and media queries in inline style (in component JS file). After you download it you need to import it in you component and then wrap the default export the following way:

export default Radium(App);

## Sudo selectors inline

Example for hover. Further sudo selectors work the same way.

*const* style = {  
 ':hover': {  
 backgroundColor: '#81c784',  
 }  
};

## Advanced features (Media-Queries, Keyframes)

To access the advanced features of Radium you need to wrap the JSX of you Root component in an StyleRoot-element like this:

<StyleRoot>

<!----App----->

</StyleRoot>

Then you can use the advanced feature like the example below:

*const* style = {  
 '@media (min-width:500px)' :{  
 width:'450px'  
 }  
};

Assign the style const to an element as usual.

## CSS Modules

-----------------------------------------------------------------------------------------------------------------

**IMPORTANT**

If you are using react-scripts 2.x or higher the, below the line described, steps do not apply. You only need to import the CSS classes. Example below:

import classes from ‘./App.module.css’;

-----------------------------------------------------------------------------------------------------------------

**For versions of react-scripts below 2.x**

CSS modules is a way of making the stylesheets not global. The first step on achieving this is by ejecting the project as follows:

npm run eject

Afterwards you go in the *webpack.config.dev.js* and extend the options of the *css-loader* with the following commands:

models: true,

localIdentName: ‘[name]\_[local]\_[hash:base64:5]

Then add the same lines in the *webpack.config.prod.js* file.

Then you can import your CSS files like this:

import classes from ‘./App.css’;

That way you import all classes of that specific stylesheet.

After this step you can set classes as the example below:

<p className={classes.ClassName}>Hello Mars</p>

# Error-handling

There is a type of error handling in react specifically for code that you know might fail, e.g. external API calls and similar. For this use case you can use ErrorBoundary if you use a React version 16+.

You can create an ErrorBoundary component looking something similar like this:

*import* React, {Component} *from* 'react';  
  
*class* ErrorBoundary *extends* Component {  
  
 state = {  
 hasError: *false*,  
 errorMessage: ''  
 }  
  
 componentDidCatch(error, errorInfo) {  
 *this*.setState({  
 hasError: *true*,  
 errorMessage: error  
 })  
 }  
  
 render() {  
 *if* (*this*.state.hasError){  
 *return* (<h1>Oooopsie error</h1>)  
 } *else* {  
 *return this*.props.children  
 }  
 }  
}  
  
*export default* ErrorBoundary;

Then wrap the child, that might throw the error in this component and in production (**only in production**) the ErrorBoundary component will be loaded.

# Components & React internals Deep dive

## Containers

Containers such as *App.js* should be lean (less JSX) and manage/manipulate the state.

## Class-Based vs Functional Components

Figure 3 Class-based vs Functional Components

If you use a version of React, that doesn’t support React Hooks and you work with the state or need the Lifecycle Hooks, then you should work with Class-Based components. Use Functional Components in all other cases.

## Component Lifecycle

The Lifecycle is only available in class-based components. Functional Components have a similar equivalent, but it’s not the same. The following table lists those methods in the order of execution.

|  |  |
| --- | --- |
| Lifecycle Method | Explanation |
| constructor(props) | ES6 class Feature. If you create it you need to call *super(props)*. Can be used to set initial state. Shouldn’t be used for so called side effects (http-requests, local-storage etc.) those can trigger re-renders which we don’t want. |
| getDerivedStateFromProps  (props,state) | Used very rarely. In case your props can change and then you want to change state of that component. Again, you shouldn’t cause side effects here. |
| render() | Use to prepare & structure your JSX code. Once more no side effects, time-outs and similar. (This method calls all lifecycle method of its children components, so the next lifecycle-method in your parent component is executed once all lifecycle-methods of the children finished executing.) |
| componentDidMount() | Here you can cause side-effects. This method is perfect for http-requests. However you shouldn’t update state unless data coming from a server or similar. |

Component Lifecycle-methods call order when updating component props.

|  |  |
| --- | --- |
| getDerivedStateFromProps… | //As previous |
| shouldComponentUpdate (nextProps, nextState) | For performance optimization. |
| render() | //As previous |
| getSnapshotBeforeUpdate  (prevProps, prevState) | Used for DOM interaction, like getting the scrolling position of the used. For example during a component update you can get the location of the scroll where a use was and then when the component is finished updating get the user scroll to where he was before. |
| componentDidUpdate() | For http-request and other side effects. Attention to entering an endless loop, where you call http-request that triggers an update and you start the update-methods again and end up with the http-request call executing again and again. |
| componentWillUnmount() | When component gets unmounted from the DOM. For example, when you want to clean up something |

Component Lifecycle-methods call order when updating component state.

|  |  |
| --- | --- |
| shouldComponentUpdate (nextProps, nextState) | //As previous |
| render() | //As previous |
| componentDidUpdate() | //As previous |

## Functional-Component Lifecycle

Thanks to React Hooks functional components can have state now. Therefore they should have lifecycle methods to handle changes properly

|  |  |
| --- | --- |
| useEffect() | Executes for every Lifecycle. |
| useEffect() as componentDidMount | *useEffect*(() => {  }, []);  Empty array to make the method run only once. This way we say that this method has no dependencies and the method reruns only if the dependencies are changed. |
| useEffect() to run when something changes | *useEffect*(() => {  *//Http request simulation  setTimeout*(() => {  *alert*("Saved data")  }, 1000); }, [props.data]); |
| useEffect() as componentWillUnmount() | *useEffect*(() => {  return () => {  //Code here gets executed before the //component unmounts  }; }, []); |
| useEffect() as componentWillUnmount() without array-argument | *useEffect*(() => {  return () => {  //Executed whenever the component //re-renders/after it updated  }; }); |

## React.memo()

To imitate *shouldComponentUpdate* in a functional component you’d need to wrap the export in *React.memo().* It basically stores the component as it is and when the component is re-rendered if the inputs change it will re-render it. If no differences appear React will give back the store component.

Export wrapper:

export default React.memo(component);

## PureComponent

If you need to check if all of your props changed on the *shouldComponentUpdate* method before making sure whether you need to update or not, you can make the Component extend *PureComponent*(of course this only works for class components). *PureComponent* is a basic Component that you’d extend usually with the difference that it already implements the *shouldComponentUpdate-*method and makes full props comparison.

## How does React update the DOM

React compares two Virtual DOMs with one each other. It keeps two copies of the DOM, one of the old Virtual DOM and one of the Re-rendered Virtual DOM (this is the one that is created when the *render*­*­*-method is called). If there are differences the “real” DOM gets updated. It doesn’t completely replace the “real” DOM only swaps what’s new. If there are no differences the real DOM isn’t touched.

## Higher Order Component (HOC)

A HOC is a function that takes a Component as one of its parameters and enhances that component in some way. They just add something to the component, HTML, CSS or JS.

## Rendering Adjacent JSX elements

To return multiple HTML-elements without any outer div you can haven a HOC (basically empty functional component with no JSX) that wraps the adjacent elements.

Example of HOC:

*import* React *from* 'react';  
  
*const aux* = props => props.children;  
  
*export default aux*;

Then wrap you children as usual.

React also offers the same functionality on its own with the *React.Fragment*. Just import *Fragment* from the ‘react’ package and wrap the children with it.

## PropTypes

PropTypes allow you to use data-types on the props so you can output some kind of warning/error in case the wrong prop-type is passed. This is especially useful for open source packages that are used by developers. To use PropTypes you first need to download the package as this is not delivered with the react-package.

npm install --save prop-types

Then import:

import PropTypes from 'prop-types';

Afterwards just before the export of the class you can define your props:

Person.propTypes = {  
 click: PropTypes.func,  
 name: PropTypes.string,  
 age: PropTypes.number,  
 changed: PropTypes.func  
};

## Refs

Ref is used to access DOM elements (similar to document.getElementById(“id”)). Currently there are two possibilities on how to make a ref to an HTML-element in class-based component.

1. The older one works by adding the *ref* attribute to the HTML-element and pass an arrow function the following way:

<div ref={(inputEl) => {  
 this.inputEl = inputEl;  
}}/>

In the function you have a class property which gets the value of the HTML-element and can control it.

1. The other way (newer) requires a constructor in which you assign an empty ref to class-property like this:

constructor(props) {  
 super(Zprops);  
 this.inputElRef = React.createRef();  
}

Then the HTML-element doesn’t get a function passed, but a this specific property.

<div ref={this.inputElRef}/>

And again the element is accessible by the class-property.

1. In functional components that use React Hooks you can import the method *useRef()* to have a similar approach as the second one shown.

import React, {useRef} from 'react';

After you can assign a ref to a variable:

const btnRef = useRef();

At last you can assign the HTML-element to the ref-variable:

<button ref={btnRef} onClick={doSomething}>Switch Name</button>

## Context API

In case of long chains of passing props you can use React Context. Let’s say you have Components from A to D which are wrapped hierarchically and you want to pass props to the D component from the A component. With the usual method you would have to pass them to each child component until you reach the desired one. The Context API has a different method.

First you create a context-component:

import React from 'react';  
  
const contextName = React.createContext({

randomPropName: 0  
 //Here you put the props you’d like to pass with key: default value (not required, optional) annotation  
});  
  
export default contextName;

To pass the props to the Context-component you do the following:

<ContextName.Provider value={{ randomPropName: this.state.randomVar}}>  
 // Here you wrap the child-components which should be able to access the context values  
</ContextName.Provider>

The child Components of the components wrapped in *ContextName.Provider* also have access to the context-values.

To get the values in the child components:

< ContextName.Consumer>  
 {(context) => <div propName={context.randomPropName }>Hello</div>  
 }  
</ ContextName.Consumer>

In class components instead of wrapping whatever needed the property you have the following possibility as well:

static contextType = ContextName;

Then to access the context values:

this.context.randomPropName

This way you can access the values in the whole class not only in the JSX.

In functional components you can use React Hooks the following way to get the context saved in a variable and make it accessible in the whole file:

const context = useContext(AuthContext);

Use as usual JS object.

# Reaching out to the Web (Http / Ajax)

## Axios

Axios is a library that facilitates the making of http requests. It’s installed by npm and once imported in the file it can be used in any JS code.

All of the following examples are done with axios.

The *.then* is the promise that’s returned by the method because this method is asynchronous and waits for the server response. Any logic that requires the response should be executed in the function that *.then* takes as an argument.

On the place of the console.logs you can execute any logic that manipulates the server-response in any way.

#### GET Request

axios.get('https://jsonplaceholder.typicode.com/posts').then((response) => {  
 console.log(response);  
});

#### POST Request

In this example the argument *post* is a JSON object that’s passed to the backend.

axios.post('/posts', post).then(response => {  
 console.log(response);  
});

#### DELETE Request

axios.delete('/posts/'+this.props.id).then(response => {  
 console.log(response);  
});

#### Error Handling

axios.get('/posts').then((response) => {  
 console.log(response);  
}).catch(error => {  
 console.log(error);  
})

#### Instances

Instances are used in case you need different global configurations for different things. For example, environments like dev, prod etc.

The instances are created in a new file (you can have as much as you want). In those instances, you set the configuration similarly.

Example:

import axios from 'axios';  
  
const instance = axios.create({  
 baseURL: 'https://jsonplaceholder.typicode.com',  
});  
  
instance.defaults.headers.common['Authorization'] = 'Auth token form instace';  
  
export default instance;

Then you import the file in the component you want to use this instance in and use it like usual axios.

#### Global config

Any type of global configuration is set in index.js except in cases the configs have to differ at specific conditions (see Instances-chapter).

Example for global configurations:

axios.defaults.baseURL = 'https://jsonplaceholder.typicode.com';  
axios.defaults.headers.common['Authorization'] = 'Auth token form index';  
axios.defaults.headers.post['Content-Type']='application/json';

#### Interceptors

Interceptors are used the intercept with the request and add or manipulate it in any way. For example, if you need to add headers on every request you’d use an interceptor. This is again done either in an instance of axios or in the index if you have only one global configuration.

Example:

axios.interceptors.response.use(request => {  
 console.log(request);  
 // Add header or similar  
 return request;  
}, error => {  
 console.log(error);  
 return Promise.reject(error);  
});

# Redux

“Redux has **one** main advantage, and that's the efficiency it provides. Redux allows you to store your state in what is called a "Redux Store" and uses actions to call reducers, which in turn manipulate your state however you see fit.”

Dylan Mestyanek, [Redux Basics](https://dev.to/dylanmesty/redux-basics-explained-from-a-beginner-s-perspective-abm)

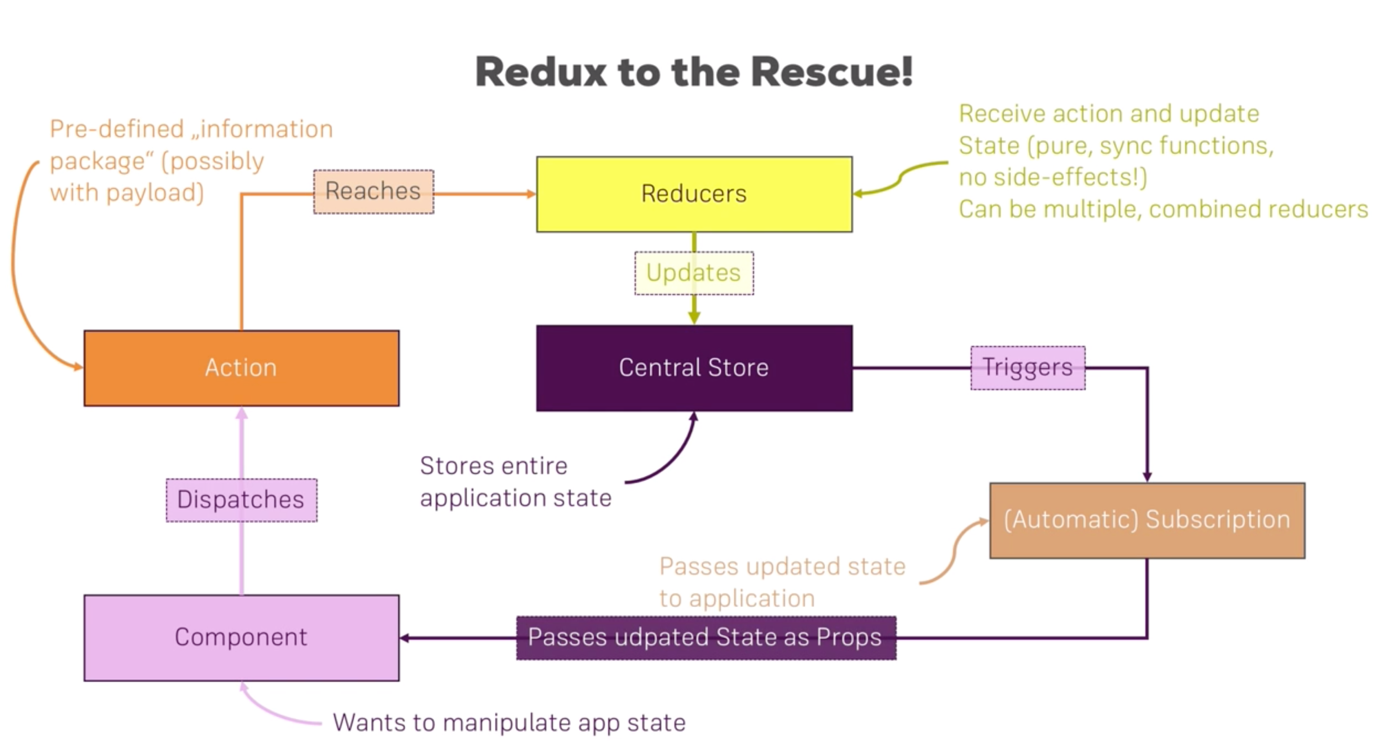


Figure 4 Redux Flow (starting with the Component)

To be able to use Redux in you React project there are two libraries you need to install with you package manager:

npm install --save redux

npm install –save react-redux

Installing Redux is needed to have the independent Redux library and React-Redux is used to connect React with Redux.

## The Redux Store

The Redux Store is where the state is stored globally, so we need to initialize the store globally in the *index.js* of the project.

Following lines of code are required for the imports:

import {createStore} from 'redux';

import {Provider} from 'react-redux';

import reducer from "./store/reducer";

After the imports initialize a constant the following way:

const store = createStore(reducer);

At last wrap you *App*-Component (in the *index.js* file) in a *Provider* and pass the store constant as props:

ReactDOM.render(<Provider store={store}><App/></Provider>, document.getElementById('root'));

## Actions & Connection component to reducer and store

Dispatching an action in the component is required so they can be detected by the reducer which then handles the state. This chapter explains how to connect your component to the reducer and the store and how to define actions.

At first manage the imports:

import {connect} from 'react-redux';

Next you need to create a constant that holds a function, which later will be managed by *connect* from React-Redux. The function will get the state from the redux-store passed as an argument. Then you can return an object (this object will be mapped to the props of the component) with the values you require from the state.

Example:

const mapStateToProps = state => {  
 return {  
 counter: state.counter  
 };  
};

However, this is to only get something from the state. If you want to update the state you need to create a second constant that maps the dispatch (which triggers actions) to your props. Again here you return an object (most likely with a function that is triggered with a certain action).

Example:

const mapDisplatchToProps = dispatch => {  
 return {  
 onIncrementCounter: () => dispatch({  
 type:'INCREMENT'  
 })  
 }  
};

The function *onIncrementCounter* can then be used like a usual function in the props.

The last thing that you need to do is manage you import the following way:

export default connect(mapStateToProps, mapDisplatchToProps)(Counter);

## Reducers

Reducers are like execution guidelines. They are basically functions full of *if…else* conditions, depending on the condition they change the state a differently.

A reducer is conventionally created in separate directory (called store) in the *src* folder. The JS file then has the following structure:

const initialState = {  
 counter: 0  
};  
  
const reducer = (state = initialState, action) => {  
 if (action.type === 'INCREMENT'){  
 return {  
 counter: state.counter + 1  
 }  
 }  
 return state;  
};  
  
export default reducer;

The reducer should always return a state in every condition and this state shouldn’t be undefined. So, in the arguments the *state* is set to have an intial value of *initialState*. The *action* shows what type of action was executed and depending on that we can decide what to do with the state. Usually though you’d export your action name as constants in a separate file.

## Immutably updating state

Mutable updating means directly altering a value in the state. When you mutate the internal state, React does not know what's changed, and even worse; it's notion of the current state is incorrect. So the DOM and virtual DOM will become out of sync.

Example:

const newState = state;

newState.value = state.value+1;

return newState;

This [link](https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns/) explains the immutable update patterns and how to properly update your state.

## Outsourcing Action Types

Create a file with constants to hold all your action-types so you can reduce the chance of typos and unpredicted errors.

## Multiple Reducer

Splitting reducers is a good practice, so you don’t have too large files, which can quickly become confusing with all the conditions. The separate reducer files don’t differentiate with the usual reducer file, they are structured the same way. However, important note is that you don’t have access of the global state from the reducers now. One separate reducer has access only of its own state, other values should be passed as an argument to the dispatcher.

To combine the reducers you need to do the following in the *index.js* file:

First import them and the *combineReducers* method (combines multiple reducers into one):

import counterReducer from "./store/reducers/counter";  
import resultReducer from "./store/reducers/result";

import {createStore, combineReducers} from 'redux';

Then use the combine method and pass it as argument to the *createStore* method:

const store = createStore(combineReducers({  
 counterReducer: counterReducer,  
 resultReducer: resultReducer,  
}));

Then you can import the *store* in your component just like before. However, depending on which state-values you want to access (state of one or other separate reducer) after *state. c*omes the reducer name as you named it in the *combineReducers* arguments. See example:

const mapStateToProps = state => {  
 return {  
 ctr: state.counterReducer.counter,  
 storedResults: state.resultReducer.results  
 };  
};

From this point on the state-value is passed to the props and its usage is the same as previously described (see [Actions & Connection component to reducer and store](#_Actions_&_Connection)).

# Advanced Redux

## Middleware

Term used for code that’s hooked into a process which then gets executed as a part of the process without stopping it, but you can do something with that actions in the middleware before it reaches the reducer.

To add middleware to your code you first need to create a function like this:

const logger = (store) => {  
 return (next) => {  
 return (action) => {  
 //Logic to be executed  
 }  
 }  
};

The *store* argument is passed automatically once you apply the middleware (see next step). The *next* and *action* arguments and both of their functions are also managed by Redux. So basically, this function is the basic configuration for a middleware to work.

To apply the middleware, you need to first import a method from Redux:

import {applyMiddleware} from ‘redux’;

Next you need to pass your middleware as an argument in the *applyMiddleware* method, which is then passed as an argument in the *createStore* method:

const store = createStore(reducer, applyMiddleware(logger));

## Action Creators

Action Creators basically shift the responsibility of creating the *action*-object from the component to a separate file (the file where all the constants with the action-types are).

Once in that file, you can create method which return an action object just like in you’d return it in the Component’s *mapDispatchToProps*-method.

export const ADD\_NUMBER = 'ADD\_NUMBER';

export const addNumber = (number) => {  
 return {  
 type: ADD\_NUMBER,  
 payload: {  
 number: number  
 }  
 };  
};

Afterwards instead of creating the object in the component you can import the action creator method and have it return the desired object:

const mapDispatchToProps = dispatch => {  
 return {  
 onIncrementCounter: () => dispatch(addNumber(number))

}

}

## Action Creators for Asynchronous code

To handle asynchronous code in React-Redux you need to install a further package called *react-thunk*.

Therefore:

npm install --save react-thunk

Afterwards import it in the *index.js* file:

import thunk from 'redux-thunk';

*Thunk* is basically a middleware with its own logic, that interrupts the returning of an action-objects and allows the action to return a function which dispatches an action. Due to returning a function which dispatches and not dispatching by itself it can execute asynchronous code.

Example:

//Method that dispatches the action-object

export const saveResult = (someData) => {  
 return {  
 type: STORE\_RESULT,  
 payload: {  
 someData: someData  
 }  
 };  
};

//Method that calls something asynchronously  
export const storeResult = (someData) => {  
 //You get dispatch by redux-thunk  
 return (dispatch) => {  
 setTimeout(() => {  
 //Logic Logic  
 dispatch(saveResult(someData));  
 }, 2000);  
 }  
};

## Restructuring Actions

Usually you export the action-creators into separate files just like with reducers. You can leave the constants with the action types in one file or also split them. Both however, is best-practice in big projects. To keep things simple for the coders you can export everything from the single files through a further file (which kind of wraps the separate file functions).

Example of the wrapper:

export {deleteResult, storeResult} from './result';  
export {addNumber,subtractNumber,increment,decrement} from './counter';

## Where should put data transforming logic?

Both reducers and actions are kinda correct. However, **reducers** are there to **reduce** which data goes in the state, so they are more or less more opted for data transforming logic.

## Accessing the state from action creators

The following action-creator, which was used in one of the previous chapters, show how to get the state in the action in case it’s required. The *getState*-method is again received from thunk. However, you should try to use it less and try to get the required data by arguments.

export const storeResult = (result) => {  
 //You get dispatch by redux-thunk  
 return (dispatch, getState) => {  
 setTimeout(() => {  
 const someState = getState().someState;  
 dispatch(saveResult(result));  
 }, 2000);  
 }  
};

## Utility Function

To have a leaner code when updating state in reducers you can create an utility method in a separate *utility.js* file in the *store* directory.

The method gets the state and the updated property of the state and returns them:

export const updateObject = (oldObject, updatedValues) => {  
 return {  
 ...oldObject,  
 ...updatedValues  
 };  
};

Then use the method as follows:

switch (action.type) {  
 case actionTypes.INCREMENT:  
 return updateObject(state, {counter: state.counter + 1});

For further logic outsourcing (for example some data that you transform)you can create method in the corresponding file and put the logic as well as the return-statement there, then execute it in the switch-case.

# Testing

## Testing Tools

Test Runner -> Executes Tests and provides Validation Library -> This chapter is going to use Jest

Testing Utilities -> Simulating the React App (mounting components, allowing digging in the DOM) -> We are going to use React Test Util and Enzyme

## Writing Tests

First, we need to install enzyme and its dependencies:

npm install --save enzyme react-test-renderer enzyme-adapter-react-16

#### Testing functional component

In the same folder as the component you can create a new file name something like this: *componentName.test.js.*

There you can configure enzyme so you can use it in you test. What enzyme does, is simply making it possible to render only one component during tests and not the whole application.

import { configure, shallow } from ‘enzyme’;

import Adapter from ‘enzyme-adapter-react-16’;

configure({adapter: new Adapter()});

The *shallow* import is something that will help us with the rendering of one component and not a whole component tree. It basically renders the component empty without any of its contents.

Let’s say we have the following component and we want to test if the component has one p-element. Of course the test would contain logic that changes the rendered elements but for example purposes this was not implemented.

import React from 'react';  
  
import './CounterOutput.css';  
  
const counterOutput = (props) => (  
 <div className="CounterOutput">  
 <p>For tests</p>  
 </div>  
);  
  
export default counterOutput;

The test method for this component would look like this:

describe('<CounterOutput`>', () => {  
 it('should have a p-elemnt', () => {  
 const wrapper = shallow(<CounterOutput/>);  
 expect(wrapper.find('p')).toHaveLength(1);  
 });  
});

The methods *describe*, *it* and *expect* are jest methdos that are available automatically without imports.

*describe* -> takes a string, which describes what component is tested, as the first argument and a function as the second argument.

*it ->* takes a string, which describes what is the expected result of the test, as the first argument and a function as the second. You can put as many *it-*functions in the *describe ­*method as you want.

*const wrapper* -> is the rendered component without any further ReastJS components

*expect* -> defines what exactly is expected, in our case we expect to find one p-element.

For further methods provided by Enzyme and Jest and how to use them:

* Enzyme API: <http://airbnb.io/enzyme/docs/api/>
* Jest Docs: <https://facebook.github.io/jest/>

If you have the same logic you use in multiple tests you can separate it in the *beforeEach –* function (also provided by jest) which gets executed before each test.

## Test Redux

Redux tests are simpler than component tests because they are basic JS functions, so no enzyme methods are used.

You can basically import the reducer and test it as it is.

Example:

import counterReducer from "./counter";  
  
describe('auth reducer', () => {  
  
 it('should return initial state', () => {  
 expect(counterReducer(undefined, {})).toEqual({  
 counter: 0,  
 });  
 })  
});

In this test case we pass an undefined state and an empty action to the reducer and expect the initial state to be returned. The initial state is passed as an object in the *toEqual* method. This method compares the passed argument with the returned object from the method that is tested.

# Working with Webpack

OH BOY WASN’T THIS A HELL ON EARTH.

## What is Webpack?

Webpack is a bundler that allows you to optimize your files and let’s you hook in various plugins, loaders and similar to transform your files and transpile next-generation JS to current generation JS. Its core idea is to have multiple files and combine them together, so out of 5 JS files there is one *bundle.js* file.

## How does Webpack work?

Webpack has 4 important features.

It always needs at least one entry-point, however you can have multiple ones. In a React app this would be the *App.js*, which hold its dependencies. Webpack analyzes them and their dependencies and in this way, it figures out what files and dependencies make up our application.

Afterwards it bundles all the dependencies together and outputs them somewhere (most probably dist/bundle.js).

We can also have loaders. Loaders are applied on a per file level. In this way, we can say that JS files should be handled by loader-x (bable-loader is a real example) for example.

Plugins are applied to the bundle before it’s written to the output. They alter the whole bundle in a way depending on what plugins are imported.

## Webpack Project Initialization and configuration

To start you will need an empty folder with your project name. Afterwards you can initialize it to be an npm project with the following line in the terminal:

npm init

Then you will need to install the required Webpack dependencies:

npm install --save-dev webpack webpack-dev-server

From this point on you recreate the React project-structure (folders, files etc.). After you are done creating the project you can configure the webpack server. This is done by managing your *package.json* file and adding the *start*-command in the file under scripts. Your *scripts* object should look like this:

"scripts": {  
 "test": "echo \"Error: no test specified\" && exit 1",  
 "start" : "webpack-dev-server"  
},

Afterwards install the *webpack-cli* package:

npm install --save-dev webpack-cli

This tool is needed to build the project properly.

Next we need to create a file that holds the whole Webpack-configuration. This file is stored on the same level as the *package.json* file and is called *webpack.config.js.* The following configuration is required for the proper functioning of Webpack with the React project. For further explanation of the config check this [link](https://webpack.js.org/guides/getting-started/).

const path = require('path');  
  
module.export = {  
 mde: 'development',  
 entry: './src/index.js',  
 output: {  
 path: path.resolve(\_\_dirname, 'dist'),  
 filename: 'bundle.js',  
 publicPath: ''  
 },  
 devtool: 'cheap-module-eval-source-map'  
};

#### Support JS

Now this alone won’t be enough for Webpack to understand new-genearation JS, that’s why we need to install these babel-packages:

npm install --save-dev @babel/core @babel/preset-env @babel/preset-react @babel/preset-stage-2 babel-loader @babel/plugin-proposal-class-properties

Babel is a 3rd party package that transforms new generation JavaScript into an older generation in case of an older browser. It also transforms JSX code into regular JavaScript.

Now we have to configure Babel. The Babel configuration is located in a separate file again on the same level as the *package.json* file. It is called *.babelrc*.

The code snippet below represents the basic configuration of babel, however this documentation won’t go in further detail on what these configs do/mean. For reference use this [link](https://babeljs.io/docs/en/configuration).

{  
 "presets": [  
 ["@babel/preset-env", {  
 "targets": {  
 "browsers": [">1%", "last 2 versions"]  
 }  
 }],  
 "@babel/preset-react"  
 ],  
 "plugins": [  
 "@babel/plugin-proposal-class-properties"  
 ]  
}

For Webpack to recognize Babel add the following code to the *webpack.config.js* file after the *devtool* key:

module: {  
 rules:[  
 {  
 test: /\.js$/,  
 loader: 'babel-loader',  
 exclude: /node\_modules/  
 }  
 ]  
}

#### Support CSS

To support CSS files you need to install a new loader:

npm install --save-dev style-loader css-loader

CSS loader analyzes the CSS imports and while the style-loader gets all the CSS code and injects it in the HTML page.

Afterwards add a second rule in the *webpack.config.js* file.

{  
 test: /\.css$/,  
 exclude: /node\_modules/,  
 use: [  
 {loader: 'style-loader'},  
 {  
 loader: 'css-loader', options: {  
 importLoaders: 1,  
 modules: {  
 localIdentName: '[name]\_\_[local]\_\_[hash:base64:5]'  
 }  
 }  
 }  
 ]  
}

Finally, for the CSS install these two package and add them to the config file:

npm install --save-dev postcss-loader

npm install --save-dev autoprefixer

Then in the Webpack file create a constant with the *autoprefixer*.

const autoprefixer = require('autoprefixer');

Then add the loader to the CSS rule:

{  
 loader: 'postcss-loader',  
 options: {  
 ident: 'postcss',  
 plugins: () => [autoprefixer()]  
 }  
}

At last add the following line to you *package.json* right after the *license* key:

"browserslist": "> 1%, last 2 versions",

#### Support Images

We need to install yet another package to enable image support:

npm install --save-dev url-loader

Again add the loader in the Webpack configuration:

{  
 test: /\.(png|jpe?g|gif)$/,  
 loader: 'url-loader?limit=8000&name=images/[name].[ext]'  
}

Now to basically connect this configuration to the indedx.html guess what? WE NEED MORE PACKAGES ☺ ☺ ☺ :

npm install --save-dev html-webpack-plugin

npm install --save-dev file-loader

To use this plugin we need to create a constant in the Webpack configurations:

const HtmlWebpackPlugin = require(‘html-webpack-plugin’);

After that you can add the code snippet in the Webpack configuration below the *module* object:

plugins: [  
 new HtmlWebpackPlugin({  
 template: \_\_dirname + '/src/index.html',  
 filename: 'index.html',  
 inject: 'body'  
 })  
]

# Next.js

Next.js is a framework for server-rendered React applications. React’s SSR (server-sided-rendering) can get quite confusing and Next.js helps by automating the routing and making the whole process easier.

## Configuring Next.js Project

You start with an empty folder which you can open in your IDEA. Afterwards open the folder in the terminal and run:

npm init

This initializes the folder as a npm project. Once done you can install Next.js, React and React-DOM in the project.

npm install --save next react react-dom

After the installation you can configure the script methods in the *package.json* file. Just replace whatever *script* key you have in the object with this one:

"scripts": {  
 "dev": "next",  
 "build": "next build",  
 "start": "next start"  
},

## Links

Now to be able to use the automatic routing you need to follow certain directory-order. It’s one directory, which order should be followed and that is the pages folder (you should create this one in the project folder). In there you can create a folder for each of your components and in there create your component-files. In the page-folder you have to create your index.js so you have a default page, that is loaded for empty routes.

#### Clickable Links

It’s similar to vanilla HTML except that you need to wrap your anchor tag in a Link tag. The *href* attribute is passed to the anchor by Next.js

<p> Go To <Link href="/auth"><a>Auth</a></Link></p>

If you want to navigate with a button it’s also very simple. First you import the Next.js router and then you just push the link you want to go to, either in a function or onClick:

import Router from 'next/router'

<button onClick={() => Router.push('/auth')}>Auth</button>

## Components

You can also create a separate folder for all components that are not involved in the routing whatsoever. You can create you *components* directory and add you components just like you’d add them in pure ReactJS. To use them in other components you just need to import them like usual.

## Styling

Inline styles work as they work in React.

Scoped styling however can be achieved by adding *style-*elementin you JSX code.

Example:

<style jsx>  
 {`  
 div {

border: 1px solid #eee;

}   
 `}  
</style>

## Handling (404) Errors

Creating and styling an *\_error.js* file in the *pages* directory automatically transfers all 404s to this file.

## Lifecycle Hook

The *getIntialProps* hook is especially useful to do initial data population on the server-side and deliver the populated page to the browser.

To implement it in a functional component:

Page.getInitialProps = async ctx => {

const res = await fetch('https://api.github.com/repos/zeit/next.js')

const json = await res.json()

return { stars: json.stargazers\_count }

}

And for a class-based component:

static async getInitialProps(ctx) {

const res = await fetch('https://api.github.com/repos/zeit/next.js')

const json = await res.json()

return { stars: json.stargazers\_count }

}

# React Hooks

Previously we touched on React Hooks and how they enable Lifecycle-Hooks and state in functional components. This chapter is a more detailed explanation of React-Hooks.

React Hooks basically allows functional components to have state and lifecycle-methods. Something that wasn’t possible earlier. Now you can use only functional components in projects (don’t have to, but you can).

## useState

*useState* is method that need to be imported from React and it enables functional components to have a state. This method takes one argument, which is set to be the initial state. However, in *useState* this can be a string a number, object or further datatypes, while in class-based components state it’s always an object. The method also returns something, an array with two elements.

* The first-element is a snapshot of the current state. As we know components rerender when the state is changed and the *useState* method is executed again. However, the values that were initially passed as an argument in *useState* won’t be set again because react internally saves that the state has been reconfigured and doesn’t require a new initialization.
* The second element is a function that allows you the set the state.

Example:

First the import:

import React, {useState} from 'react';

Then the method is executed and the values stored in a new constant. We store it in a constant so we can access the array in the whole component. And because we know that the method always return an array with two elements, we destruct them into two separate constants.

const [state, setState] = useState({  
 title: '',  
 amount: ''  
});

1. Example of two-way binding by setting the state. It is good to know it this way but there is also something simpler that is better. However, if you want to understand the process and why something behave the way they do read the following

Long and very detailed explanation that is kinda confusing:

{ This part requires a more detailed explanation due to some things being managed by React and some not. In the *onChange*-Listener we get the event automatically from React. Then we execute the second array-element (the function for setting the state), which automatically receives an argument with the updated state. This is required because when we set the state, React doesn’t merge it but overwrites it. So, if we have only one attribute of the object passed to be set, then it will become the whole state. However, if we set the updated attribute to its new value and the not-updated attribute to its value in the state we risk of getting an old value from the state because the React manages the state asynchronously.

So, a use case for the explained would be that we typed something in the *title* input and then something in the *amount* input. However, the *title* isn’t yet set in the state, because React decided that now wouldn’t be a good time for that. Then when we typed something in the *amount* and again set the *title* to its value from the state object (state.title), because we always have to point what value each attribute receives, we receive an old value, because the state hasn’t been updated yet. That’s when the argument *updatedState* comes in play and provides us with the latest state values.}

TL; DR:

The following way always provides and updates the state with the most recent state-values even when they haven’t been updated into the state.

<input type="number" id="amount" value={state.amount}  
 onChange={event => {  
 const newAmount = event.target.value;  
 setState(( updatedState) => {  
 return {title: prevState.title, amount: newAmount};  
 })  
 }}/>

<input type="text" id="title" value={state.title}  
 onChange={event => {  
 const newTitle = event.target.value;  
 setState((prevState) => {  
 console.log(prevState);  
 return {title: newTitle, amount: prevState.amount};  
 })}}/>

1. Making sure you don’t lose an attribute of you state can get very long, complicated and confusing, you can use multiple states. So one state for *title*, one state for *amount* etc. In that way you can always call whatever state you need and only set it. This also facilitates the two-way binding and is a cleaner way to do it.

The code for the state would be:

const [title, setTitle] = useState('');  
const [amount, setAmount] = useState('');

And the inputs:

<input type="text" id="title" value={title}  
 onChange={event => {  
 setTitle(event.target.value)  
 }}/>

<input type="number" id="amount" value={amount}  
 onChange={event => {  
 setAmount(event.target.value);  
 }}/>

## useEffect

This hook gets executed after every lifecycle, and that’s how it imitates them. The first argument they get passed is a function where the logic lies. The hook is used to recreate the lifecycle-methods and can be used for side-effect. You can have multiple *useEffect* hooks and control their execution order with the second argument they get passed, which is an array of their dependencies. Then the hook gets executed only when this dependency is changed.

For example this executes only when the variable *x* is changed:

useEffect(()=>{

console.log(“Executed”);

}, x);

This [chapter](#_Functional-Component_Lifecycle) shows you how to recreate each life-cycle in functional components and explains their usage.

## useRef

This hook allows you to create a reference to an HTML-element in your component. It’s very basic and needs no detailed explanation, hence here is the code:

const refInput = useRef();

<input ref={refInput} type="text"/>

## useCallback

This hooks saves the functions that don’t change on re-renders so they won’t re-initialized. This optimizes the performance.

const filteredIngredientsHandler = useCallback((filter) => {  
 setIngredients(filter);  
}),

## useMemo

This hook recreates the React.memo wrapper, which saves a component and only re-renders it when some dependencies are update.

Example:

const ingredientList = useMemo(() => {  
 return <ComponentName/>  
}, [dependency]);

Just as *useContext*, *useMemo* accepts two arguments. The first is a function that composes of the component and some logic and the second one are the dependencies.

## Common Gotchas for infinite loops

* Passing functions, which sets a state, as props to a child component, which then uses this prop for *useEffect* execution.
  + Solution:

Wrap the method in *useCallback* (also imported from rect). This caches your method, so it won’t be reinitialized when the component renders. Example:

const filteredIngredientsHandler = useCallback((filter) => {  
 setIngredients(filter);  
}),

## useReducer

“useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. useReducer also lets you optimize performance for components that trigger deep updates because [you can pass dispatch down instead of callbacks](https://reactjs.org/docs/hooks-faq.html#how-to-avoid-passing-callbacks-down).”

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

You usually create the reducers before the components, so they won’t re-initialized with the every re-render. You initialize a constant with a function (that is the dispatch function) that holds all switch-cases (same as a Redux reducer):

const ingredientReducer = (currentIngredients, action) => {  
 switch (action.type) {  
 case 'SET':  
 return action.ingredients;  
 case 'ADD':  
 return [...currentIngredients, action.ingredient];  
 case 'DELETE':  
 return currentIngredients.filter(ing => ing.id !== action.id);  
 default:  
 throw new Error("WTF you doin here boi")  
 }  
};

Then in you component you can use the reducer the following way. The hook returns your state and a dispatch method to trigger the reducer’s function.

const [state, dispatch] = useReducer(ingredientReducer, []);

As we defined the dispatch function to take the *currentIngredients* and an *action.* So we pass them when we want to trigger:

dispatch({type: 'SET', ingredients: filteredIngredients})

## useContext

This has been already explained in detail. Check [this](#_Context_API) chapter.