**React**

Contents

[Pros 3](#_Toc96516300)

[Single Page Applications 3](#_Toc96516301)

[JSX 3](#_Toc96516302)

[Babel 4](#_Toc96516303)

[Flow of React 4](#_Toc96516304)

[Dataflow in React 4](#_Toc96516305)

[Virtual DOM 4](#_Toc96516306)

[Components 4](#_Toc96516307)

[Default Props 5](#_Toc96516308)

[Wrapper component (Composition, children props) 5](#_Toc96516309)

[Controlled component vs Uncontrolled component 5](#_Toc96516310)

[Stateless vs Stateful components 5](#_Toc96516311)

[Adding dynamic styles 5](#_Toc96516312)

[Styling React Components 6](#_Toc96516313)

[Conditional and Dynamic Styles 6](#_Toc96516314)

[CSS Classes Dynamically 6](#_Toc96516315)

[Scoping Styles with Styled Components 6](#_Toc96516316)

[Media Queries 6](#_Toc96516317)

[CSS Modules 7](#_Toc96516318)

[Lists and Conditional content 7](#_Toc96516319)

[Conditional content 7](#_Toc96516320)

[Props 8](#_Toc96516321)

[State 8](#_Toc96516322)

[Update state depending on previous state 8](#_Toc96516323)

[Two-way binding 8](#_Toc96516324)

[Fragments, Portals and Refs 8](#_Toc96516325)

[JSX Limitations & Fragments 8](#_Toc96516326)

[Getting a cleaner DOM with Portals 9](#_Toc96516327)

[Working with Refs 9](#_Toc96516328)

[Effects, Reducers and Context 9](#_Toc96516329)

[Working with side effects (effect) 9](#_Toc96516330)

[Debounce 9](#_Toc96516331)

[Managing more Complex state with reducers 10](#_Toc96516332)

[Managing app wide or component wide state with context 10](#_Toc96516333)

[Hooks 10](#_Toc96516334)

[useState 10](#_Toc96516335)

[useEffect 10](#_Toc96516336)

[useReducer 11](#_Toc96516337)

[useContext 11](#_Toc96516338)

[useImperativeHandle 12](#_Toc96516339)

[useCallback 12](#_Toc96516340)

[useMemo 12](#_Toc96516341)

[Custom hooks 13](#_Toc96516342)

[Building Custom Hooks 13](#_Toc96516343)

[JavaScript bind 13](#_Toc96516344)

[Handling Forms and User Input 14](#_Toc96516345)

[Context API 14](#_Toc96516346)

[Rules of Hooks 14](#_Toc96516347)

[Debugging React Apps 14](#_Toc96516348)

[React – Behind the scenes 15](#_Toc96516349)

[Virtual DOM and DOM updates 15](#_Toc96516350)

[State and State Updates 15](#_Toc96516351)

[Routing 16](#_Toc96516352)

[Redux 16](#_Toc96516353)

[Redux vs React Context 16](#_Toc96516354)

[Authentication 16](#_Toc96516355)

[Deploying React Apps 16](#_Toc96516356)

[Testing React Applications 17](#_Toc96516357)

[Next.js 17](#_Toc96516358)

[Server-side rendering support 17](#_Toc96516359)

[File based routing 17](#_Toc96516360)

[Fullstack capabilities 17](#_Toc96516361)

[Course 18](#_Toc96516362)

JS library for building user interfaces. Front end framework.

React runs on the client as a SPA. Used for full-stack (MERN).

Declarative approach, define the final state and React takes care of the JS DOM instructions.

# Pros

Structure the “view” layer of your application

Reusable components with their own state

JSX – Dynamic markup

Interactive UIs with Virtual DOM

Performance & testing

Very popular

# Single Page Applications

React can be used to control parts of HTML pages or entire pages.

Widget approach on multi page application, some pages are still rendered by the server.

SPA, server only sends one HTML page, and after, React takes control of the UI.

# JSX

Is syntax sugar for React createElement(elem,props,children). React is creating elements under the hood. JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods. In the past React was imported in all the component files.

We can store JSX values in variables.

# Babel

Under the hood babel is doing the transpilation turning the JSX into JavaScript and class is a reserved keyword so we have to use className.

# Flow of React

Starts in the index.js file. React calls the component functions, and then the functions that are returned, until there are no functions left. So custom components are treated as functions by JSX. React never repeats this, it is done when the application starts.

# Dataflow in React

Unidirectional, from parent to child components. Props vs state. Avoid prop drilling with global state, context, redux.

# Virtual DOM

Update parts of the DOM without reloading. Much more faster, dynamic.

# Components

All user interfaces are created using components. Components are a core concept in React.

Reusability and Separation of Concerns.

Components can be created using both functions and classes.

Components render/return JSX (JavaScript Syntax Extension)

Components can also take in “props”. Lower case is built in html, upper case is custom element.

Function components with hooks are the focus nowadays.

Components can contain state, which is an object that determines how a component renders and behaves.

App or global state refers to the state that is available to the entire UI, not just a single component.

Prior to React 16.8 we had to use class based components to use state. Now we can use functional components with hooks.

import Button from './Button'

const Header = ({title}) => { // destructuring

    const onClick = (e) => {

        console.log("on click");

    }

    return (

        <header className='header'>

            <h1 style={{color: 'blue'}}>{title}</h1>

            <Button color='green' text='Hello' onClick={onClick} />

        </header>

    )

}

## Default Props

// if we don't pass props these will be used

Header.defaultProps = {

    title: "Task Tracker",

}

Prop Types

// don't need to use this with typescript

Header.propTypes = {

    title: PropTypes.string.isRequired

}

## Wrapper component (Composition, children props)

If we want to create a custom component that acts as a wrapper/container for other components, then we need to use props.children that is set automatically by React.

Also to style this wrapper, we have to use props.className to get the className that we passed to the wrapper, and then use it inside the componenet.

## Controlled component vs Uncontrolled component

Both the value as well as changes to the value are not handled in the component itself, but in a parent component.

If we use ref then we have an uncontrolled component (input) because the state of the input is not managed by React (even if we use ref), we change the value using the DOM, so it’s uncontrolled. If we use state for a component then it’s controlled.

## Stateless vs Stateful components

Stateless, presentational, dumb component

Stateful, smart

## Adding dynamic styles

{{ }} because styles wants an object, it’s not a special double curly brace syntax.

# Styling React Components

## Conditional and Dynamic Styles

<label style={{ color: !isValid ? 'red' : 'black' }}>Course Goal</label>

We can have ternary operator in style object.

Inline styles have the highest priority, so you overwrite all other styles with that. We can have duplicate CSS because we have a color in the CSS file and then we overwrite just because we want to have a conditional style.

## CSS Classes Dynamically

<div className={`form-control ${!isValid ? 'invalid' : ''}`}>

## Scoping Styles with Styled Components

styled-components

is a npm package that can be installed and it provides the possibility to create components that are styled using tagging template function and the CSS provided will be transformed into classes with UNIQUE ids and added into the global CSS, so classes will never spill to other components.

// Tagged templates

const Button = styled.button`

  font: inherit;

  padding: 0.5rem 1.5rem;

  border: 1px solid #8b005d;

  color: white;

  background: #8b005d;

  box-shadow: 0 0 4px rgba(0, 0, 0, 0.26);

  cursor: pointer;

&:focus {

  outline: none;

}

&:hover,

&:active {

  background: #ac0e77;

  border-color: #ac0e77;

  box-shadow: 0 0 8px rgba(0, 0, 0, 0.26);

}

`;

## Media Queries

@media (min-width: 768px) {

    width: auto;

  }

## CSS Modules

import styles from './Button.module.css';

It changes the class name of the classes you are importing to be unique.

Good separation between CSS and JavaScript and scoped styles.

<button type={props.type} className={styles.button} onClick={props.onClick}>

We can then access the styles object to get the classes and apply them.

Dynamic styling using CSS modules

<div className={`${styles['form-control']} ${!isValid && styles.invalid}`}>

# Lists and Conditional content

Childs of a list need to have an unique id so that React can update them fast. If we don’t provide keys to the children, then for react all the items look the same, it renders the new added item in the list, and then it updates all the values in the list, this is bad for performance.

<ul>

            {expenses.map((expense) => (

                <ExpenseItem

                    key={expense.id}

                    title={expense.title}

                    amount={expense.amount}

                    date={expense.date}

                />

            ))}

</ul>

Besides performance issues we could also run into bugs (loss of state).

With a key React is able to update a list in the most efficient way, and only render the item added.

## Conditional content

{filteredExpenses.length === 0 && <h3>No expenses for the selected year</h3>}

Or we can use the ternary operator.

If we want to have a very clean JSX we can store JSX in JavaScript variables:

  let expensesContent = <h3>No expenses for the selected year</h3>;

  if (filteredExpenses.length > 0) {

    expensesContent = <ExpenseList expenses={filteredExpenses} />;

  }

And then use it as a single expression:

        {expensesContent}

# Props

Components can have props. It’s a way to transfer data from a parent component to a child component.

We can also communicate bottom-up, from child component to parent component using props with functions, but still by sending function from parent to child (unidirectional).

The problem of prop drilling can arise if we have a lot of nested custom components and we need to pass data from a component close to the root. To avoid it, we use the global state.

# State

We use state so that the UI is reactive. State is introduced to change the ui after it was rendered initially.

State variables are managed by React and trigger the re-execution of the component function and only that component. The setState function schedules the change but it doesn’t happen instantly, but it happens before the component is re-evaluated by React, so the const title has the new value. React keeps track of when useState was executed the first time in a component and it takes the default parameter, but after that it will not re-initialize the state.

If we have data that might change and that change/data should be reflected in the ui then we have to use state.

Each component has it’s own state. We can have multiple pieces of state in a component or a single state with multiple properties (is is not recommended by React, multiple pieces of state is better for memorization, and you don’t risk loosing data).

## Update state depending on previous state

Whenever you update state and you depend on a previous state you should use an arrow function with parameter (prevState), this is called the function form of the state updating function.

setExpenses(prevExpenses => [expense, ...prevExpenses]);

## Two-way binding

When using forms, bind the value of input to state and then you can reset it after submit.

# Fragments, Portals and Refs

## JSX Limitations & Fragments

You can’t return 2 root elements. We wrap the 2 elements in a div. A new problems: div soup, too many empty divs rendered in the DOM. To fix this problem we can use an empty component that just renders the children:

const Wrapper = (props) => {

    return props.children

}

Because this is a common problem, this element is already in React and it’s called Fragment. It can be used in 2 ways: by importing Fragment from react and using <Fragment> or by using <> </> empty tags. In order for <> </> to work it needs to be set up in the project. <Fragment> always works.

We can return an array of JSX elements but we get an error because anytime we use an array of elements in react we need to add keys.

## Getting a cleaner DOM with Portals

While using modals, we show it on top of the whole site, but in the DOM it’s nested somewhere, where we used it. This is not correct semantically and from a clean html structure perspective. We can portal the HTLM to a div that we want, we create a div in index.html and give it an id, so that we can use it as the second parameter of createPortal.

import ReactDOM from 'react-dom';

{ReactDOM.createPortal(<Backdrop onConfirm={props.onConfirm} />,

        document.getElementById('backdrop-root'))}

## Working with Refs

References, they allow us to get access to other DOM elements and work with them. They should be not used to manipulate the DOM, only read from it. Refs have a current property.

const nameInputRef = useRef();

          <input id="username" type="text" value={enteredUsername}

            ref={nameInputRef}

          />

const enteredName = nameInputRef.current.value;

# Effects, Reducers and Context

## Working with side effects (effect)

What is a side effect? Everything else besides the main job of react: UI & react to user input.

Examples: Store data in browser storage, send https requests, set and manage timers

## Debounce

Used with the useEffect hook. In order to reduce the number of executions of use effect ( for example when its triggered by a keystroke) we can debounce it. Use effect can return a function, and it’s called clean-up function. It doesn’t run before the first execution of useEffect, but it runs before every execution after that.

  useEffect(() => {

    const indetifier = setTimeout(() => {

      console.log("check validity");

      setFormIsValid(enteredEmail.includes('@') && enteredPassword.trim().length > 6);

    }, 500);

    // it doesn't run before the first execution

    // it runs before every execution after that

    return () => {

      console.log('CLEANUP');

      clearTimeout(indetifier);

    };

  }, [enteredEmail, enteredPassword]); // can ommit setFormIsValid

If we have a cleanup function without dependencies in useEffect then it runs when the component is removed from the DOM.

  useEffect(() => {

    console.log('Component login');

    // it runs when the component is removed from the DOM

    return () => {

      console.log('Component login removed from the DOM');

    };

  }, []);

## Managing more Complex state with reducers

## Managing app wide or component wide state with context

# Hooks

React hooks are functions that les us hook into the React state and lifecycle features from function components.

## useState

Returns a stateful value and a function to update it

## useEffect

Perform side effects in function components. 2 parameters:

A function that should be executed AFTER every component evaluation IF the specified dependencies changed.

The dependencies, when one changes, the function will re-run.

If there are no dependencies, then the function runs once.

Some things that don’t need to be added as a dependency: state update functions, built-in apis or functions like fetch or localStorage, variables or functions defined outside of components.

When we have to add a function as a dependency to useEffect, we have to wrap it in useCallback because functions are objects and they will change every time to component is re-evaluated, hence creating an infinite loop.

## useReducer

Sometimes you have more complex state, for example multiple states, multiple ways of changing it or dependencies to other states.

useReducer is used as a replacement for useState

3 parameters:

Reducer function

Initial state

Optional: function for initial state

const [emailState, dispatchEmail] = useReducer(emailReducer, { value: '',

isValid: null });

const [passwordState, dispatchPassword] = useReducer(passwordReducer, { value:'', isValid: null });

Reducer function gets the most current state and an action. It returns a new state.

const emailReducer = (state, action) => {

  if (action.type === "EMAIL\_INPUT") {

    return { value: action.val, isValid: action.val.includes("@") };

  }

  if (action.type === "EMAIL\_BLUR") {

    return { value: state.value, isValid: state.value.includes("@") };

  }

  return { value: '', isValid: false }

};

Dispatch call

dispatchEmail({ type: 'EMAIL\_INPUT', val: event.target.value });

When using in combination cu useEffect:

  const {isValid: emailIsValid} = emailState;

  const {isValid: passwordIsValid} = passwordState;

Extract the properties so that useEffect doesn’t run for any change of the state, but only for the properties that we want it to run for.

## useContext

It’s used to get the values from a context. We import the context and then:

const ctx = useContext(AuthContext);

Much more elegant than AuthContext.Consumer { (ctx) => {JSX} }

## useImperativeHandle

Allows us to use / expose this component or functionalities of component imperatively. Not through the state management, instead by directly calling or manipulating something in the component programmatically.

A function that returns an object, that object contains all the data that should be accessible from outside through that name.

  useImperativeHandle(ref, () => {

    return {

      focus: activate,

    };

  });

We need to declare the component in a special way, the ref here will be the ref that we send in the parent component.

const Input = React.forwardRef((props, ref) => {

        <Input ref={emailInputRef}>

We also need a ref inside the <Input> component

  const inputRef = useRef();

  const activate = () => {

    inputRef.current.focus();

  };

      <input ref={inputRef}>

We should avoid this at all cost, but its ok for focus and scrolling.

## useCallback

Used to memoize a function so it’s not redefined every time the component is re-evaluated.

  const changeTitleHandler = useCallback(() => {

    setListTitle('New Title');

  }, []);

Dependency just like useEffect, to re run when the [ ] change.

## useMemo

Used to memoize a result so it’s not redefined every time the component is re-evaluated.

  const sortedList = useMemo(() => {

    return items.sort((a, b) => a - b);

  }, [items]);

Dependency just like useEffect, to re run when the [items] change.

## Custom hooks

We can also create our own custom hooks.

# Building Custom Hooks

Custom hooks are in the end just regular functions but they are functions which can contain stateful logic into re-usable functions.

Unlike regular functions, custom hooks can use other React hooks and React state.

The state that Is used in a custom hook is attached to the component that uses the custom hook. So when that state changes, the component is re-evaluated.

const useCounter = (ascending = true, step = 1) => {

    const [counter, setCounter] = useState(0);

    let increment;

    if (ascending) {

        increment = step;

    } else {

        increment = -step;

    }

    useEffect(() => {

        const interval = setInterval(() => {

            setCounter((prevCounter) => prevCounter + increment);

        }, 1000);

        return () => clearInterval(interval);

    }, [increment]);

    return counter;

}

And then we use it just like a normal hook.

const counter = useCounter();

return <Card>{counter}</Card>;

JavaScript bind method can be used to set a this keyword to a function OR to set a default first parameter from somewhere.

// Create a function with a preset first argument.

const addThirtySeven = addArguments.bind(null, 37); or addArguments.bind(null, 37);

const result2 = addThirtySeven(5);

// 37 + 5 = 42

# Handling Forms and User Input

# Context API

GlobalContext

Wrap the components that we want to have access to some app state to the Provider from our Context created with createContext.

Create context

const AuthContext = React.createContext({

    isLoggedIn: false,

    onLogout: () => { },

    onLogin: (email, password) => { },

});

Create provider

export const AuthContextProvider = ({ children }) => {

return (<AuthContext.Provider value={{

        isLoggedIn,

        onLogin: loginHandler,

        onLogout: logoutHandler

    }}>

        {children}

    </AuthContext.Provider>);

}

React Context is NOT optimized for high frequency changes!

Customizable components should still have props.

# Rules of Hooks

Only call React hooks in react Functions

React component functions

Custom hooks

Only call React hooks at the top level. Don’t call in nested functions. Don’t call in any block statements.

# Debugging React Apps

Understanding error messages, debugging and analyzing react apps, react dev tools.

Changing the state in the react dev tools shows changes in the application.

# React – Behind the scenes

React manages components and state. It sends the information to React DOM. React handles Context, Props and State, if a component wants to show something new on the screen, React DOM is notified.

React determines how the component tree currently looks like and what it should look like.

React DOM manages the DOM and what the user sees. ReactDOM receives the differences and then manipulates the real DOM.

React does virtual DOM diffing, finding differences between 2 snapshots. Re-Evaluating components is not equal to Re-Rendering the DOM. Re-Rendering the DOM happens only when necessary, for differences between evaluations.

## Virtual DOM and DOM updates

Preventing unnecessary re-evaluations with React.memo()

export default React.memo(DemoList);

It tells React that for this component React should look at the props this component gets and check the new value for all those props and compare it to the old values those props got. And only if the value of a prop changed, the component should be re-evaluated.

Memo has a performance hit because it needs to compare props.

Memo doesn’t work when we send functions as props to the component, because the function changes everytime the parent component is re-evaluated and functions are not primitive data types, they are objects. Memo works fine for primitive data types where the === operator works.

We can make memo work for props that are not primitive as well with the help of an extra hook. useCallback

Tell React that we want to save a function and not re-create it with every execution. We save the function somewhere in React internal storage so === works.

  const changeTitleHandler = useCallback(() => {

    setListTitle('New Title');

  }, []);

Anything that we use in the function needs to be added in the second parameter of useCallback, param for dependencies, just like with useEffect.

The dependency is needed because functions in JavaScript are closures. So when we use useCallback, the variables are captured and stored for that function. If something changes, the value is not reflected in the function. If we declare the dependency, then it will update the variable value by recreating the function.

## State and State Updates

React takes care of state management and components. The useState is tied to the component. useState creates a new state but the default value is taken into account only the first time, after that, no new state is being created, and it simply updates it.

If something is removed from the DOM, then the state is removed and reinitialized when the component is added again. (conditional component)

setState schedules a state update, it doesn’t happen instantly, but the order of the updates is guaranteed by React. This is why it is recommended to use the functional version of setState((prevState) => {}) when we depend on the previous snapshot of the state.

useEffect is guaranteed to use the latest state too because of the dependencies mechanism and because it runs for every component re-evaluation.

React batches together all the setState calls in a synchronous function block, so the component is re-evaluated only once.

# Routing

SPA – single page app

MPA – multi page app

Dynamic and Nested routes

<Routes>

          <Route path='/' element={<Navigate replace to='/welcome' />} />

          <Route path='/welcome' element={<Welcome />}>

            <Route

              path='new-user'

              element={<p>Welcome, new user!</p>}

            ></Route>

          </Route>

          <Route path='/products' element={<Products />} />

          <Route path='/products/:productId' element={<ProductDetail />} />

          <Route path='/form' element={<BlockingForm />} />

</Routes>

# Redux

Redux is a state management system for cross component or app-wide state. It helps us manage state across multiple components or the complete app. We have 3 types of state: local state, cross component state and app wide state. Local state belongs to a single component. Cross component affects multiple components.

## Redux vs React Context

React Context can become hard to setup and create heavy JSX code in a very big application where we have a lot of different contexts, so a lot of different providers.

React Context is bad for high frequency updates, but good for rare updates.

Redux has one central data store. One store of all your state for your entire application.

Components subscribe to the central data store and the store notifies the components and give them the data they need. Components don’t manipulate the data in the store. The reducer function mutates the store data. Components dispatch actions that are forwarded to the reducer function.

Providing the store, to provide the store to our react app, we go into the index.js file where we render our root component.

import store from "./store/index";

ReactDOM.render(

  <Provider store={store}>

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

  </Provider>

);

And then in order to use it in a component:

const counter = useSelector((state) => state.counter);

useSelector automatically subscribes the component to the store. Changes to the redux store will cause the component to re-render. When the component is unmount, it is unsubscribed from the store as well.

YOU SHOULD NEVER MUTATE THE EXISTING STATE WHEN WORKING WITH REDUX. ALWAYS OVERWRITE IT WITH A BRAND NEW STATE.

# Authentication

Server side sessions: Store unique identifier on server, send same identifier to client. Client sends identifier along with requests to protected resources.

Authentication Tokens: Create, but not store, permission token on server, send token to client. Client sends token along with requests to protected resources.

# Deploying React Apps

Steps and Pitfalls

Server side routing vs client side routing

# Testing React Applications

Manual testing vs Automatic testing

## Unit Tests

Test the individual building blocks (functions, components) in isolation. The most common.

## Integration Tests

Test the combination of multiple building blocks.

## End to End Tests

Test entire workflows, entire scenarios in your app as the user would experience them.

## Jest

Is a tool for running the tests and asserting the result.

For simulating the react app, we use the React Testing Library.

Both come installed when creating a app with the create-react-app

We can group tests in test suites, using the describe function.

describe("Greeting component", () => {

  test("Greeting text test", () => {

    // Arrange

    render(<Greeting />);

    // Act

    // Assert

    const linkElement = screen.getByText(/hello world/i);

    const linkElement2 = screen.getByText("Hello World", { exact: false });

    expect(linkElement).toBeInTheDocument();

  });

});

# Next.js

The React framework for production.

A fullstack framework for React.

Next.js solves common problems and makes building React apps easier. Adds a lot of features to your React app. For production.

Features:

## Server-side rendering support

React does client side rendering, so the initial html file is almost empty and the Search Engine Optimization, SEO crawlers can’t see what the user sees.

Server-side rendering allows us to preload react pages and components on a server. So the user doesn’t see the loading flicker and the SEO crawlers see the page.

This can be achieved in React too with React DOM Server, but it requires work on your side and it is tricky.

Next.js does this by default. (for the first load, after that it acts normally)

Blending client side and server side: fetch data on the server and render finished pages

## File based routing

Defines pages and routes with files and folders instead of code.

Less code, less work, highly understandable.

## Fullstack capabilities

Easily add backend (server-side) code to your next / react app.

Storing data, getting data, authentication etc. can be added to your React projects.

No need for 2 different projects, can work only on 1.

Project structure:

3 important folders: pages, public and styles

In public we don’t have index.html like we do in React, because pages are pre-rendered.

In pages we have the file based routing.

# Course

Components & Building UIs

Working with events and data props and state.

Styling react apps and components.

React Hooks, Side effects, Refs

React’s Context API and Redux

Forms, Http Requests and Custom Hooks

Routing, Deployment, NextJS