Overview training

React + Express

21.11 - 25.11

sages

Główne obszary działania

Szkolenia

Oferujemy szeroki katalog szkoleń z technologii mainstreamowych i specjalistycznych, wschodzących i legacy. Zajęcia prowadzimy w trybie warsztatowym, a programy są oparte o praktyczne know-how. Specjalizujemy się w prowadzeniu dedykowanych szkoleń technologicznych, których agendę dostosowujemy do potrzeb naszych klientów i oczekiwań uczestników.

Kursy rozwojowe

Posiadamy kursy otwarte (Kodołamacz.pl) i dedykowane akademie dla firm, pozwalające na zdobycie nowych kompetencji w ramach kompleksowych programów rozwojowych dla pracowników. Oferujemy również wsparcie w rekrutacji i edukacji przyszłych pracowników naszych klientów.

E-learning połączony z warsztatami

Jako uzupełnienie szkoleń tradycyjnych oraz formę nauki samodzielnej proponujemy kursy e-learningowe. Aktualnie w naszej ofercie dostępne są szkolenia typu masterclass, rozumiane jako szkolenia wideo, uzupełnione o spotkania/warsztaty na żywo (zdalnie) z autorem kursu.

Studia podyplomowe

Współpracujemy z uczelniami wyższymi wspierając realizację zaawansowanych kierunków studiów z zakresu specjalistyki IT. Jesteśmy partnerami studiów podyplomowych:

na Politechnice Warszawskiej:
Data Science: Algorytmy, narzędzia i aplikacje dla problemów typu Big Data
Big Data: Przetwarzanie i analiza dużych zbiorów danych
Wizualna analityka danych.

na Akademii Leona Koźmińskiego:
Data Science i Big Data w zarządzaniu
Chmura obliczeniowa w zarządzaniu projektami i organizacją.

Wydarzenia IT

Inspirujemy i szerzymy wiedzę o technologiach z różnych obszarów na kilkugodzinnych, praktycznych warsztatach w ramach naszej inicjatywy Stacja IT. Organizujemy konferencje m.in. AI & NLP Day, Testaton. Występujemy na konferencjach wewnętrznych naszych klientów np. Orange Developer Day.

Who am I?

Mateusz Jabłoński

- programmer since 2011
- React / Angular on Frontend
- NodeJS / Java on Backened
- sometimes blogger / trainer / mentor

mateuszjablonski.com
mail@mateuszjablonski.com

From the beginning

The arrangements

- our goal and agenda of workshops
- mutual expectations
- questions and discussions
- flexibility
- openness and honesty

What awaits us?

Day 1.

- React
  - Architecture
  - Idea around the Virtual DOM
  - Components
  - Object programming in React
  - Functional programming in React
  - State management
  - Events
  - Life cycle methods
  - React Hooks

What awaits us?

Day 2.

- React in practice
  - Create React App
  - Complex views
  - Typing components
  - Styling components
    - CSS Modules
    - CSS in JS
    - Styled Components
  - Forms
  - Routing
  - Communication with API
  - Envs in React
  - React Portal

- SSG in Gatsby
- SSR in NextJS
- SSR in React
- React 18 - what’s new?
- Server components

What awaits us?

Day 3.

- State management
    - Context API
    - FLUX
    - Redux
    - Mobx
- Programming technics
    - MPA
    - SPA
    - SSR
    - SSG

- NodeJS
    - CommonJS
    - NodeJS as a server
    - Global objects in Node
    - Modules
    - File System

What awaits us?

Day 4.

- ExpressJS
    - Introduction
    - Installation and configuration
    - Project structure
    - Routing
    - Communication
    - Error handling
    - HTML in ExpressJS
    - Response and Request
    - Templates engines
    - Complex views: Partial, Layout, Blocks
    - Forms
    - Validation
    - Upload
- Authorization and authentication
    - Cookies and user session
    - Register and login
    - Middlewares
- Databases
- Rest API
- Swagger
- SocketIO

What awaits us?

Day 5.

- Testing
    - Definition and scope of unit tests
    - What and how test?
    - Black box vs. white box
    - Possibilities of React Testing Utilities
    - Testing React components
    - Events in tests
    - Shallow / full rendering
    - Contract testing
- Storybook
- Performance
- Proxy

github.com/matwjablonski/bsh-training

React

Babel

Babel in React

- Babel is one of the basic tools used in React app development process
- Configuration is kept in .babelrc file
- Configuration contains two groups of rules:
  - presets - rules of code transformations
  - plugins - allows extend Babel engine

```json
{
  "presets": ["@babel/preset-react"],
  "plugins": ["react-hot-loader/babel"]
}
```

React

Basics

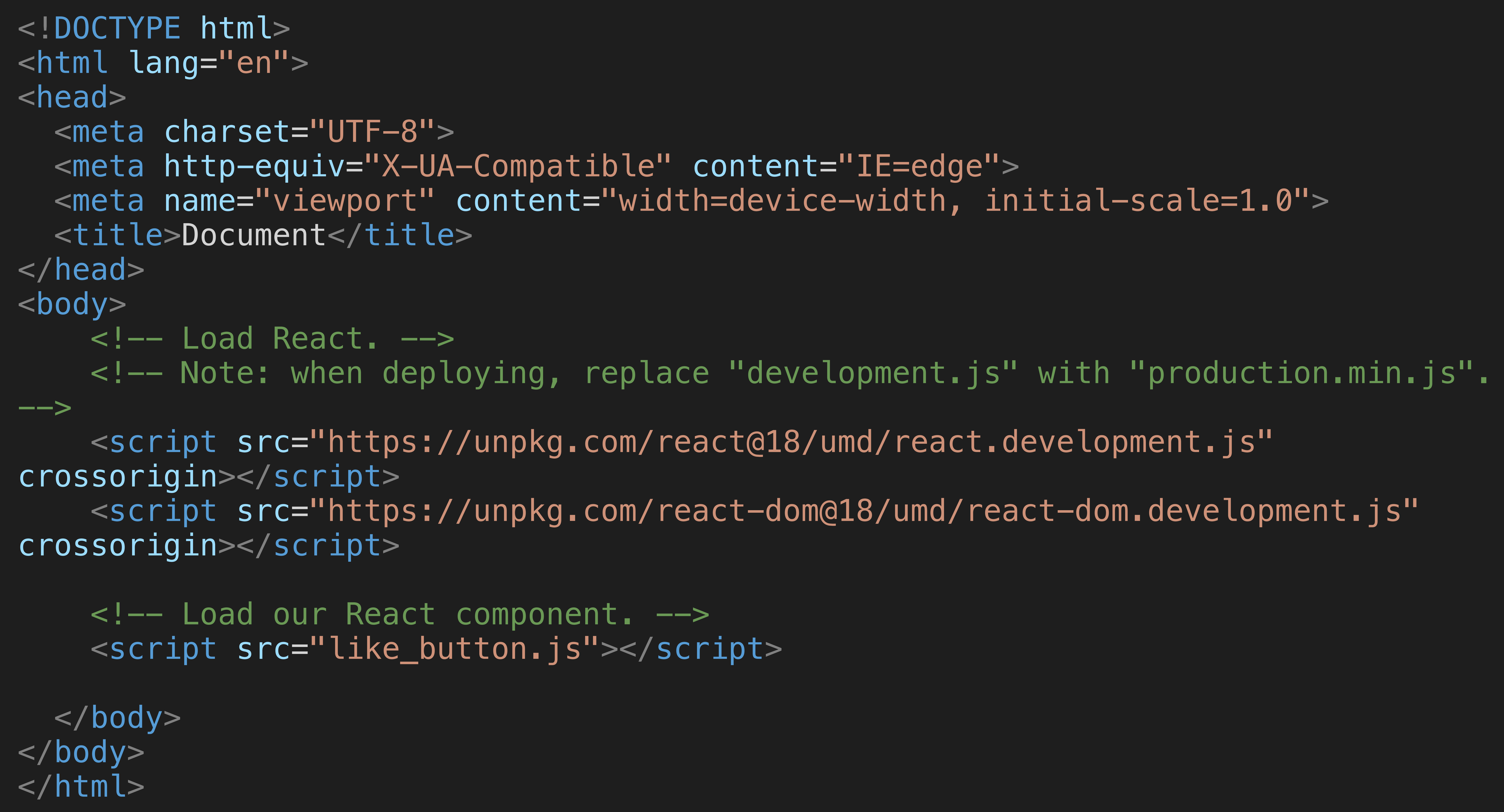
React

React

- React is a library which can be used for building dynamic, complex user interfaces in declarative and module way
- It takes off responsibility for rendering and updating state of DOM tree from programmer
- Allows declaring structure and logic of displaying content in more declarative way (more natural than in object oriented methods or in imperative programming)
- Allows to hold structure and logic in one place (**only with Javascript** - programmer can use all possibilities of Javascript, without touching and learning syntax of markup languages)

!

**React doesn’t render changes directly,
but through Virtual-DOM**



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
    <!-- Load React. -->
    <!-- Note: when deploying, replace "development.js" with "production.min.js".
-->
    <script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>

    <!-- Load our React component. -->
    <script src="like_button.js"></script>

  </body>
</body>
</html>
```

React

Virtual DOM

Virtual DOM

- VirtualDOM is simplified representation of DOM (Document Object Model) objects, which are in the browser. Rendering cycle looks like:
    - React will build VirtualDOM tree in a declarative way, in the next step render it as a browser DOM
    - In case of state change React will build new Virtual DOM tree of changed component
    - ReactDOM is comparing actual and new tree, after that it updates only those places, which really need to be change

!

Due to the small number of DOM operations, React updates the view instantly!

Virtual DOM

- Allows to generate abstract DOM, which can be rendered as UI on a lot of platforms:
    - **react-dom** - browser DOM
    - React Native - native apps for iOS and Android
    - react-blessed - terminal
    - react-canvas - HTML Canvas element
    - react-vr / react 360 - 3D applications.

React

Attributes and classes

```
React.createElement(
  'div',
  {
    id: 'root_element',
    className: 'root-element-class',
    style: {
      borderTop: '1px solid black',
    }
  },
  "Hello!"
)
```

React

JSX

```
const Section = <section>
  <h1>title</h1>
  <h2 className="subtitle" id="hook_to_title">daily</h2>
  <ul>
    <li>ts</li>
    <li>react</li>
  </ul>
</section>;

React.render(Section, document.getElementById('app'));
```

```
const section = {
  title: 'title',
  subtitle: 'daily',
}

const Section = <section>
  <h1>{section.title}</h1>
  <h2 className="subtitle" id="hook_to_title">{section.subtitle}</h2>
  <ul>
    <li>ts</li>
    <li>react</li>
  </ul>
</section>;

React.render(Section, document.getElementById('app'));
```

```
const section = {
  title: 'title',
  subtitle: 'daily',
  items: [
    {
      id: 1,
      name: 'ts',
    },
    {
      id: 2,
      name: 'react',
    },
  ]
};

const Section = <section>
  <h1>{section.title}</h1>
  <h2 className="subtitle" id="hook_to_title">{section.subtitle}</h2>
  <ul>
    {section.items.map(item => <li key={item.id}>{item.name}</li>)}
  </ul>
</section>;

React.render(Section, document.getElementById('app'));
```

React

Components

Components

- Basic block of React application is Component
- Component is an element in DOM tree, which is manage by React
- **Function / Class of the component** contains Javascript code which controls look and behavior of the element.
- Instance of component is an element, which had been re-render by React in usage of its class or function.
- Components can be nested in any structures, same as HTML or XML
- We can pass to component data as attributes same as it is in HTML. In contrast to HTML we can pass any objects (not only strings). Passed params are known as **component properties** (in short: **props**)

```
<MyComponent option={variable} title="text">
  Text or other components
</MyComponent>
```

```
// Pseudoclass ES6

var createReactClass = require('create-react-class');
var Greetings = createReactClass({
  render: function() {
    return React.createElement('h1', {}, ,Hello,' + this.props.name)
    // return <h1>Hello, {this.props.name}</h1>
  }
});
```

```jsx
// ES6 class

class Greetings extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

```
// TS class

interface GreetingsProps {
  name: string;
}

class Greetings extends React.Component<GreetingsProps> {
  public render(): ReactNode {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

```
// function component

const Greetings = ({ name }) => {

  return <h1>Hello, {name}</h1>
}
```

!

Function component - previously known as stateless function component. From introduction of Hooks function component can have its own state and manage it.

```tsx
// function component in ts

interface GreetingsProps {
  name: string;
}

const Greetings = ({ name }: GreetingsProps) => {

  return <h1>Hello, {name}</h1>
}

const Greetings2: React.FC<GreetingsProps> = ({ name }) => {

  return <h1>Hello, {name}</h1>
}
```

React

Diffing Algorithm

Elements Of Different Types

- In diffing process React takes two trees and check at the beginning types of the root elements
  - When they are different - React purge old tree and create new based on Virtual DOM
  - While tearing down:
    - DOM elements are destroyed
    - Component instances will receive **componentWillUnmount** event (lifecycle method)
    - New tree will be rendered in place of old one
    - New components will receive information that components are mounted

DOM Elements Of Same Type

- When they are same:
  - React takes arguments of both and compare them and in next step it update only those who needs to be updated, like **styles**, **className** etc.
  - Without destroying tree it goes to children and start diffing on children elements.

Components Elements Of Same Type

- When they are same:
  - Between re-renders the instance of component stays the same
  - Component will be informed that something changed and it should re-render itself

Recursing on children

- By default React iterates over lists of children at the same time and generate mutations whenever there is a difference
- If React found anything new in the end of the list it will just add it to the list
- Worse if we are adding item to the beginning of the list (it has worst performance because every next item will be different, so React need to update whole list)
    - To handle that situation we have **key** attribute

Key

- Element in collection should have an **key** attribute.
- Key should be unique and stable between every component render. Due to that React can update lists in most optimal way, avoiding unnecessary re-renders, using as low number of operation as possible on DOM.
- If key will have new value, React re-render whole list once again.
- For the DOM updates is responsible Diffing Algorithm. Process of DOM update is reconciliation.

!

Diffing algorithm is just an implementation detail. We can avoid using it. Then whole app will be re-render on every change.

React

Create React App

```json
{
  "name": "test2",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.16.5",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Task

25 minutes
materials/TASK_14.md

React

State and Props

State and Props

- React „reacts” on data changes and in case of change execute next render of VirtualDOM.
- Rendering can be executed in two scenarios:
  - When component received new properties (**props**)
  - When internal state of component had changed (**state**)
- Passing new properties to element or change of its internal state case re-render of component.
- We shouldn’t update manually props, either state

```
import React from 'react';

const Todo = props => (
  <div>
    <h3>{props.title}</h3>
  </div>
);

const App = () => (
  <Todo title="Say hello” />
)
```

```
import React from 'react';

interface TodoProps {
  title: string;
}

class Todo extends React.Component<TodoProps> {
  render() {
    return (
      <div>
        <h3>{this.props.title}</h3>
      </div>
    )
  }
}

class App extends React.Component {
  render() {
    return (<Todo title="Say hello” />)
  }
}
```

```
import React, { Component } from 'react';

function Clock(props) {
  return <h3>{props.name}</h3>
}

Clock.defaultProps = {
  name: 'Hello'
}
```

```
import React, { Component } from 'react';

class Clock extends Component {
  // default props
  static defaultProps = {
    name: 'Hello',
  }

  constructor(props) {
    super(props);

    // default state
    this.state = { date: new Date() }
  }
}
```

Updating state

- React have to know, when state of component changed. **Don't modify state directly.**
- For changing state we should use function: **setState** or hook **useState**.
- State methods work asynchronously.
- When state is dependant on previous state we should use callback as first argument of changing state method.

- **Class Way:**
  - Changes are merged (when changing only one value of state object, previous stays the same)
  - Using value of state just after update should cause an error (component could not re-render in time)
  - **this.replaceState({})** - instantly overwrite whole **this.state** object

- **Hook Way:**
  - Changes are overwrited

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

```
interface AProps {

}

interface AState {

}

class A extends React.Component<AProps, AState> {

}
```

```
class Comp extends React.Component {

}

Comp.propTypes = {
  name: PropTypes.string,
}
```

Events

- On events in React can be listen directly on rendered elements
- We can manage on them in components code, object of event is passed as param (prop)
- We can execute code directly

```
const Input = () => {
  const [value, setValue] = useState()

  return (
    <input value={value} onChange={(e: SyntheticEvent) => setValue(e.target.value)}
  )
}
```

```
class Input extends React.Component {
  handleChange(e) {
    this.setState({value: e.target.value})
  }

  render() {
    return (
      <input value={this.state.value} onChange={this.handleChange} />
    )
  }
}
```

Forms

- By adding to form field an value attribute, field is changing into **controlled component**
- **Controlled component** means that value of it, is controlled by React component (in most cases its connected with state of component)
- When field is controlled component only component can change it (on UI input wouldn’t react without implementation of change functionality)
- By default all fields are uncontrolled components (browser is responsible for changing value inside inputs) (without attribute value)
- Code `<input value={this.state.myValue} />` makes field impossible to change manually
- To change value of controlled field we should use one of two methods: **setState** or **useState**
- Elements of type **<select>** or **<textarea>** also have attribute **value**

Task

25 minutes

materials/TASK_15.md

Components nesting

- Components can be nested
- Any code which is JSX Element passed between open and close tags of component will be accessible as JSX Object as variable **props.children**
- We can nest also **strings**, **arrays** of **jsx elements** and everything which fulfills type **ReactNode**

```
const LastItem = () => <p>I am last!</p>

const List = ({ data, children }) => {
  return (
    <ul>
      {data.map(item => <li key={item.id}>{item.title}</li>)}
      {children}
    </ul>
  )
}

const App = () => (
  <List data={[]}>
    <LastItem />
  </List>
)

export default List;
```

!

Prop drilling (also: threading) its a process of passing data by props to the bottom of components tree to target component.

React

Lifecycle methods

Life cycle of component

- Every component has implemented life cycle process.
- Life cycle contains methods which are stages of existence component in React VirtualDOM structure.
- Those stages represents different points in life of component from mount to unmount (remove from UI).
- Mounting:
  - **UNSAFE_componentWillMount()** - before mounting in DOM
  - **componentDidMount()** - after mounting in DOM
- Updating:
  - **UNSAFE_componentWillReceiveProps(newProps)** - component will receive new props
  - **getDerivedStateFromProps(newProps)** - component received new props
  - **shouldComponentUpdate(newProps, newState)** - If return false, React omit next render
  - **UNSAFE_componentWillUpdate()** - component will be render, don’t change the state
  - **componentDidUpdate()** - component is rendered, DOM is stable
  - **getSnapshotBeforeUpdate(prevProps, prevState)** - just before update browser DOM

Life cycle of component

- Unmounting:
  - **componentWillUnmount()** - before remove component from DOM
- Exceptions:
  - Methods are executed in case of exceptions while rendering process, in lifecycle methods or constructor methods of child components
  - **getDerivedStateFromError(newProps)**
  - **componentDidCatch()**

React

Operating on DOM

Its not advisable manipulating DOM generated by React, but there is such a possibility. To access DOM elements we can use references.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.inputRef = React.createRef();
  }

  componentDidMount(): void {
    this.inputRef.current.focus();
  }

  render(): React.ReactNode {
    return (
      <form>
        <input ref="inputRef"/>
      </form>
    )
  }
}
```

```
const Form = ({ data }) => {
  const inputRef = useRef<HTMLInputElement>(null);

  const handleSubmit = (e) => {
    e.preventDefault();

    inputRef.current && inputRef.current.focus();
  }

  return (
    <form onSubmit={handleSubmit}>
      <input ref={inputRef}/>
    </form>
  )
}
```

Dealing with DOM

- To have an access to reference we can use:
  - **ReactDOM.findDOMNode(this.refObj.current);**
  - Above example is kind of emergency exit, docs advise against using that function, because it distorts abstraction of components structure
  - Remember that references will be available only when component is rendered
  - Use references carefully, they should be called only in specific life cycle methods (also keeping data inside references will omit re-render process after change that data)

```
return (
      <form>
        <input ref={element => element.focus()}/>
      </form>
    )
```

React

Hooks

React Hooks

- Functions in Javascript doesn’t have state, it means that every execution has access only to it arguments (props in React) and variables from upper scope (closure)
- In React components are functions - so by default they cannot keep state between next calls (re-renders)
- Hooks are functions, which can be places inside component, but only in top scope of component
- Hooks add possibilities to use references, state and side effects inside functional components
- Hooks can be called only in React components
- We can create our own hooks (name of this kind of function should be prefixed by **use** keyword)

!

Order of executing Hooks in important!

```
const Item = ({ data }) => {
  const [ clicked, setClicked ] = useState(0);

  const handleClick = () => {
    setClicked((prevValue) => prevValue + 1);
  }

  return (
    <div onClick={handleClick}>I am clicked {clicked} times</div>
  )
}
```

useState

- It is used for changing state of component
- While calling this hook we can pass default value
- In the component we can have more then one calls of this hook to handle different state for different uses
- **useState** overwrites whole state (in contrast to **setState**)
- Default state can be a function, then while creating state default value will be value returned from this function

Task

25 minutes
materials/TASK_16.md

```
function MyComponent(props) {
  const [state, setState] = React.useState(() => {
    return calculations(props);
  });
}
```

useEffect

- **useEffect** allows to create “side effects”
- After every DOM render we can execute imperative operations on / outside component using hook **useEffect**
- For example we can set title of browser window, fetch data from server / api etc.

```
const WindowTitleHandler = () => {
  const [title, setTitle] = useState('Default title')

  useEffect(() => {
    document.title = title;
  })

  return <div>
    <button onClick={() => setTitle('New title’)}></button>
  </div>
}
```

Every render of component cause new execution of effect. It works similar to life cycle methods: componentDidMount() and componentDidUpdate()

```
const WindowResizeWatcher = () => {
  const [windowWidth, setWindowWidth] = useState(window.innerWidth);

  const handleResizeAction = () => {
    setWindowWidth(window.innerWidth);
  }

  useEffect(() => {
    window.addEventListener('resize', handleResizeAction);

    return () => {
      window.removeEventListener('resize', handleResizeAction);
    }
  })

  return <div>
    Moje okno ma teraz {windowWidth}px szerokości
  </div>
}
```

useEffect

- Second argument of **useEffect** is an array of dependencies (from which depends execution of effect)
  - When second argument is not passed, **useEffect** will be executed after every render
  - When second argument is an empty array, **useEffect** will be executed only once
  - When second argument array contains dependency, **useEffect** will be executed when any of it change

```
useEffect(() => {}, []);
// only after first render

useEffect(() => {});
// after every render

useEffect(() => {}, [ props.title, value ]);
// after change of dependencies
```

useLayoutEffect

- Behavior similar to **useEffect**
- Difference is that it is executed synchronously after mutation in DOM structure
- Useful when we want our view complies with data in situation of mutation DOM structure directly

Other hooks

- useContext
- useRef
- useReducer
- **useMemo** - memoization of values
- **useCallback** - memoization of callbacks (functions)
- **useImperativeHandle** - customizes the instance value that is exposed to parent components when using ref, similar to useRef, but gives control over the value that is returned and allows to replace native functions like blur, focus with custom implementations
- **useDebugValue** - can be used to display a label for custom hooks in React DevTools
- **useId** - generates unique ID, which is stable between every render
- **useTransition** - we can specify, which state update should have bigger priority
- **useDefferedValue** - can show old value, before new will be ready (usefull when view depends on values from external source, like libraries, other components)

Task

25 minutes

materials/TASK_17.md

React

Communication with API

```
const List = (props: any) => {
  const [data, setData] = useState([]);

  const fetchData = async () => {
    const res = await fetch('//example.com/api/data');
    const data = await res.json();

    setData(data);
  }

  useEffect(() => {
    fetchData();
  }, [])

  return <div>
    {data.map(item => <ListItem />)}
  </div>
}
```

```
const [code, setCode] = useState([]);
  const [isLoading, setIsLoading] = useState(false);

  const fetchCode = () => {
    setIsLoading(true);
    fetch('//example.com/api/data')
      .then(res => res.json())
      .then(data => {
        if (data.status === 404) {
          throw new Error()
        }

        setCode(data);
      })
      .catch((err) => {
        // handle error
      })
      .finally(() => {
        setIsLoading(false);
      })
  }
```

```
const [isLoading, setIsLoading] = useState(false);

const fetchData = async () => {
  setIsLoading(true);
  try {
    const res = await fetch('//example.com/api/data');
    const data = await res.json();

    if (data.status === 404) {
      throw new Error()
    }

    setData(data);
  } catch(err) {

  } finally {
    setIsLoading(false);
  }
}

useEffect(() => {
  fetchData();
}, [])
```

Task

25 minutes

materials/TASK_18.md

React

React Router DOM

Router

- Allows to „replace” rendered component on view based on current url in browser
- Operate on object location (of course in customized version)
- Allows to add params to paths
- Allows using dynamic links (based on variables)

```
import React from "react";
import ReactDOM from "react-dom/client";
import { BrowserRouter } from "react-router-dom";

import "./index.css";
import App from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

```
<Routes>
  <Route path="/" element={<Layout />}>
    <Route index element={<Home />} />
    <Route path="about" element={<About />} />
    <Route path="dashboard" element={<Dashboard />} />
    <Route path="post/:id" element={<Post />} />
    <Route path="*" element={<NoMatch />} />
  </Route>
</Routes>
```

```
<nav>
  <Link to="/">Home</Link>
  <Link to="/about">About</Link>
  <Link to="/dashboard">Dashboard</Link>
  <Link to={`/post/${id}`}>New post</Link>
</nav>
```

```
const Post = () => {
  const { id } = useParams()

  return (
    <div>
      Post ID: ${id}
    </div>
  )
}
```

```
const navigate = useNavigate();

navigate(`/transaction/${id}`);
```

Task

25 minutes

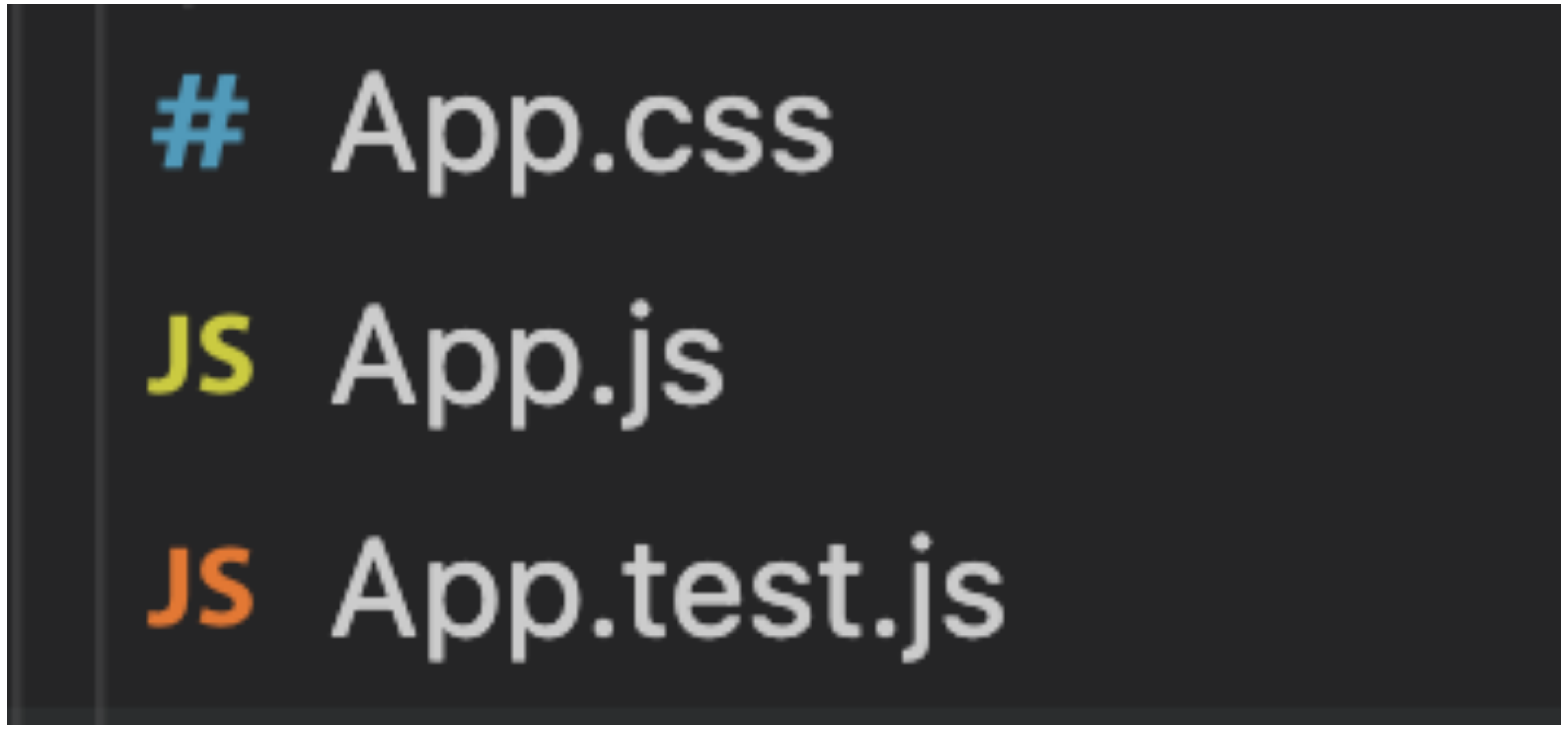
materials/TASK_19.md

React

css

CSS Modules

- In CRA default way of declaring styles
- Allow to use preprocessors, like: Sass
- File with styles are treated as modules
- Every component should have own file / module with styles
- Name of every styles module should be suffixed with **.module.${ext}**
- Easy way to have encapsulated styles
- Global styles are kept in separate directory



App.css

JS App.js

JS App.test.js

JSS

- First concept of creating styles with JS
- It was proposed and developed in 1996 by Netscape company (this solution hadn’t have enough support, so it died)
- Currently JSS is a similar concept of keeping CSS in JS, based on creating styles with javascript objects in reusable way
- JSS is framework - agnostic
- To use JSS in React project we can use package: **react-jss**

```
jss.setup(preset())

const styles = {
  '@global': {
    a: {
      textDecoration: 'underline'
    }
  },
  withTemplates: `
    background-color: green;
    margin: 20px 40px;
    padding: 10px;
  `,
  button: {
    fontSize: 12,
    '&:hover': {
      background: 'blue'
    }
  },
  ctaButton: {
    extend: 'button',
    '&:hover': {
      background: color('blue').darken(0.3).hex()
    }
  },
  '@media (min-width: 1024px)': {
    button: {
      width: 200
    }
  }
}
```

Styled Components

- Based on JSS
- Library for creating styles as components
- Visual component can be created as dedicated for component or as separate visual components for out app
- Behave and work as other JSX elements
- Allow to extend React components by adding styles to them
- Styled components are reusable, they can use props and keep inside logic connected with styles changes

```
import styled from 'styled-components';

const Header = styled.div`
  background-color: #ddd;
  color: blue;
  border-bottom: 2px solid;
  padding: ${({ padding }) => `${padding}px`}
`

// usage

<Header padding={20}/>
```

```
const functionA = value => {
  return value[0] + 'string';
}

functionA`test`; // teststring
```

Task

25 minutes

materials/TASK_20.md

React

Environment variables

.env

- In CRA project we can declare and use environment variables
- Names of that variables should be prefixed by **REACT_APP_**
- Declared variables shouldn’t keep secrets
- Never keep secrets in repository
- Remember that those value will be passed into build code

!

Never keep secrets inside React app.
(especially private API keys)

```
const buildUrl = (endpoint: string): string =>
          `${process.env.REACT_APP_API_URI}/${endpoint}`;
```

React

Managing permissions

```
enum Roles {
  ADMIN = 'admin',
  USER = 'user',
}

interface ProtectedRouteType<T = {}> {
  children: ReactNode;
  necessaryRole?: Roles;
  userRole?: Roles;
  auth: boolean;
}

const ProtectedRoute = ({ userRole, auth, necessaryRole, children }: ProtectedRouteTy
=> {
  const canActivate = () => !!(auth === true && userRole === necessaryRole);

  if (!canActivate()) {
    return <Navigate to="/" replace />
  }

  return <>{children}</>;
}
```

```
<Routes>
  <Route path="/" index element={<HomePage />}/>
  <Route path="/history" element={<TransactionsPage />}/>
  <Route path="/protected" element={
     <ProtectedRoute auth={false}>
         <TransactionsPage />
     </ProtectedRoute>
   } />
</Routes>
```

Task

25 minutes

materials/TASK_21.md

React

Portals

Portale

- It is mechanism that allows render React elements outside the place in structure where it should to be.
- Allows to render elements in any place in DOM (Document Object Model) or even in dependant windows (new Window).
- Portal can be useful, when styling of component restrict possibilities to proper render element, e.g. when component has set z-index or overflow with value hidden.
- Element rendered through Portal is over React control.
- Components rendered in Portal can share state, have access to passed props or to Context API.
- Event bubbling will work same as without Portal, so events fired inside Portal will be propagated to top of the React tree.

```
import { createPortal } from 'react-dom';

const Modal = ({ children }) => {
  return createPortal(
    <div>{children}</div>,
    document.getElementById('modal'),
  )
}
```

```html
<body>
  <div id="app"></div> /* React app */
  <div id="modal"></div> /* Portal */
</body>
```

Task

25 minutes

materials/TASK_22.md

React

Context API

Context API

- Context enable passing data inside component tree without passing them through props of child components
- If purpose is only avoid prop drilling through few levels of React tree, simpler and better solution will be use of components composition.
- When Context is making decision about what should be rendered once again, its checking references. It means that in some cases next render of Providers parent can result in unwanted re-render all consumers for specific context.
- Every context object has its own Provider component, which enable components to subscribe on changes in that context.
- React component subscribed on changes in context is Consumer. Consumer can listen on changes from inside of component.

React

Composition

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}
```

So What About Inheritance? At Facebook, we use React in thousands of components, and we haven’t found any use cases where we would recommend creating component inheritance hierarchies.

reactjs.org

React

Code splitting

Code splitting

- With growing of codebase of application, grows also size of packages.
- Code Splitting is a functionality supported by tools like: Webpack, Rollup, Browserify
- Code Splitting allows creating a lot of small packages of data, which can be loaded in time of app working (when they are needed)
- Good place to split an app with Code Splitting are router paths
- Way to split code in React app:
    - import()
    - React.lazy()

```
const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading…</div>}>
      <Routes>
        <Route path="/" element={<Home />}/>
        <Route path="/about" element={<About />}/>
      </Routes>
    </Suspense>
  </Router>
);
```

```
import React, { Component } from 'react';

class App extends Component {
  handleClick = () => {
    import('./moduleA')
      .then(({ moduleA }) => {
        // użyj modułu A
      })
      .catch(err => {
        // obsługa błędów
      });
  };

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Load</button>
      </div>
    );
  }
}

export default App;
```

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

React

Strict Mode

<React.StrictMode>

- Added to React in 2018 (at the beginning only for class components)
- It should helps developers to update application by resign from old (unsupported APIs)
- Adds set of tips and warnings for helping to avoid typical traps in development process
- Helps to avoid keeping in code / using impure component functions
- Double-invoke functions that are expected to be idempotent
- Works only in development mode
- Side effect of it is double render of application in development process (it allows to catch async side effects, which shouldn’t exist in an app)

Techniques

SPA, SSG, SSR

SPA

- SPA - Single Page Applications
- SPA is an alternative for MPA (Multi Page Applications)
- Server return simple HTML file which contains only one div element and informations about most necessary styles and scripts
- Response from the server is very fast
- All logic of an application and navigation is working on browser side
- Problematic SEO
- Problem with sharing content in social media (meta tags)
- SPA can build apps in modular and components way
- No big requirements from server (only html, css and js static files)
- Cost of infrastructure moved partially to end user

SSG

- Static Generation describes the process of compiling and rendering a website or app at build time
- The output is a bunch of static files, including the HTML file itself and assets like JavaScript and CSS
- Gives us the ability to serve the entire content on first load.
- Page is fast, scalable and no need custom server (it can be hosted almost everywhere)
- Tools:
  - NextJS
  - Gatsby

Gatsby

- React-based, open source framework
- It offers built-in performance, scalability and security
- Its good for SEO
- There is quite big community around it
- Content can be added in Markdown, JSX or through CMS by API
- Its not best options for big apps with a lot of changing data
- Apps are static, so its not good choice for completely dynamic apps

SSR

- Server Side Rendering is a new MPA
- Application is rendered on server side on demand. Server response contains partially prepared view in HTML file.
- Better for SEO, there is possibility to add meta tags (specific for every page)
- SSR allows optimize views with a lot of components
- Sometimes SSR is not best approach (usually for small components with lazy loading)
- SSR websites don’t have access to browser objects (like window)
- It cannot be easily used with PWA (Progressive Web Applications)

NextJS

- React-based framework
- Development is easier and quicker because of additional abstractions (like directory-based routing)
- Support for typescript
- It use hybrid technique for serving apps (SSR + SSG)
- Give an opportunity to create own API

NextJS
CLI

- Tool from Next team to create and manage a nextjs app
- npx create-next-app (without arguments) it runs interactive configurator to setup an app
- —example api-routes - allows to create app based on an example (in NextJS repository is a bunch of examples)
- —ts, —typescript - support for typescript
- next [build, start, export, dev, lint, telemetry, info]

NextJS
Project structure

- **pages/ (!)**
- *pages/api/ (?)*
- *styles/ (?)*
- *layouts/ (?)*
- ***components/ (?) (!)***
- public/
- **lib/ (!)**

  - **(!) - eslint**
  - *(?) - opcjonalny*

NextJS
Project structure

- We can overwrite base app file by create file **_app.tsx** in directory **pages/**
- Useful when we want to keep state between views (and it should not depend of navigation), add global styles or scripts, add additional props to views
- When we want to have access to tags like: <html>, <body> we can create file **_document.tsx** and manage them there
- Next.js omit definition of tags <html> and <body>
- **pages/404.tsx** - own 404 error page
- directories like **pages/** can be keep in directory src (default support)

NextJS
Routing

- Routing based on directories structure
- **pages/index.tsx** - home page
- **pages/contact.tsx** or **pages/contact/index.tsx** - page contact
- **pages/category/[categoryId].tsx** - category page identified by categoryId
- **pages/user/[…all].tsx** - for all paths, e.g.:
  - **/user/1/settings**
  - **/user/1/settings/newsletters**

NextJS
SSG or SSR

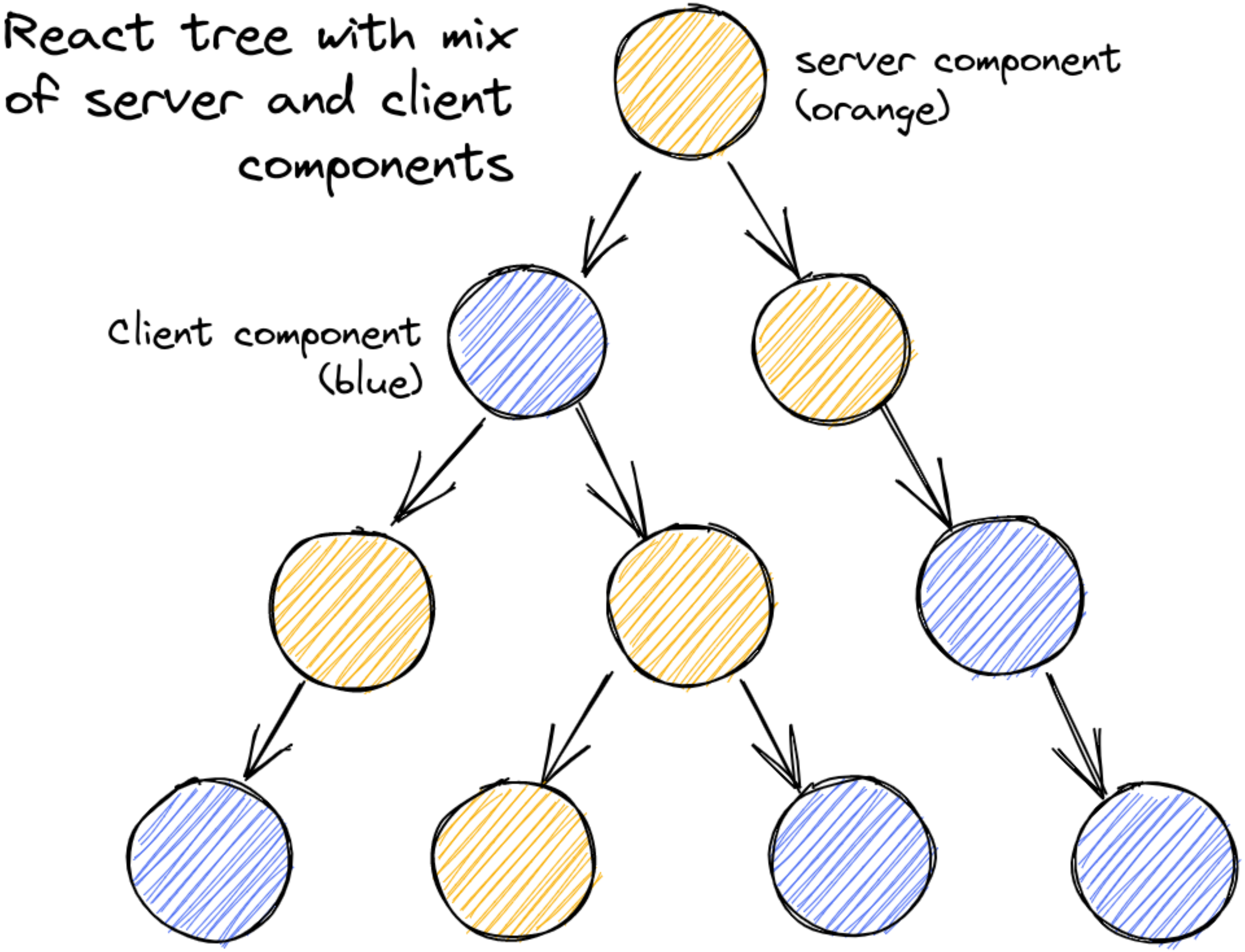
- We can choose which view should be render with which technique by exporting additional function from Next Page file
- We have access to params of request from arguments of functions:
  - getServerSideProps - for SSR
  - getStaticProps - for SSG

NextJS
API Routes

- Allows to build own API
- We can use environment variables
- Allows to manage different HTTP methods and responses like in NodeJS
- It is different way to fetch data (apart from SSR and SSG)
- NextJS can be also backend application

React Server Components

- React Server Components allows the server and the client (browser) to collaborate in rendering your React application
- RSC makes it possible for some components in this tree to be rendered by the server, and some components to be rendered by the browser
- React Server Component is not server-side rendering (SSR)
- We can do server-side rendering with server components and hydrate them properly in the browser
- The server has more direct access to your data sources
- The server can cheaply make use of “heavy” code modules, like an npm package for rendering markdown to html


React tree with mix of server and client components
server component (orange)
Client component (blue)

ReactDOMClient

- Package provides client-specific methods used for initializing an app on the client
- Most of our components should not need to use this module
- Has exported two methods:
  - **createRoot** - create a React root for the supplied container and return the root
    - Root can be **render** and **unmount**
  - **hydrateRoot** - similar to createRoot, but is used to hydrate a container whose HTML contents were rendered by ReactDOMServer

ReactDOMServer

- ReactDOMServer APIs let to render React components to HTML
  - renderToPipeableStream
  - renderToReadableStream
  - renderToNodeStream
  - renderToStaticNodeStream
  - renderToString
  - renderToStaticMarkup

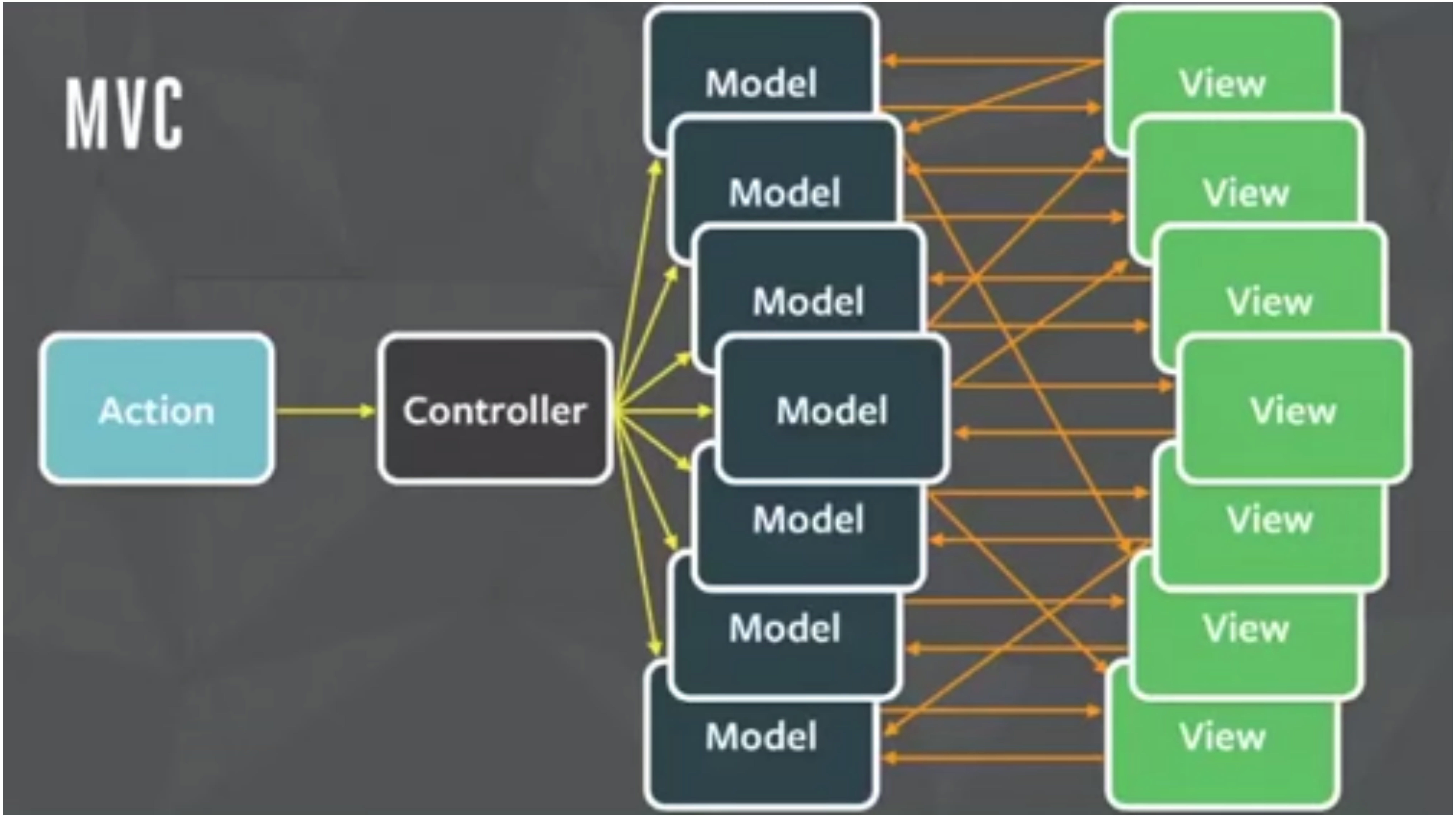
FLUX

Architecture

Problems with MVC

- Models and views create many bi-directional connections
- User action can have impact to many views and models, which can change other models… etc.
- It difficult to define direction of data flow
- It’s easier to make mistake

MVC



Action

Controller

Model

Model

Model

Model

Model

Model

Model

View

View

View

View

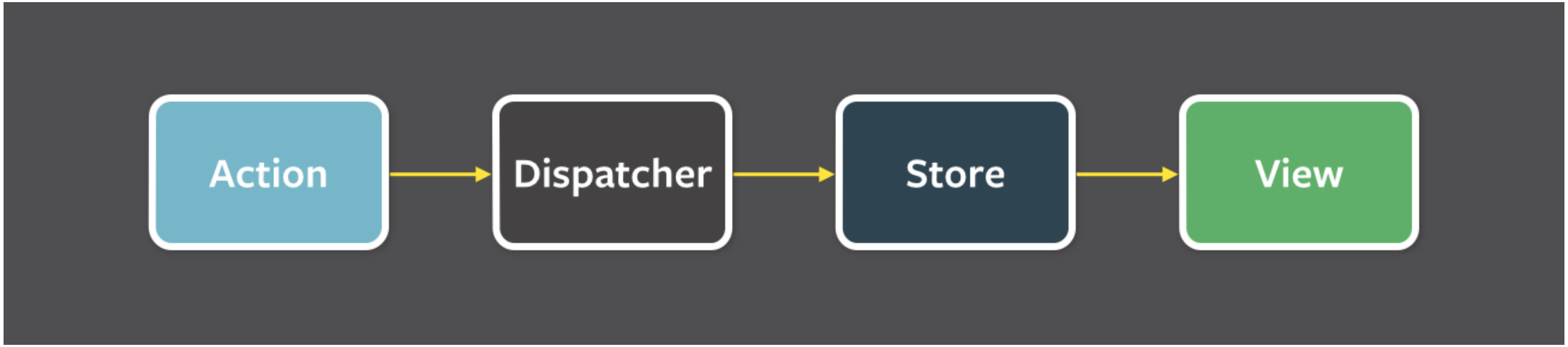
View

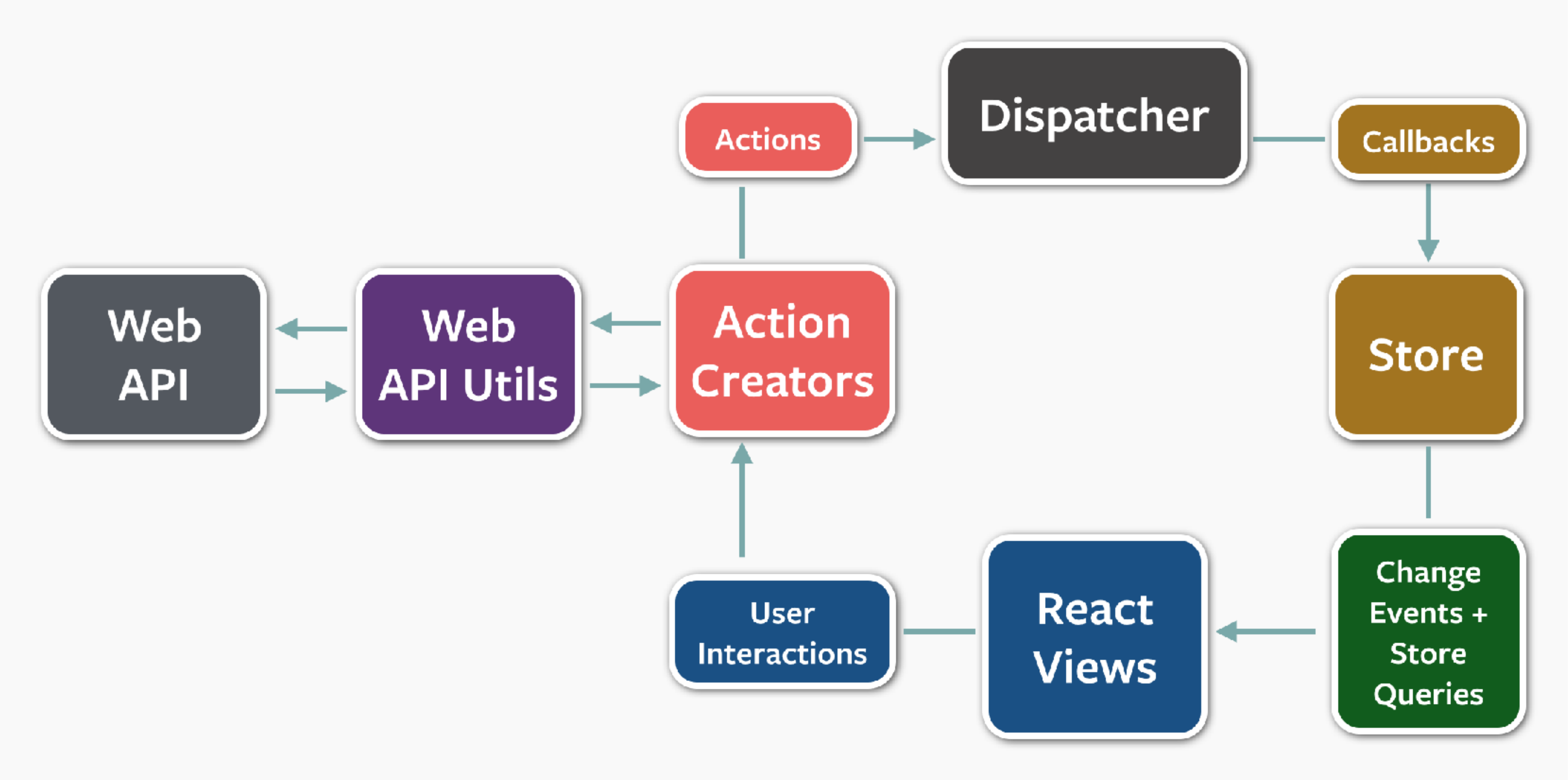
View

View

FLUX

- FLUX architecture assumes one-directional data flow
  - Actions are only one way to change state of an application
  - Dispatcher pass actions to specific Stores
  - Store is one source of truth, it change its own state in reaction on action. Application state is fetched from stores and passed to views
  - View observe stores and renders changes in application


Action
Dispatcher
Store
View


Actions
Dispatcher
Callbacks
Web API
Web API Utils
Action Creators
Store
User Interactions
React Views
Change Events + Store Queries

Actions

- Actions arę objects which contains at least two properties:
  - **type** - action type, based on it store know how to deal with payload
  - **payload** - data connected to the action

```
const ADD_FLIGHT = {
  type: 'ADD_FLIGHT',
  payload: {
    id: 1235,
    name: 'WAW/SFO',
  },
};

const REMOVE_FLIHT = {
  type: 'REMOVE_FLIGHT',
  payload: {
    id: 1235,
  },
};

const UPDATE_FLIGHT = {
  type: 'UPDATE_FLIGHT',
  payload: {
    id: 1235,
    name: 'WAW/SFO',
    status: 'canceled',
  },
};
```

Store

- Store has two parts:
  - Object which keep state
  - Function which handle incoming actions, modifying state of store and notifying about changes
- Good practice is creating an object (class), which has those roles

```
class FlightStore extends EventEmitter {
  constructor() {
    this.store = {
      flights: [],
    }
  }

  handleAction(action) {
    switch(action.type) {
      case 'ADD_FLIGHT':
        this.store.flights.push(action.payload);
        break;
    }

    this.notifyViews(this.state);
  }
}
```

Advantages

- Because of immutable stuctures of data, detecting changes becomes clear and efficient - we just need to compare state object with it's previous version
- When object is the same, nothing changed and there is not need to update view

```
if (newState !== state) {
  state = newState;

  this.notifyViews(this.state);
} else {
  // no changes - without update
}
```

Redux

Redux

Architecture

Redux

- Redux is similar architecture to Flux, inspired functional programming
- Main assumptions:
  - One source of application state
  - Immutable state

```
import { configureStore, ThunkAction, Action } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/counterSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});
```

Reducers

- Reducer can work with nested state
- Reducer should always return new object when there was any change in it
- It’s not allowed to modifying state (excluding reducers)

```
function reducer (state, action) {
  switch (action) {
    case 'INC':
      return { ...state, counter: state.counter + 1}
  }
}
```

```
export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      // Redux Toolkit pozwala na tworzenie "mutującej" logiki w reducerze.
      // Właściwie nie mutujemy stanu ponieważ pod spodem użyta jest biblioteka
      // Immer która wykrywa zmiany w „tymczasowym magazynie” i na jego podstawie
      // tworzy całkiem nowy niemutowany stan zawierający już zmiany
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
```

Action creators

- To simplify creating actions and not forget about important fields we are creating functions - action creators
- This way is more effective for adding changes to state of application

Slice

- Additional separation level
- Slice is an element responsible for keeping logic and reducers connected with one business domain / functional part of app
- Name comes from splitting main reducer to lot of „slices” of state

```
export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
    incrementByAmount: (state, action: PayloadAction<number>) => {
      state.value += action.payload;
    },
  },
  extraReducers: (builder) => {
    builder
      .addCase(incrementAsync.pending, (state) => {
        state.status = 'loading';
      })
      .addCase(incrementAsync.fulfilled, (state, action) => {
        state.status = 'idle';
        state.value += action.payload;
      })
      .addCase(incrementAsync.rejected, (state) => {
        state.status = 'failed';
      });
  },
});
```

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}
```

```
import { addTodo } from './actionCreators'

dispatch(addTodo('Use Redux'))
```

Nesting reducers

- We can have many different reducers grouped by functionalities
- Reducers can be nested in main reducer, or
- Combined with themselves (internal method **combineReducers**)

Middlewares

- **Middleware’s** can be useful in many cases, e.g. for executing requests to API
- **Middleware** is a function, which will be executed after dispatching action and before reducer
- To manage middlewares in Redux exists library **redux-thunk**
- Common middlewars:
  - **redux-thunk**
  - **redux-promise**
  - **redux-saga**

```
const logger = (store) => (next) => (actio) => {
  console.log('dispatching', action);
  let result = next(action);
  console.log('next state', store);
  return result;
}

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(logger),
});
```

configureStore… instead of everything

- One call configureStore can replace executing many operations:
  - Allows to nest (combining) our reducers
  - Creates store
  - Adds middleware’s from thunk by default
  - Adds middleware’s for checking common mistakes and verifying immutability and cohesion of store
  - Connect app with **Redux DevTools Extension** by default (easier development)

```
export function Counter() {
  const count = useAppSelector(selectCount);
  const dispatch = useAppDispatch();

  return (
    <div>
      <div className={styles.row}>
        <button
          className={styles.button}
          aria-label="Decrement value"
          onClick={() => dispatch(decrement())}
        >
          -
        </button>
```

Task

25 minutes
materials/TASK_23.md

MVC vs FLUX vs REDUX

- **MVC:**
  - bi-directional data flow
  - No store
  - Logic in Controller
  - Debugging hard because of bi-directional data flow
  - Used on BE (Django) and FE (AngularJS, Ember)

- **FLUX:**
  - Uni-directional data flow
  - Many stores
  - Store is responsible for logic
  - Debugging quite easy, on dispatcher level
  - Used only on FE: (React, Angular, Vue)

- **REDUX:**
  - Uni-directional data flow
  - One store
  - Reducers are responsible for logic
  - Debugging quite fast, all data in one place
  - Used only on FE: (React, Angular, Vue)

Context API

React

```
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    // Use a Provider to pass the current theme to the tree below.
    // Any component can read it, no matter how deep it is.
    // In this example, we're passing "dark" as the current value.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
```

```
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}
```

```
function CountDisplay() {
  const {count} = React.useContext(CountContext)
  return <div>{count}</div>
}

ReactDOM.render(<CountDisplay />, document.getElementById('⚛️'))
```

Task

25 minutes

materials/TASK_24.md

Redux vs Context

- **ContextAPI**
  - Its part of React library
  - Required minimal setup
  - Design for static data (that is not often refreshed or updated), like:
    - Theme
    - User data
    - Simple dictionary values
  - UI logic and state management logic in one place
  - Debugging can be hard

- **Redux**
  - External library
  - Required extensive setup to integrate it with React
  - Works like a charm with both static and dynamic data
  - Better code organization with separate UI logic and State Management Logic
  - Has a great tool for debugging
  - Good for storing bunch od data

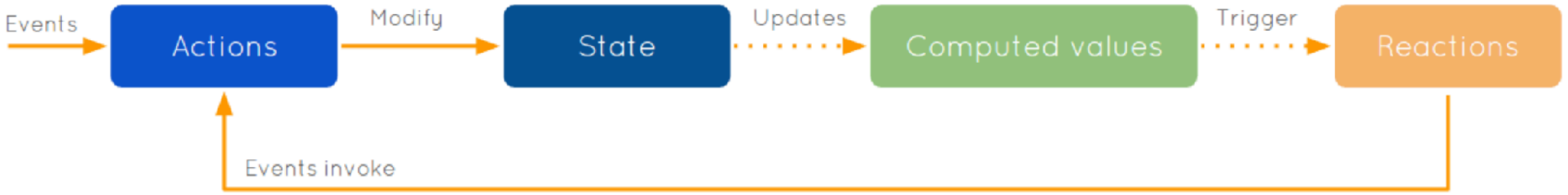
You might not need Redux

Dan Abramov, creator of Redux, 2016

Mobx

Mobx

- MobX is an open source state management tool
- It is not depend on any frontend library or framework
- Application state refers to the entire model of an application, and can contain different data types including array, numbers, and objects.
- Actions are methods that manipulate and update the state. These methods can be bound to a JavaScript event handler to ensure a UI event triggers them.
- MobX store should be reactive, and therefore respond to changes



Events invoke actions. Actions are the only thing that modify state and may have other side effects.

```
@action onClick = () => {
  this.props.todo.done = true;
}
```

State is observable and minimally defined. Should not contain redundant or derivable data. Can be a graph, contain classes, arrays, refs, etc.

```
@observable todos = [{
  title: "learn MobX",
  done: false
}]
```

Computed values are values that can be derived from the state using a pure function. Will be updated automatically by MobX and optimized away if not in use.

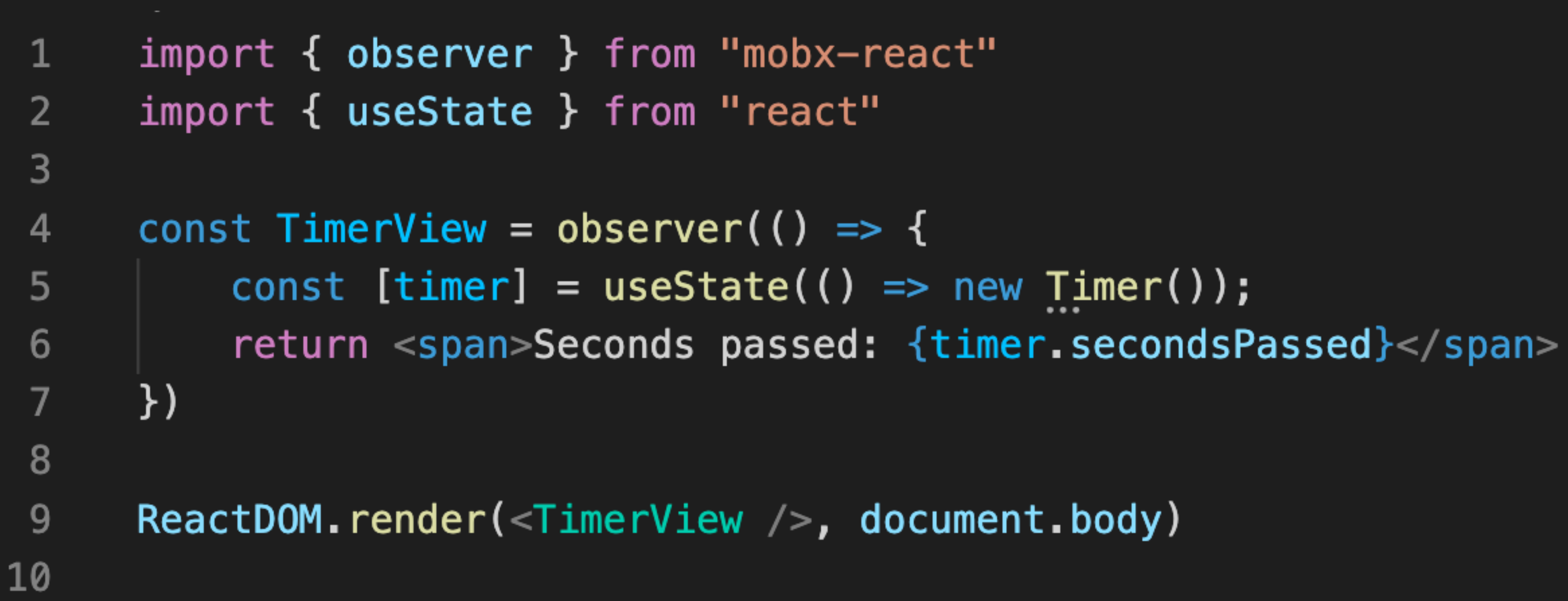
```
@computed get completedTodos() {
  return this.todos.filter(
    todo => todo.done
  )
}
```

Reactions are like computed values and react to state changes. But they produce a side effect instead of a value, like updating the UI.

```
const Todos = observer({ todos } =>
  <ul>
    todos.map(todo => <TodoView ... />
  </ul>
)
```

```
class PetOwnerStore {
  pets = [];
  owners = [];
}
```

```
import { action, observable, reaction } from 'mobx';

// A class to store some data inside
class Data {
    @observable
    value = 0;

    @action
    incrementValue = () => this.value += 1;

    constructor() {
        // Create a reaction that listens to the value property changing
        // and log a message to the console when that happens
        reaction(() => this.value, () => console.log('Value Changed'));
    }
}

// Create a new instance of the class
const myData = new Data();

// After this function is called, 'Value Changed' will be printed to the console
myData.incrementValue();
```



```
import { observer } from "mobx-react"
import { useState } from "react"

const TimerView = observer(() => {
    const [timer] = useState(() => new Timer());
    return <span>Seconds passed: {timer.secondsPassed}</span>
})

ReactDOM.render(<TimerView />, document.body)

```

Dodatkowe pytania?

kahoot.it

XXXXX

☸

Ankieta

tinyurl.com/mtp3xym8

mail@mateuszjablonski.com
mateuszjablonski.com
linkedin.com/in/mateusz-jablonski/

Dziękuję za uwagę

JABŁOŃSKI

sages