Mateusz Jabłoński

Tworzenie aplikacji z React

27.02 - 01.03

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.

sages

Kim jestem?

Mateusz Jabłoński

- programista od 2011 roku
- React / Angular na frontendzie
- NodeJS / Java na backendzie
- czasami blogger / trener / mentor

mateuszjablonski.com
mail@mateuszjablonski.com

Od początku

Ustalenia

- Cel i agenda warsztatów
- Wzajemne oczekiwania
- Pytania i dyskusje
- Elastyczność
- Otwartość i uczciwość

Co nas czeka?

Dzień 1.

- Wprowadzenie
  - Charakterystyka i zasada działania biblioteki
  - JavaScript / TypeScript - powtórzenie elementów istotnych w kontekście biblioteki React
  - Idea Virtual DOM
  - Konfiguracja środowiska i omówienie wykorzystywanych narzędzi deweloperskich
- Podstawy React
  - Wprowadzenie do składni JSX
  - Tworzenie, konfigurowanie i renderowanie komponentów
  - Zarządzanie stanem i jego współdzielenie
  - Obsługa zdarzeń
  - Cykl życia komponentów
  - React hooks
  - Debugowanie błędów i rozwiązywanie problemów

Co nas czeka?

Dzień 2.

- React w praktyce
  - Budowanie złożonych widoków
  - Stylowanie - przegląd rozwiązań, implementacja motywów
  - Praca z formularzami
  - Routing
  - Dobre praktyki
  - Komunikacja z backend
- Redux - zarządzanie stanem aplikacji
  - Omówienie założeń architektury
  - Modelowanie stanu
  - Reduktory
  - Actions oraz action creators
  - Integracja z React
  - Metody implementacji niemutowalnych zmian
  - Kiedy Redux a kiedy Context?
  - Praca z Redux Dev Tools

Co nas czeka?
Dzień 3.

- MobX
  - Omówienie zasady działania
  - MobX State Tree
  - Praktyczne zastosowania
  - Praca z DevTools
  - Porównanie z Redux

- Testowanie
  - Definicja i zakres odpowiedzialności testów jednostkowych
  - Cechy dobrych testów jednostkowych
  - Jak i co testować?
  - Testowanie black box vs. white box
  - Izolacja zależności
  - Możliwości React Testing Utilities
  - Testowanie komponentów
  - Symulowanie zdarzeń przeglądarki
  - Testowanie akcji
  - Testowanie reduktorów
  - Komunikacja z API w testach
  - Shallow / full rendering komponentów

github.com/matwjablonski/rossmann-react-2

Rozkład dnia

- **8:00 - start**
- 9:30 - 9:45 - przerwa
- **11:30 - 12:30** - przerwa obiadowa
- 13:20 - 13:35 - przerwa
- 14:50 - 15:05 - przerwa
- **16:00 - koniec**

Projekt

- aplikacja bankowa

	Angular.js	React.js	Vue
Business Benefits			
Flexibility	Somewhat Flexible	Most Flexible	Somewhat Flexible
Learning Curve	High	Low-Medium	Low-Medium
Framework Size	Heavyweight, suited for dense and complex apps.	Lightweight library, smaller than Angular's	Lightweight, suited for small apps
Community Support	Very good because of Google's support	Quite popular among developers because of extensibility and Facebook's support	Less popular among the top corporations but still quite popular
Top Use Cases	Google, The Guardian, etc.	Facebook, Twitter, Instagram	9GAG, Gitlab
Performance	Uses Real DOM, more efficient	Uses Virtual DOM, Faster than real DOM	Uses Virtual DOM, Faster than real DOM
Background	Typescript based JS, created by Google in 2010.	Founded by Facebook in 2013 to address high traffic on their sites.	A progressive architecture created by ex-Google staff in 2014.
Popularity	Most Popular and Used	Most Popular & Used	Not as popular as these 2 but still widely used.

Javascript

Transpilacja kodu

Transpilacja to proces przepisywania kodu do jego odpowiednika w innym języku (lub w tym samym, ale w innej wersji)

Babel

- Darmowy transpiler javascriptowy
- Pierwszy raz wydany w 2014 roku
- Transpiluje kod z ES2015+ do starszych wersji, zgodnie ze wsparciem
- Zezwala na transformacje niestandardowych technik jak: JSX do kodu JS
- Zawiera zestaw polyfiili aby zezwolić na używanie funkcji, które nie są dostępne w standardzie ES5

React

Babel

Babel w React

- Babel jest jednym z podstawowych narzędzi wykorzystywanych w procesie development aplikacji React’owych
- Konfiguracja jest trzymana w pliku **.babelrc**
- Konfiguracja jest podzielona na dwie grupy:
  - presets - reguły transformacji kodu
  - plugins - rozszerzenia dla silnika Babel’a
-

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

React

Podstawy


React

SPA

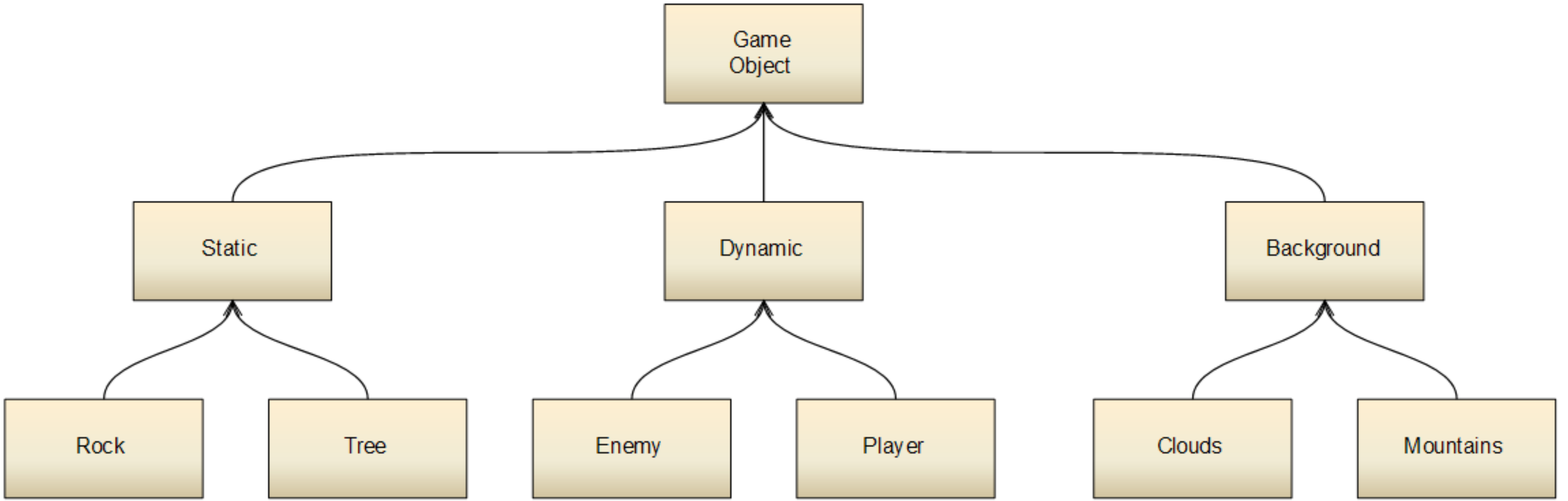
- SPA - Single Page Applications
- SPA jest alternatywą dla MPA (Multi Page Applications)
- Serwer zwraca prosty plik HTML, który zawiera tylko jednego diva i informacje o wymaganych stylach i skryptach
- Odpowiedź z serwera jest bardzo szybka
- Cała logika aplikacji i nawigacji jest przeniesiona na przeglądarkę
- Problematyczne SEO
- Problem z udostępnianiem treści w mediach społecznościowych (meta tags)
- Aplikacje SPA mogą być budowane w sposób modułowy i komponentowy
- Nie ma dużych wymagań od serwera (tylko pliki statyczne: html, css. js)
- Kosz infrastruktury został przeniesiony na klienta końcowego

React

- React jest biblioteką która może być wykorzystana do budowania dynamicznych i złożonych interfejsów użytkownika w sposób deklaratywny i modułowy
- Zdejmuje odpowiedzialność za renderowanie i aktualizację stanu drzewa dom z programisty
- Zezwala na tworzenie struktury widoku i logiki wyświetlania treści w sposób bardziej deklaratywny (bardziej naturalny niż w przypadku programowania zorientowanego obiektowo czy imperatywnego)
- Zezwala na trzymanie struktury i logiki w jednym miejscu (**tylko z wykorzystaniem Javascript** - programista może wykorzystywać wszystkie możliwości języka Javascript, bez konieczności dotykania składni języków strukturalnych)
- React buduje widok w architekturze komponentowej
    - Komponent to element oprogramowania posiadający dobrze wyspecyfikowany interfejs oraz zachowanie

Architektura komponentowa

- Komponenty mogą być wykorzystane w wielu aplikacjach
- Skraca czas budowy nowych aplikacji i obniża koszty projektu - poprzez zastosowanie istniejących komponentów.
- Wykorzystując istniejące komponenty po pierwsze, nie musimy ich pisać od nowa, po drugie testować i dokumentować.
- Poszczególne komponenty mogą być tworzone równolegle. Poprzez dobre wyspecyfikowanie interfejsów poszczególnych komponentów można łatwo zlecać implementacje komponentów na zewnątrz.
- Wspiera tzw. modyfikowalność aplikacji - konkretna funkcjonalność skupia się w dedykowanych komponentach programowych, więc w przypadku konieczności wprowadzenia zmiany, z reguły proces ten dotyczy modyfikacji jednego lub co najwyżej kilku komponentów a nie całej aplikacji.
- Spójność (ang. coherence) komponentów wspiera modyfikowalność - im większa spójność tym łatwiej modyfikować aplikację.


Game Object
Static
Dynamic
Background
Rock
Tree
Enemy
Player
Clouds
Mountains

React

- **react** - podstawowa biblioteka obsługująca model kompozycyjny React’a
- **react-dom** - biblioteka udostępniająca specyficzne dla DOM metody
  - **react-dom/client** - moduł, który zezwala na tworzenie i używanie aplikacji React po stronie klienta
  - **react-dom/server** - moduł, który zezwala na budowanie komponentów React w wersji serwerowej

!

React nie renderuje zmian bezpośrednio, ale poprzez 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 to uproszczona reprezentacja struktury DOM (Document Object Model)
  - React buduje drzewo Virtual DOM w sposób deklaratywny, a następnie przygotowuje na jego podstawie drzewo DOM w przeglądarce
  - W przypadku wystąpienia zmian w strukturze, React na nowo przygotuje drzewo Virtual DOM dla zmienionych komponentów
  - Biblioteka ReactDOM porównuje aktualne i nowe drzewo Virtual DOM, a następnie aktualizuje tylko te miejsca, które wymagają zmiany

!

Dzięki małej liczbie operacji na strukturze DOM,
React aktualizuje widok błyskawicznie!

Virtual DOM

- Zezwala na generowanie abstrakcyjnego DOM, który może być renderowany jako UI (User Interface) na wielu platformach:
  - **react-dom** - przeglądarkowy DOM
  - React Native - natywne aplikacje dla iOS i Android
  - react-blessed - terminal
  - react-canvas - elementy HTML Canvas
  - react-vr / react 360 - aplikacje 3D.

React

Atrybuty i klasy

```
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

Komponenty

Komponenty

- Komponent to podstawowy blok aplikacji Reactowej
- Komponent to element w strukturze DOM, który jest zarządzany przez React'a
- **Funkcja / Klasa komponentu** zawiera kod Javascriptowy, który kontroluje wygląd i zachowanie elementu
- Instancja komponentu to element, który został wyrenderowany przez Reac'a za pośrednictwem swojej klasy / funkcji
- Komponenty mogą być zagnieżdżane w dowolne struktury, podobnie jak HTML czy XML
- Możemy przekazać do komponentu dowolne dane jako atrybuty, w taki sam sposób jak w HTMLu. W przeciwieństwie do HTMLa poprzez atrybuty możemy przekazywać również obiekty (a nie tylko stringi).
- Przekazane w ten sposób parametry nazywają się właściwościami komponentu (ang. **Component properties**, w skrócie **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>
  }
});
```

```
// ES6 class

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

```tsx
// 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>
}
```


!
Komponent funkcyjny - wcześniej nazywany jako bezstanowy komponent funkcyjny. Od wprowadzenia Hook’ów komponenty funkcyjne mogą mieć swój stan i nim zarządzać.

```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

Algorytm różnicujący

Elementy o różnych typach

- W procesie różnicowania React porównuje dwa drzewa Virtual DOM (aktualne i nowe). Proces sprawdzania rozpoczyna od sprawdzenia typów elementów najwyższego poziomu (root)
  - Kiedy typy są różne (np. SECTION i DIV) - React niszczy stare drzewo, wraz z instancjami wcześniejszych komponentów i na podstawie nowego Virtual DOM renderuje na nowo aplikację.
  - Podczas tworzenia drzewa na nowo:
    - Stare elementy DOM są niszczone
    - Instancje komponentów otrzymają informację, że komponent zostanie odmontowany
    - Nowe drzewo zostanie wyrenderowane w miejsce starego
    - Nowe komponenty otrzymają informację, że zostały zamontowane

Elementy DOM o tym samym typie

- Kiedy elementy DOM mają ten sam typ:
  - React sprawdza argumenty (właściwości) obu porównywanych elementów i w następnym kroku aktualizuje tylko te wartości, które wymagają zmiany, np. **styles**, **className**
  - Bez niszczenia drzewa idzie do dzieci i rozpoczyna różnicowanie na elementach potomnych

Komponenty elementów o tym samym typie

- Kiedy są takie same:
  - Pomiędzy re-renderowaniami instancja komponentu pozostaje niezmieniona
- Kiedy w komponencie wykryto zmianę:
  - Komponent zostanie poinformowany o potrzebie wykonania re-renderowania.

Rekurencja na dzieciach

- Domyślnie React iteruje się poprzez listę dzieci w tym samym czasie i generuje mutacje w przypadku napotkania różnic
- Jeśli React znajdzie nowy element na końcu listy - doda go do niej
- Gorzej jeśli nowy element pojawi się na początku lub w środku listy - React porównując elementy nie będzie w stanie wykryć, który element jest nowy - w związku z tym wyrenderuje całą listę od momentu wystąpienia różnicy.
  - Tak przeprowadzona aktualizacja ma bardzo negatywny wpływ na wydajność aplikacji.
  - Aby uniknąć takich sytuacji - elementy listy powinny otrzymać atrybut **key**

Key

- Element kolekcji powinien mieć dodany atrybut **key**
- Key powinien być unikalny i stabilny pomiędzy re-renderowaniami komponentu. Na jego podstawie React jest w stanie rozpoznać elementy, odpowiednio je porównać i podjąć decyzję czy powinny być ponownie wyrenderowane.
- Dzięki kluczom React jest w stanie optymalnie wyrenderować listę i pominąć zbędne re-renderowania komponentu, a zatem wykonać na drzewie DOM możliwie najmniej operacji (mutacji)
- Jeśli key otrzyma nową wartość, React przerenderuje całą listę ponownie.

!

Za aktualizację drzewa DOM odpowiada algorytm różnicujący (Diffing Algorithm). Proces aktualizacji drzewa DOM to rekoncyliacja.

!

Algorytm różnicujący jest tylko szczegółem implementacyjnym. Możemy go pominąć. Wówczas każda zmiana będzie powodowała re-render całej aplikacji.

React

Create React App

```
{
  "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"
    ]
  }
}
```

Zadanie

15 minutes
tasks/TASK_01.md

React

Stan i propsy

Stan i props’y

- React „reaguje” na zmiany danych w komponentach (w przypadku zmiany zostaje wywołany kolejny render z Virtual DOM)
- Renderowanie może zostać wykonane w dwóch przypadkach:
  - Kiedy komponent otrzyma nowe dane poprzez właściwości (**props**)
  - Kiedy zmienił się wewnętrzny stan komponentu (**state**)
- **Przekazanie nowych props do komponentu lub zmiana wewnętrznego stanu komponentu spowoduje re-render komponentu**
- Nigdy nie powinniśmy aktualizować stanu i props’ów ręcznie (taka zmiana nie zostanie wykryta przez Reacta)

```
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() }
  }
}
```

Aktualizacja stanu

- React musi wiedzieć kiedy stan komponentu się zmienił. **Nigdy nie zmieniaj stanu bezpośrednio!**
- Do zmiany stanu w React służą dwie funkcje: **setState** i hook **useState**
- Funkcje zmiany stanu działają asynchronicznie.
- Kiedy stan jest zależny od poprzedniej wartości stanu powinniśmy użyć funkcji wywołania zwrotnego jako pierwszy argument w funkcji zmiany stanu.

- **Podejście klasowe:**
  - Zmiany są scalane (kiedy zmieniamy tylko jedną wartość obiektu stanu, pozostałe pozostaną bez zmian)
  - Użycie wartości stanu zaraz po jego aktualizacji może spowodować błąd (komponent może nie zdążyć się wyrenderować ponownie)
  - **this.replaceState({})** - metoda do bezpośredniego nadpisania całego obiektu stanu

- **Podejście funkcyjne (hook):**
  - Zmiany stanu nadpisują poprzedni stan (w przeciwieństwie do komponentów klasowych)

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

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

```
class Comp extends React.Component {

}

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

Zdarzenia

- Zdarzenia w React możemy rejestrować bezpośrednio na elementach za pośrednictwem specjalnych props’ów, prefixowanych słowem **on**
- Zdarzeniami możemy zarządzać z poziomu kodu komponentu
- Funkcję możemy również wykonać bezpośrednio podczas deklaracji zdarzenia
- Przykładowe propsy dla zdarzeń:
  - **onClick** - zdarzenia kliknięcia w element
  - **onChange** - zdarzenie zmiany wartości pola formularza
  - **onSubmit** - zdarzenie wysłania formularza

Synthetic Event

- Obiekty zdarzeń w React mają inny typ niż obiekty zdarzeń w przeglądarce
- SyntheticEvent to klasa, która opakowuje natywny obiekt zdarzenia
- Interfejs SyntheticEvent jest zgodny z interfejsem natywnego zdarzenia
- W przypadku konieczności odwołania się do natywnego zdarzenia, możemy wykorzystać właściwość **nativeEvent**
- React normalizuje zdarzenia, tak by ich właściwości były jednakowe w różnych przeglądarkach
- Aby zarejestrować procedurę obsługi zdarzenia w fazie przechwytywania (ang. *capturing phase*), dodaj na końcu nazwy **Capture** (np. **onClickCapture**)

Zadanie

20 minutes
tasks/TASK_02.md

Zadanie

20 minutes
tasks/TASK_03.md

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

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

```
class Input extends React.Component {
  handleChange(e) {
    let value = e.currentTarget.value;
    this.setState({value: value})
  }

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

Formularze

- Domyślnie pola formularza w React oznaczone są jako pole niekontrolowane (uncontrolled components)
- Pola niekontrolowane nie są kontrolowane w pełni przez mechanizm Reacta (za zmianę ich wartości odpowiada przeglądarka)
- Jeśli do pola formularza dodany atrybut value, wówczas pole zmienia się w pole kontrolowane (**controlled component**)
- **Pole kontrolowane** jest kontrolowane przez Reacta a zatem zmiana jego wartości może odbyć się tylko za pośrednictwem mechanizmu Reacta (tj poprzez zmianę stanu lub propsów - w większości przypadków poprzez zmianę stanu)
- Kiedy pole jest kontrolowane tylko komponent może zmienić jego wartość (na UI pole nie będzie reagowało bez implementacji funkcji zmiany wartości)
- Kod `<input value={this.state.myValue} />` sprawia że pole staje się niemożliwe do edycji ręcznie.
- Elementy typu **<select>** I **<textarea>** również mogą przyjmować atrybut **value**

Zagnieżdżanie komponentów

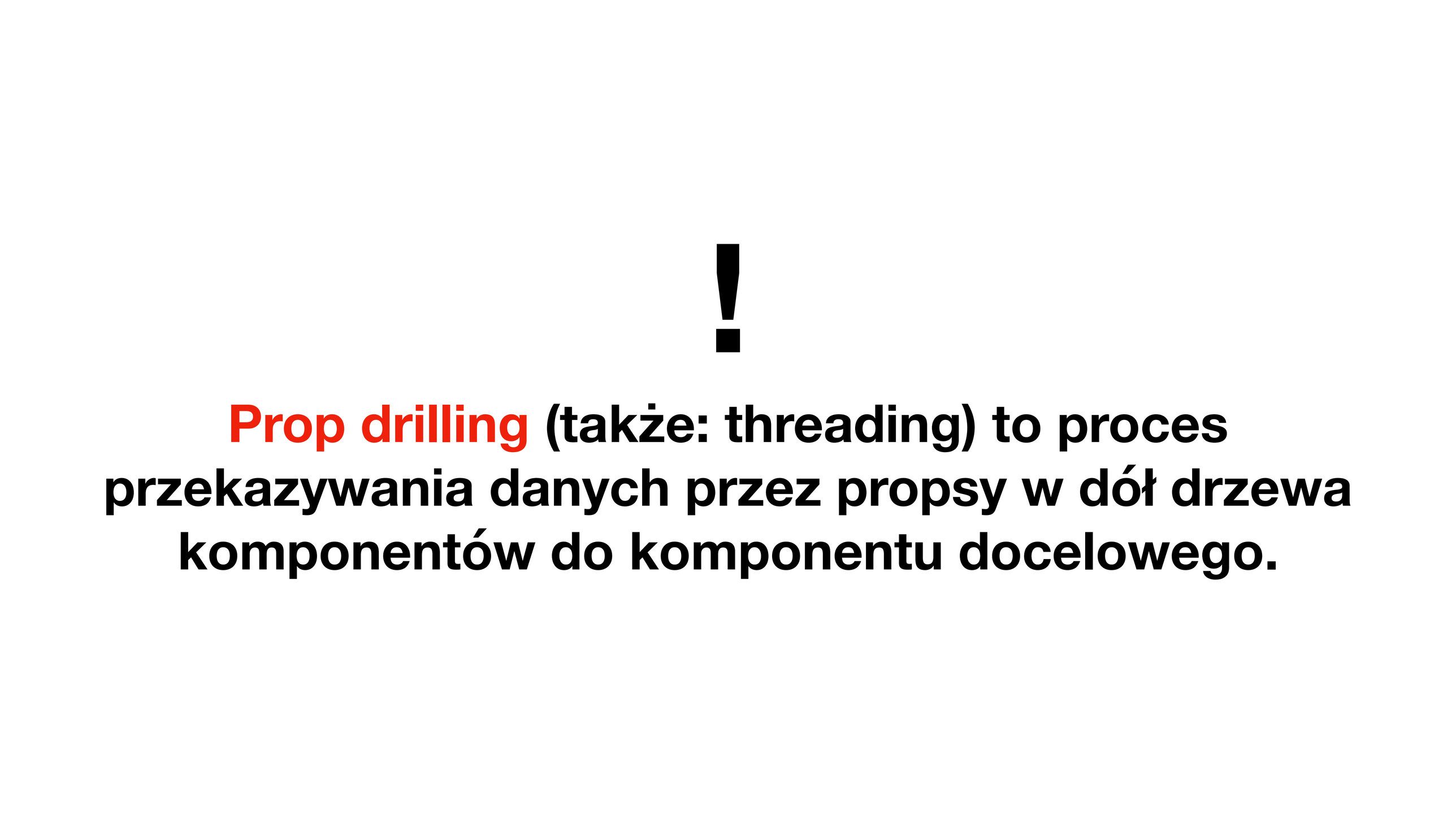
- Komponenty mogą być zagnieżdżane
- Każdy kod JSX, który jest przekazany pomiędzy znacznikiem otwarcia i zamknięcia komponentu, będzie dostępny w tym komponencie jako props **children**
- Jako dzieci komponentu możemy zagnieżdżać stringi, tablice elementów JSX i wszystko co wypełnia typ **ReactNode**

```jsx
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 (także: threading) to proces przekazywania danych przez propsy w dół drzewa komponentów do komponentu docelowego.

Zadanie

20 minutes
tasks/TASK_04.md

React

Metody cyklu życia komponentu

Cykl życia komponentu

- Każdy komponent ma zaimplementowany mechanizm cyklu życia.
- Cykl życia zawiera metody, które połączone są z poszczególnymi etapami istnienia komponentu w strukturze Virtual DOM.
- Etapy te pozwalają nam kontrolować komponent od jego zamontowania do odmontowania w drzewie Virtual DOM. Pozwalają nam np. na wykonywanie dodatkowych operacji w wybranych przez nas momentach istnienia komponentu.
- Montowanie:
  - **UNSAFE_componentWillMount()** - przed zamontowaniem w DOM
  - **componentDidMount()** - po zamontowaniu w DOM
- Aktualizacja:
    - **UNSAFE_componentWillReceiveProps(newProps)** - komponent otrzyma nowe propsy
    - **getDerivedStateFromProps(newProps)** - komponent otrzymał nowe propsy
    - **shouldComponentUpdate(newProps, newState)** - jeśli zwróci false, React pominie następny render
    - **UNSAFE_componentWillUpdate()** - komponent będzie renderowany, nie zmieniaj stanu
    - **componentDidUpdate()** - komponent jest wyrenderowany, DOM jest stabilny
    - **getSnapshotBeforeUpdate(prevProps, prevState)** - tuż przed aktualizacją struktury DOM w przeglądarce

Cykl życia komponentu

- Odmontowywanie:
  - **componentWillUnmount()** - przed usunięciem komponentu z DOM
- Obsługa wyjątków:
  - Metody te są wykonywane w przypadku wystąpienia błędów podczas procesu renderowania, w metodach cyklu życia lub w metodach konstruktora komponentów potomnych.
    - **getDerivedStateFromError(newProps)**
    - **componentDidCatch()**

React

Operacje na DOM

Nie jest zalecane manipulowanie strukturami DOM wygenerowanymi przez Reacta, ale istnieje taka możliwość. Aby uzyskać dostęp do elementów DOM możemy użyć mechanizmu referencji.

```
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>
  )
}
```

Manipulacje na DOM

- Aby uzyskać dostęp do elementu DOM możemy użyć:
  - Hook’a **useRef**
  - Stworzyć referencję w komponencie klasowych za pomocą **createRef**
  - **ReactDOM.findDOMNode(this.refObj.current);**
    - Powyższy przykład jest rodzajem wyjścia awaryjnego. React odradza używanie tej funkcji, ponieważ zaburza abstrakcję struktury komponentów
- Pamiętajmy, że wartości referencji będą dostępne tylko gdy komponent zostanie wyrenderowany
- Używaj referencji rozważnie, powinny być wykorzystywany tylko w specyficznych metodach cyklu życia.
- Przechowywanie danych w referencji spowoduje pomięcie re-renderowania komponentu po zmiany tych danych

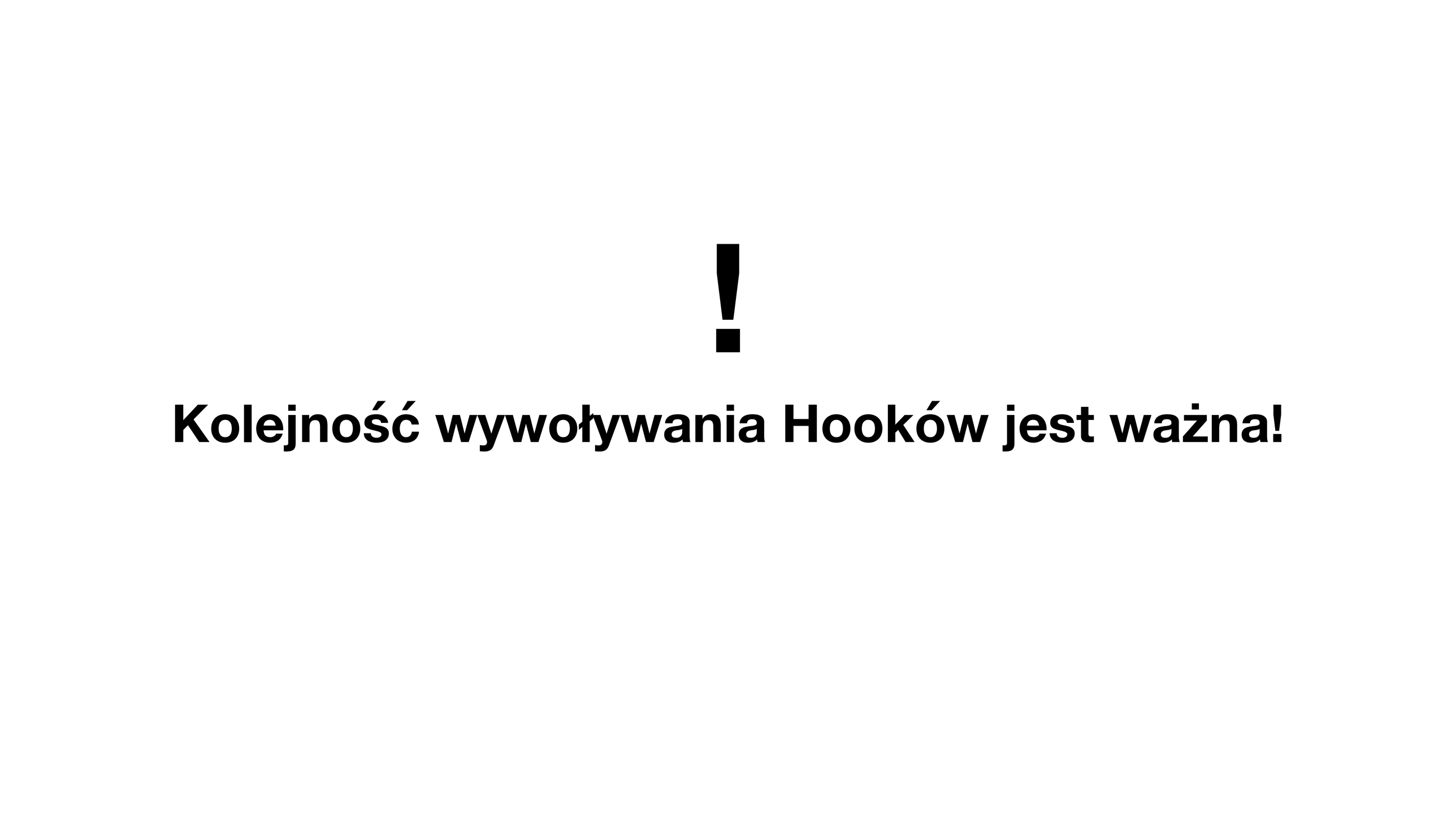
```
return (
      <form>
        <input ref={element => element.focus()}/>
      </form>
    )
```

React

Hooks

React Hooks

- Funkcje w Javascript nie posiadają stanu. Oznacza to że każde wywołanie funkcji powoduje stworzenie jej instancji na nowo, a zatem funkcja ma dostęp tylko do danych które przekazywane są przez parametry lub do zmiennych z zasięgów wyższych (closure)
- W React komponenty są funkcjami - dlatego domyślnie nie mogą trzymać stanu pomiędzy kolejnymi wywołaniami (re-renderami)
- Hooki są funkcjami, które mogą być umieszczane wewnątrz komponentu, ale tylko na najwyższym poziomie (top scope)
- Hooki dodają do komponentów funkcyjnych możliwości używania referencji, obsługi stanu i wywoływania efektów ubocznych
- Hooki mogą być wywoływane tylko w komponentach Reactowych lub w innych własnych hookach
- Możemy tworzyć własne hooki (nazwa funkcji hook’a powinna rozpoczynać się od słowa kluczowego **use**)


!
Kolejność wywoływania Hooków jest ważna!

```
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

- Jest wykorzystywany do zarządzania stanem komponentu
- Podczas wywoływania tego Hook’a możemy przekazać do niego wartość (będzie to wartość, która zostanie ustawiona jako domyślny stan)
- W komponencie możemy wywołać ten hook więcej niż jeden raz do zarządzania różnymi stanami dla różnych celów
- Dobrą praktyką jest grupować elementy stanu wg domeny biznesowej
- **useState** nadpisuje cały stan (w przeciwieństwie do klasowego **setState**)
- Wywołanie **useState** zwraca tablicę dwuelementową - pierwsza wartość do zmienna przechowująca bieżący stan, druga to funkcja zmieniająca stan).
- Domyślny stan może być również funkcją. Wówczas podczas tworzenia domyślnego stanu domyślna wartość będzie wartością zwróconą z wywołania tej funkcji.

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

Zadanie

20 minutes
tasks/TASK_05.md

useEffect

- **useEffect** zezwala nam na wykonywanie „side effect’ów”
- Po każdym renderowaniu DOM możemy wywołać imperatywne operacje na komponencie lub mające wpływ na aspekty nie dotyczące komponentu za pomocą Hooka **useEffect**
- Przykładowe zdarzenia uboczne:
  - Zmiana tytułu okna przeglądarki
  - Pobranie danych z serwera / api

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

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

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

Każde renderowanie komponentu powoduje nowe wywołanie efektu. Działa to podobnie do metod cyklu życia: componentDidMount() i 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

- Drugim argumentem hooka **useEffect** jest tablica zależności (wartości od których zależy ponowne wywołanie danego useEffectu)
  - Jeśli drugi argument nie został przekazany, **useEffect** wywoła się przy każdym renderze
  - Jeśli drugi argument jest pustą tablicą, **useEffect** wywoła się tylko raz (podczas pierwszego renderu)
  - Jeśli drugi argument zawiera zależności, **useEffect** wywoła się jeśli którakolwiek z zależności ulegnie zmianie (**Uwaga na wartości referencyjne!)**

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

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

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

useLayoutEffect

- Zachowanie podobne do **useEffect**
- Różnica polega na tym, że jest wywoływany synchronicznie po mutacji na strukturze DOM
- Przydatny kiedy chcemy uspójnić widok z danymi w sytuacji bezpośredniej mutacji struktury DOM (np. Wszelkiego rodzaju animacje)

Inne hooki

- useContext
- useRef
- useReducer
- **useMemo** - memoizacja wartości
- **useCallback** - memoizacja funkcji
- **useImperativeHandle** - dostosowuje wartość instancji, która została przykazana do rodzica komponent podczas użycia referencji, zastosowanie podobne do useRef, ale daje większą kontrolę nad wartością która jest zwracana i zezwala na zastąpienie natywnych funkcji takich jak: blur, focus własnymi ich implementacjami
- **useDebugValue** - może być użyty do wyświetlenia etykiet dla własnych Hooków w React DevTools
- **useId** - generuje unikalne ID, które jest stabilne pomiędzy każdym renderem, przydatne gdy musimy uspójnić komponent renderowany po stronie klienta i po stronie serwera
- **useTransition** - pozwala nam ustawić priorytet na zmianie stanu, tak aby wykonała się w pierwszej kolejności
- **useDefferedValue** - pozwala na pokazanie starej wartości, dopóki nowa nie będzie gotowa (przydatne gdy widok jest zależny od wartości pochodzących spoza komponentu, tj. biblioteki, inne komponenty)

Zadanie

20 minutes
tasks/TASK_06.md

React

Komunikacja z 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();
}, [])
```

Zadanie

20 minutes
tasks/TASK_07.md

React

React Router DOM

Router

- Pozwala zastąpić renderowany komponent na widoku w zależności od bieżącego adresu URL w przeglądarce
- Operuje na obiekcie location (oczywiście dostosowanego do potrzeb aplikacji SPA)
- Zezwala na dodanie parametrów do ścieżek
- Zezwala na wykorzystywanie dynamicznych linków (bazujących na zmiennych)

```jsx
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>
);
```

```jsx
<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}`);
```

Zadanie

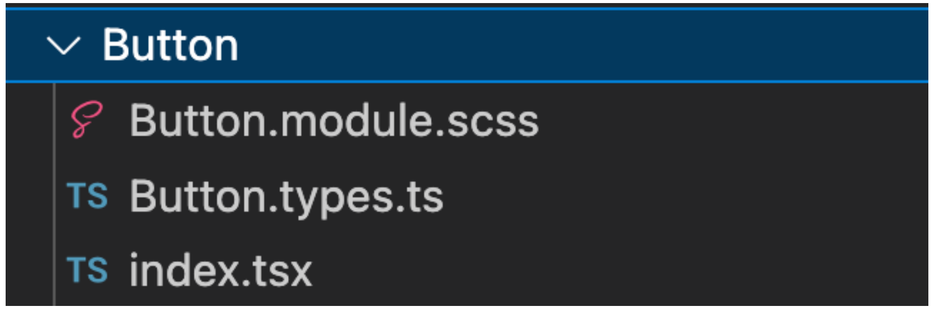
20 minutes
tasks/TASK_08.md

React

CSS

CSS Modules

- Domyślny sposób deklarowania styli w CRA
- Pozwala na wykorzystanie preprocessor’ów takich jak Sass
- Pliki ze stylami są traktowane jako moduły
- Każdy komponent posiada swój własny plik / moduł ze stylami
- Nazwa każdego modułu ze stylami powinna być zakończona rozszerzeniem **.module.${ext}**
- Bardzo łatwy sposób na enkapsulacje styli (enkapsulacja za pomocą generowane ID dodawanego do nazw klas)
- Bardzo dobra adaptacja w przeglądarce - nie ma większych problemów z wydajnością, ponieważ docelowo style są traktowane tak samo jak zwykłe style css
- Klasy css są dostępne na poziomie komponentu w obiekcie ze stylami
- Style globalne powinny być trzymane w osobnym pliku / katalogu


Button
Button.module.scss
Button.types.ts
index.tsx

```
import React, { FunctionComponent } from 'react';
import styles from './Counter.module.scss';

const Counter: FunctionComponent<CounterProps> = ({ days, nextItemName }) => {
    return (
        <div className={styles.counter}>
            <div className={styles.counterText}>
                <span className={styles.nextTitle}>Następny {nextItemName}</span>
                <span className={styles.remainingDays}>{prepareDaysLabel()}</span>
            </div>
            <div className={styles.icon}>
                <Image src={calendar || `/icons/calendar.svg`} width={24} height={24}
alt=""/>
            </div>
        </div>
    )
}

export default Counter;
```

JSS

- Pierwsza próba tworzenia styli z wykorzystaniem Javascript
- Koncepcja została zaproponowana i przygotowana w 1996 roku przez firmę Netscape (wówczas to rozwiązanie nie uzyskało wystarczająco dużego wsparcia przeglądarek i upadło)
- Współcześnie JSS jest podobnym konceptem trzymania CSS w JS za pomocą obiektów Javascriptowych. Zasadność wykorzystania JSa do tworzenia CSS opiera się na dużej reużywalności obiektów i sporymi możliwościami enkapsulacji styli
- JSS nie jest powiązany z żadnym frameworkiem
- Aby używać JSS w React możemy użyć pakietu **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

- Bazuje na rozwiązaniach JSS
- Biblioteka do tworzenia styli jako komponentów
- Wykorzystuje Javascriptowy mechanizm **tagged templates**
- Styled komponent może być tworzony jako dedykowany dla innego komponentu React lub jako samodzielny działający dla całej aplikacji
- Zachowują się i działają w sposób taki jak inne elementy JSX
- Zezwala na rozszerzania komponentów React poprzez dodawania styli do nich
- Styled komponenty są reużywalne, mogą korzystać z propsów i trzymać logikę powiązaną z zarządzaniem stylami

```
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
```

Zadanie

20 minutes
tasks/TASK_09.md

React

Zmienne środowiskowe

.env

- W projekcie CRA możemy deklarować i używać zmiennych środowiskowych
- Nazwy zmiennych środowiskowych powinny być prefixowane za pomocą **REACT_APP_**
- Zadeklarowane zmienne nie powinny przechowywać sekretów
- Nigdy nie trzymaj sekretów w repozytorium
- Pamiętaj, że wartości ze zmiennych środowiskowych w komponentach klienckich zostaną przeniesione do kodu produkcyjnego

!

Nigdy nie trzymaj sekretów w aplikacji React.
(przede wszystkim prywatnych kluczy API)

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

React

Zarządzanie uprawnieniami

```tsx
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}</>;
}
```

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

Zadanie

20 minutes
Tasks/TASK_10.md

React

Portale

Portale

- Mechanizm, który zezwala na renderowanie komponentów React poza miejscem w strukturze, gdzie powinny one zostać wyrenderowane.
- Zezwala na renderowanie elementów w dowolnym miejscu w DOM lub w oknach zależnych (new Window)
- Portale mogą być użyteczne, gdy stylowanie komponentów ogranicza możliwości poprawnego renderowania komponentów, np. Gdy komponent ma ustawiony z-index lub overflow z wartością hidden.
- Element renderowany za pośrednictwem Portalu jest nadal pod kontrolą React.
- Komponenty renderowane w Portalu mogą współdzielić stan, mają dostęp do przekazywanych dropsów i do context'u.
- Event bubbling (bąbelkowanie) będzie działało w ten sam sposób jak bez Portalu, dlatego zdarzenia odpalone wewnątrz Portalu będą propagowaną w górę drzewa React.

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

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

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

Zadanie

20 minutes
tasks/TASK_11.md

React

Context API

Context API

- Context pozwala na przekazywanie danych w drzewie komponentów bez przekazania ich poprzez propsy elementów potomnych.
- Jeśli użycie Context API ma pozwolić nam uniknąć prop drillingu przez kilka poziomów drzewa React, łatwiejszych i lepszych rozwiązaniem będzie użycie **kompozycji komponentów**.
- Kiedy Context podejmuje decyzję co powinno być wyrenderowane ponownie sprawdza referencje elementów. Oznacza to że w niektórych przypadkach kolejne rendery rodziców Providera Contextu mogą wywołać kolejne niechciane re-rendery wszystkich konsumentów (Consumer) dla konkretnego contextu.
- Każdy obiekt context'u ma swój własny komponent dostawcy (Provider'a), które zezwala komponentom subskrybować się na zmiany w tym context'cie.
- Komponent Reactowy, który subskrybuje się na zmiany w context'cie to Consumer. Cunsumer może nasłuchiwać na zmiany w ramach swojej struktury.

React

Kompozycja

```
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.

React

Strict Mode

<React.StrictMode>

- Dodany do React w 2018 roku (pierwotnie dotyczył tylko komponentów klasowych)
- Ma za zadanie pomagać deweloperom w aktualizacji aplikacji poprzez rezygnację ze starego, niewspieranego API
- Dodaje zestaw podpowiedzi i ostrzeżeń, aby pomóc uniknąć typowych pułapek związanych z procesem Developmentu
- Pomaga uniknąć trzymania w kodzie „nieczystych: funkcji
- Sprawdza czy funkcje komponentów React są idempotentne (pomaga w tym podwójne wywołania kodu)
- Działa tylko w trybie deweloperskim
- Efektem ubocznym jest podwójne renderowanie aplikacji w procesie development (pomaga dzięki temu wyłapać błędy w asynchronicznie działających API Reacta)

React

Code splitting

Code splitting

- Wraz z rozrastającym się kodem aplikacji, rośnie również rozmiar plików produkcyjnych.
- Code Splitting jest funkcjonalnością wspieraną przez takie narzędzia jak: Webpack, Rollup, Browserify
- Code Splitting zezwala na tworzenie dużej ilości małych paczek kodu, które mogą być ładowane na żądanie w trakcie działania aplikacji
- Dobrym miejscem na podział aplikacji za pomocą Code Splittingu są ścieżki w routerze
- Sposoby na Code splitting w aplikacji React:
  - 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;
```

```jsx
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>
  );
}
```

Zadanie

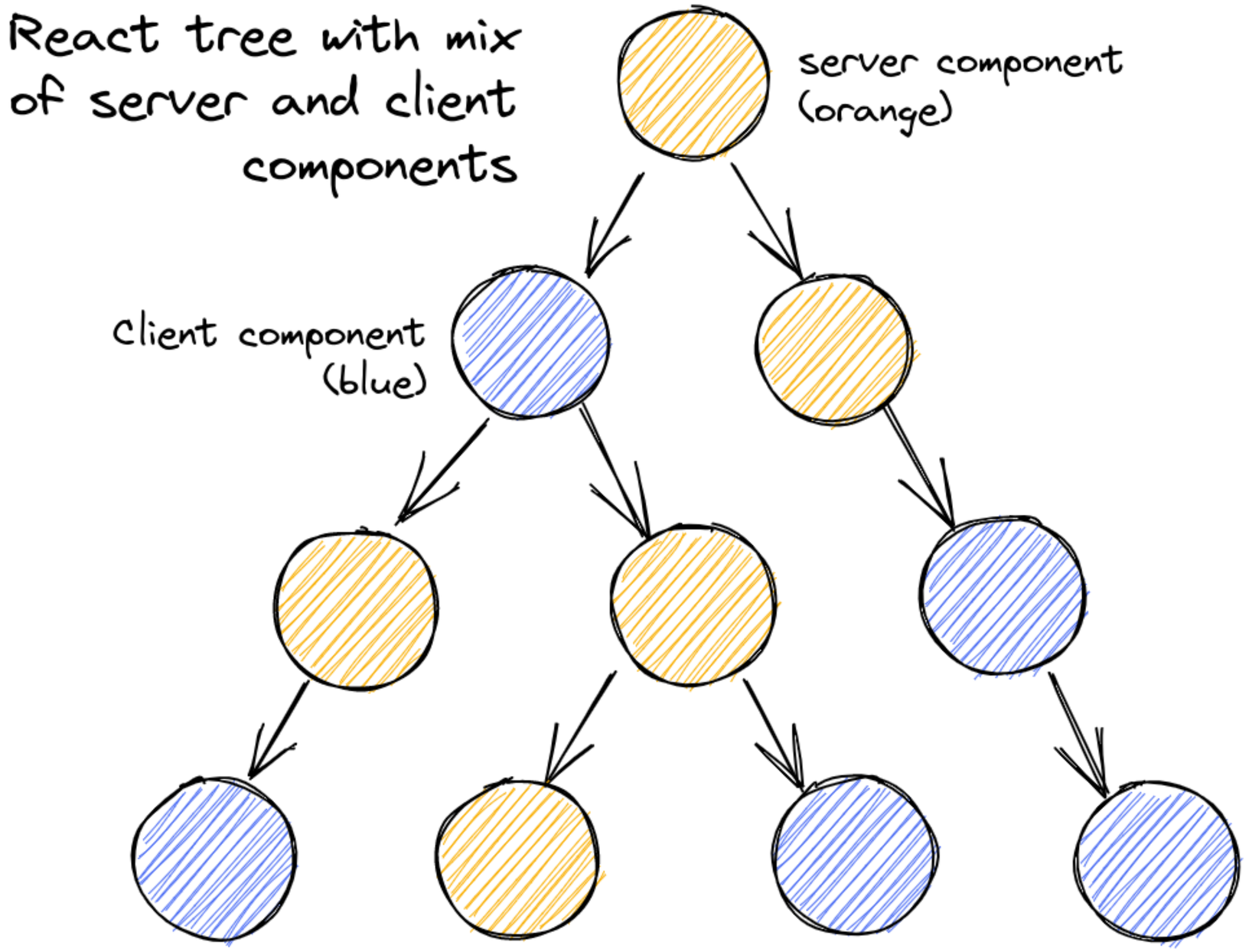
10 minutes
tasks/TASK_12.md

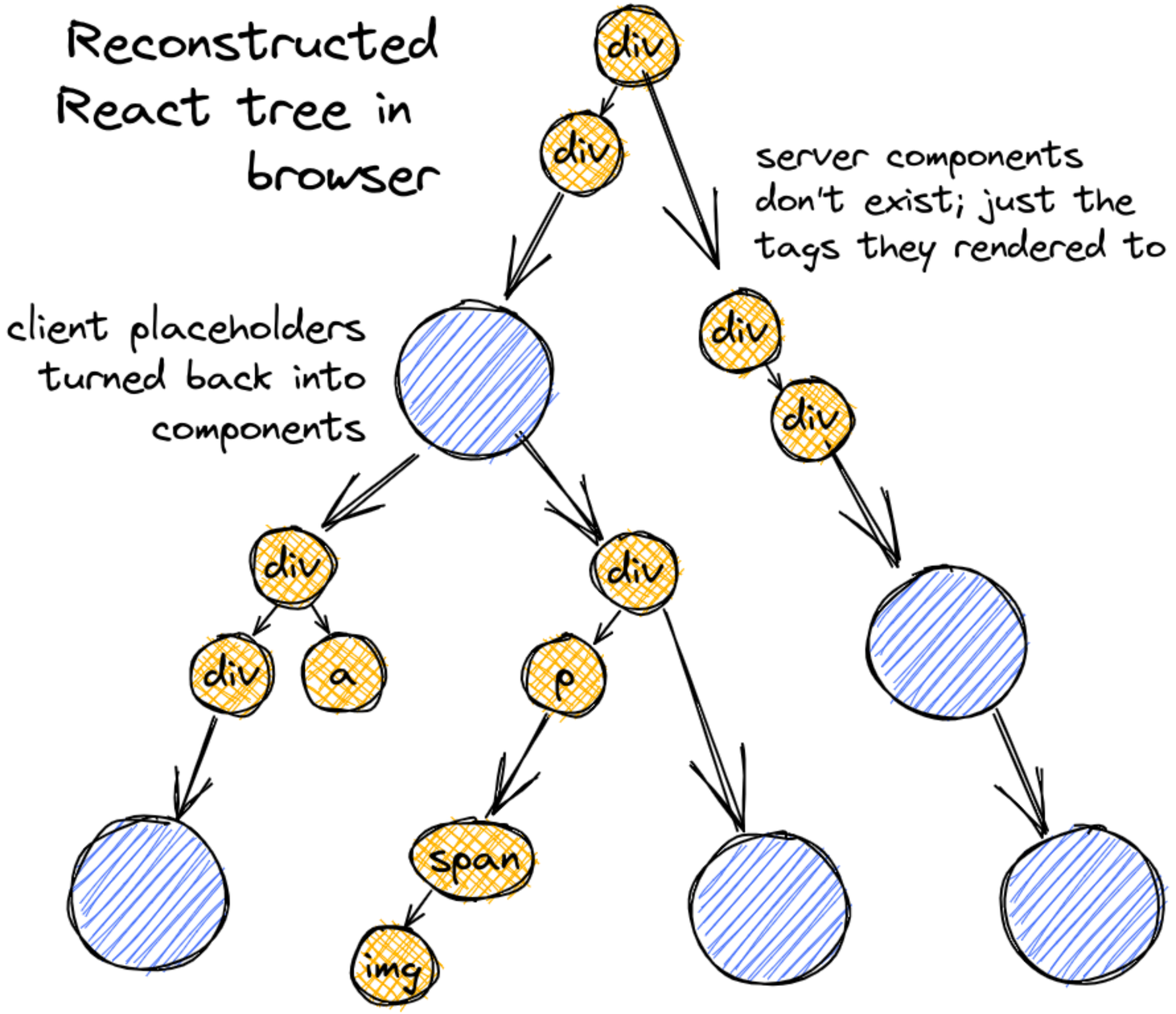
React

Server Components

React Server Components

- React Server Components pozwalają wykorzystywać możliwości serwera oraz klienta (przeglądarki) w procesie renderowania aplikacji React
- RSC sprawia, że niektóre komponenty w drzewie zostaną wyrenderowane przez serwer, a inne wyrenderowane przez przeglądarkę
- RSC to nie jest SSR (może to być mylące, ale mechanizm nie ma za zadanie zastąpienia SSR)
- Możemy wykorzystać SSR wraz RSC i dokonać odpowiedniej hydratacji z poziomu przeglądarki
- Serwer jest zazwyczaj bliżej dostępu do źródeł danych, takich jak baza danych
- RSC mogą niskim kosztem zwiększyć użyteczność ciężkich modułów, takich jak pakiety npm'owe wykonujące operacje renderowania markdown do html
- Na ten moment jedyną implementację tego mechanizmu przygotował NextJS (niestety nie jest ona do końca stabilna)


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


Reconstructed
React tree in
browser
client placeholders
turned back into
components
server components
don't exist; just the
tags they rendered to
div
div
div
div
div
div
div
a
p
span
img

ReactDOMClient

- Pakiet zawiera metody specyficzne dla klienta - przydatne do inicjalizacji aplikacji po stronie klienta
- Większość naszych komponentów nigdy nie będzie musiała użyć tego modułu
- Ma wyeksportowane dwie metody:
  - **createRoot** - tworzy root aplikacji React i go następnie zwraca
    - Root może być **render**’owany **i unmount**’owany
  - **hydrateRoot** - działa podobnie do createRoot, ale dodatkowo jest wykorzystywane do hydratacji kontenera, który został wyrenderowany przez ReactDOMServer

ReactDOMServer

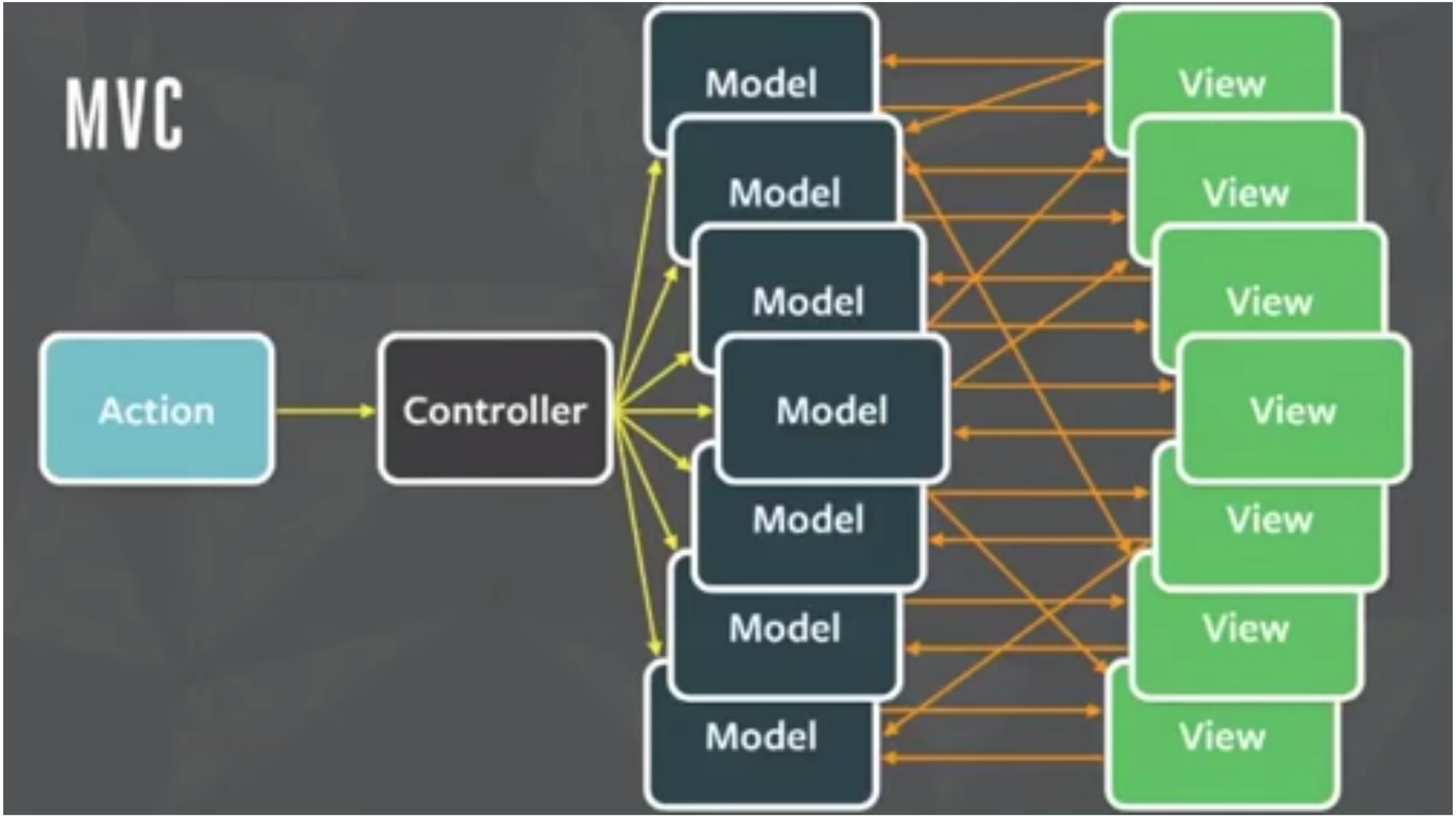
- API ReactDOMServer zezwala na rederowanie komponentów React do wskazanej formy:
  - renderToPipeableStream
  - renderToReadableStream
  - renderToNodeStream
  - renderToStaticNodeStream
  - renderToString
  - renderToStaticMarkup

FLUX

Architektura

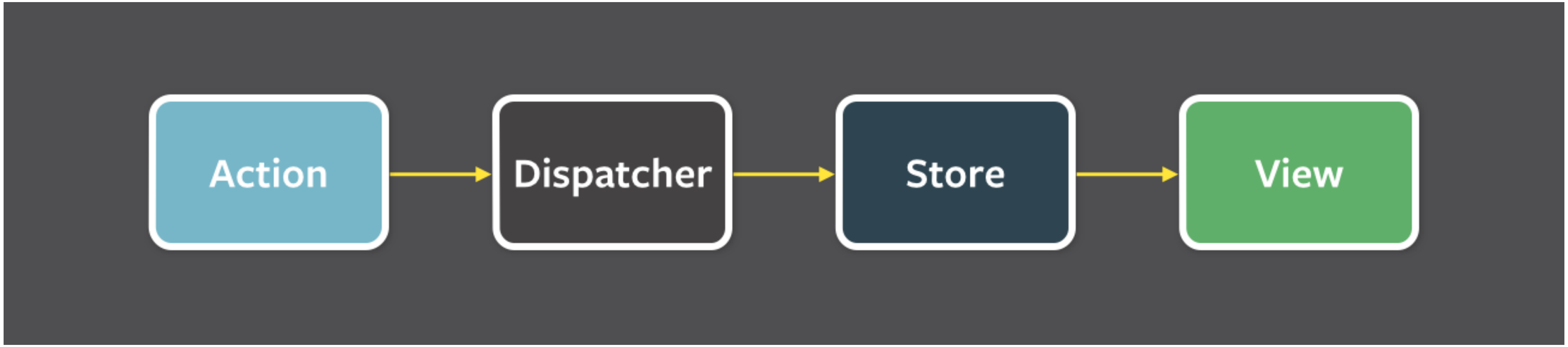
Problemy z MVC

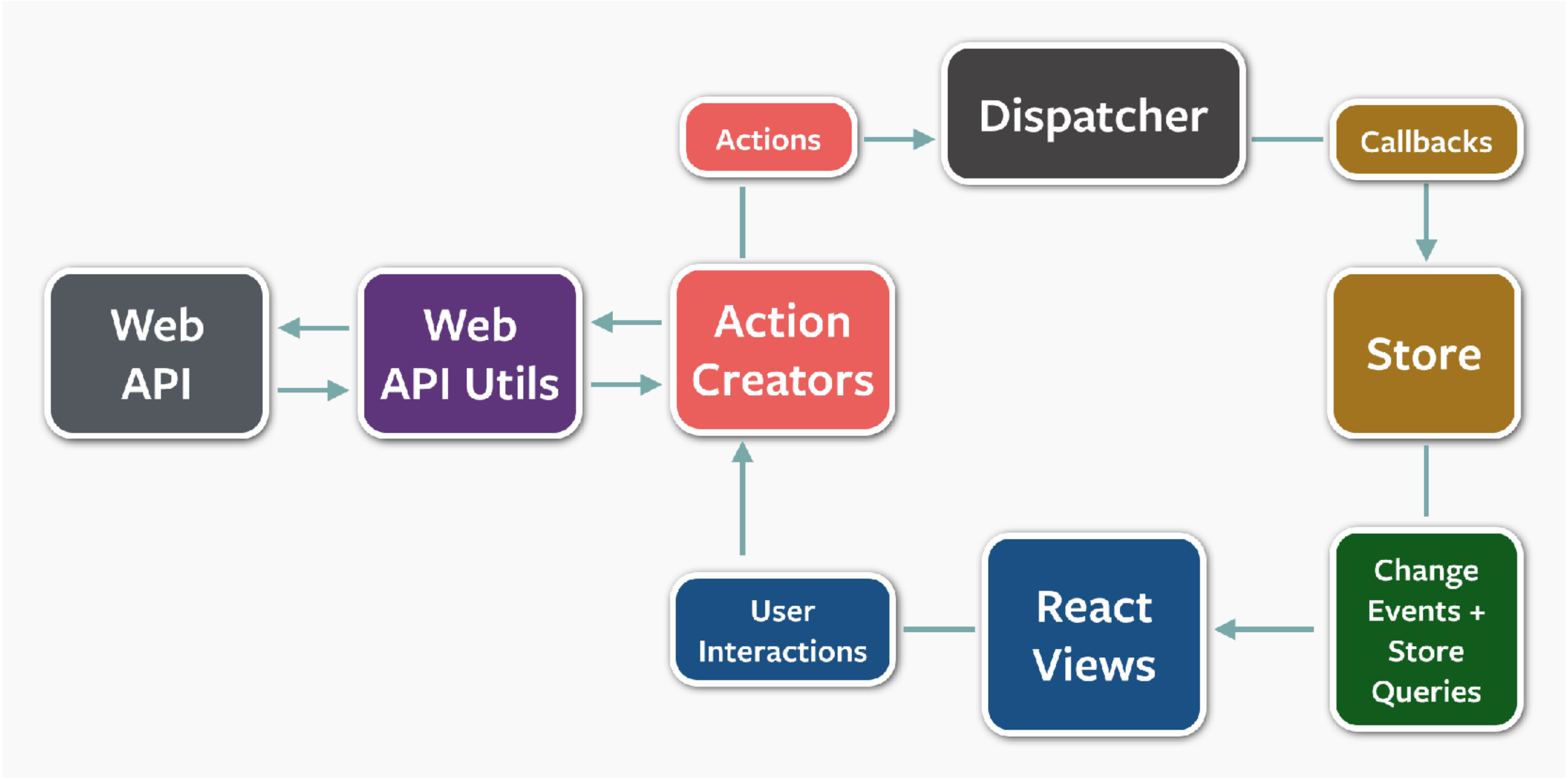
- Modele i Widoki tworzą wiele dwukierunkowych powiązań
- Akcje użytkownika wpływają na wiele widoków i modeli, które mogą zmieniać kolejne modele… itd.
- Utrudnia to określenie z którego kierunku pochodzą dane (komplikuje to przepływ danych)
- Zwiększa to ryzyko popełnienia błędu


MVC
Action
Controller
Model
Model
Model
Model
Model
Model
Model
View
View
View
View
View
View
View

FLUX

- Architektura FLUX zakłada jednokierunkowy przepływ danych:
  - Akcje są jedynym sposobem na zmianę stanu aplikacji
  - Dispatchery przekazują akcje do specyficznego Magazynu (Store’a)
  - Magazyn (Store) jest jedynym źródłem prawdy, może zmienić swój stan w reakcji na akcje. Stan aplikacji jest pobierany z magazynu i przekazywany do widoku
  - Widok obserwuje magazyny i renderują zmiany w aplikacji


Action
Dispatcher
Store
View


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

Akcje

- Akcje są obiektami, które zawierają przynajmniej dwie właściwości:
  - **type** - typ akcji, na nim bazuje magazyn, na jego podstawie wie co powinien zrobić z payloadem
  - **payload** - dane przekazane do akcji

```
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',
  },
};
```

Magazyn (Store)

- Magazyn składa się z dwóch części:
  - Obiekt który zawiera stan aplikacji
  - Funkcje, które obsługują przychodzące akcje, modyfikuję stan magazynu i powiadamiają widoki o zmianach
- Dobrą praktyką jest tworzenie magazynu jako obiektu (klasy), który spełnia te role

```
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);
  }
}
```

Zalety

- Wykrywanie zmian jest proste i wydajne, w związku z niemutowanymi strukturami danych. Musimy po prostu porównać obiekty stanu - nowy ze starym.
- Kiedy obiekt jest taki sam, nic się zmieniło, nie ma potrzeby żeby aktualizować widok

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

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

Redux

Redux

Architektura

Redux

- Redux jest architekturą podobą do Flux, inspirowaną programowaniem funkcyjnym
- Główne założenia:
  - Jeden magazyn (jedno źródło stanu)
  - Niemutowalny stan
  - Magazyn nie modyfikuje danych (logika aktualizacji stanu jest wydzielona do reduktorów)

Reduktory

- Reducer może pracować z zagnieżdżonym stanem
- Reducer powinien zawsze zwracać nowy obiekt stanu, kiedy wystąpiła jakakolwiek jego modyfikacja
- Nie jest dozwolone modyfikowanie stanu inaczej niż poprzez reduktory

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

Kreatory akcji

- Dla uproszczenia kreatory akcji to funkcje, które wykorzystywane są po to, aby nie zapomnieć o ważnych polach podczas dispatch’owania akcji
- Taki sposób jest dużo bardziej efektywny podczas wprowadzania zmian do stanu aplikacji

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

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

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

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

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

Slice

- Dodatkowy poziom abstrakcji
- Slice to element odpowiedzialny za przechowywanie logiki i reduktorów z jednego obszaru biznesowego (domeny / części funkcjonalnej aplikacji)
- Nazwa pochodzi od dzielenia głównego redux’a na wiele mniejszych „plasterków” (slice’ów) stanu

```
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;
    },
```

```
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';
      });
  },
});
```

Zagnieżdżanie reducer

- Możemy posiadać w aplikacje wiele różnych reducer’s pogrupowanych według funkcjonalności
- Reducery mogą być zagnieżdżane w jednym reducerze, lub
- Kombinowane ze sobą (za pomocą wewnętrznej metody **combineReducers**)

Middleware’y

- **Middleware’y** mogą być użyteczne w wielu przypadkach, np. podczas pobierania danych z API
- **Middleware** to funkcja, która będzie wykonywana po wyzwoleniu akcji, ale przed reducerem
- Do zarządzania middleware’ami w Redux możemy użyć biblioteki **redux-thunk**
- Popularne biblioteki do middleware’ów:
  - **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… jeden zamiast wszystkiego

- Jedno wywołanie configureStore ma zastąpić wywoływanie wielu osobnych operacji:
  - Pozwala na zagnieżdżanie (kombinowanie) reduktorów
  - Tworzy magazyn
  - Dodaje obsługę middleware’ów (domyślnie redux thunk)
  - Dodaje middleware’y odpowiedzialne za sprawdzanie popularnych błędów
  - Weryfikuje niemutowalność i spójność magazynu
  - Domyślnie łączy aplikację z **Redux DevTools Extension** (łatwiejszy development)

```
export function Counter() {
  const count = useSelector(selectCount);
  const dispatch = useDispatch();

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

MVC vs FLUX vs REDUX

- **MVC:**
  - Dwukierunkowy przepływ danych
  - Brak magazynu
  - Logika w Controller’ach
  - Debuggowanie utrudnione z związku z dwukierunkowym przepływem danych
  - Wykorzystywany na BE (Django) i FE (AngularJS, Ember)
- **FLUX:**
  - Jednokierunkowy przepływ danych
  - Wiele magazynów
  - Magazyn jest odpowiedzialny za logikę
  - Debuggowanie łatwe, na poziomie dispatcher’ów
  - Wykorzystywana tylko na FE: (React, Angular, Vue)
- **REDUX:**
  - Jednokierunkowy przepływ danych
  - Jeden magazyn
  - Reduktory są odpowiedzialne za logikę
  - Debuggowanie szybsze, ponieważ wszystkie dane w jednym miejscu
  - Wykorzystywana tylko na FE: (React, Angular, Vue)

Zadanie

20 minutes
tasks/TASK_13.md

Context API

React

```jsx
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>
    );
  }
}
```

```jsx
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('⚛️'))
```

Redux vs Context

- **ContextAPI**
  - Jest częścią biblioteki React
  - Wymagana minimalna konfiguracja
  - Zaprojektowany dla danych statycznych (takich, które nie są często odświeżane lub aktualizowane) np.:
    - Theme
    - Dane użytkownika
    - Proste wartości słownikowe
  - Logika widoku I logika zarządzania stanem są w jednym miejscu
  - Debuggowanie może być trudne
- **Redux**
  - Zewnętrzna biblioteka
  - Wymagana dodatkowa konfiguracja aby zintegrować Redux z React
  - Świetnie nadaje się do pracy zarówno z danymi statycznymi, jak i dynamicznymi
  - Lepsza organizacja kodu (separacja logiki UI i logiki zarządzania stanem)
  - Posiada bardzo dobre narzędzie do debuggowania
  - Nadaje się do przechowywania dużych ilości danych

You might not need Redux

Dan Abramov, twórca Reduxa, 2016

Zadanie

20 minutes
tasks/TASK_14.md

Mobx

Mobx

- MobX to biblioteka open source’owa do zarządzania stanem
- Nie jest powiązana z jakimkolwiek frameworkiem
- Stan aplikacji odnosi się do całego modelu aplikacji i może zawierać różne typy danych, w tym tablice, liczby i obiekty
- Akcje są metodami, które manipulują i aktualizują stan. Akcje można powiązać z modułem obsługi zdarzeń JavaScript, aby mieć pewność, że zdarzenie w interfejsie użytkownika je wyzwoli.
- Magazyn MobX jest reaktywny, a zatem reaguje na zmiany.

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)

```

Podstawy/idea RxJS

Asynchroniczność

RxJS

- Biblioteka która wykorzystuje koncepcje reaktywnego programowania.
- Opiera swoje założenia na Observable’ach
- RxJS to taki Lodash dla zdarzeń
- Strumienie możemy obserwować, przekształcać a następnie subskrybować w celu otrzymania informacji, kiedy nastąpiła zmiana.
- RxJS posiada bogatą bibliotekę operatorów umożliwiających przekształcenia:
  - przekształcające (np. delay, map, debounce, scan)
  - łączące (np. merge, sample, startWith, zip)
  - filtrujące (np. distinctUntilChanged, filter, skip)
  - i wiele innych.

Promise vs Observable

- Promise
  - Jest zawsze asynchroniczny
  - Może zwrócić jedną wartość
  - Promise inicjowany jest podczas tworzenia
  - Nie można anulować Promise’a
  - W przypadku wystąpienie błędu Promise się kończy

- Observable
  - Może być synchroniczny lub asynchroniczny
  - Może zwrócić jedną lub wiele wartości
  - Na observable się subskrybujemy, subskrypcja jest aktywna dopóki ręcznie jest nie anulujemy
  - Observable możemy anulować
  - Observable możemy ponawiać w przypadku wystąpienia błędu

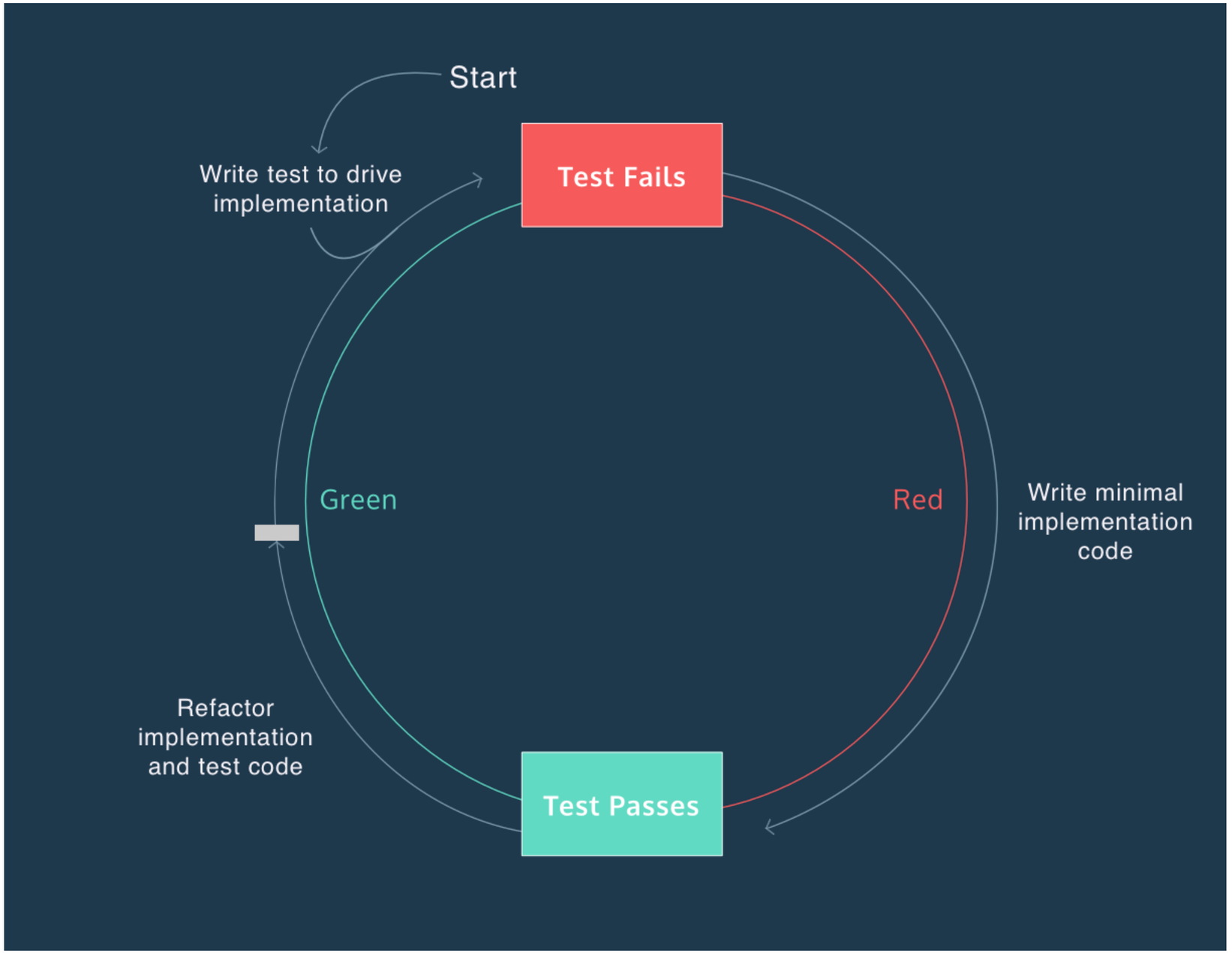
Testowanie

Testowanie

- Możemy testować różne przypadki, nie tylko scenariusze testowe, ale również sytuacje w których aplikacja otrzymała złe lub puste wartości
- Testy mają nam zagwarantować wyłapywanie wszystkich problemów, które mogą być wysyłane przez przyszłe zmiany (najwyższa wartość testów objawi się w przyszłości)
- Testy dostarczają nam informacji o błędach w kodzie (szybko)
- Testy są także dokumentacją kodu dla innych programistów w projekcie - dzięki nim mamy informacje o odpowiedzialnościach poszczególnych fragmentów aplikacji

TDD

- Test Driven Development
- Testy bazują na cyklu: Red / Green / Refactoring
  - **Red** — w tej fazie myślimy o tym co chcemy zaprogramować
  - **Green** — ta faza pokrywa odpowiedź na pytanie jak zrobić, żeby przeszły testy
  - **Refactor** — myślimy o tym jak poprawić napisaną w poprzednich fazach implementację
- W tym podejściu do testowania testy piszemy zanim zostanie napisana rzeczywista implementacja


Start
Write test to drive implementation
Test Fails
Green
Red
Write minimal implementation code
Refactor implementation and test code
Test Passes

TDD: Red

- Czerwona faza to zawsze punkt wyjściowy od którego zaczynamy testowanie
- Celem tej fazy jest napisanie testów, które informują nas o tym co powinna robić implementacja funkcjonalności (definiują cel funkcjonalności)
- Na końcu tej fazy wszystkie testy powinny być czerwone (funkcjonalność jeszcze nie istnieje)
- Testy będą przechodzić dopiero kiedy wymagania zostaną pokryte.

```
1) sortArray returns an array sorted in ascending order:
    TypeError: Calculate.arraySort is not a function
     at Context.it (test/index_test.js:10:32)
```

TDD: Green

- W zielonej fazie piszemy kod implementacji, tak aby testy napisane w fazie czerwonej przeszły (stały się zielone)
- Celem jest znalezienie rozwiązania, bez głębszego wchodzenia w optymalizację naszej implementacji.
- Na końcu tej fazy „jesteśmy na zielono”. Możemy zacząć myśleć o optymalizacjach naszego kodu.

TDD: Refaktoryzacja

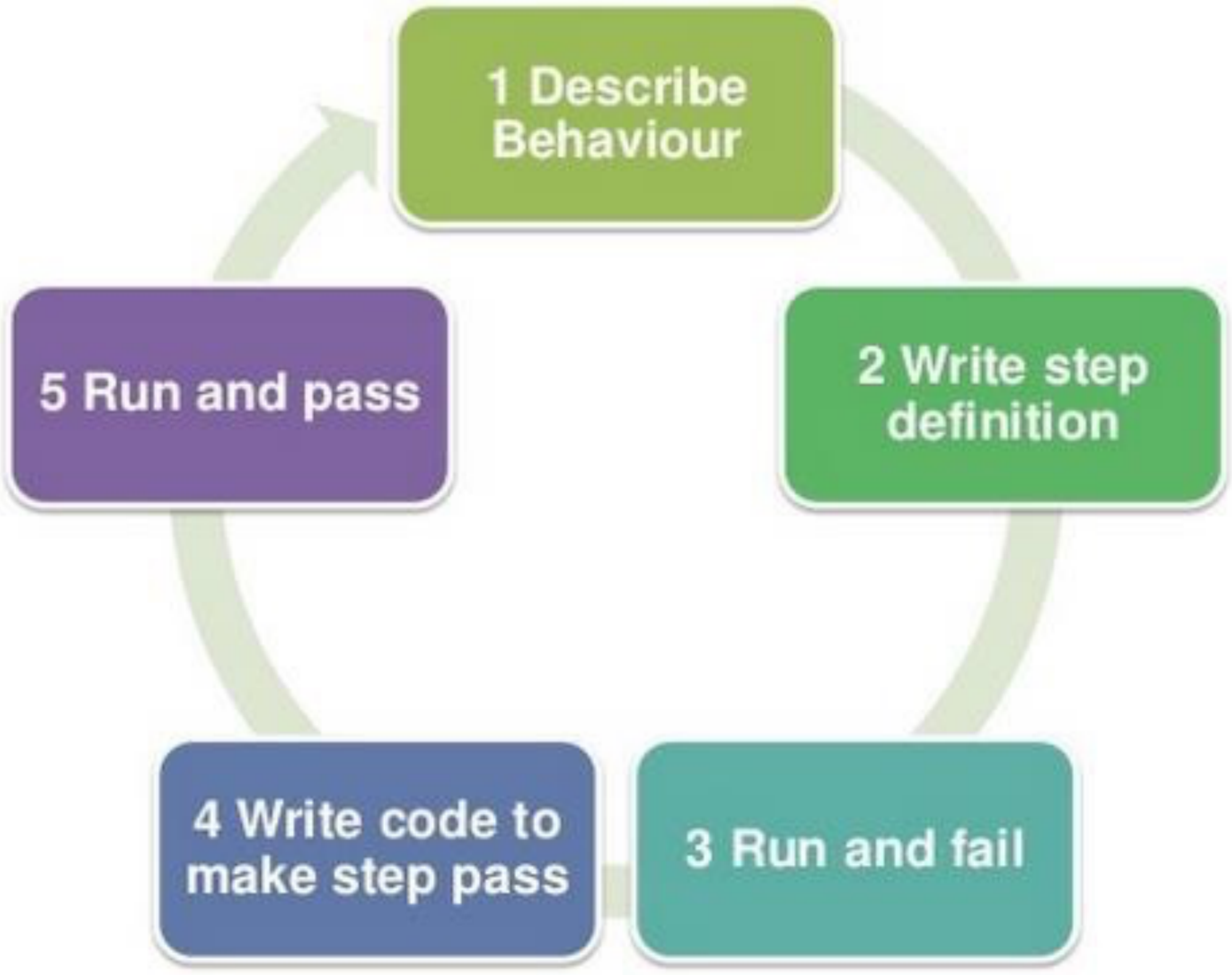
- W tej fazie nadal „jesteśmy na zielono”.
- Ostatnia faza to moment na zastanowienie się jak napisać nasz kod lepiej lub bardziej wydajnie.
- Podczas refaktoryzkcja nie chodzi o zmiany wyniku naszej funkcji, a o osiągnięcie tego samego efektu przy bardziej opisowym lub szybszym kodzie.
- Pytania, które powinniśmy sobie zadać kiedy myślimy o refaktoryzacji:
  - Czy mogę poprawić moje testy, aby były lepiej dopasowane?
  - Czy moje testy zwracają wartościowy feedback na temat funkcjonalności?
  - Czy moje testy są odpowiednio odizolowane od innych testów / modułów?
  - Czy mogę zredukować duplikację kodu w moich testach lub w samej implementacji?
  - Czy moja implementacja może być czytelniejsza?
  - Czy moja implementacja może być bardziej wydajna?

BDD

- BDD - Behavior Driven Development
  - Zwinne podejście do testowania
  - Dokumentacja i testy są pisane możliwie najbardziej naturalnym językiem
  - W idealnym świecie pisanie testów jest jedną z najważniejszych rzeczy w procesie wytwarzania oprogramowania
  - Funkcjonalności powinny być opisane w językach służących do opisywania scenariuszy / feature’ów (np. Gherkin) jeszcze zanim rozpocznie się faza developmentu
  - Dokumentacja może się zmienić (lub raczej rozszerzyć) w fazie projektowania i implementacji.
  - Dokumentacja jest ważna dla developmentu i dla testowania (manualnego i automatycznego).

BDD: Popularne błędy

- Biznes nie dostarcza gotowych scenariuszy: Given / When / Then
- Scenariusze testowe są przygotowywane po developmencie
- Biznes nie jet zainteresowany wynikiem i treścią testów


1 Describe Behaviour
2 Write step definition
3 Run and fail
4 Write code to make step pass
5 Run and pass

BDD: Narzędzia

- Dla generowania raportów:
  - Serenity BDD
  - Allure
- Frameworki:
  - Cucumber - jeden z najpopularniejszych framework do test BDD na rynku, oryginalnie napisany w Ruby
  - Quantum - open-source, napisany w Javie, zaprojektowany przez Perfecto
  - SpecFlow - .NET BDD framework
  - JBehave - główny framework BDD dla Javy and innych języków JVM
  - Codeception - popularny framework dla PHP, jest narzędziem, które doskonale nadaje się do unit testów, testowania API i testowania funkcjonalności, w tym również wspiera BDD

Zalety testów BDD

- Zakłada podejście „shift-left” - które mówi o tym, że testy są wykonywane wcześniej niż tworzenie kodu (przesunięte w lewo na timeline projektu)
- Łatwe w utrzymaniu
- Oferowane w wielu różnych językach programowania

Edit | View tests

Raw version

1	Given the account balance is $100
2	and the card is valid
3	and the machine contains enough money
4	When the Account Holder requests $20
5	Then the ATM should dispense $20
6	and the account balance should be $80

and the card should be

Create action

- the card should be returned
- the ATM should say the card has been retained
- the account balance should be "p1"
- the ATM should retain the card
- the ATM should dispense "p1"
- the ATM should not dispense any money
- the ATM should say there are insufficient funds
- the card is disabled
- the card is valid
- the account balance is "p1"

Język Gherkin

- Gherkin jest językiem czytelnym dla biznesu, specyficznym dla domeny
- Został stworzony w celu opisywania zachowań w ramach projektu / aplikacji
- Daje możliwość usunięcie szczegółów logiki z testów behawioralnych
- Gherkin spełnia dwa cele:
  - Dostarcza dokumentację projektu
  - Automatyzuje testy
- Składnia języka Gherkin jest zorientowana liniowo (co oznacza, że używa wcięć do definiowania struktur podobnie jak Python czy YAML)

```
Feature: Some terse yet descriptive text of what is desired
  In order to realize a named business value
  As an explicit system actor
  I want to gain some beneficial outcome which furthers the goal

  Scenario: Some determinable business situation
    Given some precondition
      And some other precondition
     When some action by the actor
      And some other action
      And yet another action
     Then some testable outcome is achieved
      And something else we can check happens too

  Scenario: A different situation
```

Typy testów

- Testy jednostkowe
  - Testujemy małe fragmenty kodu (pojedyncze funkcje, komponenty). Na tym poziomie nie powinniśmy myśleć dlaczego użytkownik coś robi, ale czy wynik działania funkcji / komponentu jest zgodny z oczekiwaniami.
  - Ten typ testów jest uruchamiany w separacji od reszty aplikacji. Oznacza to, że komponent może wykorzystywać tylko te dane, które zostały przekazane do testu.
- Testy integracyjne
  - Testujemy współpracę pomiędzy różnymi częściami aplikacji. Testy integracyjne pozwalają na wyłapywanie sytuacji, kiedy komponenty działają w separacji zgodnie z oczekiwaniami, a razem już nie.
- Testy e2e
  - Określane również jako testy systemowe. W tych testach symulujemy zachowanie użytkownika pracującego z aplikacją. Ten typ testów sprawdza całościowe działanie naszej aplikacji.

Miary jakości testów jednostkowych

- Test coverage
  - Wartość jest wyrażana w % (gdzie max to 100%)
  - Pokazuje nam jak dużo linii kodu zostało pokrytych testami
  - Ta wartość to uproszczona informacja o testach (łatwo o wprowadzenie antywzorca Kłamca)
  - Kiedy testy są dobre, wówczas wysoki wynik test coverage jest efekt ubocznym
- Proporcja pomiędzy liniami kodu a liniami testów (idealnie 1:1)
- Miary empiryczne
  - Związane z raportowaniem testów i zbieraniem danych historycznych z raportów

White Box vs Black Box

- White Box
  - Testy strukturalne
  - Testy bazują na kodzie źródłowym
  - O wiele łatwiej przygotować kod do całościowej optymalizacji dzięki testom
  - Łatwiejsze do automatyzacji

- Black Box
  - Testy funkcjonalne
  - Nie mamy dostępu do kodu źródłowego
  - Testujemy tylko zakładane funkcjonalności
  - W tego typu testach mamy większą szansę na znalezienie większej liczby błędów, ale niekonicznie ich pochodzenia

Antywzorce w testach

- Obywatel drugiej kategorii
  - Kod testów traktowany jako ten gorszy.
  - Nie jest utrzymywany, wszędzie mamy do czynienia z duplikacją kodu, chaosem i mylącymi nazwami
- Brak szacunku
  - Deweloper robi jakąś zmianę i nie aktualizuje unit testów
  - Po wprowadzeniu system continuous integration zwraca błąd spowodowany nieprzechodzącymi testami
  - Zamiast poprawić testy usuwa się je, zakomentowuje albo ignoruje
- Optymistyczna ścieżka
  - Testowanie tylko podstawowego działania funkcji bez zastanowienia się nad możliwymi wyjątkami, warunkami brzegowymi, czy złośliwymi kombinacjami danych wejściowych.
- Gigant
  - Test zawierający bardzo dużo linii kodu – kilkaset, albo nawet ponad tysiąc.
  - Test jest tak duży, że nie wiadomo do końca co robi, jego utrzymanie jest bardzo trudne i może on posiadać swoje własne bugi.

Antywzorce w testach

- Pasażer na gapę
  - Zamiast stworzyć nowy test, dodajemy kolejny cykl Arrange-Act-Assert do istniejącego testu.
  - W ten sposób istniejące testy się wydłużają, a ich cel się rozmywa.
  - W końcu taki test przeobraża się w giganta.
- Chain gang
  - Kolejne testy są od siebie zależne i muszą być wykonywane w odpowiedniej kolejności.
  - Usunięcie lub zmiana jednego testu z łańcucha skutkuje wtedy zepsuciem całej grupy testów.
- Sobowtór
  - Na potrzeby testu mockujemy jakąś zależność. Jednak aby wykonać test nie wystarczy nam podstawić zwracaną wartość. Zamiast tego w mocku kopiujemy zachowanie rzeczywistej funkcji.
  - Jeżeli będziemy chcieli wprowadzić zmiany, musimy je uwzględnić zarówno w kodzie produkcyjnym, jak i w mocku.

Antywzorce w testach

- Kłamca
  - Test jest tak skonstruowany, żeby wchodził w odpowiednie ścieżki w kodzie i nabijał w ten sposób code coverage.
  - Jednak nie sprawdza on w żaden sposób poprawności wykonywania tych ścieżek
- Inspektor
  - Aby osiągnąć lepsze pokrycie, test bazuje na konkretnej implementacji i łamie zasady enkapsulacji.
  - W ten sposób otrzymujemy skomplikowane testy utrudniające refactor kodu w przyszłości.

Testowanie FE

Co powinniśmy testować?

- komponent prezentacyjny (komponent widoku)
- komponent z logiką biznesową
- serwisy
- funkcje pomocnicze, custom hooks

Domyślne pliki testów

- **.js** w katalogu __**tests**__
- **.test**.js
- **.spec**.js

Jest

- Javascript Testing Framework
- Zezwala na dostęp do struktury DOM za pomocą biblioteki **jsdom**
- **jsdom** jest tylko symulacją przeglądarki - pokrywa większość przypadków, w którymi napotkamy się testując komponenty React
- W Jest możemy mockować moduły (takie jak serwisy) czy timery.

Jest: Matchers

- Jest udostępnia „matchers" aby ułatwić porównywanie wartości na wiele różnych sposobów:
  - **toBe** - używa Object.is do testowania dopasowania ścisłego
  - **toEqual** - rekursywnie sprawdza każde pole w obiekcie lub tablicy
  - **not**
  - **toBeNull, toBeUndefined, toBeDefined, toBeTruthy, toBeFalsy** - dla sprawdzenia prawdziwości
  - **toBeGreaterThan, toBeLessThan, toBeGreaterThanOrEqual, toBeLessThanOrEqual**, **toBeCloseTo** - dla liczb
  - **toMatch** - dla stringów
  - **toContain** - dla tablic
  - **toThrow** - dla wyjątków

```
test('two plus two is four', () => {
  expect(2 + 2).toBe(4);
});

test('object assignment', () => {
  const data = {one: 1};
  data['two'] = 2;
  expect(data).toEqual({one: 1, two: 2});
});

test('adding positive numbers is not zero', () => {
  for (let a = 1; a < 10; a++) {
    for (let b = 1; b < 10; b++) {
      expect(a + b).not.toBe(0);
    }
  }
});
```

Async

- Kod JavaScript uruchamiany jest zazwyczaj asynchronicznie (zdarzenia, zapytania itp)
- Zwrócenie obietnicy z testu spowoduje, że Jest będzie oczekiwał aż obietnica zostanie rozwiązana.
- Jest posiada również matchery do kodu asynchronicznego: **resolves** i **rejects**

```
test('the data is peanut butter', () => {
  return fetchData().then(data => {
    expect(data).toBe('peanut butter');
  });
});

test('the data is peanut butter', async () => {
  const data = await fetchData();
  expect(data).toBe('peanut butter');
});

test('the fetch fails with an error', async () => {
  expect.assertions(1);
  try {
    await fetchData();
  } catch (e) {
    expect(e).toMatch('error');
  }
});

test('the data is peanut butter', async () => {
  await expect(fetchData()).resolves.toBe('peanut butter');
});
```

```
import {describe, expect, test} from '@jest/globals';
import {sum} from './sum';

describe('sum module', () => {

  describe('when sth happen', () => {

    test('adds 1 + 2 to equal 3', () => {
      expect(sum(1, 2)).toBe(3);
    });
  });

});
```

Setup testów

- Jeśli musimy wykonać operacje wielokrotnie przed i po uruchomieniu jakiegoś testu, możemy skorzystać z Hooków: **beforeEach** i **afterEach**.
- W niektórych przypadkach potrzebujesz wykonać jakąś operację raz przed / po wszystkimi testami z danej grupy. Dla takich przypadków Jest udostępnia hooki: **beforeAll** i **afterAll**
- Jeśli musimy uruchomić test jeden test (ponieważ nam nie przechodzi), tymczasowo możemy zastąpić funkcję **test** za pomocą funkcji **test.only**

```
beforeEach(() => {
  return initializeCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});
```

```
test.only('this will be the only test that runs', () => {
  expect(true).toBe(false);
});

test('this test will not run', () => {
  expect('A').toBe('A');
});
```

Mockowanie funkcji

- Mockowanie funkcji zezwala na testowanie połączeń pomiędzy różnymi częściami kodu za pomocą usunięcia aktualnej implementacji funkcji, przechwyceniem jej uruchomienia (wraz z parametrami), przychwycenie instancji konstruktora funkcji, kiedy zostały zainicjowane przez operator **new**. Pozwala ponadto na zwrócenie własnej wartości z mockowanej funkcji.
- Wszystkie mockowane funkcje mają specjalną właściwość **mock**, gdzie możemy znaleźć informacje na temat ilości i sposobu wywołania danej funkcji.
- Możemy wykorzystać mockowanie implementacji całych modułów.

```
function forEach(items, callback) {
  for (let index = 0; index < items.length; index++) {
    callback(items[index]);
  }
}

const mockCallback = jest.fn(x => 42 + x);
forEach([0, 1], mockCallback);

// The mock function is called twice
expect(mockCallback.mock.calls.length).toBe(2);
```

```
import axios from 'axios';
import Users from './users';

jest.mock('axios');

test('should fetch users', () => {
  const users = [{name: 'Bob'}];
  const resp = {data: users};
  axios.get.mockResolvedValue(resp);

  // or you could use the following depending on your use case:
  // axios.get.mockImplementation(() => Promise.resolve(resp))

  return Users.all().then(data => expect(data).toEqual(users));
});
```

Timers

- Natywne funkcje timerów (np. **setTimeout, setInterval, clearTimeout, clearInterval**) są mniej niż idealne dla testowania na środowiskach testerskich, ponieważ są zależne od prawdziwego upływu czasu.
- W naszym teście możemy włączyć fake timers poprzez wywołanie **jest.useFakeTimers()**

```
jest.useFakeTimers();
jest.spyOn(global, 'setTimeout');

test('waits 1 second before ending the game', () => {
  const timerGame = require('../timerGame');
  timerGame();

  expect(setTimeout).toHaveBeenCalledTimes(1);
  expect(setTimeout).toHaveBeenLastCalledWith(expect.any(Function), 1000);
});
```


PASS packages/diff-sequences/src/__tests__/index.test.js
PASS packages/jest-diff/src/__tests__/diff.test.js
PASS packages/jest-mock/src/__tests__/jest_mock.test.js
PASS packages/jest-util/src/__tests__/fakeTimers.test.js
PASS packages/pretty-format/src/__tests__/prettyFormat.test.js
RUNS packages/jest-haste-map/src/__tests__/index.test.js
RUNS packages/pretty-format/src/__tests__/DOMElement.test.js
RUNS packages/jest-config/src/__tests__/normalize.test.js
RUNS packages/expect/src/__tests__/matchers.test.js
RUNS packages/pretty-format/src/__tests__/Immutable.test.js
RUNS packages/expect/src/__tests__/spyMatchers.test.js
RUNS packages/jest-cli/src/__tests__/SearchSource.test.js
RUNS packages/jest-runtime/src/__tests__/script_transformer.test.js
RUNS packages/jest-cli/src/__tests__/watch.test.js
RUNS packages/jest-haste-map/src/crawlers/__tests__/watchman.test.js
RUNS packages/pretty-format/src/__tests__/react.test.js
Test Suites: 5 passed, 5 of 303 total
Tests: 332 passed, 332 total
Snapshots: 21 passed, 21 total
Time: 4s

Testowanie snapshotowe

- Testy Snapshotów są przydatnym narzędziem, gdy chcemy być pewni że nasz UI nie zmienił się w sposób nieoczekiwany.
- Typowy test snapshotowy renderuje komponent UI i porównuje go z kodem snapshot’u (artefakt), który najczęściej jest przechowywane razem z testami.
- Artefakt snapshot’u powinien być commitowany razem ze zmianami w kodzie i przechodzić review tak samo jak pozostałe części naszego kodu.
- Aby zaktualizować snapshot uruchom: j**est -u** lub **jest —updateSnapshot**

Testowanie snapshotowe

- Traktuj snapshot tak jak kod
  - Commituj snapshot’y i wykonuj na nich proces code review
- Testy snapshotowe powinny być deterministyczne
  - Uruchamianie tych samych testów wiele razy na komponencie, który nie zmienił się, powinno zawsze zwrócić ten sam wynik
  - Jesteśmy odpowiedzialni za upewnienie się, że wygenerowany snapshot nie zawiera informacji specyficznych dla platformy czy innych przypadkowych danych
- Używaj opisowych nazw dla snapshotów

```
import renderer from 'react-test-renderer';
import Link from '../Link';

it('renders correctly', () => {
  const tree = renderer
    .create(<Link page="http://www.facebook.com">Facebook</Link>)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```

React Testing Library

- Jest to biblioteka która pomaga nam pisać scenariusze testowe
- Zawiera zestaw funkcji pomocniczych dla testowania komponentów React bez szczegółów implementacyjnych
- Nie zezwala na renderowanie komponentów w sposób płytki (shallow) - czyli bez potomków
- **React Testing Library nie jest alternatywą dla Jest'a**, obie biblioteki potrzebują siebie nawzajem i każda z nich ma inne zadanie do wykonania
- RTL jest zamiennikiem dla **Enzyme**
- **React Testing Library** nie jest specyficzne dla żadnego frameworki do testowania. RTP jest zbudowany na **DOM Testing Library**.

```
import {render, screen} from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import '@testing-library/jest-dom'
import Fetch from './fetch'

test('loads and displays greeting', async () => {
  // ARRANGE
  render(<Fetch url="/greeting" />)

  // ACT
  await userEvent.click(screen.getByText('Load Greeting'))
  await screen.findByRole('heading')

  // ASSERT
  expect(screen.getByRole('heading')).toHaveTextContent('hello there')
  expect(screen.getByRole('button')).toBeDisabled()
})
```

React Testing Library

	No Match	1 Match	1+ Match	Await?
getBy	throw	return	throw	No
findBy	throw	return	throw	Yes
queryBy	null	return	throw	No
getAllBy	throw	array	array	No
findAllBy	throw	array	array	Yes
queryAllBy	[]	array	array	No

Zadanie

20 minutes
tasks/TASK_15.md

Vitest

- **Vitest** jest popularną alternatywą dla Jest'a, w szczególności z powodu wykorzystywania **Vite'a**
- **Vitest** może być bezpośrednim zamiennikiem dla Jest'a (ponieważ jest również test runner'em)
- Vitest jest kompatybilny z Jest

CyPress

- biblioteka do pisania testów end-to-end
- dzięki łańcuchowaniu wyrażeń bardzo łatwo pisze się testy
- pisanie przypomina pisanie scenariuszy testowych opartych o zachowanie użytkownika
- zezwala na testowanie całościowej integracji naszego systemu z serwisami zewnętrznymi
- działa w dwóch trybach: headless oraz widoku przeglądarki

```
describe('My First Test', () => {
  it('finds the content "type"', () => {
    cy.visit('https://example.cypress.io')

    cy.contains('type')
  })
})
```

Narzędzia

- **ESLint** - narzędzie do analizy statycznej kodu
- **Prettier**, **Editorconfig** - narzędzia automatyzujące poprawianie kodu stylistycznie podczas zapisu oraz commitowania
- **Visual Studio Code**, **Webstorm** - IDE
- **Live Share** dla VSC - narzędzie do wspólnego pisania kodu
- **Snyk** - narzędzia do analizy podatności w zależnościach w projekcie
- **Playgroundy**:
  - https://www.typescriptlang.org/play
  - https://www.sassmeister.com/
  - https://jsbin.com/?html,output
  - https://jsconsole.com/
  - https://rxmarbles.com/

Dodatkowe pytania?

kahoot.it

XXXXX

Ankieta:

tinyurl.com/bdhc8dtf

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

Dziękuję za uwagę


sages