Maciej Marczak
Numer albumu: 127047

Biblioteki, frameworki JavaScript oraz testy

Praca magisterska na kierunku **informatyka**
napisana pod opieką
dr. Grzegorza Nowaka

Poznań, maj 2022

Streszczenie

Głównym tematem tej pracy jest opisanie trzech najpopularniejszych technologii JavaScript, dodatkowo opisanie testów w React.js, które są wykorzystywane na co dzień w świecie IT. Celem tej pracy jest przybliżenie techonlogii takich jak: Angular, React oraz Vue.

Słowa kluczowe: klasa

Spis treści

Wstęp

Istnieją trzy frameworki do tworzenia aplikacji internetowych, o których słyszał każdy programista frontend: React, Vue.js i Angular. React to biblioteka interfejsu użytkownika, Angular to pełnoprawny framework front-endowy, a Vue.js to progresywny framework.

Mogą być używane prawie zamiennie do tworzenia aplikacji front-end, ale nie są w 100 procentach takie same, więc warto je porównać i zrozumieć różnice. Każda technologia jest oparta na komponentach i umożliwia szybkie tworzenie funkcji interfejsu użytkownika. Jednak wszystkie mają inną strukturę i architekturę.

Jeszcze kilka lat temu programiści debatowali głównie, czy powinni używać Angulara lub Reacta w swoich projektach. Ale w ciągu ostatnich kilku lat został zaobserwowany wzrost zainteresowania trzecim graczem o nazwie Vue.js.

Rysunek 1. Zdjęcie ilustrujące loga frameworków
Źródło: https://kruschecompany.com/angular-vs-react-vs-vue-1/

Praca została podzielona na 4 rozdziały. Każdy z rozdziałów przybliży podstawy każdej z technologii.

W pierwszym rozdziale zostaną omówione podstawy technologii React, Redux oraz wykorzystanie narzędzii do testowania owych komponentów i zdarzeń. Na podstawie tej biblioteki zostanie przybliżone czytelnikowi jej charakterystykę, historie i sposób wykorzystania w codziennej pracy programisty. Następnie zostanie powiedziane o Reduxie, w jaki sposób korzystać z niego.

Drugi rozdział będzie opisywał framework Angular. Przedstawione będzie historia, początki tej technologii, oraz zostaną opisane dobre zasady Angulara, jak projektować komponenty.

Następny rozdział prezentuję technologię, którą wykorzystuje się przy tworzeniu warstwy frontendowej, czyli Vue. Tak jak w dwóch poprzednich rozdziałach zostaną przedstawione podstawy i historia powstania owego frameworka.

Ostatni rozdział zostanie poświęcony porównaniem trzech frameworków, które wykorzystuje się w współczesnym front-endowym świecie. Zostanie opisanie także, dlaczego tak dużo profitu płynie podczas testowania kodu podczas fazy tworzenia aplikacji. Dodatkowo zostanie pokazane, w jaki sposób podejść do tematu, aby wykorzystać architektury każdej z technologii, aby pisać sprawniej testy, które pozwolą nam na dynamiczny rozwój aplikacji oraz stabilność aplikacji na produkcji.

Rozdział 1

Podstawy Reacta oraz jego najpopularniejsze biblioteki

Do stworzenia aplikacji webowych można podchodzić w różny sposób. Można wymienić między innymi: aplikację webową, mobilną oraz także na komputer. Obecnie na rynku w najwyższym stopniu pożądane są aplikacje webowe, które ze względu na jej funkcjonalność umożliwiającą przystępność na wszystkie platformy, które posiadają nowoczesne przeglądarki internetowe takie jak Google Chrome czy Mozilla Firefox. Firmy zajmujące się produkowaniem aplikacji webowych ciągle są zmuszone, aby rozwijąć swoją wiedze o nowe technologie i najnowsze informacje w temacie front-endu. W wyniku tego pozwala to na co raz błyskawiczny wzrost produkcji oprogramowania. Z każdym rokiem język JavaScript jest prężnie rozszerzany i usprawniany. Te wszystkie działania prowadzą do tego, że staje się on oczywisty podczas tworzenia nowego i nowoczesnego kodu.Na prostote zdecydowanie przyczynia się standard ECMA i jego najnowsza wersja ECMA 2021. Znaczenie frameworków w JavaScripcie na rynku IT ciągle rośnie, dodatkowo każdy kto chce być programistą zaczyna się właśnie uczy JavaScript'u. Można śmiało stwierdzić, że jest to jeżyk przyszłości. Trzema najbardziej powszechnymi oraz najwiekszymi frameworkami są: Angular, React i Vue. Te ważne kwestie zagnieżdzają się zresztą , że na aktualnym rynku jest popyt na programistów zajmujących się wyłacznie front-endem jest co raz wieksze, dodatkowo w następnych latach różnica między programistą back-endowym a programistą front-endowym będzie minimalna. Spowodowane jest to, że JavaScript staje się pokrewny do innych języków obiektowych. Bardzo ważnymi pojęciami są Single Page Application i Client Side Rendering, jeśli mówimy o nowoczesnych aplikacjach webowych, dlatego w poniższych podrozdziałach zostanie to przybliżone i dalej zostanie opisany React.

1.1. Single Page Application

Aplikacja jednostronicowa (SPA[1]) to pojedyncza strona (stąd nazwa), na której wiele informacji pozostaje bez zmian i tylko kilka elementów wymaga aktualizacji na raz. Na przykład podczas przeglądania wiadomości e-mail zauważysz, że podczas nawigacji niewiele się zmienia — pasek boczny i nagłówek pozostają nietknięte podczas przeglądania skrzynki odbiorczej. SPA wysyła tylko to, czego potrzebujesz za każdym kliknięciem, a Twoja przeglądarka renderuje te informacje. Różni się to od tradycyjnego ładowania strony, w którym serwer ponownie renderuje całą stronę za każdym kliknięciem i wysyła ją do przeglądarki.

Struktury SPA doskonale nadają się do zabawy z tymi usługami w celu tworzenia angażujących, dynamicznych, a nawet animowanych środowisk użytkownika. Ta pojedyncza metoda po stronie klienta znacznie przyspiesza czas ładowania dla użytkowników i sprawia, że ilość informacji, które serwer musi wysyłać, jest znacznie mniejsza i bardziej opłacalna.

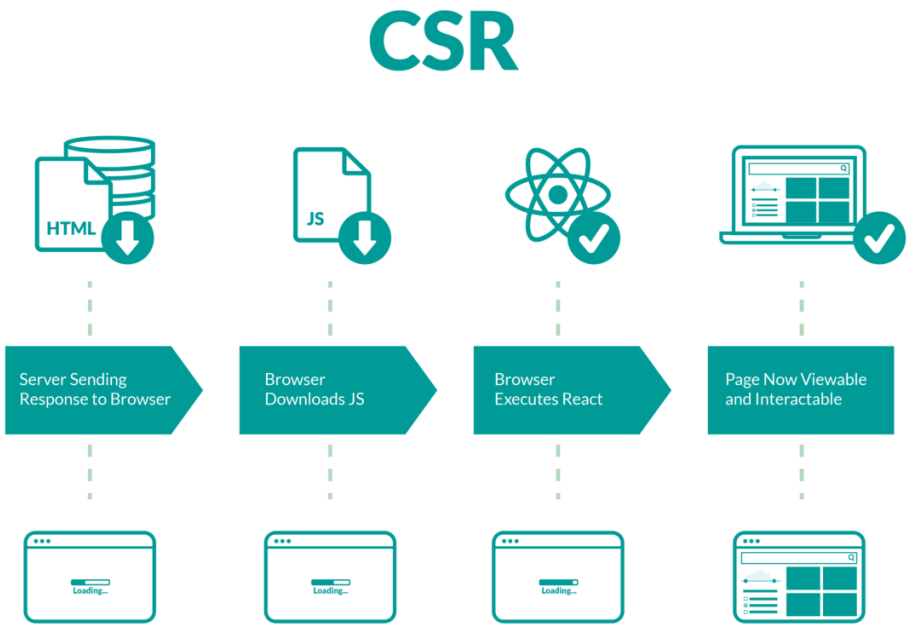
1.2. Client-Side-Rendering

CSR[2] jest całkowitym przeciwieństwem SSR[3]. W tym przypadku przeglądarka ponosi wyłączną odpowiedzialność zbytnio zadania analizowania, renderowania i wyświetlania zawartości strony. CSR stał się nadzwyczaj popularny wraz z pojawieniem się aplikacji jednostronicowych (SPA). Ta system programistyczna w dużej mierze zależy od momentu przeglądarki użytkownika (klienta) do przetwarzania, analizowania i renderowania zawartości przed wyświetleniem jej użytkownikowi.

Działa to w taki sposób, że serwer wysyła do przeglądarki prawie pusty plik HTML, tudzież wraz z zanim kod komputerowy JavaScript potrzebny do działania witryny, ponieważ serwer nie może uruchomić kodu JS. Przeglądarka analizuje i kompiluje ten kod komputerowy, i od tego czasu wypełnia stronę internetową kodem HTML, JS i CSS przy użyciu własnych zasobów, zanim zawartość strony zostanie ostatecznie wyrenderowana użytkownikowi i przed zaimplementowaniem jakichkolwiek działań użytkownika, takich jakże klik-

[1] Single Page Application
[2] Client-Side-Rendering
[3] Server-Side-Rendering

nięcie przycisk, tak aby przesłać formularz. CSR jest potrzebny ze względu na ewolucję sieci.



Rysunek 1.1. Diagram Client-Side-Rendering

Źródło:https://rockcontent.com/blog/client-side-rendering-vs-server-side-rendering/

1.3. React.js

Facebook wymyślił początkowo dla swoich potrzeb React.js w celu tworzenia interfejsów użytkownika oraz reużywalnych komponentów, jednak z czasem stał się pełnoprawną biblioteką JavaScript. Autorem i twórcą jest Jordan Walke. Celem stworzenia owej biblioteki było usprawnienie oraz rozwijanie modułów do portalu społecznościowego jakim był Facebook. Po pewnym czasie React.js zaczeły korzystać także firmy z Doliny Krzemowej. 2013 rok był przełomowy z powodu wydania na cały świat z powodu wydania React.js na 3-klauzolowej licenji BSD. Czas pokazał, że biblioteka się przyjęła wiele pozytywnych opinii na rynku i staję się przodownikiem w świecie Front-endu, co daje wielką popularność i możliwość do rozwoju.

1.3.1. Aspekty architektoniczne w React.js

DOM[4] to istotny atrybut w React.js. Aplikacja zostaje zapisana w pamięci w postaci grafu. Dzięki temu, React.js ma możliwość edycji pojendynczego węzła w DOM zamiast modyfikować cała strukture HTML. Warto wspomnieć, że operacje na DOM są bardzo kosztowne dla pamięci. Unikalną cechą jest wykorzystanie JSX, czyli JavaScript, HTML i XML w jednym miejscu, a dokładniej w komponencie.

1.3.2. Renderowanie elementów

Elementy są najmniejsze bloki, dzięki którym aplikacja Reactowa jest zbudowana. Unikalna funkcjonalnościa tej technologii jest to że, w przeciwieństwie do elementów w drzewie DOM, elementy stworzone przez Reacta są obiektami i koszt tworzenia ich jest niski. React DOM aktualizuje drzewo DOM w taki sposób, aby odpowiadało to strukturze elementów reactowych. Załózmy, że posiadamy w pliku HTML odpowiedni div[5]

Wyciąg 1-1. Korzeń DOM

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

Ten węzeł drzewa DOM jest nazywany "korzeniem". Wszystko co w nim jest zawarte, będzie zarzadzane przez React DOM. Najcześciej tylko jeden taki węzeł sie spotyka w aplikach Reactowych, jednakże jeśli istnieje potrzeba wykorzystania to nie ma żadnych przeciwskazań aby to zrobić. To wyrenderowania tego elementu w korzeniu, musimy użyć ReactDOM.render(). Poniżej zostanie pokazany przykład:

Wyciąg 1-2. Wykorzystanie funkcji ReactDOM.render()

```
1 const element = <h5>Hello World!</h5>;
2 ReactDOM.render(element, document.getElementById('root'));
```

[4] Document Object Model

[5] Div nazywany jest również pudełkiem, albo kontenerem, z tego względu, że zawiera on w sobie inne elementy.

1.3.3. JSX

JSX oznacza JavaScript XML, czyli jest to rozszerzenie składni języka JavaScript. Dzięki temu, React pozwala nam pisac HTML w kodzie JavaScript. Największa zaletą jest optymalizacja podczas tłumaczenia na zwykły JavaScript. Zamiast rozdzielać znaczniki i logikę w oddzielnych plikach, React używa do tego celu komponentów, które zostana opisane w następnym podrozdziale.

Charakterystyczne aspekty JSX:

— JSX wytwarza elementy reagujące
— JSX jest zgodny z reguła XML
— Po kompilacji wyrażenia JSX stają się zwykłymi wywołaniami funkcji JavaScript

1.3.4. Komponenty

Komponenty to elementy składowe każdej aplikacji React. Każda z nich posiada ich bardzo wiele. Istnieją dwa możliwe podejścia do pisania komponentów czyli klasowe i funkcyjne. Z punktu widzenia React jest to tym samym komponentem. Jednakże komponenty klasowe napisane za pomocą standardu ES6 są o wiele szybsze od funkcyjnych ze względu na metody cyklu życia, które zostaną później opisane. Komponent funkcyjny jest to javascriptowa funkcja, która może przyjąć tak samo jak komponent klasowy różne właściwości (ang. props). Zostaną pokazane analogicznie komponenty klasowe i funkcyjne poniżej.

Wyciąg 1-3. Komponent klasowy

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

class Name extends Component {
 render() {
   return (
     <h1>Witaj {this.props.surname} !</h1>
 );
 }
}
```

Wyciąg 1-4. Komponent funkcyjny

```
import React from 'react';

const Name = props => {
```

```
4   <h1>Witaj {props.surname} !</h1>;
5 };
```

Głównym aspektem React.js jest tworzeniu wielu małych komponentów, które posłuża do stworzenia jednego dużego komponentu. A jest to możliwe dzięki właściwości 'children'. Taka właściwość daje możliwość utworzenia komponentu w komponencie jako dziecko danego rodzica. W tym przypadku zostanie pokazany następujący przykład:

Wyciąg 1-5. Komponent potomny

```
1 import React from 'react';
2
3 const ComponentChildren = props => {
4   <div>{props.children}</div>;
5 };
```

Wyciąg 1-6. Komponent rodzica

```
1  import React from 'react';
2
3  const Component = () => {
4    return (
5      <>
6        <Name name="Maciek" />
7        <ComponentChildren>
8          {' '}
9          <h1>Hello</h1>
10       </ComponentChildren>
11     </>
12   );
13 };
```

1.3.5. Props

Propsy to właściwości danego komponentu. Możemy je przekazać jako argumenty funkcji w JavaScript i atrybuty w HTML. Dzięki nim, mamy możliwość przekazywania danych pomiędzy komponentami. Wartość atrybutu zostaje przekazywana komponentu rodzica do komponentu potomka lub używając callbacku w drugą stronę.Co najważniejsze atrybuty są tylko do odczytu, bez

możliwości zmiany stanu. Komponent może posiadać domyślne argumenty komponentu. Wartości początkowe są ustawiane w komponencie potomka dla bezpieczeństwa renderowania strony. I w tym celu został pokazany przykład wykorzystania:

Wyciąg 1-7. Przykład komponentu, w którym wykorzystujemy atrybuty.

```

import React from 'react';

const Component = ({ name }) => {
  <h1>Name {name}</h1>;
};
```

1.3.6. State

Stan(ang. State) jest to miejsce w pamięci w którym, są trzymane dane komponentu. Stan jest dynamiczny, co w wyniku umożliwia aktualizację wartości bez potrzeby odświeżania strony. Komponent posiada stan lokalny i dzięki temu możemy w nim zarządzać stanem ale także wysłać niżej do komponentu potomnego. Aby zaktualizować dany stan trzeba użyć metody setState w przypadku komponentów klasowych. React spostrzeże użycie tej metody co będzie skutkowało aktualizacja stanu,po w dalszej kolejność wyrenderuje dany komponent z nowymi danymi. Nie co inaczej jest w komponentach funkcyjnych ponieważ istnieją specjalne React hooks[6], które odwzorowują metody z komponentów klasowych. Do przetrzymwania stanu w komponentach funkcyjnych używamy hook useState.

Wyciąg 1-8. Przykład wykorzystania state i setState w komponencie klasowym

```
import React from 'react';
class Clicks extends React.Component {
  constructor() {
    super();
    this.state = {
      amountClicks: 0
    };
  }
```

[6] React hooks - React hooks to funkcje, które pozwalają podpiąć się do stanu React i cech cyklu życia z komponentów funkcji.

```
9
10    render() {
11      return (
12        <div>
13        <button onClick={this.remmoveClick.bind(this)}> -
14          </button>
15          <h1>Amount: {this.state.amountClicks}</h1>
16          <button onClick={this.addClick.bind(this)}> +
17          </button>
18        </div>
19      );
20    }
21
22    addClick() {
23      this.setState({
24        amountClicks: this.state.clicks + 1
25      }, () => {
26      });
27    }
28
29    remmoveClick() {
30      this.setState({
31        amountClicks: this.state.clicks - 1
32      }, () => {
33      });
34    }
35  }
```

Wyciąg 1-9. Przykład wykorzystania hooka useState w komponencie funkcyjnym

```
1  import React, { useState } from "react";
2
3  const AmountClicks = () => {
4    const [amountClicks, setamountClicks] = useState(0);
5
6    const handleAddClicks = () => {
7    setamountClicks(prevCount => prevCount + 1);
8    }
9    const handleRemoveClicks = () => {
10   setamountClicks(prevCount => prevCount - 1);
11   }
12
```

```
  return (
    <div>
      <div>
        <button onClick={handleAddClicks}>+</button>
        <h5>Clicks is {clicks}</h5>
        <button onClick={handleRemoveClicks}>-</button>
      </div>
    </div>
  );
}

export default AmountClicks;
```

1.3.7. Destrukturyzacja

Destrukturyzacja to wyrażenie JavaScript, które pozwala nam wyodrębnić dane z tablic, obiektów i map oraz ustawić je w nowych, odrębnych zmiennych. Destrukturyzacja pozwala nam na jednoczesne wyodrębnienie wielu właściwości lub elementów z tablicy.

Wyciąg 1-10. Przykład wykorzystania destruktryzacji

```
const animals = {
  tiger: 'B',
  elephant: 'A'
};

const { tiger, elephant } = animals;

console.log(tiger);     // => 'B',
console.log(elephant; // => 'A'
```

1.3.8. Cykl życia komponentu

Komponent posiada cykl własnego życia. Jako trzy główne fazy życia definiuje się: montowanie, aktualizowanie i odmontowanie komponentu.

Metody, które zostały wykorzystane do montowania komponentu wraz z ich krótkim opisem:

— constructor - Wywołana funkcja przed zamontowaniem komponentu. Przydatny dla inicjalizacji stanu lub bindowania metod komponentu. Warto wspomnieć konstruktor nie jest związany z montowaniem komponentu do DOM.

— render - Polega na tym, że znaczniki komponentu są widoczne w interfejsie użytkownika, czyli jest metodą obowiązkową w każdym komponencie klasowym
— componentDidMount - Metoda wywoływana po zamontowaniu komponentu w drzewie DOM. w tym miejscu, zostają wywołane żądania do uzyskania danych z API,

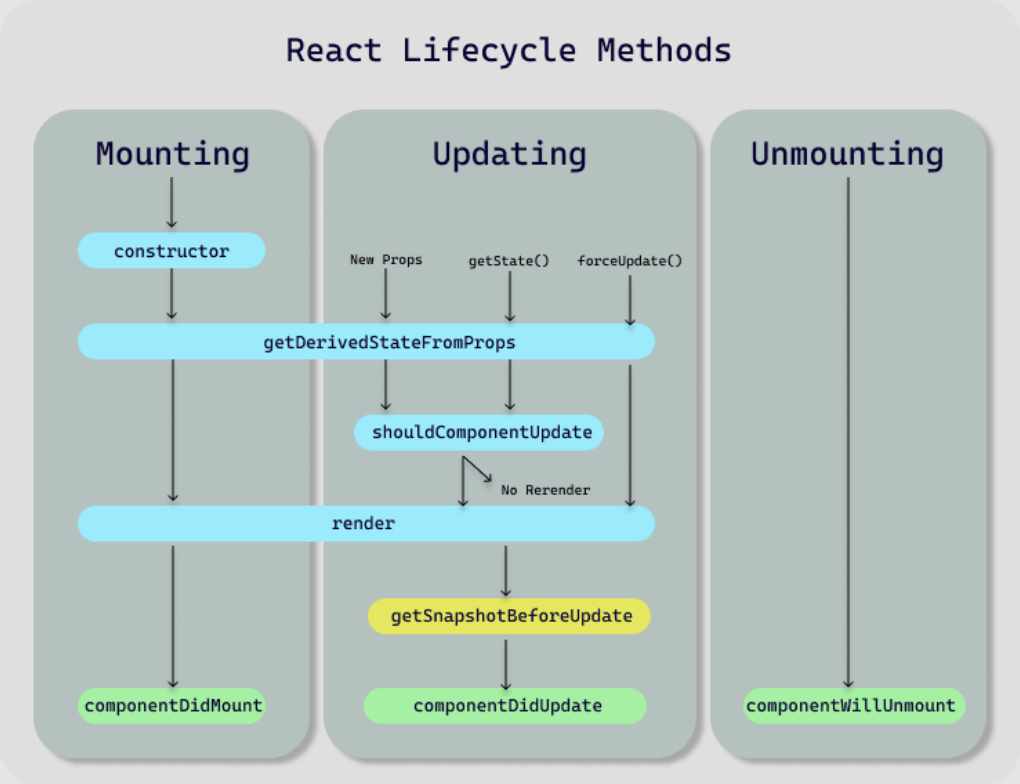
Metody, które wykorzystywane zostały w momencie aktualizacji komponentu oraz ich krótki opis:

— shouldComponentUpdate - Metoda, dzięki której istnieje możliwość sprawdzenia czy stan zostanie zaaktualizowany czy nie. Domyślnym zachowaniem metody jest aktualizowanie stanu za każdym razem, gdy wykryje zmianę stanu. Dzięki temu, bez odświeżania strony, stan będzie miał nowe wartości i interfejs będzie pokazywał aktualny stan.

— getSnapshotBeforeUpdate - Metoda, dzięki której mamy możliwość przechwycenia danych i w jakiś sposób edycji przed ponownym wyrenderowaniem na stronie.

— componentDidUpdate - Metoda, dzięki której zaraz po wyrenderowaniu jest wywołana i zmianie stanu. W komponencie componentDidUpdate można wykonać wiele czynności bez ponownego renderowania komponentu.

Metody, które wykorzystujemy w momencie odmontowania komponentu, oraz ich krótki opis:
— componentWillUnMount - Metoda, dzięki której mamy możliwość przed wymazaniem komponentu z DOM przeprowadzić wszelkie niezbędne porządki po przejściu komponentu.



Rysunek 1.2. Diagram cyklu życia komponentu

Źródło: https://dev.to/aditya278/react-components-lifecycle-methods-wth-are-they-2lh5

1.3.9. Hooki

Konferencja "React Conf 2018" była przełomowa dla frameworka React. Na tej konferencji po raz pierwszy hooki zostały pokazane na świat. Przed zaprezentowaniem hooków, nie było możliwości korzystania z komponentów funkcyjnych w taki sam sposób jak klasowych. Aktualnie nowe aplikacje webowe zostają napisane za pomocą komponentów funkcyjnych z hookami. Hooki nie współpracują z komponentami klasowymi z powodu, że klasowe posiadają własne metody do trzymania stanu, lub aktualizacji stanu itp. Poniżej, zostaną zaprezentowane hooki, które są wykorzystywane najczęściej:

— useState - hook, dzięki któremu komponent funkcyjny może posiadać stan lokalny

Wyciąg 1-11. Wykorzystanie hooka useState

```
1 import React, { useState } from 'react';
```

```

const Click = () => {
  const [click, setClick] = useState(0);
  return (
    <div>
      <p>Amount click {click} </p>
      <button onClick={() => setClick(click + 1)}>
        Click me
      </button>
    </div>
  );
}
```

— useEffect - hook, dzięki któremu możemy zaaktualizować stan danymi np. z API

Wyciąg 1-12. Wykorzystanie hooka useEffect

```
const CountRendering = ({ amount }) => {
    useEffect(() => {
        console.log(‘Amount render - ${amount}‘);
        return () => {
            console.log(‘Unmount amount - ${amount}‘);
        };
    });

    return <p>{count}</p>;
};
```

— useMemo - hook, dzięki któremu istnieje możliwość zwrócenia zapamiętanej wartości.

Wyciąg 1-13. Wykorzystanie hooka useMemo

```
const updateValue = useMemo(() => {
    updateValues(amount, click), [amount, click]);
}
```

— useRef -hook, dzięki któremu istnieje możliwość zwrócenie mutowalnego obiektu, którego właściwość ’.current’ jest wartość przekazaną jako argument.

Wyciąg 1-14. Wykorzystanie hooka useRef

```

```

```
const ButtonWithRef = () => {
   const mountRef = React.useRef()


   const handleClick = () => {
      mountRef.current.focus()
   }

   return (
      <>
         <input ref={mountRef} type="text" />
         <button onClick={handleClick}></button>
      </>
   )
}
```

Po przedstawieniu kilku hooków trzeba przedstawić zalety hooków:
— Poprawa czytelności drzewa komponentów
— Hermetyzacja skutków ubocznych (ang. Encapsulating Side Effects)
— Logika wielokrotnego użytku

1.4. Redux

React.js posiada możliwości przesyłania danych między komponentami jednakże czasem jest to niemożliwe. Potrzebny nam jest state globalny a w takim przypadku przychodzi nam z pomocą Redux. Idealnie się sprawdza do zarządzania stanem globalnym aplikacji w React.js. Pamietać należy, że przy prostych aplikach i mało rozbudowanych Redux nie jest potrzebny. Najważniejsze zasady Reduxa:

— Pojedyncze źródło prawdy - stan całej aplikacji przetrzymywany jest w drzewie obiektów wewnątrz pojedynczego obiektu store.

— Stan jest tylko do odczytu - jedynym sposobem na zmianę stanu jest wywołanie akcji, która zwraca obiekt opisujący co powinno się stać.

— Zmiany wykonywane są czystymi funkcjami - aby określić jak drzewo stanu transformowane jest przez akcje należy tworzyć “czyste reducery”.

Redux nie jest stricte połączony z React, można z niej korzystać z innych frameworkami. Redux stworzył biblioteke dla Reacta o nazwie react-redux. Własnie ona nam dostarcza możliwości, dzięki którym możemy ich użyc w React. Aby stworzyć instancje singletonu, czyli pierwszy store należy wywołać createStore, dzięki której będziemy mieć możliwość zarządzaniem danymi. Aby zaimplementować w aplikacji potrzebujemy następujących metod:

— 'createStore' - dzięki tej metodzie zostanie stworzenie drzewo stanu, które przetrzymuje stan globalny w naszej aplikacji. Aplikacja może posiada tylko jedno drzewo, ale może posiadać kilka reducerów. Aby zamknąć wszystkie reducery w drzewie przedstawimy metodę combineReducers:

Wyciąg 1-15. Wykorzystanie łacznika combineReducers

```
1    const rootReducer = combineReducers({
2   LinkReducer,
3   TransitionReducer,
4   ShoppingReducer,
5   WorkflowReducer
6 });
```

— 'Provider' - Dzięki, temu możemy zamknać cała naszą aplikacje, który udostępnia stan globalny dla całej aplikacji.

Wyciąg 1-16. Wykorzystanie komponentu Provider

```
    <Provider store={store}>
        <App />
    </Provider>
```

Nazwa akcji zostaje zdefiniowana w pliku actionTypes.js lub actionTypes.ts jednakże zależy to od konwecji programistów i czy jest to JavaScript lub TypeScript.

Wyciąg 1-17. Definiowanie nazwy akcji

```
export const FILTER_SEARCH_API_REQUEST =
'FILTER_SEARCH_API_REQUEST';
```

Po zdefiniowaniu nazwy akcji należy zaimplementować, co dokładnie ma robić akcja.

Wyciąg 1-18. Implementacja wysyłanie danych

```
import * as actionTypes from './actionTypes';


export const FilterSearchPayload = {
search: string;
amount: number;
}
export const filterSearchApiRequest = filterSearch => {
  return {
    type: actionTypes.FILTER_SEARCH_API_REQUEST,
    filterSearch
  };
};

export const filterSearchApiSuccess = () => {
return {
    type: actionTypes.FILTER_SEARCH_SUCCESS
    };
}
export const filterSearchApiRequest =
(filterSearch: FilterSearchPayload) => {
  return dispatch => {
```

```
    axios
      .post(‘/filterSearch‘, filterSearch)
      .then(response => {
        dispatch(filterSearchApiSuccess);
      })
      .catch(() => {
        console.log(’error’);
      });
  };
};
```

Jeśli akcja przejdzie pozytywnie i serwer zwróci status 200, w reducerze zostaną zaaktualizowane stany:

Wyciąg 1-19. Wykorzystanie reducera

```
import * as actionTypes from ’../actions/actionTypes’;

const initialState = {
  isReady: false
};


const reducer = (state = initialState, action) => {
  switch (action.type) {
  case actionTypes.FILTER_SEARCH_API_REQUEST:
      return {...state, isReady: fals}
    case actionTypes.FILTER_SEARCH_SUCCESS:
      return {...state, isReady: true}
    default:
      return state;
  }
};

export default reducer;
```

Aby wywołać akcję w komponencie klasowym potrzebujemy metody dispatch, która dostarcza nam react-redux.

Wyciąg 1-20. Wykorzystanie metody dispatch

```
dispatch(filterSearchApiRequest({ search, amount }))
```

UseSelector przychodzi nam z pomocą jak potrzebujemy wyciagnąć w dowol-

nym miejscu stan globalny. Selektor będzie za każdym razem pobierał stan do którego się odnosi, gdy komponent funkcji będzie renderowany.

Wyciąg 1-21. Przykład wykorzystania useSelector

```
const isLoading = useSelector(state => state.utils.isLoading);
```

1.4.1. Redux Saga

Istnieje specjalny middleware, dzięki któremu praca z API staję sie dużo łatwiejsza. A w tym przypadku mowa Redux-Saga. Ten middleware korzysta z generatorów z ES6. Dzięki temu ułożenie poprawnego wykonywanie akcji staję się proste.

Wyciąg 1-22. Wykorzystanie Redux-Saga

```
import { call, put, takeEvery } from 'redux-saga/effects';
import axios from '../../utils/axiosGet';
import { MENU_ROUTES } from '../../constants/routes/Routes';
import { saveConfig } from '../../store/config/action';
import { errorActions } from '../../store/ActionApps/action';

function* configRequest({ payload: { id } }) {
  try {
    const configUrl = MENU_ROUTES.CONFIG + id;
    const response = yield call(axios.get, configUrl);
    const config = response.data;
    yield put(saveConfigAsync.success({ config }));
  } catch (error) {
    yield put(saveConfigAsync.failure({ error }));
    const errorData = error.response.data;
    yield put(showModalError({ error: errorData }));
  }
}

export default function* configRequestSaga() {
  yield takeEvery(saveConfigAsync.request, configRequest);
}
```

1.5. Testy

Testowanie to przeglądanie wiersz po wierszu sposobu wykonania kodu. Zestaw testów dla aplikacji składa się z różnych fragmentów kodu w celu sprawdzenia, czy aplikacja działa pomyślnie i bez błędów. Testowanie przydaje się również w przypadku aktualizacji kodu. Po zaktualizowaniu fragmentu kodu możesz uruchomić test, aby upewnić się, że aktualizacja nie zepsuje funkcjonalności już w aplikacji.

1.5.1. Dlaczego warto testowac?

Testowanie jest to istotna część pisania komercyjnego kodu. Takie podejście ma swoje zalety takie jak :

— Pierwszym celem testowania jest zapobieganie regresji. Regresja to ponowne pojawienie się błędu, który został wcześniej naprawiony.

— Testowanie zapewnia funkcjonalność złożonych komponentów i aplikacji modułowych.

— Testowanie jest wymagane do efektywnego działania aplikacji lub produktu.

1.5.2. Typy testów

Testy jednostkowe - W tego typu teście testowane są poszczególne jednostki lub komponenty oprogramowania. Jednostką może być pojedyncza funkcja, metoda, procedura, moduł lub obiekt. Test jednostkowy izoluje fragment kodu i weryfikuje jego poprawność w celu sprawdzenia, czy każda jednostka kodu oprogramowania działa zgodnie z oczekiwaniami.

W testach jednostkowych poszczególne procedury lub funkcje są testowane, aby zagwarantować ich prawidłowe działanie, a wszystkie komponenty są testowane indywidualnie. Na przykład testowanie funkcji lub sprawdzenie, czy instrukcja lub pętla w programie działa poprawnie, wchodzi w zakres testów jednostkowych.

Testy komponentów (ang. Component Test) - Testy modułowe weryfikują funkcjonalność poszczególnych części aplikacji. Testy są przeprowadzane na każdym komponencie w oderwaniu od innych komponentów. Ogólnie rzecz

biorąc, aplikacje React składają się z kilku komponentów, więc testowanie komponentów zajmuje się testowaniem tych komponentów indywidualnie.

Wyciąg 1-23. Przykład testu jednostkowego

```
import { deleteApps, getApps } from './customFunction';

describe('test', () => {
  describe('returnCheckedAppToDelete', () => {
    it('should return empty array', () => {
      const mockedApplication = [
        { name: 'Balancer', applications: [{ name: '-api',
        checked: false }] },
      ];
      const mockedResult = [];
      expect(deleteApps(mockedApplication))
      .toEqual(mockedResult);
    });

    it('should return one element of array', () => {
      const mockedApplication = [
        { name: '', id: 3, applications: [{ name: '-api',
        checked: true }] },
      ];
      const mockedResult = [{ name: '-api',
      checked: true, id: 3 }];
      expect(deleteApps(mockedApplication))
      .toEqual(mockedResult);
    });

    it('should return -api', () => {
      const mockedApplication = [
        {
          name: '',
          id: 3,
          applications: [
            { name: '-api', checked: true },
            { name: '-web', checked: false },
          ],
        },
      ];
      const mockedResult = [{ name: '-api',
      checked: true, id: 3 }];
      expect(deleteApps(mockedApplication))
```

```
40        .toEqual(mockedResult);
41      });
42    });
43  });
44
45  describe('test', () => {
46    describe('getApps', () => {
47      it('should array of two elements', () => {
48        const mockedApplication = [
49          {
50            name: '',
51            modules: [
52              { name: '-api', checked: true },
53              { name: '-web', checked: true },
54            ],
55          },
56        ];
57        const mockedResult = [
58          { name: '-api', id: 2, checked: true },
59          { name: '-web', id: 2, checked: true },
60        ];
61        expect(getApps(mockedApplication, 2))
62        .toEqual(mockedResult);
63      });
64      it('should array one element', () => {
65        const mockedApplication = [
66          {
67            name: '',
68            modules: [
69              { name: '-api', checked: false },
70              { name: '-web', checked: true },
71            ],
72          },
73        ];
74        const mockedResult = [{ name: '-web', id: 2,
75        checked: true }];
76        expect(getApps(mockedApplication, 2))
77        .toEqual(mockedResult);
78      });
79      it('should array zero element', () => {
80        const mockedApplication = [
81          {
82            name: '',
83            modules: [
```

```
            { name: '-api', checked: false },
            { name: '-web', checked: false },
          ],
        },
      ];
      const mockedResult = [];
      expect(getApps(mockedApplication, 2))
      .toEqual(mockedResult);
    });
  });
});
```

Testy migawki (ang. Snapshot Test) - Test migawki zapewnia, że interfejs użytkownika (UI) aplikacji sieci Web nie zmienia się nieoczekiwanie. Przechwytuje kod komponentu w danym momencie, dzięki czemu istnieje możliwość porównać komponent w jednym stanie z dowolnym innym możliwym stanem, jaki może przyjąć.

Wyciąg 1-24. Przykład testu migawki

```
import React from 'react';
import { render, act } from '@testing-library/react';
import DetailsPopup from './index';

const axiosResponseMock = {
   data: {
      Id: 123,
      Type: '',
      Name: '',
      fileName: null,
      language: 'pl_PL',
      participants: '',
      priority: '',
      projectName: null,
      reference: '',
      segmentText: null,
      unitId: null
   }
};
jest.mock('axios', () => ({
   defaults: {
      headers: {
         common: ''
      },
```

```
25    },
26    create: () => ({
27       get: () => Promise.resolve(axiosResponseMock)
28    })
29 }));
30
31 jest.mock('react-intl', () => ({
32    useIntl: () => {
33       const locale = require('localization.json');
34       return {
35          formatMessage: ({ id }) => locale[id]
36       };
37    }
38 }));
39
40 describe('DetailsPopup test', () => {
41    const closePopupHandlerMock = () => {};
42    let component;
43
44    beforeEach(async () => {
45       await act(async () => {
46          component = render(
47             <DetailsPopup
48             isOpen={true}
49             selectedChatId="123"
50             closePopupHandler={closePopupHandlerMock} />
51          );
52       });
53    });
54
55    afterEach(() => {
56       component.unmount();
57       component = null;
58    });
59
60    it('It should match snapshot.', async () => {
61       expect(component.asFragment()).toMatchSnapshot();
62    });
63
64    it('should render proper dd values.', () => {
65       expect(component.container.
66       querySelectorAll('dl dd')[0].innerHTML).toEqual('123');
67       expect(component.container.
68       querySelectorAll('dl dd')[1].innerHTML).toEqual('-');
```

```
69    });
70 });
```

1.5.3. Testowanie aplikacji w React

Zależnie od wersji zaleca się różne narzędzia do testowania aplikacji webowych. Dla Reacta w wersji 16 lub niższych jest polecana biblioteka Enzyme. Natomiast przy wersji jest sugerowana biblioteka React Testing Library. Dzięki niej istnieje możliwość pisania testów, które testują komponenty w taki sam sposób jak użytkownik korzysta z aplikacji. Dodatkowo istnieje narzędzie takie jak ReactTestUtils. Pozwala na testowanie komponentów reactowych. Poniżej zostaną pokazane najważniejsze metody z jakich należy korzystać podczas testowania komponentów w raz przykładami.

— act() - W celu przygotowania komponentu do testowania, należy cały kod komponentu otoczyć wywołaniem funkcji act(). W ten sposób zostanie odtworzone w najlepszy sposób zachowanie Reacta w przeglądarce.

Wyciąg 1-25. Komponent Licznik

```
1  class Licznik extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = {amount: 0};
5      this.handleClick = this.handleClick.bind(this);
6    }
7    componentDidMount() {
8      document.title = ‘KliknińŹto ${this.state.count} razy‘;
9    }
10   handleClick() {
11     this.setState(state => ({
12       count: state.count + 1,
13     }));
14   }
15   render() {
16     return (
17       <div>
18         <p>KliknińŹto {this.state.amount} razy</p>
19         <button onClick={this.handleClick}>
20           Kliknij mnie
21         </button>
22       </div>
```

```
    );
  }
}
```

Wyciąg 1-26. Test komponentu Licznik

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Licznik from './Licznik';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('Aktualizowanie licznika', () => {
  act(() => {
    ReactDOM.createRoot(container).render(<Licznik />);
  });
  const button = container.querySelector('button');
  const p = container.querySelector('p');
  expect(p.textContent).toBe('Kliknieto 0 razy');
  expect(document.title).toBe('Kliknieto 0 razy');
```

— mockComponent() - Dzięki tej funkcji, mamy możliwośc przekazania atrapy komponentu.

Wyciąg 1-27. Atrapa komponentu

```
mockComponent(
  component,
  [mock]
)
```

— isElement() - zwraca wartość true, jeśli element jest w drzewie Reactowym.

Wyciąg 1-28. Test elementu

```
isElement(element)
```

— scryRenderedDOMComponentsWithClass() - Dzięki tej funkcji, mamy możliwość wyszukiwania wszystkich elementów CSS w drzewie DOM

Wyciąg 1-29. Wykorzystanie funkcji scryRenderedDOMComponentsWithClass

```
findRenderedDOMComponentWithClass(
  tree,
  className
)
```

— renderIntoDocument() - Dzięki tej funkcji, mamy możliwość wyrenderowania elementu w węźle drzewie DOM. Najważniejszym aspektem tej funkcji jest, że działa tylko w ramach dokumentu.

Wyciąg 1-30. Wykorzystanie funkcji renderIntoDocument

```
renderIntoDocument(element)
```

— Simulate - Dzięki temu obiektowi, mamy możliwość przesłania zdarzenia do węźła DOM, takich jak kliknięcia etc. Simulate posiada odpowiednie metody dla każdego ze zdarzeń.

Wyciąg 1-31. Wykorzystanie obiektu Simulate

```

const popover = this.popover;
ReactTestUtils.Simulate.click(popover);
```

Rozdział 2

Historia i podstawy Angular'a

2.1. Historia

Angular jest rozwijany cały czas. Angular posiada aktualnie sześć wersji, takich jak :
— Angular 1.0 - Wersja została wydana w 2010 roku.
— Angular 2.0 - Wersja została wydana w 2014 roku.
— Angular 4.0 - Wersja została wydana w 2016 roku.
— Angular 5.0 - Wersja została wydana w 2017 roku.
— Angular 6.0 - Wersja została wydana w 2018 roku.
— Angular 7.0 - Wersja została wydana w 2018 roku.

Angular 7 wprowadził wiele nowych funkcjonalności takich jak :
— Aktualizacje dotyczące wydajności aplikacji
— Angular Material CDK
— Virtual Scrolling
— Została poprawiona dostępność selekcji
— Obsługuje projekcję treści przy użyciu standardu internetowego dla elementów niestandardowych
— Aktualizacje zależnosci dotyczących TypeScript etc.

Do zrozumienia frameworka Angular, należy na początku się nauczyć co to jest komponent, szablon, dyrektywy oraz wstrzykiwanie zależności (ang. Dependency injection)

2.2. Komponenty

Komponenty jest to podstawowy element podczas tworzenia interfejsu aplikacji Angulara. Aplikacja jest stworzona z drzewa komponentów. Komponenty angularowe są zbiorem dyrektyw. W przeciwieństwie do dyrektyw, tylko jeden składnik może tworzyć dany element w szablonie.

Wyciąg 2-1. Przykład komponentu w Angularze

```
export class ListComponent implements OnInit {
  lists: list[] = [];
  selectedList: selectedList | undefined;

  constructor(private service: ListService) { }

  ngOnInit() {
    this.lists = this.service.getLists();
  }

  selectList(list: List) { this.selectedList = list; }
}
```

W Angularze istnieje możliwość deklaracji styli, z jakich komponent ma wybrać style i zaciągnąć do środka. Istnieje także możliwość bez importowania pliku tylko bezpośrednio w tablicy podać atrybuty, jakie mają być wykorzystane.

Wyciąg 2-2. Przykład deklaracji styli bezpośrednio z pliku style.css w Angularze

```
@Component({
  selector: 'overview',
  templateUrl: './overview.component.html',
  styleUrls: ['./style.component.css']
})
```

Wyciąg 2-3. Przykład deklaracji styli bezpośrednio jako tablica w Angularze

```
@Component({
  selector: 'overview',
  templateUrl: './overview.component.html',
  styleUrls: ['h1 { background-color: #fff; }']
})
```

2.2.1. Cykl życia haków (ang. Lifecycle hooks)

Po uruchomieniu aplikacja w Angularze tworzy i renderuje składnik główny. Następnie generuje swoje rodziców i ich dzieci, po czym tworzy drzewo składników. Po załadowaniu komponentów Angular rozpoczyna proces renderowania widoku. Aby to zrobić, musi sprawdzić właściwości wejściowe, powiązania danych i wyrażenia, wyrenderować przewidywaną zawartość itp. Angular usuwa również komponent z DOM, gdy już go nie potrzebuje. Angular informuje nas, kiedy te zdarzenia mają miejsce, używając hooków cyklu życia. Hooki cyklu życia Angulara to nic innego jak funkcja zwrotna, którą Angular wywołuje, gdy podczas cyklu życia komponentu wystąpi określone zdarzenie. Poniżej zostaną wyjaśnione niektóre hooki.

— ngOnChanges() - Hook, dzięki któremu Angular ustawia lub resetuje właściwości wejściowe. Metoda odbiera aktualne i poprzednie wartości właściwości.

— ngOnInit() - Hook, dzięki któremu Angular zainicjalizuje dyrektywę lub składnik po czym, ustawi właściwosci po czym ustawi wejściowe rzeczy.

— ngDoCheck() - Hook, dzięki któremu wykrywa i działa na podstawie zmian, których Angular nie możę lub nie wykryje samodzielnie.

— ngAfterContentInit() - Hook, dzięki któremu wyświetla zawartość zewnętrzną w widoku składnika lub w widoku, w którym znajduje się dyrektywa.

— ngAfterContentChecked() - Hook, dzięki któremu Angular potrafi sprawdzić zawartość wyświetlanej w dyrektywie lub składniku

— ngAfterViewInit() - Hook, dzięki któremu Angular zainicjuje widoki składnika i widoki podrzędne lub widok zawierający dyrektywę.

— ngAfterViewChecked() - Hook, dzięki któremu Angular sprawdzi widoki składnika i widoki podrzędne lub widok zawierający dyrektywę.

— ngOnDestroy() - Hook, dzięki któremu Angular czyści dyrektywę lub komponent.

2.3. Szablony (eng. Template)

W Angularze istnieje możliwość stworzenia szablonu, czyli fragment kodu HTML. Dzięki specjalnej składni w szablonie, jest szansa wykorzystać wiele funkcji Angulara. Każdy szablon Angulara jest sekcją HTML, która powinno się uwzględnić jako część strony wyświetlanej w przeglądarce. Taka struktura renderuje widok lub cały interfejs użytkownika, jednakże pozwala o wiele większą użyteczności. Dodatkowo używając Angular CLI szablon jest tworzony automatycznie.

Wyciąg 2-4. Przykład szablonu w Angularze

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app',
  template: `
            <p>
            hello world
            </p>
            `,
  styleUrls: ['./a.component.css']
})

export class Component implements OnInit {

  constructor() { }

  ngOnInit() { }
}
```

2.4. Dyrektywy (eng. Directives)

Dyrektywy to klasy, dzięki któremu dodaja specjalne zachowanie do elementów w aplikacji Angular. Istnieją trzy rodzaje dyrektyw:

— Dyrektywy składowe

— Dyrektywy strukturalne

— Dyrektywy atrybutów

2.4.1. Dyrektywy składowe

Definiują właściwości, tworzą instancję oraz sposób wykorzystania danego komponentu w aplikacji webowej.

Wyciąg 2-5. Przykład dyrektywy składowej w Angularze

```
import { Component } from '@angular/core';

@Component({
  selector: 'root',
  template: ''./app.component.html‘,
  styleUrls: ['./app.component.css']
})

export class Component {
button = 'button';
}
```

2.4.2. Dyrektywa strukturalna

Dyrektywy, które zmieniają strukturę DOM poprzez usuwanie lub dodawanie elementów. Występują trzy dyrektywy strukturalne wbudowane w Angularze.

— NgIf

Wyciąg 2-6. Przykład dyrektywy NgIf

```
<p class="text"
   *ngIf="!data.hide">
  {{ data.button }}
</p>
```

— NgFor

Wyciąg 2-7. Przykład dyrektywy NgFor

```
<smart *ngFor="let i of smartes"
      [smart]="i">
</smart>
```

— NgSwitch

Wyciąg 2-8. Przykład dyrektywy NgSwitch

```
<ng-template> [ngSwitch]="switch_expression">
   <p *ngSwitchCase="expression">Content</p>
</ng-template>>
```

Te dyrektywy korzystają tagów HTML5 <ng-template>. HTML5 dodał nowy tag, który został specjalnie zaprojektowany do przechowywania kodu szablonu.

2.4.3. Dyrektywy atrybutów

Dyrektywy atrybutów są używane jako atrybuty elementów. Na przykład wbudowana dyrektywa NgStyle w przewodniku po składni szablonu może zmieniać kilka stylów elementów jednocześnie.

Wyciąg 2-9. Przykład dyrektywy atrybutów

```
import { Directive, ElementRef, Input } from '@angular/core';
@Directive({ selector: '[myHighlight]' })
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'red';
    }
}
```

2.5. Wstrzykiwanie zależności (eng. Dependency Injection)

Wstrzykiwanie zależności to wzorzec projektowy, w którym żąda zależności z innych miejsc, zamiast stworzyć je ponownie. Istnieje framework Angular DI, który dostarcza zależność od razu po stworzeniu klasy instancji.

Wyciąg 2-10. Przykład tworzenia serwisu ClassService za pomocą Angular CLI

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class ClassService {
  constructor() { }
}
```

Dzięki dekoratorowi @Injectable() Angular posiada możliwość użycia klasy w systemie DI. ProvidedIn przypisany root, oznacza, że ClassService jest widoczny w całej aplikacji. Następnie istnieję możliwość metody do pobierania klas, która je zwróci.

Wyciąg 2-11. Przykład tworzenia metody getClass

```
import { Injectable } from '@angular/core';

export const CLASS = [{id:1 class: 1},
{id: 2 class: 2}]
@Injectable({

  providedIn: 'root',
})
export class ClassService {
  getClass() { return CLASS; }
}
```

Aby wstrzyknąć zależność należy w konstruktorze dodać argument z typem zależności. Dzięki temu, komponent ma dostęp do zależności.

Wyciąg 2-12. Przykład wstrzykiwania zależności

```
constructor(classService: ClassService)
```

2.5.1. Angular CLI

Angular CLI jest to terminalowy interfejs z specjalnymi komendami dla Angulara. Bardzo użyteczne narzędzie, jeśli patrząć na pryzmat programowania, inicjalizowania aplikacji itp. Poniżej zostanie wymienione najczęściej używane komendy.

— ng add

— ng build

— ng deploy

— ng generate

— ng new

— ng lint

— ng update

— ng serve

— ng run

— ng test

Rozdział 3

Vue.js

Framework JavaScript, dzięki któremu można budować interfejs użytkownika. Podpierając się HTML, CSS oraz JavaScriptem zapewnia możliwość budowania komponentów, które są wydajne i stosunkowo proste do tworzenia, jeśli patrząc na stronę programistyczną.

Wyciąg 3-1. Przykład aplikacji Vue.js

```
import { createApp } from 'vue'

createApp({
  data() {
    return {
      value: 0
    }
  }
}).mount('#app')
```

Wyciąg 3-2. Przykład wstrzyknięcia aplikacji Vue.js w miejsca id

```
<div id="app">
  <button @click="value++">
    Value is: {{ value }}
  </button>
</div>
```

Najważniejszymi cechami Vue jest:

— Renderowanie deklaratywne (eng. Declarative Rendering) - Vue powiększa standardowy HTML o składnie szablonu
— Reaktywność (eng.Reactivity) - Vue posiada mechanizm, który śledzi sam

zmiany stanu JavaScript i dzięki temu aktualizuje DOM, ale tylko wtedy gdy zachodzi taka potrzeba.

3.1. Dyrektywy

Dyrektywy jest to atrybut, dzięki któremu dodajemy do znaczników HTML, z tym, że nie są one częścią standardu HTML, a są intepretowane bezpośrednio przez framework Vue.js. Często się je określa jako pseudo-funkcje, które zmieniają zachowanie i strukturę DOM. Każda dyrektywa rozpoczyna się od prefiksu v-, po którym dodajemy nazwę dyrektywy. Istnieje możliwość tworzenia oprócz standardowych dyrektw, także autorskie.

3.1.1. Dyrektywa v-model

Dyrektywa v-model jest wykorzystywana do powiązania wartości danego tagu HTML z polem danych.

Wyciąg 3-3. Przykład dyrektywy v-model w Vue.js

```
<div id="app">
  <input v-model="title" />
  <p>{{ title }}</p>
</div>
```

3.1.2. Dyrektywa v-bind

Niektóre dyrektywy przyjmują po dwukropku parametr, dzięki czemu dany element HTML staję kontrolowany przez Vue.js.

Wyciąg 3-4. Przykład dyrektywy v-bind w Vue.js

```
<div
  class="following"
  v-bind:class="{ active: isActive }"
>
</div>
```

3.1.3. Dyrektywa warunkowa v-if

Najczęściej wykorzystywana w celu postawienia warunku na danym elemencie jest dyrektywa v-if. Dyrektywa umożliwa na podstawie warunku, potrafi np. dodać element do DOM lub go usunąć.

Wyciąg 3-5. Przykład dyrektywy v-if w Vue.js

```
<div id="app">
  <p v-if="text === ''">Write something...</p>
</div>
```

3.1.4. Dyrektywa v-else

Jeśli istnieje sytuacja, kiedy potrzeba wyrenderować element od razu po dyrektywie v-if, wtedy idealnie pasuje dyrektywa v-else.

Wyciąg 3-6. Przykład dyrektywy v-else w Vue.js

```
<div id="app">
  <p v-if="text === ''">Write something...</p>
  <p v-else>Thanks!</p>
</div>
```

3.1.5. Dyrektywa v-show

Podczas projektowania danego interfejsu użytkownika, czasem programista może napotkać sytuacje, w której nie chce usuwać ani pokazywać elementu. W takim celu została stworzona dyrektywa v-show. Najważniejszą cechą jest, że element nie zostaje usunięty lecz nadaję sie temu elementowi styl CSS "display:none".

Wyciąg 3-7. Przykład dyrektywy v-show w Vue.js

```
<div id="app">
  <p v-show="value">Value</p>
</div>
```

3.1.6. Dyrektywa v-for

Ta dyrektywa używa specjalnej składni element in tablicaDanych, gdzie tablicaDanymi jest tablica z danymi a element zmienną lokalną.

Wyciąg 3-8. Przykład dyrektywy v-for w Vue.js

```
<ul>
  <li v-for="element in elements">{{ element }}</li>
</ul>
```

3.2. Atrybuty class i style w Vue.js

Podobnie jak W React.js możemy do komponentu przekazać class, w której się zawierają style dla danego fragmentu kodu, ale dodatkowo istnieje możliwość wstrzyknięcia styli bezpośrednio do danego tagu HTML.

Wyciąg 3-9. Przykład wykorzystania class w Vue.js

```
.correct {
  color: green;
}

new Vue({
  data() {
    return {
      isCorrect: true,
    };
  },
}).$mount('#app');

<div id="app">
  <label :className="{ correct: isCorrect }">
    <input type="checkbox" v-model="isCorrect" />
    Click me!
  </label>
</div>
```

Wyciąg 3-10. Przykład wykorzystania style w Vue.js

```
<div id="app">
  <label :style={{"{'backgroundColor'":"black}"}}>
    Choose a color:
    <select v-model="selectedColor">
      <option disabled>Choose...</option>
      <option>red</option>
      <option>blue</option>
      <option>pink</option>
      <option>grey</option>
    </select>
  </label>
  <label :style={{"{textAlign":"center ? 'center' : 'none' }"}}>
    <input type="checkbox" v-model="center" /> Center?
  </label>
</div>
```

3.3. Metody komponentów

Framework Vue.js pozwala na definiowanie metod, które mogą być wykorzystane potem w szablonie, ale także w innych metodach. Aby taką metodę stworzyć należy zdefiniować w komponencie obiekt "methods". W obiekcie data należy określic pola tej metody, aby potem móc się w samej metodzie do nich odnieść.

Wyciąg 3-11. Tworzenie metody w Vue.js

```
new Vue({
  data() {
    return {
      name: 'Maciej',
      surname: 'Marczak',
    };
  },
  methods: {
    getFullName() {
            return this.name + ' ' + this.surname;
    },
  },
}).$mount('#app');
```

Wyciąg 3-12. Wykorzystanie metody getFullName w Vue.js

```
<div id="app">
  <input type="text" v-model="name" />
  <p>Imie i nazwisko: {{ getFullName() }}</p>
</div>
```

3.3.1. Pola wyliczone (eng. Computed properties)

W Vue.js umożliwia obliczanie pól, tworzenie pól, których można używać do modyfikowania, manipulowania i wyświetlania danych w komponentach w czytelny i wydajny sposób. Obliczone pola można używać do obliczania i wyświetlania wartości na podstawie wartości lub zestawu wartości w modelu danych. Może również mieć pewną niestandardową logikę, która jest buforowana na podstawie jego zależności, co oznacza, że nie ładuje się ponownie, ale ma zależność, która się zmienia, pozwalając mu nieco słuchać zmian i odpowiednio działać.

Wyciąg 3-13. Tworzenie computed properties w Vue.js

```
new Vue({
 data() {
    return {
      Value: 2
    }
  },
  computed: {
    valueCount: () => {
      return 'Value' + this.Value
    }
  }
}).$mount('#app');
```

Wyciąg 3-14. Wykorzystanie computed properties w Vue.js

```
<div id="app">
  <input type="text" v-model="Value" />
  <p>Value: {{ valueCount }}</p>
</div>
```

3.4. Dyrektywa v-on, obsługa zdarzeń

Framework Vue.js posiada specjalną dyrektywę 'v-on', dzięki której można nasłuchiwać zdarzeń DOM. Wartość dyrektywy 'v-on' jest albo metodą dostępna w komponencie lub wyrażeniem JavaScript.

Wyciąg 3-15. Tworzenie metod dla dyrektywy v-on w Vue.js

```
new Vue({
  data() {
    return {
      value: 0,
    };
  },
  methods: {
    increment() {
      this.value = this.value + 1;
    },
    decrement() {
      this.value = this.value - 1;
    },
  },
}).$mount('#app');
```

Wyciąg 3-16. Wykorzystanie dyrektywy v-on w Vue.js

```
<div id="app">
  <p>Value: {{ Value }}</p>
  <button v-on:click="increment">+</button>
    <button @click="Value -= 1">-</button>
</div>
```

3.5. Vue CLI

Podobnie jak Angular, Vue.js także posiada terminalowy interfejs z specjalnymi komendami. Jedynie póki co React nie posiada żadnego CLI. Jest to bardzo użyteczne narzędzie, które bardzo przyspiesza pracę programistów.

Wyciąg 3-17. Tworzenie projektu w Vue.js za pomocą Vue CLI

```
vue create hello-world
```

Po tej komendzie, terminal zacznie wypytywać programistę o np. czy projekt podstawowy ma posiadać wsparcie dla TypeScripta, czy ma posiadać testy jednostkowe i E2E oraz wiele więcej.

Wyciąg 3-18. Opcje vue create z flagą –help

```
Usage: create [options] <app-name>

create a new project powered by vue-cli-service


Options:

  -p, --preset <presetName>
  Skip prompts and use saved or remote preset
  -d, --default
  Skip prompts and use default preset
  -i, --inlinePreset <json>
  Skip prompts and use inline JSON string as preset
  -m, --packageManager <command>
  Use specified npm client when installing dependencies
  -r, --registry <url>
  Use specified npm registry when installing dependencies
  -n, --no-git
  Skip git initialization
  -f, --force
  Overwrite target directory if it exists
  --merge
  Merge target directory if it exists
  -c, --clone
  Use git clone when fetching remote preset
  -x, --proxy
  Use specified proxy when creating project
  -b, --bare
  Scaffold project without beginner instructions
  --skipGetStarted
  Skip displaying "Get started" instructions
  -h, --help
  Output usage information
```

Zakończenie

Reasumując, zależnie od wielu czynników należy wybrać taki framework. Każdy frameworków posiada inne podejście w kontekście tworzenia komponentów. Aktualnie w świecie IT, przoduje React, ze względu na swoją prostotę. Jednakże należy uważać na tą "prostotę". Dlaczego? Z powodu, że pisząc prosty moduł istnieje możliwość rozwiązania problemów na n różnych sposobów. Ale należy pamiętać o nastepstwach jakie za tym idą.

Natomiast Angular zawsze będzie stabilnym frameworkiem i idealnym dla ogromnych projektów, ze względu na swoje sztywne zasady, podczas projektowania interfejsu użytkownika. Zawsze istnieje możliwość, aby część projektu napisać w innym frameworku. Co raz cześciej jest to spotykane, że kilka zespołów piszę jeden projekt, jednakże są podzieleni na moduły, które są pisane w innych frameworkach.

Vue.js jako technologia co raz mocniej progresuję na rynku. Wielu programistów zaczyna swoją drogę z frameworkami, właśnie od Vue.js. Bierze się to stąd, że jest bardzo podobny w składni do czystego JavaScriptu, z oczywiście dodatkami. Jednak, React i Angular przez kolejne lata będą wiodły prym na rynku IT.

Spis rysunków

Bibliografia

[1] DOM
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

[2] React.js
https://reactjs.org

[3] Licencje BSD
https://pl.wikipedia.org/wiki/Licencje_BSD

[4] Conditional (ternary) operator
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

[5] Prop-types
https://reactjs.org/docs/typechecking-with-proptypes.html

[6] Cykl życia komponentu
https://pl.reactjs.org/docs/react-component.html

[7] ESLint
https://eslint.org ESLint

[8] React Redux
https://react-redux.js.org

[9] TDD W React.js
https://typeofweb.com/tdd-react-testing-library

[10] Testing In React
https://javascript.plainenglish.io/testing-in-react-part-1-types-tools

[11] Angular
https://angular.io/

[12] Angular-template
https://www.javatpoint.com/angular-template

[13] Computed-properties
https://blog.logrocket.com/understanding-computed-properties-in-vue-js/

[14] Computed-properties
https://vuejs.org//