Wykorzystanie biblioteki React do tworzenia interaktywnego interfejsu użytkownika

Julia May

Spis treści

1. Wprowadzenie

W dzisiejszych czasach aplikacje webowe są jednym z najbardziej rozpowszechnionych typów oprogramowania. Współcześnie praktycznie każda firma staje przed koniecznością posiadania własnej witryny internetowej, między innymi po to, aby zostać zauważoną przez potencjalnych nowych klientów, umożliwić łatwe i szybkie wyszukiwanie informacji na swój temat oraz świadczyć usługi online, takie jak na przykład sprzedaż swoich produktów. Zwiększa to zapotrzebowanie na narzędzia umożliwiające szybkie i wygodne tworzenie aplikacji webowych, zarówno spełniających wymagania klientów i użytkowników, jak i będących łatwymi do utrzymania i rozwijania przez developerów. Bardzo ważnym elementem dobrej aplikacji webowej jest warstwa interfejsu użytkownika. To właśnie z nią użytkownik dokonuje interakcji i zazwyczaj jej działanie ma kluczowe znaczenie dla zadowolenia lub niezadowolenia użytkownika z danej aplikacji webowej. Spowodowało to, że w ostatnich latach powstało wiele bibliotek umożliwiających szybkie i intuicyjne tworzenie interfejsów użytkownika w aplikacjach webowych. Najpopularniejszymi z nich wśród współczesnych programistów są: React, Angular oraz Vue.js, posiadające szeroką społeczność użytkowników oraz ciągle rozwijane przez swoich twórców, aby sprostać stale zmieniającym się wymaganiom i oczekiwaniom. Wszystkie z nich wykorzystane zostały do budowy warstwy interfejsu wielu popularnych witryn internetowych.

W ostatnich latach rośnie również popularność różnorodnych serwisów społecznościowych, na których użytkownicy mogą komunikować się między sobą, wymieniać informacje na różnorodne tematy oraz publikować różnorodne treści, takie jak posty, komentarze, zdjęcia oraz filmy. Pojawiło się między innymi wiele serwisów umożliwiających użytkownikom wzajemne motywowanie się w osiąganiu celów oraz publikowanie swoich osiągnięć dotyczących różnych dziedzin życia, od osiągnięć zawodowych po sukcesy sportowe. Przykładem takiego serwisu jest projekt GoaLeaf, którego interfejs użytkownika został zbudowany w oparciu o bibliotekę React, ponieważ uznaliśmy ją za najbardziej intuicyjną w obsłudze oraz umożliwiającą łatwe utrzymywanie aplikacji i dodawanie do niej nowych funkcjonalności.

Celem tej części pracy jest przedstawienie najważniejszych elementów biblioteki React na przykładzie interfejsu użytkownika serwisu GoaLeaf oraz pokazanie poszczególnych kroków na drodze do stworzenia działającej funkcjonalnej aplikacji z wykorzystaniem biblioteki React. W drugim rozdziale omówione zostaje, czym jest biblioteka React, jaki jest jej związek z managerem pakietów NPM oraz wskazane zostają jej podstawowe założenia. W trzecim rozdziale poruszony zostaje temat wykorzystania JSX, czyli rozszerzonego języka JavaScript, umożliwiającego używanie składni HTML wewnątrz kodu JavaScript. Następnie omówione zostają najważniejsze elementy biblioteki niezbędne to tworzenia interfejsu użytkownika. Rozdział czwarty prezentuje ideę komponentu, czyli podstawowej jednostki służącej do budowy aplikacji w React. W rozdziale piątym omówione zostają właściwości *state* i *props* przypisane do komponentu w React. W rozdziale szóstym omówiony zostaje cykl życia komponentu, który składa się z trzech faz: *Mounting*, *Updating* i *Unmounting*. Każda z tych faz posiada specjalne metody umożliwiające programiście sterowanie tym, co dzieje się w każdej z faz cyklu życia

komponentu. W rozdziale siódmym omówiony zostaje mechanizm obsługi zdarzeń w bibliotece React oraz sposób wykorzystania go w aplikacji do reagowania na czynności wykonane przez użytkownika. W rozdziale ósmym poruszony zostaje temat obsługi formularzy w React. W następnej kolejności omówione zostaną najważniejsze dodatkowe biblioteki należące do React, które zostały użyte w projekcie, czyli React Router i Axios. React Router, omówiony w rozdziale dziewiątym, służy do nawigowania w aplikacji i właściwej synchronizacji UI wyświetlanego w przeglądarce z aktualnie wybranym adresem URL. Biblioteka Axios, omówiona w rozdziale dziesiątym, służy do komunikacji z API serwera. W projekcie wykorzystana została do pobierania i edytowania danych na serwerze udostępniającym REST API.

2. React i NPM

Celem tego rozdziału jest omówienie czym jest biblioteka React, krótkie przedstawienie historii jej powstania oraz problemów występujących przy tworzeniu aplikacji webowych przed wprowadzeniem React, które biblioteka ta rozwiązuje. Następnie przedstawiony zostaje NPM, czyli manager pakietów dla języka JavaScript, który umożliwia pobieranie pakietów i bibliotek z rejestru online w celu wykorzystania ich w projekcie. Umożliwia on również pobranie i korzystanie z React. Zaprezentowane zostają najważniejsze komendy NPM, niezbędne przy tworzeniu aplikacji oraz przykładowa budowa pliku package.json, zawierającego informacje o pakietach i ich wersjach wykorzystywanych w aplikacji.

React jest biblioteką do tworzenia interaktywnych interfejsów użytkownika, stworzoną w języku JavaScript, wydaną na licencji MIT. Stworzona została przez Jordana Walke, programistę Facebooka i wydana jako biblioteka *open source* przez Facebook 29 maja 2013 roku, jako wersja 0.3.0. W 2015 roku wprowadzony został również React Native, służący do tworzenia aplikacji mobilnych na Android i iOS. Aktualnie najnowszy *release* React został wprowadzony 22 października 2019 i jest to wersja 16.11.0. [1]

React służy do tworzenia tak zwanych *single-page apps*. Idea ta polega na tym, że zamiast nawigowania pomiędzy podstronami lub przeładowywania danej strony, żeby uaktualnić jej zawartość, odpowiednie widoki danej strony są ładowane dynamicznie w odpowiedzi na działania użytkownika. Przed wprowadzeniem React istniały trzy główne problemy utrudniające tworzenie *single-page apps*. Jednym z nich była konieczność bezpośredniego kontrolowania przez programistę poprawnej synchronizacji zmian w danych i tego, co jest widoczne w UI (*User Interface*). Kolejnym problemem była konieczność wprowadzania wielu zmian w modelu DOM (*Document Object Model*), takich jak dodawanie i usuwanie elementów, ponieważ bezpośrednie operacje na modelu DOM są bardzo wolne. Trzecią główną trudnością napotykaną przy tworzeniu *single-page apps* była praca z szablonami HTML, które są zarządzane i wypełniane odpowiednimi danymi przez kod JavaScript, co często prowadzi do bardzo skomplikowanego i mało czytelnego kodu. [2]

Wprowadzenie biblioteki React miało na celu między innymi rozwiązanie powyższych problemów. React automatycznie zarządza stanem UI, co stanowi duże ułatwienie dla programisty. W celu zwiększenia szybkości działania aplikacji React nie modyfikuje bezpośrednio modelu DOM, tylko korzysta z umieszczonego w pamięci *virtual* DOM. Operacje przeprowadzane na *virtual* DOM są bardzo szybkie. React porównuje zmiany w *virtual* DOM ze zmianami we właściwym modelu DOM i decyduje, które zmiany należy również wprowadzić w modelu DOM, aby zachować spójność pomiędzy oboma modelami oraz jednocześnie wykonać jak najmniej czasochłonnych modyfikacji. React umożliwia dzielenie elementów widocznych na stronie na mniejsze komponenty, co zwiększa czytelność kodu i umożliwia tworzenie aplikacji o konstrukcji modułowej. Aby poradzić sobie z problemem zarządzania szablonami HTML, React wykorzystuje składnię JSX, co umożliwia definiowanie UI w kodzie JavaScript, dzięki czemu nie potrzeba dodatkowo definiować połączeń między szablonem strony napisanym w HTML a kodem, który nim zarządza napisanym w JavaScript. [2]

NPM jest managerem pakietów dla języka JavaScript, który umożliwia pobieranie pakietów i bibliotek z rejestru online. W skład NPM wchodzą: strona internetowa, na której znajdują się informacje o pakietach, *Command Line Interface* po stronie klienta, który umożliwia interakcję z NPM oraz rejestr, który jest bazą danych zawierająca biblioteki i pakiety. Najważniejszymi komendami *Command Line Interface*, które są niezbędne w procesie tworzenia aplikacji w React, są `npx create-react-app`, `npm install`, `npm start` i `npm run build`. Komenda `npx create-react-app`, przedstawiona na Rys. 2.1, służy do utworzenia nowej *single-page app* bazującej na React. Komenda ta, oprócz utworzenia szablonu aplikacji, dokonuje również automatycznej konfiguracji dodatkowych narzędzi, takich jak na przykład Babel, dzięki czemu programista nie musi wykonywać konfiguracji własnoręcznie. Do uruchomienia tej komendy niezbędne jest posiadanie Node.js w wersji co najmniej 8.10 oraz npm w wersji co najmniej 5.6.

```
npx create-react-app my-app
cd my-app
npm start
```

Rys. 2.1 Utworzenie szablonu aplikacji React o nazwie my-app i uruchomienie go (https://reactjs.org/docs/create-a-new-react-app.html)

Komenda `npm install` służy do zainstalowania w lokalnym folderze node_modules wszystkich modułów i bibliotek wymienionych jako `dependencies` w pliku package.json.

Komenda `npm start` wywołuje komendę `start` umieszczoną w pliku package.json w części `scripts`, w przykładzie na Rys. 2.2 jest to `react-scripts start`, która to komenda należy do zbioru skryptów pakietu create-react-app i służy do konfiguracji środowiska oraz uruchamia serwer aplikacji.

Komenda `npm run build` tworzy folder build, który zawiera zagregowane wszystkie pliki CSS w jeden plik oraz wszystkie pliki JavaScript, również w jednym pliku. Jest ona używana, gdy aplikacja ma zostać wdrożona. Zagregowanie plików ma wpływ na zwiększenie szybkości ładowania aplikacji po stronie klienta. [3]

```
{
"name": "goaleaf",
"version": "0.1.0",
"private": true,
"proxy": "http://localhost:8080",
"dependencies": {
"@material-ui/core": "^3.9.3",
"axios": "^0.18.0",
"materialize-css": "^1.0.0",
"node-sass": "^4.11.0",
"react": "^16.8.4",
"react-avatar-edit": "^0.8.2",
"react-avatar-editor": "^11.0.7",
"react-dom": "^16.8.4",
"react-dropzone": "^10.1.3",
"react-images-upload": "^1.2.6",
"react-images-uploader": "^1.2.0-rc1",
"react-materialize": "^3.3.1",
"react-redux": "^6.0.1",
"react-router-dom": "^5.0.0",
"react-scripts": "2.1.8",
"reactjs-popup": "^1.4.0",
"redux": "^4.0.1",
"redux-thunk": "^2.3.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test",
"eject": "react-scripts eject"
},
"eslintConfig": {
"extends": "react-app"
},
```

```
"browserslist": [
">0.2%",
"not dead",
"not ie <= 11",
"not op_mini all"
]
}
```

Rys. 2.2 Zawartość pliku package.json na przykładzie projektu GoaLeaf

Rys. 2.2 przedstawia przykładową budowę pliku package.json. Na początku pliku znajdują się parametry `name` i `version`, które określają nazwę i wersję aplikacji JavaScript. Następny parametr `private` ustawiony na wartość *true* zabezpiecza przed opublikowaniem prywatnego repozytorium przez npm. Pole `proxy` określa adres, pod który mają być kierowane zapytania, np. generowane przez bibliotekę Axios do łączenia się z REST API serwera. Pole `dependencies` zawiera nazwy wraz z wersjami wszystkich używanych w aplikacji zewnętrznych modułów i bibliotek. Parametr `scripts` wskazuje na komendę, która ma zostać wywołana w odpowiedzi na wpisanie odpowiedniej komendy npm, w tym przypadku `start`, `build`, `test` lub `eject`. Zawartość pola `eslintConfig` definiuje konfigurację narzędzia ESLint do statycznej analizy kodu JavaScript, w tym przypadku parametr `extends` określa, że powinien zostać użyty *plugin* obsługujący React. Pole `browserslist` określa jakie przeglądarki i ich wersje mają być obsługiwane przez aplikację, według reguł określonych przez pakiet Browserslist, będący częścią npm. [4]

3. JSX

Celem tego rozdziału jest przedstawienie JSX, czyli rozszerzonej składni JavaScript, która stanowi istotny element biblioteki React. Poruszony zostaje również temat pakietu Babel, służącego do tłumaczenia znaczników JSX na czysty React i JavaScript, co jest niezbędne, aby aplikacja React mogła zostać uruchomiona przez przeglądarkę internetową. Przedstawiona zostaje również budowa pliku konfiguracyjnego .babelrc dla pakietu Babel.

JSX to rozszerzenie języka JavaScript, umożliwiające korzystanie ze składni języka HTML wewnątrz kodu napisanego w JavaScript. Każdy element ma przypisany odpowiedni *tag*, który użyty w kodzie wskazuje, że w tym miejscu wykorzystany ma być dany typ elementu, natomiast podane atrybuty określają jego argumenty. Ewentualne elementy potomne są umieszczane pomiędzy znacznikami otwierającym i zamykającym, co umożliwia tworzenie zagnieżdżonych komponentów. Wyrażenia JavaScript umieszczone wewnątrz kodu JSX muszą być otoczone nawiasami klamrowymi, które oznaczają, że wynik ma zostać zwrócony po wykonaniu zawartego wewnątrz kodu. Aby zdefiniować atrybut `class` elementu, należy użyć `className`, ponieważ `class` jest słowem zarezerwowanym w JSX. Rys. 3.1 przedstawia przykładowy element JSX. [5]

```
<li className="collection-item">
<span className="member-login">{props.position}</span>
<img src={TempPic} alt="User avatar" title="User avatar" width="128" height="128" />
<span className="member-login">{props.userLogin}</span>
<span className="member-login">{props.points}</span>


</li>
```

Rys. 3.1 Przykładowy element JSX

```
let members = this.state.members;
let foundMembers = false;
let memberCards = [];

Object.keys(this.state.members).map(pos => {

foundMembers = true;
let member = this.state.members[pos];
memberCards.push(<LeaderboardCard key={member.id} position={pos} userID={member.userID}
userLogin={member.userLogin} profilePic={member.imgName} points={member.points}/>)
})
```

Rys. 3.2 Przykład wykorzystania JSX wewnątrz kodu JavaScript

Na Rys. 3.2 pokazane jest, w jaki sposób JSX może zostać użyty w kodzie JavaScript na przykładzie funkcji mapującej elementy słownika na elementy JSX. Zmienna `members` zawiera słownik, w którym umieszczony jest ranking użytkowników według zdobytych przez nich punktów. Liczba punktów stanowi klucz, obiekt zawierający dane użytkownika – wartość. Dla każdego klucza pobierany jest przypisany mu obiekt użytkownika i na podstawie zawartych w nim danych tworzony jest element `LeaderboardCard`, zapisywany następnie w tablicy `memberCards`. Obiekty w niej zawarte zostaną później umieszczone w komponencie będącym rankingiem użytkowników.

Aby kod JavaScript z rozszerzeniem JSX mógł zostać właściwie zinterpretowany przez przeglądarkę, musi zostać przeprowadzony proces transpilacji, czyli tłumaczenia znaczników JSX na czysty React i JavaScript, formę zrozumiałą dla przeglądarki. Do tego celu służy Babel. Wszystkie elementy JSX zawarte w kodzie zostają przekształcone na wywołania funkcji `React.createElement`, która służy do tworzenia odpowiedniego elementu z podanymi argumentami. Konfiguracja Babel umieszczona jest w pliku .babelrc, który określa, jakie *presets*, czyli zestawy wtyczek, mają zostać użyte oraz definiuje dodatkowe opcje.

```
{
 "presets": [
  [
   "@babel/preset-react",
   {
    "pragma": "dom", // default pragma is React.createElement
    "pragmaFrag": "DomFrag", // default is React.Fragment
    "throwIfNamespace": false // defaults to true
   }
  ]
 ]}
```

Rys. 3.3 Przykładowa zawartość pliku .babelrc (https://babeljs.io/docs/en/babel-preset-react)

Rys. 3.3 przedstawia przykładową zawartość pliku .babelrc. `presets` określa zbiory wtyczek Babel, które mają zostać użyte. *Presets* powodują, że w konfiguracji nie potrzeba wymieniać każdej wtyczki z osobna, tylko można skorzystać z gotowych zestawów. `pragma` służy do zdefiniowania funkcji używanej do kompilacji wyrażeń JSX (domyślnie `React.createElement`). `pragmaFrag` określa komponent używany do kompilacji fragmentów JSX (domyślnie `React.Fragment`). `throwIfNamespace` określa czy ma zostać zgłoszony błąd w sytuacji użycia *tagu* definiującego przestrzeń nazw XML. [6]

4. Komponenty

Celem tego rozdziału jest przedstawienie idei komponentu na przykładzie pochodzącym z aplikacji GoaLeaf. Komponent to podstawowa jednostka służąca do budowy aplikacji w React. Poszczególne komponenty umożliwiają podział aplikacji na mniejsze fragmenty, odizolowane i niezależne od siebie, które mogą zostać użyte wielokrotnie w różnych sytuacjach. Ułatwia to pracę nad aplikacją i zwiększa czytelność kodu, poprzez podzielenie jego zawartości na poszczególne funkcjonalne komponenty. Omówione zostają również różne sposoby importowania i eksportowania komponentów w React w celu wykorzystania ich w innych częściach aplikacji.

```
class CommentCard extends Component {

state = {
userLogin: ''
}

render() {
return (
<li className="comment-card collection-item col s10 offset-s2">
<div className="comment-profile">
<img className="comment-profile-pic" src={TempPic} alt="User avatar" title="User avatar" />
<p className="comment-profile-login">{this.props.userLogin}</p>
<p className="comment-profile-date">{changeDateFormat1(this.props.date)}</p>
</div>
<div className="comment-content">
<span>{this.props.commentText}</span>
</div>
{this.props.currentUserLogin === this.props.userLogin ?
<Dropdown trigger={<a href="#!" className='comment-nav dropdown-trigger' data-
target={this.props.id}><img src={MoreIcon}></img></a>}>
<a href="#!" className="dropdown-item dropdown-delete" onClick={() =>
this.props.handleCommentCardDeleted(this.props.id)}>Delete</a>
</Dropdown>
: null}
```

```
</li>
)}}
export default CommentCard;
```

Rys. 4.1 Przykładowa budowa komponentu

Komponenty w React to fragmenty kodu JavaScript opakowane w klasę dziedziczącą po klasie `Component`, które poprzez funkcję `render` zwracają (za pośrednictwem JSX) elementy HTML do wyświetlenia na stronie. W przykładzie z Rys. 4.1 komponent `CommentCard` reprezentuje komentarz dodany przez użytkownika do konkretnego postu. Pole `state` reprezentuje stan komponentu, czyli zmienne, które mogą być przechowywane wewnątrz niego. W przypadku komponentu `CommentCard` jest to łańcuch znaków `userLogin`, reprezentujący nazwę użytkownika, który jest autorem danego komentarza. Funkcja `render` zwraca poprzez `return` element HTML do wyświetlenia na stronie internetowej, reprezentujący kartę z danymi o danym komentarzu oraz użytkowniku, przez którego został dodany. Danymi tymi są: zdjęcie profilowe użytkownika, który dodał komentarz, nazwa tego użytkownika, data dodania komentarza (która przed wyświetleniem jest przekształcana do odpowiedniego formatu za pomocą funkcji `changeDateFormat1`) oraz treść komentarza. Pobierane są one z pola `props` komponentu, które reprezentuje dane lub funkcje podawane podczas wywoływania danego komponentu w komponencie wyższego rzędu. Umożliwia to wielokrotne wywoływanie komponentu z innymi wartościami przekazywanych do niego zmiennych, dzięki czemu dany komponent może być wykorzystywany wielokrotnie z innymi argumentami. W przypadku komponentu `CommentCard` umożliwia to utworzenie wielu kart z danymi o różnych komentarzach na podstawie tego samego szablonu komponentu. Następnie w komponencie `CommentCard` sprawdzane jest, czy aktualnie zalogowany użytkownik jest autorem danego komentarza. Jest to sprawdzane na podstawie zgodności nazw użytkowników, które w aplikacji GoaLeaf są unikalne dla każdego użytkownika. Jeśli jest to ten sam użytkownik, w jego interfejsie jest widoczna lista rozwijana zawierająca przycisk `Delete`, który umożliwia usunięcie dodanego przez siebie wcześniej komentarza. Usuwanie odbywa się poprzez wywołanie odpowiedniej funkcji usuwającej komentarz podanej jako jeden z elementów pola `props` komponentu, która służy do usuwania komentarza na podstawie jego unikalnego ID. W przypadku innych użytkowników niż autor komentarza lista i przycisk nie wyświetlają się w ich interfejsie użytkownika.

Aby użyć konkretnego komponentu w React, należy najpierw go zaimportować, aby umożliwić odwoływanie się do niego w kodzie. Istnieją trzy dozwolone typy składni do importowania komponentów, które przedstawione są na Rys. 4.2. [7]

```
import GIVEN_NAME from ADDRESS
import { PARA_NAME } from ADDRESS
import GIVEN_NAME, { PARA_NAME, ... } from ADDRESS
```

Rys. 4.2 Trzy możliwe sposoby na importowanie komponentów w React (https://www.geeksforgeeks.org/reactjs-importing-exporting/)

Przykładowe użycia pierwszego z nich, czyli `import` GIVEN_NAME `from` ADDRESS, w projekcie GoaLeaf, widoczne są na Rys. 4.3 i służą do zaimportowania domyślnego eksportu komponentu o nazwie GIVEN_NAME z pliku ADDRESS. W React nie jest konieczne umieszczanie rozszerzenia „.js” w nazwie pliku, jest ona przyjęta domyślnie, co widać poniżej dla pliku TaskCard.js.

```
import TaskCard from './TaskCard'
import Popup from "reactjs-popup"
import ReactPaginate from 'react-paginate'
```

Rys. 4.3 Przykładowe importy typu `import` GIVEN_NAME `from` ADDRESS

Kolejny sposób importowania komponentów, widoczny na Rys. 4.4 na przykładzie użycia w projekcie GoaLeaf, czyli `import` { PARA_NAME } `from` ADDRESS służy do zaimportowania tylko wymienionego nazwanego parametru PARA_NAME z pliku ADDRESS. Można w ten sposób importować wiele parametrów naraz, oddzielając je przecinkami wewnątrz nawiasów klamrowych. W poniższym przypadku plik helpers.js zawiera kilka funkcji pomocniczych do formatowania dat, ale importowana jest tylko wybrana funkcja `changeDateFormat1` z danego pliku.

```
import { changeDateFormat1 } from '../../../../js/helpers'
import { Dropdown } from 'react-materialize'
import { withRouter } from 'react-router-dom'
```

Rys. 4.4 Przykładowe importy typu `import` { PARA_NAME } `from` ADDRESS

Ostatni ze sposobów importowania komponentów, czyli `import` GIVEN_NAME, { PARA_NAME } `from` ADDRESS jest połączeniem dwóch poprzednich i służy do zaimportowania w jednym wywołaniu zarówno

komponentu o nazwie GIVEN_NAME, jak i nazwanego parametru PARA_NAME z pliku ADDRESS. Przykładem użycia tego sposobu importowania komponentów w projekcie GoaLeaf jest `import React, {Component} from 'react'`.

Z drugiej strony, aby dany komponent mógł zostać zaimportowany w innych miejscach w aplikacji, najpierw musi zostać w odpowiedni sposób wyeksportowany, czyli udostępniony do importu pod konkretną nazwą. Istnieją dwa dozwolone typy składni do eksportowania komponentów, które przedstawione są na Rys. 4.5.

```
export default GIVEN_NAME
export { PARA_NAME }
```

Rys. 4.5 Dwa możliwe sposoby na eksportowanie komponentów w React
(https://www.geeksforgeeks.org/reactjs-importing-exporting/)

Pierwszy z nich, widoczny na Rys. 4.6 na przykładzie użycia w projekcie GoaLeaf, czyli `export default` GIVEN_NAME służy do wykonania domyślnego eksportu komponentu o nazwie GIVEN_NAME. Komponent może mieć co najwyżej jeden domyślny eksport.

```
export default AddTask;
export default LeaderboardCard;
export default AddTask;
```

Rys. 4.6 Przykładowe domyślne eksporty komponentu

Drugi z nich, widoczny na Rys. 4.7 na przykładzie użycia w projekcie GoaLeaf, czyli `export` { PARA_NAME } służy do wyeksportowania tylko określonego nazwanego parametru o nazwie PARA_NAME. Tutaj również można eksportować wiele parametrów naraz, oddzielając je przecinkami wewnątrz nawiasów klamrowych.

```
export { changeDateFormat1 }
export { fetchUsers }
```

Rys. 4.7 Przykładowe eksporty nazwanego parametru

5. Props i state

Celem tego rozdziału jest omówienie właściwości *state* i *props* przypisanych do komponentu w React. *State* jest obiektem wbudowanym w każdy komponent. Przechowuje on wartości zmiennych przypisanych do danego komponentu i wykorzystywanych przez niego. Zmiana wartości w *state* powoduje ponowne renderowanie komponentu w celu uwzględnienia wprowadzonych zmian w interfejsie użytkownika. *Props* służą do przekazywania wartości do komponentów, aby mogły zostać później w nich wykorzystane.

```
let tasks = this.state.tasks;
let foundTasks = false;
let taskCards = [];

tasks.forEach(task => {

foundTasks = true;
taskCards.push(<TaskCard key={task.id} id={task.id} description={task.description} points={task.points}
habitID={this.props.habitID}/>)

})

let tasksToDisplay = taskCards;

if (!foundTasks) {
tasksToDisplay = <li style={{display: 'flex', justifyContent: 'center', marginTop: '110px'}}>There are no tasks yet ?
¯\_(ツ)_/¯</li>
}

if (localStorage.getItem('token')) {
return (
<div className="row">
<ul className="tasks">
{tasksToDisplay}
</ul>
</div>
```

```
)
} else {
return null
}
```

Rys. 5.1 Przekazywanie argumentów do komponentu z wykorzystaniem *props*

W aplikacji składającej się z wielu zależnych komponentów musi istnieć możliwość komunikacji między nimi oraz przekazywania argumentów. W React do tego służą *props*, czyli właściwości komponentu, które umożliwiają przekazywanie danych między komponentami oraz wielokrotne używanie tych samych szablonów komponentów dla różnych danych. *Props* pozwalają na przekazywanie informacji tylko w jednym kierunku: z komponentu nadrzędnego do komponentu podrzędnego, nie można ich wykorzystać do przesyłania danych w drugą stronę. Ważną cechą *props* jest to, że stanowią pole tylko do odczytu, nie ma możliwości ich modyfikacji w komponencie podrzędnym. [8]

Rys. 5.1 przedstawia przykład przekazywania argumentów do komponentu z wykorzystaniem *props*. Jest to zawartość funkcji `render` komponentu `Tasks`, służącego do wyświetlania wszystkich zadań dostępnych do wykonania dla członków danej grupy. Wewnątrz pola `tasks`, należącego do `state`, czyli stanu komponentu, przechowywane są dane o dostępnych zadaniach do wykonania. Dla każdego zadania tworzony jest nowy komponent typu `TaskCard`, reprezentujący dane zadanie i informacje go dotyczące. ID zadania, jego opis, liczba punktów możliwych do uzyskania za wykonanie go oraz ID grupy, w której zadanie jest dostępne, są przekazywane do komponentu `TaskCard` jako `props`, którym nadane zostają odpowiednie nazwy. Poprzez podane nazwy można odwoływać się do konkretnych właściwości w komponencie podrzędnym. Jeśli użytkownik jest zalogowany oraz w grupie dostępne są zadania do wykonania, wyświetlana jest odpowiednia lista zadań, a w wypadku ich braku pojawia się odpowiedni komunikat. Informacje te nie są wyświetlane, jeśli użytkownik nie jest zalogowany.

Rys. 5.2 prezentuje korzystanie z właściwości *props* wewnątrz komponentu podrzędnego na przykładzie zawartości funkcji `render` komponentu `TaskCard`. Jeśli użytkownik jest zalogowany, to widzi pole z opisem zadania oraz ilością punktów do niego przypisanych, które pobierane są z właściwości `props` komponentu `TaskCard` poprzez nazwy nadane im przy przekazywaniu w komponencie nadrzędnym, czyli w tym przypadku w komponencie `Tasks`. Po kliknięciu pola z zadaniem wyświetla się `Popup`, który umożliwia ukończenie danego zadania wraz z odpowiednim komentarzem dodanym przez użytkownika.

```
if (localStorage.getItem('token')) {
return (
<Popup trigger={

<div className="task-card">
<a>⚡</a>
<div className="task-text-con">
<span className="task-title">{this.props.description}</span>
<span className="task-points">+{this.props.points}</span>
</div>
</div>

} modal closeOnDocumentClick
contentStyle={{
maxWidth: '80%',
width: '500px',
backgroundColor: '#f2f2f2',
borderRadius: '30px',
border: "none",
padding: '10px'
}}
overlayStyle={{
background: "rgb(0,0,0, 0.4)"
}}
>
{close => (
<div className="task-popup">
<form className="task-popup-form" onSubmit={e => {this.completeTask(e, this.props.id); close()}}>
<span className="task-popup-title">{this.props.description}</span>
<span className="task-popup-points">? {this.props.points} pts</span>
<input className="task-popup-input" id="taskComment" type="text" placeholder="Add a comment"
autoComplete="off" onChange={this.handleChange} />
<button className="btn task-popup-btn" type="submit" value="Complete task">⚡ Task completed</button>
</form>
```

```
</div>
)}
</Popup>
)
} else {
return null
}}
```

Rys. 5.2 Korzystanie z właściwości *props* wewnątrz komponentu podrzędnego

Przechowywanie danych wewnątrz komponentu umożliwia *state*, czyli pole reprezentujące stan komponentu. Stan przechowuje dynamiczne dane należące do danego komponentu i wykorzystywane przez niego w ciągu jego cyklu życia, na przykład wyświetlane w aplikacji. Determinuje on również często zachowanie danego komponentu. Z polem *state* nierozerwalnie związana jest metoda *render* należąca do metod cyklu życia komponentu, która wywoływana jest w odpowiedzi na każdą zmianę w zawartości pola *state* i powodująca ponowne renderowanie zaktualizowanej wersji komponentu.[8]

```
state = {
taskComment: ''
}

handleChange = e => {
this.setState({
[e.target.id]: e.target.value
})
}
```

Rys. 5.3 Przykład wykorzystania pola *state* komponentu

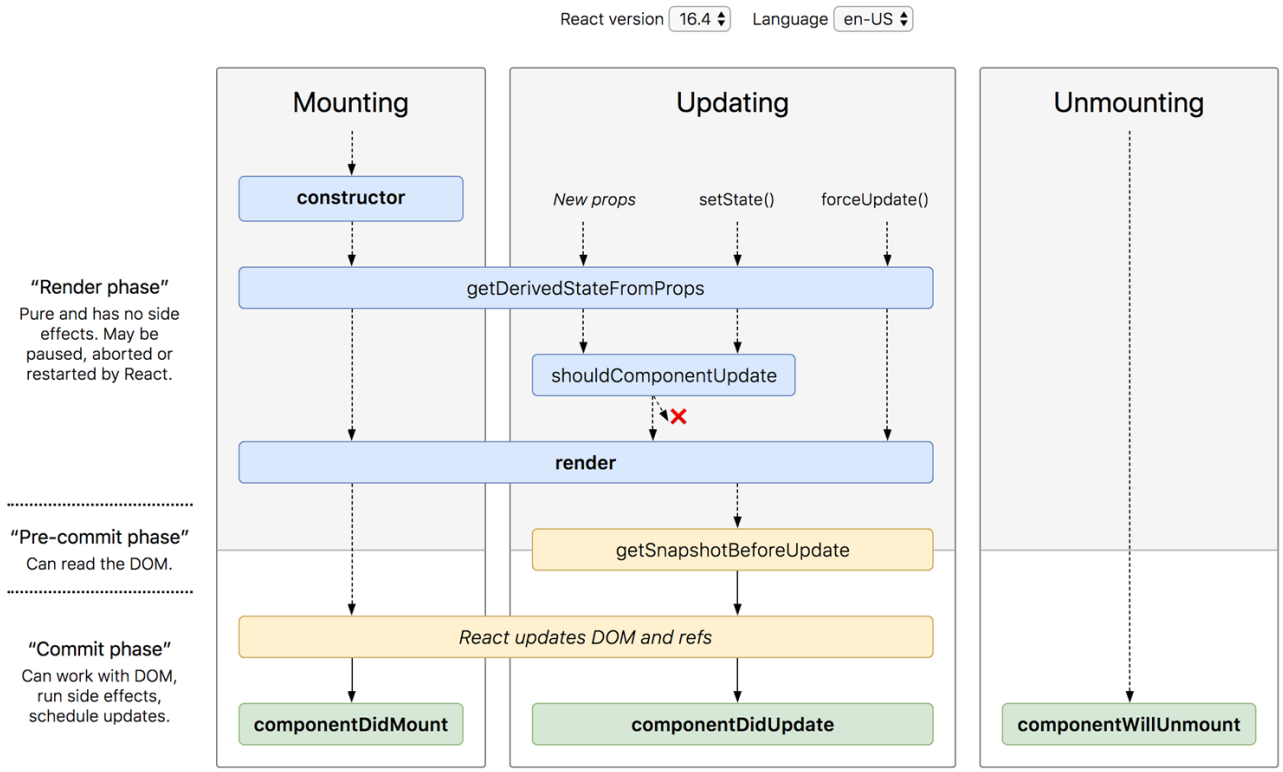
Na Rys. 5.3 przedstawione jest użycie *state* w komponencie `TaskCard`. Pole `taskComment`, należące do stanu komponentu, reprezentuje komentarz dodany przez użytkownika po ukończeniu danego zadania. Funkcja `handleChange` wykorzystywana jest do uaktualnienia zawartości stanu po wpisaniu komentarza przez użytkownika w polu o id `taskComment` w formularzu widocznym na Rys. 5.2. Do uaktualniania zawartości stanu komponentu służy funkcja `setState`. Należy w niej podać pole, które trzeba uaktualnić oraz jego nową

wartość. W tym przypadku pole stanu o nazwie takiej, jak id pola formularza, czyli `taskComment` jest uaktualniane wartością podaną przez użytkownika w tym polu formularza.

6. Cykl życia komponentu

Celem tego rozdziału jest przedstawienie, czym jest cykl życia komponentu w React. Omówione zostają trzy fazy cyklu życia, kolejność ich występowania oraz znaczenie dla poprawnego działania aplikacji. Zaprezentowane zostają również specjalne metody umożliwiające sterowanie tym, co dzieje się w każdej z wymienionych faz, dzięki czemu programista ma dużą kontrolę nad tym, w którym momencie działania programu wykonany zostanie napisany przez niego kod.

Cykl życia komponentu w React składa się z trzech faz: *Mounting*, *Updating* i *Unmounting*. Każda z nich posiada unikalne dla siebie metody, których nadpisanie umożliwia uruchamianie kodu zdefiniowanego przez programistę w konkretnych momentach cyklu życia komponentu. Diagram prezentujący cykl życia komponentu przedstawiony jest na Rys. 6.1.



Rys. 6.1 Cykl życia komponentu i występujące w nim metody (https://medium.com/@nancydo7/understanding-react-16-4-component-lifecycle-methods-e376710e5157)

6.1 Mounting

Mounting jest fazą, w której nowy komponent zostaje utworzony i umieszczony w modelu DOM. Należą do niej następujące metody, wywoływane w podanej kolejności: `constructor`, `getDerivedStateFromProps`, `render`, `componentDidMount`.

Metoda `constructor` jest wywoływana na początku fazy *Mounting*. Może być ona używana w sytuacji, gdy programista potrzebuje zainicjalizować lokalny stan komponentu poprzez przypisanie mu konkretnych obiektów lub podpiąć wybrane metody obsługi zdarzeń do danej instancji komponentu. Wywołanie tej metody powinno zawsze rozpoczynać się od wywołania `super(props)`, aby zabezpieczyć się przed problemami spowodowanymi niezdefiniowanymi właściwościami *props*. Metoda `constructor` jest jedynym miejscem w kodzie, gdzie polu *state* można przypisywać wartość bezpośrednio, nie używając metody `setState`. Jeżeli nie ma potrzeby inicjalizacji stanu komponentu ani podpięcia konkretnych metod obsługi zdarzeń, nie jest konieczne implementowanie własnego konstruktora dla danego komponentu. Przykładowe wywołanie metody `constructor` prezentuje Rys. 6.2.

```
constructor(props) {
  super(props);
  // Don't call this.setState() here!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

Rys. 6.2 Przykładowe wywołanie metody `constructor` (https://reactjs.org/docs/react-component.html#constructor)

Statyczna metoda `getDerivedStateFromProps` należy zarówno do fazy *Mounting*, jak i *Updating*, ale jest rzadko używana w praktyce, w związku z czym nie została przedstawiona w przykładowym kodzie. Jest wywoływana przed każdym wywołaniem metody `render` i służy do wprowadzania zmian w stanie komponentu, które zależą od wartości *props*, które zmieniają się w czasie.

Metoda `render` występuje zarówno w fazie *Mounting*, jak i *Updating* i jest jedyną metodą cyklu życia, której zaimplementowanie jest konieczne podczas tworzenia komponentu w React. Zazwyczaj w metodzie `render` zwraca się elementy JSX, które mają zostać umieszczone w modelu DOM aplikacji lub *null*, jeśli nic nie ma być renderowane. Wewnątrz funkcji `render` nie powinno się modyfikować stanu komponentu.

W następnej kolejności React uaktualnia model DOM aplikacji oraz parametry *refs*, które w React służą jako odnośniki i mogą zostać użyte wewnątrz komponentu nadrzędnego, aby uzyskać dostęp do komponentu podrzędnego (umożliwia to pobieranie z niego danych lub modyfikowanie go).
Metoda `componentDidMount` jest wywoływana zaraz po tym, jak komponent zostaje umieszczony w drzewie komponentów. Jest odpowiednim miejscem na przykład na załadowanie początkowych danych ze zdalnego serwera poprzez odpowiednie zapytanie HTTP. [9]

```
componentDidMount() {
axios.get(`/api/tasks/habit?habitID=${this.props.habitID}`)
.then(res => {
this.setState({
tasks: res.data
})
}).catch(err => console.log(err.response.data.message))
}
```

Rys. 6.3 Przykładowe wykorzystanie metody `componentDidMount` w projekcie GoaLeaf

Rys. 6.3 prezentuje przykładowe wykorzystanie metody `componentDidMount` w komponencie `Tasks`. Wykonywane jest tutaj zapytanie HTTP GET do zdalnego serwera z wykorzystaniem biblioteki Axios. Ma ono na celu pobranie wszystkich aktualnie dostępnych zadań dla grupy o podanym ID i umieszczenie ich wewnątrz pola *state* komponentu, w celu późniejszego wyświetlenia ich na stronie.

6.2 Updating

Updating jest kolejną po *Mounting* fazą cyklu życia komponentu. Rozpoczyna się ona w trzech przypadkach: gdy komponent otrzymuje nowe *props*, gdy zmienia się jego *state* lub gdy wywołana zostanie metoda `forceUpdate`. Należą do niej następujące metody, wywoływane w podanej kolejności: `getDerivedStateFromProps` (omówiona w poprzednim akapicie, poświęconym fazie *Mounting*), `shouldComponentUpdate`, `render` (również omówiona w poprzednim akapicie), `getSnapshotBeforeUpdate`, `componentDidUpdate`.
Metoda `shouldComponentUpdate` wywoływana jest w sytuacji, gdy komponent otrzymuje nowe *props* lub zmienia się jego *state*. Sprawdza ona czy wprowadzone zmiany wymagają uaktualnienia wyświetlanego UI. Jeśli metoda ta zwróci wartość logiczną *true*, wtedy wywoływana jest metoda `render`, powodująca ponowne renderowanie komponentu ze zaktualizowanymi elementami. W przeciwnym wypadku nie następuje ponowne

renderowanie komponentu. W przypadku użycia metody `forceUpdate` wymuszane jest ponowne renderowanie komponentu poprzez wywołanie metody `render` (z pominięciem `shouldComponentUpdate`).
Metoda `getSnapshotBeforeUpdate` może zostać wywołana tuż przed tym, jak uaktualniona wersja komponentu zostanie ponownie renderowana. Jej celem jest pobranie i zapisanie zdefiniowanych przez programistę danych z modelu DOM przed wywołaniem funkcji `render`, w celu zachowania ich jeśli ponowne renderowanie może doprowadzić do ich utraty.
W następnej kolejności, tak jak w fazie *Mounting*, React uaktualnia model DOM aplikacji oraz parametry *refs*.
Metoda `componentDidUpdate` jest wywoływana zaraz po zakończeniu uaktualniania modelu DOM i renderowania komponentu. Można dokonywać w niej operacji na modelu DOM lub wykonywać zapytania sieciowe, jeżeli istnieje taka potrzeba.[9]

6.3 Unmounting

Unmounting jest fazą, w której komponent jest usuwany z modelu DOM i stanowi zarazem zakończenie jego cyklu życia. Zawiera tylko jedną metodę: `componentWillUnmount`. Wewnątrz niej mogą zostać przeprowadzone czynności mające na celu uporządkowanie środowiska przed usunięciem komponentu, takie jak na przykład zresetowanie *timera* lub odwołanie zapytania sieciowego. [9]

7. Obsługa zdarzeń

Celem tego rozdziału jest omówienie obsługi zdarzeń w bibliotece React. Zdarzeniami mogą być na przykład kliknięcie myszką czy wprowadzenie danych z klawiatury, poprzez które użytkownik komunikuje się z programem. Przedstawione zostaje wykorzystanie mechanizmu obsługi zdarzeń w aplikacji GoaLeaf w celu odpowiedniego reagowania na czynności wykonane przez użytkownika.

W każdej aplikacji webowej niezbędne jest umożliwienie użytkownikowi dokonywania interakcji ze stroną, takich jak klikanie myszką lub wpisywanie znaków z klawiatury. Do odpowiedniej obsługi zdarzeń użytkownika służą *event handlers*, czyli specjalne funkcje, w których zaimplementowane jest, co powinno się wydarzyć w odpowiedzi na konkretne zdarzenie.

Rys. 7.1 przedstawia wykorzystanie mechanizmu obsługi zdarzeń w React. Jest to fragment funkcji `render` komponentu `InviteMember`, służącego do zapraszania innych użytkowników do dołączenia do grupy. Widoczny jest tutaj formularz, który w odpowiedzi na zdarzenie `submit` wywołuje funkcję `addMember` komponentu, co powoduje wysłanie zaproszenia podanemu użytkownikowi. Zdarzenie `submit` polega na kliknięciu przycisku o typie `submit` wewnątrz danego formularza lub wciśnięciu klawisza *Enter*. Pole `input` formularza w odpowiedzi na zdarzenie `change` wywołuje funkcję `handleChange` komponentu, która zapisuje aktualną zawartość tego pola w stanie komponentu. Zdarzenie `change` polega na zmianie zawartości danego elementu, w tym przypadku na wpisaniu przez użytkownika tekstu lub zmianie tekstu wpisanego wcześniej. Przycisk o typie `submit` tego formularza dodatkowo w odpowiedzi na zdarzenie `click`, czyli kliknięcie myszką, wywołuje funkcję `handleDisableBtn`, która gwarantuje, nawet jeśli użytkownik kliknie dany przycisk kilkukrotnie, zaproszenie do grupy zostanie wysłane do docelowego użytkownika tylko raz. [10]

```
<div className="invite-user-box">
<div className="row">
<form className="col s10 offset-s1 l8 offset-l2 center-align" onSubmit={(e) => this.addMember(e, this.props.habitID)} autoComplete="off">
<h4 className="">Send invitation</h4>
<div className="input-field inline">
<input id="userInvited" maxLength="30" type="text" placeholder="username" onChange={ this.handleChange } />
<span className={this.state.msg === 'Invitation sent' ? "helper-text green-text" : "helper-text red-text"}>{this.state.msg}</span>
</div>
<button className={this.props.disableBtn ? "btn disable-btn" : "btn"} onClick={ this.handleDisableBtn } type="submit" value="Invite user">
<span>Invite user</span>
</button>
</form>
</div>
</div>
```

Rys. 7.1 Przykład obsługi zdarzeń w React

Rys. 7.2 przedstawia zawartość funkcji `addMember` komponentu, wykorzystanej w kodzie widocznym na Rys. 7.1. Przyjmuje ona dwa argumenty: `e`, czyli obiekt zdarzenia oraz `id`, czyli identyfikator grupy. Wywołanie `e.preventDefault` w React służy do wyłączenia domyślnych mechanizmów reakcji na zdarzenie, takich jak na przykład ponowne załadowanie strony. Następnie wykonywane jest zapytanie do serwera, w celu wysłania wybranemu użytkownikowi (o loginie podanym w polu `input` na Rys. 7.1) zaproszenia do grupy o danym id razem z adresem URL strony grupy pobranym z pola `window.location.href`. Zależnie od tego, czy zapytanie zakończy się poprawnie, czy pojawią się błędy, do stanu komponentu dodawana jest odpowiednia wiadomość, wyświetlana wewnątrz elementu `span` formularza z Rys. 7.1.

```
addMember = (e, id) => {
e.preventDefault();

this.setState({
msg: <i className="fas fa-spinner fa-spin grey-text"></i>
})

axios.post('https://glf-api.herokuapp.com/api/habits/invitemember', {
"habitID": id,
"token": localStorage.getItem('token'),
"url": window.location.href,
"userLogin": this.state.userInvited
})
.then(res => {
this.setState({ msg: "Invitation sent" });
this.setState({ disableBtn: false });
}
).catch(err => {
this.setState({ msg: err.response.data.message });
this.setState({ disableBtn: false})
})
}
```

Rys. 7.2 Przykładowy *event handler* w React

8. Obsługa formularzy

Celem tego rozdziału jest przedstawienie mechanizmu obsługi formularzy w React. Aplikacja webowa musi umożliwiać użytkownikowi wprowadzanie danych, w celu na przykład założenia konta w serwisie lub dodania postu na blogu. W tym celu wykorzystuje się formularze, które umożliwiają szybkie wprowadzanie do systemu danych z wielu pól naraz. Zostanie omówiony mechanizm ich obsługi w bibliotece React oraz pokazany przykład ich użycia w projekcie GoaLeaf.

Formularze wykorzystuje się, aby umożliwić użytkownikowi wprowadzanie danych do aplikacji webowej, takich jak na przykład login czy treść postu. W React do obsługi formularzy wykorzystuje się *controlled components*. Są to elementy formularza takie jak na przykład `input` lub `textarea`, służące do wprowadzania tekstu przez użytkownika, których atrybut `value` jest pobierany bezpośrednio z pola `state` komponentu, w którym się znajdują, a każda zmiana wprowadzona przez użytkownika jest również zapisywana w polu `state`. Przykład użycia prezentuje Rys. 8.1, gdzie widoczny jest fragment funkcji `render` komponentu `AddPrize`, który służy do ustawienia ilości punktów potrzebnych do wygrania danego wyzwania w aplikacji GoaLeaf. [11]

```
<form className="col s10 offset-s1 l8 offset-l2 center-align" autoComplete="off">
<h4 className="">{this.props.pointsToWin === 0 ? 'Set' : 'Update'} goal</h4>
<div className="input-field inline">
<div style={{display: 'flex', justifyContent: 'center', marginTop: '30px'}}>
<button className="task-points-btn task-points-btn-subtract" onClick={ this.subtractPrizePoint }>-</button>
<input title="Set goal between 1 and 1000" id="prizePoints" maxLength="4" style={{width: '50px',
textDecoration: 'none', textAlign: 'center' }} className="task-points" value={this.state.prizePoints}
onChange={(e) => {this.handleChange(e)}}/>
<button className="task-points-btn task-points-btn-add" onClick={ this.addPrizePoint }>+</button>
</div>
<span className={this.state.msg === 'Goal set' ? "helper-text green-text" : "helper-text red-text
"}>{this.state.msg}</span>
</div>
<button className="btn" onClick={(e) => this.addPrize(e, this.props.habitID)} type="submit" value="Set goal">
<span>Set goal</span>
</button>
</form>
```

Rys. 8.1 Obsługa formularzy w React

Na Rys. 8.1 element `input` formularza jest przykładem użycia *controlled components* w React. Jako `value`, czyli zawartość wyświetlająca się w miejscu, w którym użytkownik może wpisać dane, podana jest `this.state.prizePoints`, czyli pole należące do `state` komponentu, które zawiera ilość punktów do ustawienia jako nagrodę. W odpowiedzi na zdarzenie `change`, czyli w tym przypadku wpisanie liczby przez użytkownika w polu `input`, wywoływana jest funkcja `handleChange`, która zapisuje w polu `state` komponentu wpisaną wartość jako zmienną o nazwie takiej jak id danego elementu, czyli w tym przypadku `prizePoints`. Zależnie od wartości zmiennej `pointsToWin` pola `state` wyświetlany jest odpowiedni napis, *Set goal* (gdy `pointsToWin` jest równe 0) lub *Update goal* (w pozostałych przypadkach). Oprócz wpisania wartości nagrody w polu `input` można też zmniejszać lub zwiększać jej wartość o 1 za pomocą przycisków. Na dole formularza znajduje się przycisk o typie `submit`, który umożliwia ustawienie punktów dla danego wyzwania poprzez wywołanie funkcji `addPrize`, która wykonuje zapytanie POST do serwera aplikacji.

9. React Router i nawigacja w aplikacji

Celem tego rozdziału jest zaprezentowanie biblioteki React Router oraz przedstawienie przykładu jej wykorzystania w aplikacji GoaLeaf. Biblioteka ta służy do nawigowania w aplikacji i umożliwia właściwą synchronizację UI wyświetlanego w przeglądarce z aktualnie wybranym adresem URL.

Pierwszym krokiem podczas używania React Router jest umieszczenie głównego komponentu aplikacji wewnątrz komponentu `BrowserRouter`, co przedstawione jest na Rys. 9.1. Ten fragment kodu znajduje się wewnątrz funkcji `render` komponentu `App`, który jest głównym komponentem aplikacji w React.

`Navbar` jest komponentem reprezentującym pasek z przyciskami, umieszczony na szczycie strony, umożliwiający przełączanie pomiędzy podstronami aplikacji, takimi jak na przykład profil użytkownika czy lista wyzwań, w których uczestniczy. To, jakie przyciski aktualnie się na nim wyświetlają, zależy od tego, czy użytkownik jest zalogowany, czy nie. W komponencie `Main` ładowane są odpowiednie komponenty, zależnie od aktualnego adresu URL.

```
return (
<BrowserRouter>
<div className="App">
<Navbar />
<Main />
</div>
</BrowserRouter>
)
```

Rys. 9.1 Wykorzystanie React Router na przykładzie projektu GoaLeaf

```
render(){
return (
<main className="main">
<Switch>
<Route exact path='/' component={Dashboard}/>

<Route exact path='/login' component={LogIn}/>
<Route exact path='/signin' component={SignIn}/>
<Route exact path='/reset-password' component={ResetPassword}/>
<Route exact path='/resetpassword/:token' component={ResetPasswordValidate}/>
<Route exact path='/new-password' component={NewPassword}/>

<PrivateRoute exact path='/new-habit' component={NewHabit}/>
<PrivateRoute exact path='/profile' component={Profile}/>
<PrivateRoute exact path='/habit/:id' component={HabitPage}/>
<Route exact path='/browse' component={BrowseHabits}/>

<Route path="*" component={Dashboard} />
</Switch>
</main>
)
}
```

Rys. 9.2 Wykorzystanie komponentu `Route` z biblioteki React Router

Rys. 9.2 przedstawia funkcję `render` komponentu `Main`, który jest głównym komponentem w aplikacji React . Zdefiniowane zostały w nim zasady nawigacji pomiędzy podstronami w aplikacji GoaLeaf. Najważniejszym komponentem w bibliotece React Router jest `Route`, który umożliwia mapowanie adresów URL na konkretne komponenty, które mają zostać wyświetlone. Adres URL podany jest jako parametr `path`, natomiast parametr `component` określa komponent, który ma zostać załadowany w odpowiedzi na podanie danego adresu URL. Adresy te są podane jako adresy względne wewnątrz aplikacji (*relative URLs*). Oznaczenie `exact` występujące przed parametrem `path` oznacza, że aby załadowany został dany komponent, wybrany adres URL musi dokładnie odpowiadać parametrowi `path`. W przypadku braku tego oznaczenia do parametru `path` */browse* dopasowane zostałyby zarówno adresy */browse*, jak i na przykład */browse/profile*. Część parametru `path` występująca po

dwukropku jak na przykład w adresie */habit/:id* to tak zwane *path params*. Ta część adresu, w tym przypadku identyfikator wyzwania, zostaje zapisana jako parametr `match.params.number` i może zostać użyta do załadowania zawartości komponentu odpowiadającej konkretnemu obiektowi. W tym przypadku obiektem tym jest wyzwanie o danym identyfikatorze (id), który może zostać pobrany poprzez odwołanie do `props.match.params.number` w komponencie `HabitPage`, aby załadować odpowiednie dane z serwera. `PrivateRoute` w aplikacji GoaLeaf służy do tego, aby część zawartości strony była widoczna tylko dla zalogowanych użytkowników. Jeśli użytkownik, który nie jest zalogowany, wybierze adres URL oznaczony jako `PrivateRoute`, zostanie automatycznie przekierowany na stronę logowania. Komponent `Switch` powoduje, że w danym momencie wybrany może zostać tylko jeden z zawartych w nim komponentów `Route`. Dopasowany zostaje pierwszy z nich, którego parametr `path` pasuje do aktualnie wybranego adresu URL i ostatecznie załadowany zostaje odpowiadający mu komponent. [12]

10. Axios i łączenie z Rest API

Celem tego rozdziału jest zaprezentowanie biblioteki Axios, która służy do komunikacji z API serwera. Jest ona klientem HTTP i umożliwia działanie w trybie asynchronicznym bazujące na idei *promise*. W projekcie wykorzystana została do pobierania i edytowania danych na serwerze udostępniającym REST API. Przedstawione zostaje jej wykorzystanie do wykonania trzech podstawowych typów zapytań HTTP: GET, POST i DELETE w projekcie GoaLeaf.

Aplikacja webowa posiadająca zarówno *frontend*, jak i *backend*, aby wyświetlać w warstwie interfejsu użytkownika dane pobrane na przykład z bazy danych, musi mieć możliwość komunikacji z serwerem pobierającym dane z tej bazy. Najczęściej wykorzystywaną architekturą komunikacji z serwerem jest REST. Architektura ta składa się z zestawu zdefiniowanych operacji, które mogą być udostępniane przez serwis, co zapewnia standaryzowany sposób komunikacji z wieloma systemami, w przeciwieństwie na przykład do architektury SOAP, w której każdy serwis może definiować własne operacje. Architektura REST opiera się na zapytaniach HTTP oraz metodach HTTP, takich jak GET, POST, PUT, DELETE, HEAD, PATCH, CONNECT, OPTIONS, TRACE. W React najczęściej stosowaną biblioteką udostępniającą możliwość korzystania z REST jest Axios. Poniżej omówione zostaną trzy najczęściej wykorzystywane metody HTTP wraz z przykładową ich obsługą w bibliotece Axios.

10.1 Metoda GET

Metoda GET służy do pobrania określonych zasobów z serwera i nie powoduje zmiany stanu serwera, to znaczy nie dokonuje modyfikacji danych, którymi zarządza serwer. Zgodnie z oficjalną dokumentacją React dobrym momentem w cyklu życia komponentu, aby wykonać zapytanie w celu pobrania danych z serwera, jest metoda `componentDidMount`.

```
componentDidMount() {
axios.get(`/api/tasks/habit?habitID=${this.props.habitID}`)
.then(res => {
this.setState({
tasks: res.data
})
}).catch(err => console.log(err.response.data.message))
}
```

Rys. 10.1 Przykładowe zapytanie GET w bibliotece Axios

Biblioteka Axios umożliwia wykonanie zapytania GET poprzez wywołanie funkcji `axios.get`, co przedstawione jest na Rys. 10.1. Przedstawiona metoda `componentDidMount` należy do komponentu `TasksAll`, który służy do wyświetlenia wszystkich zadań w danym wyzwaniu. W tym przypadku URL zapytania, czyli */api/tasks/habit?habitID=${this.props.habitID},* służy na serwerze do pobierania danych o wszystkich zadaniach w danym wyzwaniu, gdzie parametrem zapytania HTTP `habitID` jest identyfikator wyzwania, dla którego mają zostać pobrane zadania, który podawany jest jako `props` komponentu `TasksAll`. Po zakończeniu pobierania danych pole `data` obiektu `res`, który oznacza odpowiedź serwera, zapisywane jest jako `tasks` w polu `state` komponentu. Umieszczenie tej części kodu wewnątrz funkcji `then` gwarantuje, że zostanie on wykonany dopiero gdy zapytanie zostanie zakończone a dane pobrane. Zawartość funkcji `catch` jest wywoływana w sytuacji, gdy wykonywanie zapytania doprowadzi do zwrócenia błędu, reprezentowanego tutaj przez obiekt `err`. W tym przypadku w razie wystąpienia błędu jego treść wypisywana jest w konsoli. [13]

10.2 Metoda POST

Metoda POST służy do przesłania na serwer danych umieszczonych w ciele zapytania, w celu ich zapisania na serwerze.

```
addTask = (e, id) => {
e.preventDefault();

if (this.state.task !== null){

axios.post('/api/tasks/add', {
"description": this.state.task,
"frequency": this.state.frequency,
"daysInterval": this.state.days,
"habitID": id,
"points": this.state.taskPoints,
"token": localStorage.getItem("token")

})
.then(res => {
window.location.reload();
}
```

```
).catch(err => console.log(err.response.data.message))
}
}
```

Rys. 10.2 Przykładowe zapytanie POST w bibliotece Axios

W bibliotece Axios do wykonania zapytania POST służy funkcja `axios.post`, jej użycie przedstawione jest na Rys. 10.2. Funkcja `addTask` znajduje się w komponencie `AddTask`, który służy do tworzenia nowego zadania w danym wyzwaniu i jest wywoływana w odpowiedzi na kliknięcie przez użytkownika odpowiedniego przycisku. Jeśli treść zadania zawarta w polu `task` stanu komponentu nie jest pusta, to wykonywane jest zapytanie POST. URL zapytania, czyli */api/tasks/add*, służy na serwerze do dodawania nowego zadania w danym wyzwaniu o parametrach podanych w ciele zapytania POST w formacie JSON. Parametry te oznaczają w kolejności: opis zadania, czy jest to zadanie jednorazowe, czy może być wykonywane wielokrotnie, wymagany odstęp czasu pomiędzy wykonywaniem danego zadania dla zadań wykonywanych wielokrotnie, identyfikator wyzwania, do którego ma być przypisane zadanie, punkty, które można uzyskać za jego wykonanie oraz *token* zalogowanego użytkownika. Jeśli zapytanie zostanie wykonane poprawnie, wtedy następuje przeładowanie strony w celu uwzględnienia zmian w danych, spowodowanych dodaniem nowego zadania. W przypadku błędu jego treść wypisywana jest w konsoli. [13]

10.3 Metoda DELETE

Metoda DELETE służy do usuwania określonych zasobów przechowywanych na serwerze.

```
deleteTask = (e, id) => {
axios.delete(`/api/tasks/task/remove?taskID=${id}`)
.then(res => {
window.location.reload();
}).catch(err => { console.log(err) })
}
```

Rys. 10.3 Przykładowe zapytanie DELETE w bibliotece Axios

W bibliotece Axios zapytania DELETE wykonuje się za pomocą funkcji `axios.delete`, przykład przedstawia Rys. 10.3. Funkcja `deleteTask` znajduje się w komponencie `Task`, który służy do wyświetlania informacji o

danym zadaniu i jest wywoływana w odpowiedzi na kliknięcie przez użytkownika przycisku służącego do usuwania zadania. URL zapytania, czyli */api/tasks/task/remove?taskID=${id},* służy na serwerze do usuwania danego zadania, gdzie parametrem zapytania HTTP `taskID` jest identyfikator zadania, które ma zostać usunięte, podawany jako argument funkcji `deleteTask`. Jeśli nie wystąpiły żadne błędy podczas realizacji zapytania, wtedy następuje przeładowanie strony. W przypadku wystąpienia błędu jest on wypisywany w konsoli. [13]

11. Bibliografia

[1] https://en.wikipedia.org/wiki/React_(web_framework)

[2] „Learning React”, Kirupa Chinnathambi, 2018, Pearson Education, Inc.

[3] https://docs.npmjs.com/cli-documentation/cli

[4] https://nodesource.com/blog/the-basics-of-package-json-in-node-js-and-npm/

[5] „Learning React”, Alex Banks, Eve Porcello, 2017, O’Reilly Media, Inc.

[6] https://babeljs.io/docs/en/configuration

[7] https://www.geeksforgeeks.org/reactjs-importing-exporting/

[8] https://itnext.io/what-is-props-and-how-to-use-it-in-react-da307f500da0

[9] https://pl.reactjs.org/docs/react-component.html

[10] https://reactjs.org/docs/handling-events.html

[11] https://reactjs.org/docs/forms.html

[12] https://blog.pshrmn.com/simple-react-router-v4-tutorial/

[13] https://designrevision.com/react-axios/