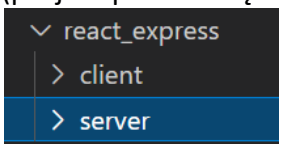
Beata Pańczyk

Szkielety programistyczne - uwagi do laboratorium 7
(MongoDB+Express+React+Node = MERN)

W zadaniach utworzone zostaną 2 aplikacje:

- **serwerowa** (REST API) w Express (jej zasoby znajdą się w folderze **server**)
- **kliencka** (frontend) w React (projekt pod nazwą **client**):



Zadanie 7.1.

Etap 1-5 Tworzenie aplikacji serwerowej w Express

W części "scripts" pliku **package.json** skonfigurować uruchamianie serwera:

```
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1",
        "start": "node index.js",
        "dev": "nodemon server.js"
      },
```

W zadaniu wykorzystano dodatkowo pakiety:

- **dotenv** – to moduł, który ładuje zmienne środowiskowe z pliku **.env**,
- **cors** - pakiet node.js zapewniający oprogramowanie pośredniczące (middleware) Connect/Express, którego można użyć do włączenia CORS z różnymi opcjami.

Polityka SOP i CORS w pigułce

Przeglądarki wprowadzają różne mechanizmy zabezpieczające a jednym z najważniejszych jest polityka SOP (ang. Same Origin Policy), która eliminuje szereg potencjalnych problemów bezpieczeństwa. SOP w pewnym uproszczeniu (w rzeczywistości polityka jest dużo luźniejsza) stanowi, że dwie aplikacje charakteryzujące się różnym pochodzeniem (ang. origin) nie mogą używać (ściągać, osadzać, odpytywać) swoich wzajemnych elementów.
CORS (ang. Cross-Origin Resource Sharing):

- to mechanizm, który zapewnia możliwość bezpiecznej wymiany danych pomiędzy stronami, które pochodzą z innych lokalizacji (origin),
- łagodzi restrykcyjną politykę SOP,
- umożliwia bezpieczne wykonywanie zapytań HTTP Cross-Origin – pozwala, aby to strona serwerowa decydowała czy ufa stronie klienckiej (wysyłającej zapytanie z innego origin) i czy w związku z tym zapytanie powinno zostać obsłużone.

W ramach laboratorium pierwsza aplikacja Rest API (express) będzie uruchamiana na serwerze np. localhost:8080 i będziemy chcieli z niej skorzystać wysyłając żądania asynchroniczne z aplikacji klienckiej (React), która z kolei będzie uruchamiana na innym serwerze - np. localhost:3000. Aby można było obsłużyć takie żądanie po stronie serwera (Express) stosuje się odpowiednie funkcje (importowane z modułu **cors**) wskazujące, czy i ewentualnie, które żądania (wysyłane asynchronicznie przez aplikację klienta) i z jakiej lokalizacji (origin) można obsłużyć po stronie serwera.
Instrukcja:

```
app.use(cors())
```

lub bardziej szczegółowo:

```
app.use(cors({
    origin: '*'
}));
```

oznacza, że nasz interfejs API jest publiczny, i wszelkie inne serwery pochodzące z różnych innych źródeł mogą uzyskiwać dostęp do tych zasobów.
Chcąc ograniczyć dostęp do API można wskazać, które serwery mogą z niego korzystać, np.:

```
app.use(cors({
        origin: ['https://pollub.pl',
                 https://www.google.com/']
}));
```

UWAGA! Przy korzystaniu z MongoDB Atlas w pliku **.env** trzeba ustawić zmienną DB na swój connection string, np.:

```
//DB=mongodb://localhost/auth_react
DB=mongodb+srv://beatap:xxxxxxxxx@cluster0.v7ftsqz.mongodb.net/?retryWrites=true&w=majority
```

Etap 6-7 Tworzenie aplikacji klienckiej React

W aplikacji React wykorzystano:

- react-router-dom - podstawowy pakiet zawierający standardowe komponenty i funkcjonalności do implementacji routingu w aplikacjach React,
- axios – bibliotekę, która do komunikacji z serwerem (backendem) wykorzystuje interfejs API Promise (Fetch API) i służy do wysyłania żądań do API, a następnie wykonywania operacji na danych (otrzymanych w odpowiedzi z API) w aplikacji React.

OBSŁUGA ZDARZEŃ W REACT - UWAGI

Obsługa zdarzeń w React jest bardzo podobna do tej z drzewa DOM. Istnieje jednak kilka różnic w składni:

- zdarzenia w React pisane są w stylu camelCase (np. onClick), a nie małymi literami. W JSX funkcja obsługi zdarzenia przekazywana jest tylko jako nazwa funkcji w **{}**, bez operatora () (jak w zwykłym JS (przykład w tabelce poniżej).
- w React nie można zwrócić *false* w celu zapobiegnięcia wykonania domyślnej akcji. Należy jawnie wywołać **preventDefault** (przykład w tabelce poniżej)

Generalnie przycisk **<button>** posiada atrybut **type**. Jeżeli nie jest ustawiony na **type="button",** to domyślnie jest typu **submit** i wtedy standardowym zachowaniem dla tego typu jest **przeładowanie strony**. Aby uniknąć przeładowywania strony można zmienić standardowe zachowanie przycisku przez:

- wywołanie **e.preventDefault()**
- albo ustawić **type="button**

HTML i JavaScript	JSX React
Obsługa zdarzeń	
<button **onclick**="**pokazDane()**"> Pokaż dane</button>	<button **onClick={pokazDane}**> Pokaż dane</button>
Zapobieganie domyślnej akcji submit dla formularza (wysłaniu danych)	

<form onsubmit="console.log('Kliknięto Wyślij.');
return false">
<button type="submit"> Wyślij</button>
</form>
	function Form() {
	function **handleSubmit(e)** {
	e.preventDefault();
	console.log('Kliknięto Wyślij.');
	}
	return (<form **onSubmit={handleSubmit}**>
	<button type="submit">Wyślij</button>
	</form>);
	}

Przekazywanie argumentów do funkcji obsługi zdarzeń w React

Jeśli potrzebujemy przekazać do procedury obsługi zdarzenia dodatkowy parametr to można to rozwiązać np. za pomocą kodu:

<button onClick={(e) => deleteRow(id, e)}>Usuń wiersz</button>

Argument **e**, reprezentujący zdarzenie w React, zostanie przekazany jako drugi w kolejności, zaraz po identyfikatorze wiersza do usunięcia.

JAK SKORZYSTAĆ Z API ZA POMOCĄ AXIOS?

Komunikacja z API może przebiegać na kilka sposobów – XHR (najstarszy sposób), Fetch, Axios:

- **Fetch** nie korzysta z żadnych bibliotek ani rozszerzeń. Ma domyślnie ustawioną metodę GET a chcąc skorzystać z innej metody trzeba ją podać w drugim parametrze, który jest obiektem pozwalającym na skonfigurowanie zapytania.
- **Axios** jest biblioteką stworzoną na potrzeby komunikacji z API. Za użyciem Axios przemawiają drobne udogodnienia sprawiające, że korzysta się z niego wygodniej niż z Fetch. Na przykład można tworzyć osobne metody dla każdej z metod HTTP, np. axios.get(), axios.post(), które są bardziej czytelne niż w Fetch. Ponadto nie musimy korzystać z metody .json(). Inną zaletą jest możliwość ustawienia globalnej konfiguracji, takiej jak domyślny URL, nagłówki itp

Zarówno w przypadku Fetch API jak i Axios do odczytu/zapisu danych wykorzystujemy **Obietnice (Promise)**.

Fetch API	Axios
fetch('http://localhost:8080/api/userlist') **.then**(response => response.json()) **.then**(data => console.log(data));	import axios from "axios" **axios**('http://localhost:8080/api/userlist') **.then**(data => console.log(response.data));

Pewnym uproszczeniem w axios jest użycie składni **async/await** (inny sposób pracy z obiektami Promise) a pełna obsługa musi zawierać reakcję na błędy (funkcja **throwError**):

```
async function getData() {
 try {
        const response = await axios(URL);
        return response.data;
 } catch (error) {
        throwError(error);
   }
}
const throwError = message => {
                console.error(message); alert(message);
                throw Error(message);
        }
```

API obsługujące komunikację może przechwytywać błędy (nie tylko związane z transmisją) i nawet w przypadku błędu kończyć działanie z sukcesem (HTTP 200), ale zwracać dane w postaci {"errors": ..., "message": ...}. Wtedy obsługa błędów będzie postaci:

```
async function getUsers(request) {
        const response = await request();
        if (response.data.errors) {
                // Błąd zwrócony jako HTTP 200, {"errors": ..., "message": ...}
                throwError(response.data.message);
        }
        return response.data;
}
```

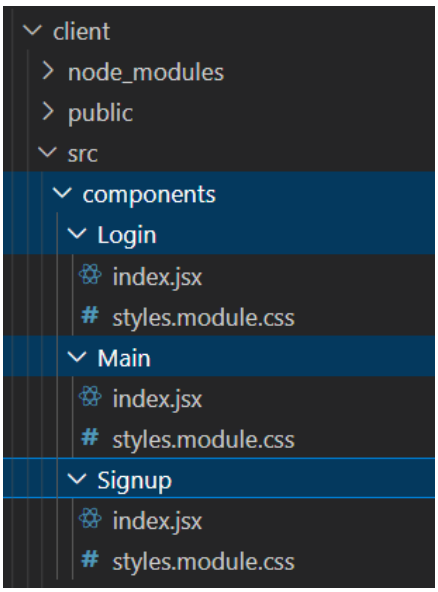
Inny sposób na stylowanie komponentów React

Do stylowania tworzonych komponentów React (**Login**, **Main**, **Signup**) zastosowano tym razem CSS modules. Taki moduł CSS to plik z regułami CSS o specjalnej nazwie ***modules.styles.css***. Sama definicja komponentu umieszczana jest wtedy w pliku o nazwie ***index.jsx***.

Zastosowanie styli z modułu CSS w komponencie (definiowanym w ***index.jsx***), polega na zaimportowaniu reguł:

```
        import styles from "./styles.module.css"
```

Każdy komponent i jego styl są umieszczone w osobnych katalogach o nazwach komponentów w folderze *components* i każdy komponent korzysta ze swojego zestawu styli:



Dzięki modułowi ***react-router-dom*** bardzo prosto można zbudować nawigację, opartą na poszczególnych komponentach. W głównym komponencie App.js zastosowano komponenty **Routes**, **Route** i **Navigate** z modułu ***react-router-dom***:

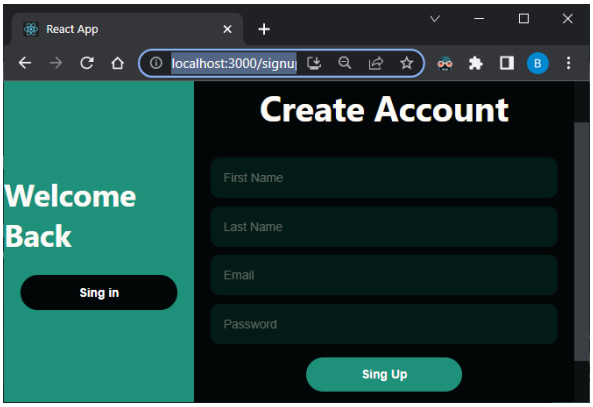
```
<Routes>
     {user && <Route path="/" exact element={<Main />} />}
     <Route path="/signup" exact element={<Signup />} />
     <Route path="/login" exact element={<Login />} />
     <Route path="/" element={<Navigate replace to="/login" />} />
</Routes>
```

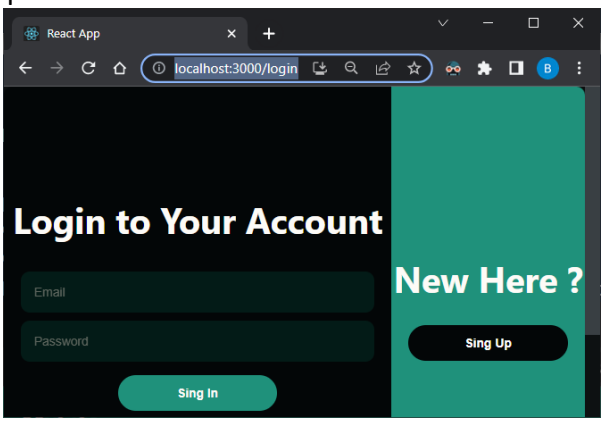
W każdym z komponentów Login, Main i Signup wysyłane są asynchroniczne żądania do aplikacji serwera (Express) w asynchronicznej metodzie (async-await) obsługującej zdarzenie onsubmit formularza za pomocą metod z modułu **axios**. Na przykład w komponencie Login zdefiniowano asynchroniczną metodę:

```
const handleSubmit = async (e) => {
        e.preventDefault()
        try {
            const url = "http://localhost:8080/api/auth"
            const { data: res } = await axios.post(url, data)
            localStorage.setItem("token", res.data)
            window.location = "/"
        } catch (error) {
            //...
        }
    }
```

Proszę zwrócić uwagę, że dopiero po otrzymaniu odpowiedzi, w localStorage (w przeglądarce) umieszczany jest token (generowany po stronie serwera w metodzie obsługującej endpoint: http://localhost:8080/api/auth, jeżeli logowanie się powiodło).

Formularz rejestracji i logowania powinien być postaci:





Po prawidłowym zalogowaniu, w localStorage powinien być zapisany token: