Programowanie w chmurze. Laboratorium 4. REST API. JSON. Komunikacja z serwerem.

REST API (Representational State Transfer Application Programming Interface) to popularny sposób komunikacji pomiędzy różnymi systemami za pomocą protokołów internetowych (np. HTTP). Jest to zbiór reguł, które pozwalają na wymianę danych pomiędzy aplikacjami w sposób prosty i zrozumiały. REST API działa na podstawie kilku podstawowych operacji, takich jak:

- **GET** – Pobieranie danych (np. danych pogodowych).
- **POST** – Wysyłanie nowych danych.
- **PUT** – Aktualizacja istniejących danych.
- **DELETE** – Usuwanie danych.

W tej karcie pracy będziemy mieć 2 zadania.

W pierwszym zadaniu stworzymy aplikację, która na podstawie wprowadzonego miasta lub aktualnej lokalizacji użytkownika pobiera dane pogodowe z OpenWeatherMap i wyświetla je w formie przyjaznej dla użytkownika. Będziemy korzystać z REST API do komunikacji z serwerem OpenWeatherMap i przetwarzać odpowiedzi w formacie JSON, aby wyświetlić informacje o pogodzie dla wybranego miasta. w naszym interfejsie.

JSON (JavaScript Object Notation) to popularny format wymiany danych, który jest łatwy do odczytania i zapisu zarówno przez ludzi, jak i maszyny. JSON jest formatem tekstowym, który wykorzystuje pary klucz-wartość (np. "name": "London") oraz zagnieżdżone obiekty i tablice.

W odpowiedzi z API OpenWeatherMap otrzymujemy dane pogodowe w formacie JSON, które zawierają informacje takie jak:

- Temperatura

- Opis pogody
- Wilgotność
- Prędkość wiatru
- Ciśnienie

Przykładowa odpowiedź JSON z API OpenWeatherMap:


```
{
  "weather": [
    {
      "description": "clear sky",
      "icon": "01d"
    }
  ],
  "main": {
    "temp": 22.5,
    "pressure": 1013,
    "humidity": 60
  },
  "wind": {
    "speed": 3.6
  },
  "sys": {
    "country": "PL"
  },
  "name": "Warsaw"
}
```


POBIERANIE DANYCH Z ZEWNĘTRZNEGO SERWERA API

OpenWeatherMap to jedno z najpopularniejszych API, które umożliwia pobieranie prognoz pogody. Oferuje ono różne funkcje, takie jak:

1. Pobieranie pogody na podstawie miasta – Za pomocą nazwy miasta, na przykład "Warszawa".

2. Pobieranie pogody na podstawie współrzędnych geograficznych – Na przykład, na podstawie długości i szerokości geograficznej.
3. Pobieranie prognoz na kilka dni do przodu.

Aby skorzystać z API OpenWeatherMap, należy:

1. Zarejestrować się na stronie OpenWeatherMap i uzyskać **klucz API** (tzw. API key).
2. Wysłać zapytanie HTTP do endpointu API z odpowiednimi parametrami (np. nazwą miasta, jednostkami miary).
3. Przetworzyć odpowiedź, która będzie w formacie JSON, aby uzyskać pożądane dane (np. temperatura, opis pogody, prędkość wiatru).
4. Wyświetlić dane w aplikacji. Po wykonaniu zapytania, API zwróci dane w formacie JSON, które możemy następnie wykorzystać w naszej aplikacji.

Przed rozpoczęciem pracy nad aplikacją, musimy zdobyć klucz API do OpenWeatherMap. Oto jak to zrobić:

1. Wejdź na stronę https://openweathermap.org/ i kliknij przycisk "Sign Up".
2. Wypełnij formularz rejestracyjny, podając swój adres e-mail i tworząc hasło.
3. Po zarejestrowaniu się, zaloguj się na swoje konto.
4. Przejdź do sekcji "API keys" na swoim koncie. Znajdziesz tam domyślny klucz API.
5. Skopiuj ten klucz - będziesz go potrzebować w swojej aplikacji w kroku 4.

   W linijce **const apiKey = 'twój_rzeczywisty_klucz_api'**; należy wstawić swój rzeczywisty klucz API:

Należy pamiętaj, że klucz API jest poufny i nie powiniśmy go udostępniać publicznie. W prawdziwej aplikacji, dostępnej publicznie najlepiej przechowywać go w zmiennych środowiskowych.

1. Utwórz nowy projekt React:

   npx create-react-app pogoda-app cd

   pogoda-app

2. W pliku src/App.js zdefiniujmy podstawową strukturę.
   import React, { useState, useEffect } from 'react'; import
   './App.css';

   function App() { **return**
   (

   <h1>Prognoza pogody</h1>

   );

```
}

export default App;
```

3. Następnie zmodyfikujmy komponent App, dodając stan dla miasta za pomocą useState, aby przechować wprowadzone przez użytkownika dane i pole input umożliwiające użytkownikowi wprowadzenie nazwy miasta. Stan zapewnia kontrolowanie tej wartości.

```
function App() {
  const [city, setCity] = useState('');

  return (
    <div className="App">
      <h1>Prognoza pogody</h1>
      <input
type="text"
        value={city}
        onChange={(e) => setCity(e.target.value)}
        placeholder="Wpisz nazwę miasta"
      />
      <button>Sprawdź pogodę</button>
    </div>
  );
}
```

4. Zaimplementujmy funkcję getWeather i spróbujmy powiązać ją z przyciskiem.Funkcja getWeather odpowiedzialna będzie za pobieranie danych pogodowych z API OpenWeatherMap. Po kliknięciu przycisku "Sprawdź pogodę", zapytanie jest wysyłane do API z nazwą miasta.

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

const apiKey = 'TWÓJ_KLUCZ_API';

function App() {
  const [city, setCity] = useState('');
  const [weatherData, setWeatherData] = useState(null);

  const getWeather = async () => {
    if (!city.trim()) {
      alert('Proszę wpisać nazwę miasta');
return;
    }

    try {
      const response = await fetch(
```

```
`https://api.openweathermap.org/data/2.5/weather?q=${city.trim()}&appid=${apiKey}&units=
metric&lang=pl`
    );

    if (!response.ok) {
      throw new Error(`Błąd HTTP: ${response.status}`);
    }

    const data = await response.json();

    // Sprawdzenie, czy dane pogodowe są poprawne      if
(!data || !data.weather || data.weather.length === 0) {
alert('Nie udało się pobrać danych pogodowych.');
return;
    }

    setWeatherData(data);
   } catch (error) {
    console.error('Błąd:', error);
    alert(error.message || 'Wystąpił błąd podczas pobierania danych');
   }
  };

  return (
   <div className="App">
    <h1>Prognoza pogody</h1>
    <input
type="text"
      value={city}
      onChange={(e) => setCity(e.target.value)}
      placeholder="Wpisz nazwę miasta"
    />
    <button onClick={getWeather}>Sprawdź pogodę</button>

//tu dodamy kod z punktu 6
   </div>
  );
}

export default App;
```

Funkcja sprawdza, czy użytkownik wpisał nazwę miasta. Jeśli tak, wykonuje zapytanie HTTP do API i przetwarza odpowiedź, zapisując dane do stanu weatherData. W przypadku błędu wyświetla alert.

5. (Opcjonalnie) Dodajmy klucz API do zmiennych środowiskowych. W głównym katalogu projektu (tam, gdzie znajduje się plik package.json), za pomocą terminalu utwórz nowy plik o nazwie .env.

- Użyj polecenia ***touch .env***
- Otwórz plik .env w edytorze tekstu i dodaj następującą linię: ***REACT_APP_OPENWEATHERMAP_API_KEY=twój_rzeczywisty_klucz_api***
- Zastąp twój_rzeczywisty_klucz_api kluczem, który otrzymałeś od OpenWeatherMap.
- Ważne: Dodaj plik .env do .gitignore, aby uniknąć przypadkowego udostępnienia klucza w repozytorium: ***echo ".env" >> .gitignore***
- Teraz w pliku App.js możesz użyć klucza API w następujący sposób: ***const apiKey = process.env.REACT_APP_OPENWEATHERMAP_API_KEY;***

Pamiętaj, że po dodaniu lub zmianie zmiennych środowiskowych musisz zrestartować serwer deweloperski. W React, wszystkie zmienne środowiskowe muszą zaczynać się od REACT_APP_, aby były dostępne w kodzie aplikacji. Jeśli używasz systemu kontroli wersji (np. Git), nigdy nie commituj pliku .env. Zamiast tego, możesz utworzyć plik .env.example z przykładowymi kluczami (bez rzeczywistych wartości) i dodać go do repozytorium.

Przy deployowaniu aplikacji na serwer produkcyjny, musimy skonfigurować zmienne środowiskowe na serwerze. Sposób konfiguracji zależy od używanej platformy hostingowej. Stosując te praktyki, zwiększamy bezpieczeństwo swojej aplikacji, chroniąc wrażliwe dane, takie jak klucze API, przed nieautoryzowanym dostępem.

6. Dodajmy teraz sekcję wyświetlającą dane pogodowe:

```
{weatherData && (
  <div id="weatherInfo">
    <h2>{weatherData.name}, {weatherData.sys.country}</h2>
    <p>{weatherData.weather[0].description}</p>
    <p>Temperatura: {weatherData.main.temp}°C</p>
    <p>Ciśnienie: {weatherData.main.pressure} hPa</p>
    <p>Wilgotność: {weatherData.main.humidity}%</p>
    <p>Prędkość wiatru: {weatherData.wind.speed} m/s</p>   </div>
)}
```

Po pomyślnym pobraniu danych pogodowych z API, aplikacja wyświetla szczegóły na temat pogody w danym mieście. **Funkcja** pokazuje nazwę miasta, jego kraj, opis pogody, temperaturę, ciśnienie, wilgotność, prędkość wiatru. Używamy operatora && do warunkowego renderowania sekcji z danymi pogodowymi tylko wtedy, gdy dane pogodowe są dostępne.

7. Następnie dodajmy obiekt weatherConditions by zmodyfikować komponent, aby używać odpowiednich kolorów i ikon dla różnych warunków pogodowych (np. deszcz, śnieg, burza).

   Dodajmy na samej górze pliku App.js
   import { WiDaySunny, WiRain, WiSnow, WiCloudy, WiThunderstorm, WiMist } from 'reacticons/wi';

8. Następnie dodajmy: const weatherConditions = {
   Thunderstorm: { color: '#616161',
     title: 'Burza',

```
    subtitle: 'Uważaj na błyskawice!',
    icon: <WiThunderstorm size={64} />
  },
  Drizzle: {     color:
'#0044CC',     title:
'Mżawka',     subtitle:
'Lekkie opady',
    icon: <WiRain size={64} />
  },
  Rain: {     color:
'#005BEA',     title:
'Deszcz',     subtitle: 'Weź
parasol',
    icon: <WiRain size={64} />
  },
  Snow: {     color:
'#00d2ff',     title:
'Śnieg',
    subtitle: 'Ubierz się ciepło',
icon: <WiSnow size={64} />
  },
  Clear: {     color: '#f7b733',
title: 'Słonecznie',     subtitle:
'Idealna pogoda!',
    icon: <WiDaySunny size={64} />
  },
  Clouds: {     color:
'#1F1C2C',     title:
'Pochmurno',
    subtitle: 'Może przejaśni się później',
    icon: <WiCloudy size={64} />
  },
  Mist: {     color:
'#3CD3AD',
    title: 'Mgła',
    subtitle: 'Uważaj na drodze',
    icon: <WiMist size={64} />
  }
};
```

9. Dalsza część kodu po naszej modyfikacji może wyglądać mniej więcej tak:

```
function App() {
  const [city, setCity] = useState('');
  const [weatherData, setWeatherData] = useState(null);

  const getWeather = async () => {
```

```
   if (!city.trim()) {
     alert('Proszę wpisać nazwę miasta');
return;
   }

   try {
     const response = await fetch(

`https://api.openweathermap.org/data/2.5/weather?q=${city.trim()}&appid=${apiKey}&units=
metric&lang=pl`
     );

     if (!response.ok) {
       throw new Error(`Błąd HTTP: ${response.status}`);
     }

     const data = await response.json();

     if (!data || !data.weather || data.weather.length === 0) {
alert('Nie udało się pobrać danych pogodowych.');         return;
     }

     setWeatherData(data);
   } catch (error) {
     console.error('Błąd:', error);
     alert(error.message || 'Wystąpił błąd podczas pobierania danych');
   }
 };

 const condition = weatherData
   ? weatherConditions[weatherData.weather[0].main] || weatherConditions.Clear     :
weatherConditions.Clear;

 return (
   <div className="App" style={{ backgroundColor: condition.color }}>
     <h1>{condition.title}</h1>
     <h2>{condition.subtitle}</h2>

     <input
type="text"
value={city}
       onChange={(e) => setCity(e.target.value)}
       placeholder="Wpisz nazwę miasta"
     />
     <button onClick={getWeather}>Sprawdź pogodę</button>
```

```
      {weatherData && (
        <div id="weatherInfo">
          <h2>{weatherData.name}, {weatherData.sys.country}</h2>
          <p>{condition.icon} {weatherData.weather[0].description}</p>
          <p>Temperatura: {weatherData.main.temp}°C</p>
          <p>Ciśnienie: {weatherData.main.pressure} hPa</p>
          <p>Wilgotność: {weatherData.main.humidity}%</p>
          <p>Prędkość wiatru: {weatherData.wind.speed} m/s</p>
        </div>
      )}
    </div>
  );
}

export default App;
```

Aplikacja używa obiektu weatherConditions, aby odpowiednio wyświetlić informacje o pogodzie, w tym tytuł, opis, ikonę oraz inne dane pogodowe (temperatura, ciśnienie, wilgotność). Przy czym aplikacja zmienia kolor tła oraz wyświetla odpowiednią ikonę dla każdego stanu pogody na podstawie aktualnych warunków pogodowych.

10. Na końcu zajmijmy się stylowaniem CSS w pliku src/App.css:

```
.App {   text-align:
center;   min-height:
100vh;   display: flex;
flex-direction: column;
align-items: center;
  justify-content: center;
  font-family: Arial, sans-serif;
  transition: background-color 0.5s ease;
}
@media (max-width: 600px) {
 #weatherInfo { width: 90%; font-size: 14px; }  input,
button { width: 80%; font-size: 16px; } }

input,  button  {
margin:    10px;
padding: 5px;
}

#weatherInfo {   margin-top:
20px;
  background-color: rgba(255, 255, 255, 0.8);   padding:
20px;
  border-radius: 10px;
```

```
}
```

Co ciekawe, dzięki transition: background-color 0.5s ease dodajemy efekt płynnej zmiany koloru tła w czasie 0.5 sekundy, co przydaje się, gdy tło zmienia się na podstawie warunków pogodowych.

11. Możemy również dodać w projekcie możliwość automatycznego pobierania pogody dla aktualnej lokalizacji użytkownika:

```
const getLocationWeather = () => {   if
(navigator.geolocation) {
   navigator.geolocation.getCurrentPosition(
    (position) => {
      const { latitude, longitude } = position.coords;
fetch(

`https://api.openweathermap.org/data/2.5/weather?lat=${latitude}&lon=${longitude}&appid=
${apiKey}&units=metric&lang=pl`
      )
        .then(response => response.json())
        .then(data => setWeatherData(data))
        .catch(error => console.error(error));
    },
    (error) => {
      alert(`Błąd geolokalizacji: ${error.message}`);
    }
   );
 } else {
   alert('Twoja przeglądarka nie wspiera geolokalizacji');
 }
};
```

Jeśli użytkownik odmówi dostępu do geolokalizacji, warto obsłużyć ten przypadek bardziej przyjaźnie:

```
(error) => {
 if (error.code === error.PERMISSION_DENIED) {
   alert('Aby pobrać pogodę dla Twojej lokalizacji, musisz zezwolić na dostęp do lokalizacji.');
 } else {
   alert(`Błąd geolokalizacji: ${error.message}`);
 }
}
```

Funkcja automatycznego pobierania pogody na podstawie aktualnej lokalizacji użytkownika działa w oparciu o.

- navigator.geolocation — sprawdzamy, czy przeglądarka obsługuje geolokalizację. Jeśli tak, używa metody getCurrentPosition do uzyskania lokalizacji użytkownika.

- Jeśli uda się uzyskać współrzędne (latitude i longitude), funkcja wysyła zapytanie do API OpenWeatherMap, aby pobrać dane pogodowe dla tej lokalizacji.
- Dane pogodowe są następnie zapisywane w stanie aplikacji za pomocą setWeatherData(data).
- Jeśli wystąpi błąd podczas pobierania danych, jest on logowany w konsoli (console.error).
- Jeśli przeglądarka nie wspiera geolokalizacji, użytkownik otrzymuje komunikat: Twoja przeglądarka nie wspiera geolokalizacji.
- Obsługuje sytuację, gdy użytkownik odmówił dostępu do swojej lokalizacji lub wystąpił inny błąd geolokalizacji.
- if (error.code === error.PERMISSION_DENIED) — sprawdzamy, czy użytkownik odmówił dostępu do lokalizacji. Jeśli tak, wyświetla komunikat: Aby pobrać pogodę dla Twojej lokalizacji, musisz zezwolić na dostęp do lokalizacji. Jeśli wystąpił inny błąd geolokalizacji, wyświetlany jest ogólny komunikat z błędem: Błąd geolokalizacji: ${error.message}.

WŁASNY SERWER REST API

Skoro już wiemy jak pobierać dane z serwera zdalnego REST API, stworzymy prosty serwer REST API w Node.js przy użyciu Express, który będzie serwował listę albumów takich zespołów jak Metallica, AC/DC i Iron Maiden. Nie będziemy używać bazy danych – dane będą przechowywane w pliku JSON lub jako obiekt w kodzie.

1. W środowisku Node.js, jeśli nie masz zainstalowanego Expressa, zainstaluj go:
   npm install express

2. Utwórzmy plik server.js i wklej do niego powyższy kod. Najpierw importujemy Express i konfigurujemy aplikację. Definiujemy tablicę albums z przykładowymi danymi. Użycie express.json() jako middleware do parsowania JSON-a w żądaniach POST/PUT jest standardową praktyką.

```
// Importujemy Express const
express = require('express'); const
app = express();
const port = 3000;

// Dane - lista albumów const
albums = [
  { id: 1, band: "Metallica", title: "Master of Puppets", year: 1986 },
  { id: 2, band: "Metallica", title: "Ride the Lightning", year: 1984 },
  { id: 3, band: "AC/DC", title: "Back in Black", year: 1980 },
  { id: 4, band: "AC/DC", title: "Highway to Hell", year: 1979 },
  { id: 5, band: "Iron Maiden", title: "The Number of the Beast", year: 1982 },
{ id: 6, band: "Iron Maiden", title: "Powerslave", year: 1984 }
];
```

```
// Middleware do obsługi JSON
app.use(express.json());
```

3. Dodajmy teraz endpointy na koniec tego pliku:

```
// Endpoint do pobierania albumów konkretnego zespołu
app.get('/albums/:band', (req, res) => {    const
band = req.params.band.toLowerCase();
  const filteredAlbums = albums.filter(album => album.band.toLowerCase() === band);

  if (filteredAlbums.length > 0) {
    res.json(filteredAlbums);
  } else {
    res.status(404).json({ message: "Nie znaleziono albumów dla tego zespołu." });
  }
});

// Endpoint do dodawania nowego albumu
app.post('/albums', (req, res) => {
  const { band, title, year, genre, cover } = req.body;
  if (!band || !title || !year || !genre || !cover) {
    return res.status(400).json({ message: "Brak wymaganych danych." });
  }
  const newAlbum = { id: albums.length + 1, band, title, year, genre, cover };
  albums.push(newAlbum);
  res.status(201).json(newAlbum);
});

// Endpoint do aktualizacji albumu
app.put('/albums/:id', (req, res) => {
const id = parseInt(req.params.id);    const
{ title, genre, cover } = req.body;
  const album = albums.find(album => album.id === id);    if
(!album) {
    return res.status(404).json({ message: "Album nie znaleziony." });
  }
  if (title) album.title = title;    if
(genre) album.genre = genre;    if
(cover) album.cover = cover;
res.json(album);
});

// Endpoint do usuwania albumu
app.delete('/albums/:id', (req, res) => {
const id = parseInt(req.params.id);
```

```
  const index = albums.findIndex(album => album.id === id);    if
(index === -1) {
      return res.status(404).json({ message: "Album nie znaleziony." });
  }
  albums.splice(index, 1);
  res.json({ message: "Album usunięty." });
});

// Uruchomienie serwera app.listen(port,
() => {
  console.log(`Serwer REST API działa na http://localhost:${port}`);
});
```

Serwer nasłuchuje na porcie 3000 i obsługuje żądania GET dla obu endpointów. Co robią poszczególne endpointy?

- **GET /albums** zwraca wszystkie albumy.
- **GET /albums/:band** filtruje albumy po nazwie zespołu i działa poprawnie, zwracając 404, jeśli brak albumów danego zespołu.
- **POST /albums** dodaje nowy album do tablicy i zwraca go po dodaniu.
- **PUT /albums/:id** umożliwia edytowanie tytułu albumu, ale warto by było dodać możliwość zmiany innych danych (np. rok, gatunek, okładka).

4. Uruchom serwer poleceniem **node server** oraz otwórz przeglądarkę lub użyj narzędzia typu Postman, aby sprawdzić dostępne endpointy:

http://localhost:3000/albums – lista wszystkich albumów
http://localhost:3000/albums/metallica – albumy Metallicy.
Zanotuj wyniki w sprawozdaniu.

5. Stwórzmy teraz nowy projekt React npx create-react-app my-album-app
cd my-album-app

6. W pliku src/App.js dodaj poniższy kod:

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

export default function AlbumList() {    const
[albums, setAlbums] = useState([]);    const
[band, setBand] = useState("");
  const [newAlbum, setNewAlbum] = useState({ band: "", title: "", year: "" });

  useEffect(() => {
```

```
    const storedAlbums = localStorage.getItem("albums");
if (storedAlbums) {
      setAlbums(JSON.parse(storedAlbums));
    } else {
      fetch("http://localhost:3000/albums")
        .then(response => response.json())
        .then(data => {
setAlbums(data);
          localStorage.setItem("albums", JSON.stringify(data));
        })
        .catch(error => console.error("Błąd pobierania danych:", error));
    }
  }, []);

  const updateLocalStorage = (updatedAlbums) => {      setAlbums(updatedAlbums);
    localStorage.setItem("albums", JSON.stringify(updatedAlbums));
  };

  const fetchBandAlbums = () => {
    fetch(`http://localhost:3000/albums/${band}`)
      .then(response => response.json())
      .then(data => updateLocalStorage(data))
      .catch(error => console.error("Błąd pobierania danych:", error));
  };

  const addAlbum = () => {
    fetch("http://localhost:3000/albums", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
body: JSON.stringify(newAlbum)
    })
    .then(response => response.json())
    .then(data => updateLocalStorage([...albums, data]))
    .catch(error => console.error("Błąd dodawania albumu:", error));
  };

  const updateAlbum = (id, newTitle) => {
fetch(`http://localhost:3000/albums/${id}`,      {
method: "PUT",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ title: newTitle })
  })
  .then(response => response.json())
  .then(() => {
    const updatedAlbums = albums.map(album => album.id === id ? { ...album, title: newTitle }
```

```
: album);
      updateLocalStorage(updatedAlbums);
    })
    .catch(error => console.error("Błąd aktualizacji albumu:", error));
};
    const deleteAlbum = (id) => {
      fetch(`http://localhost:3000/albums/${id}`, {
        method: "DELETE"
      })
      .then(() => {
        const updatedAlbums = albums.filter(album => album.id !== id);
updateLocalStorage(updatedAlbums);
      })
      .catch(error => console.error("Błąd usuwania albumu:", error));
    };

    return (
      <div className="p-4 max-w-lg mx-auto">
        <h1 className="text-xl font-bold mb-4">Lista Albumów</h1>
        <div className="flex gap-2 mb-4">
          <input
            type="text"
            placeholder="Wpisz nazwę zespołu"
value={band}
            onChange={(e) => setBand(e.target.value)}
className="border p-2 rounded w-full"
          />
          <button
            onClick={fetchBandAlbums}
            className="bg-blue-500 text-white px-4 py-2 rounded"
          >
            Szukaj
          </button>
        </div>
        <div className="mb-4">
          <input type="text" placeholder="Zespół" value={newAlbum.band} onChange={(e) =>
setNewAlbum({ ...newAlbum, band: e.target.value })} className="border p-2 rounded w-full mb-2"
/>
          <input type="text" placeholder="Tytuł" value={newAlbum.title} onChange={(e) =>
setNewAlbum({ ...newAlbum, title: e.target.value })} className="border p-2 rounded w-full mb2" />
          <input type="text" placeholder="Rok" value={newAlbum.year} onChange={(e)
=> setNewAlbum({ ...newAlbum, year: e.target.value })} className="border p-2 rounded w-
full mb-2" />
          <button onClick={addAlbum} className="bg-green-500 text-white px-4
py-2 rounded">Dodaj Album</button>
        </div>
```

```
        <ul>
          {albums.map(album => (
            <li key={album.id} className="border-b py-2 flex justify-between items-center">
              <span>
                <strong>{album.band}</strong> - {album.title} ({album.year})
              </span>
              <div>
                <button onClick={() => updateAlbum(album.id)} className="bg-yellow-500
textwhite px-2 py-1 rounded mr-2">Edytuj</button>
                <button onClick={() => deleteAlbum(album.id)} className="bg-red-500 textwhite px-
2 py-1 rounded">Usuń</button>
              </div>
            </li>
          ))}
        </ul>
      </div>
    );
}
```

Aplikacja obsługę teraz operacje GET, **POST, PUT i DELETE**, aby umożliwić dodawanie, edytowanie i usuwanie albumów. Jeśli dane nie są dostępne w localStorage, są pobierane z serwera, a następnie zapisywane w localStorage dla lepszej wydajności.

- **GET** – pobieranie albumów
- **POST** – dodawanie nowego albumu
- **PUT** – aktualizowanie tytułu albumu
- **DELETE** – usuwanie albumu

Także w tym pliku mamy renderowanie interfejsu użytkownika do interakcji z listą albumów.

7. Uruchom projekt za pomocą polecenia: **npm start** oraz otwórz w przeglądarce adres http://localhost:3000. Przetestuj dodawanie, edycję i usuwanie albumów.

9 Zmodyfikuj kod według własnego uznania np. tak by mógł zawierać kategorie muzyczne oraz okładki płyt.