POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: SYSTEMY I SIECI KOMPUTEROWE

# PRACA DYPLOMOWA INŻYNIERSKA

Projekt strony internetowej do recenzji albumów muzycznych w technologii ASP.NET Core oraz grafowej bazie danych

Design of an internet website for music albums reviewing based on ASP.NET Core and graph database

AUTOR:
Marcin Kotas

PROWADZĄCY PRACĘ:
dr inż. Mariusz Topolski, W4/K2

OCENA PRACY:

WROCŁAW, 2019

# Spis treści

# Spis rysunków

# Spis tabel

# Skróty

**OGC** (ang. *Open Geospatial Consortium*)
**XML** (ang. *eXtensible Markup Language*)
**SOAP** (ang. *Simple Object Access Protocol*)
**WSDL** (ang. *Web Services Description Language*)
**UDDI** (ang. *Universal Description Discovery and Integration*)
**GIS** (ang. *Geographical Information System*)
**SDI** (ang. *Spatial Data Infrastructure*)
**ISO** (ang. *International Standards Organization*)
**WMS** (ang. *Web Map Service*)
**WFS** (ang. *Web Feature Service*)
**WPS** (ang. *Web Processing Service*)
**GML** (ang. *Geography Markup Language*)
**SRG** (ang. *Seeded Region Growing*)
**SOA** (ang. *Service Oriented Architecture* )
**IT** (ang. *Information Technology* )

# Rozdział 1

# Wprowadzenie

## 1.1. Cel i zakres pracy

Celem pracy jest implementacja strony internetowej pozwalającej dodawać recenzje albumów muzycznych, wykorzystując najnowsze technologie z tej dziedziny zgodne z obecnymi trendami.

Główny nacisk nałożony zostanie na stworzenie architektury stanowiącej solidną bazę do dalszego rozwijania aplikacji. Projekt wykorzystywać będzie tylko oprogramowanie open source.

Zakres pracy obejmuje:

- analiza
- opracowanie struktury aplikacji
- implementacja aplikacji
- testy

## 1.2. Układ pracy

## 1.3. Przegląd technologii

### 1.3.1. Paradygmaty komunikacji z usługami sieciowymi

**RPC** to prosty protokół zdalnego wywołania procedury. Klient wysyła do serwera potrzebne dane do wywołania akcji: nazwę metody oraz parametry. W podstawowej wersji całość komunikacji odbywa się za pomocą jednego punktu dostępowego (ang. *endpoint*). Zwykle wykorzystywane są wariacje protokołu np. JSON-RPC, które przesyłają dane w formacie JSON. Przykładowe wywołanie oraz odpowiedź:

```
{
  "jsonrpc": "2.0",
  "method": "divide",
  "params": {
    "dividend": 32,
    "divisor": 4
  },
  "id": 5
}
```

```
{
  "jsonrpc": "2.0",
  "result": 8,
  "id": 5
}
```

**SOAP** to ustandaryzowany protokół służący do opakowywania przesyłanych danych. Często porównywany do wkładania danych do „koperty”, definiuje sposób kodowania i dekodowania

danych do formatu XML [7]. Paczka z danymi zawiera również informacje opisujące dane oraz w jaki sposób mają być przetworzone. Wykorzystuje przez to więcej zasobów niż czysty JSON z danymi – zarówno w kwestii rozmiaru przesyłanych danych jak i czasu ich przetworzenia.
Opcjonalny lecz zalecany jest dokument WSDL, który w ustandaryzowany sposób określa punkty dostępowe usług serwisu. Umożliwia automatyczne generowanie testów serwisu oraz funkcji wywołujących usługi.
Zapytanie w SOAP jest zawsze typu POST, a więc odpowiedź nie jest zapisywana przez przeglądarkę i za każdym razem wywoływana jest procedura w Api. Jest jednak dzięki temu bardziej zaawansowany - zapewnia wsparcie dla transakcji spełniających ACID oraz posiada rozszerzone mechanizmy bezpieczeństwa – oprócz SSL, implementuje również WS-Security. Te właściwości są istotnymi zaletami wykorzystywanymi w aplikacjach typu enterprise.
Przykładowa budowa zapytania SOAP [4]:

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

oraz odpowiedź:

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

**REST** nie jest protokołem lecz stylem architektury. Nie narzuca konkretnej implementacji ani specyfikacji. Definiuje za to wymagania, które powinno spełnić Api typu RESTful [6]:

1. Klient-Serwer: podział ról i odpowiedzialności – klient i serwer mogą być rozwijane niezależnie, o ile interfejs pozostanie bez zmian.
2. Bezstanowość: wszystkie informacje potrzebne to wykonania żądania zawarte są w zapytaniu – jego wykonanie jest niezależne od poprzedniego stanu serwera.
3. Pamięć podręczna: klient jest w stanie zapisać odpowiedź serwera w pamięci podręcznej aby ograniczać zapytania do serwera.
4. System warstwowy: klient nie musi mieć bezpośredniego połączenia z serwerem – pomiędzy może być zaimplementowany load-balancer, proxy lub inny serwis, który jest niewidoczny ani dla serwera ani klienta.
5. Kod na żądanie (opcjonalne): Serwer jest w stanie przesłać wykonywalny kod na żądanie klienta, np. skrypty w języku JavaScript.
6. Jednolity interfejs: określa wymagania dotyczące interfejsu między klientem a serwerem:

- zasoby są identyfikowane unikalnym URI
- serwer opisuje dane w postaci reprezentacji, klient modyfikuje dane poprzez przesłanie reprezentacji
- każda wiadomość zawiera informacje, w jaki sposób przetworzyć dane
- HATEOAS (ang. *Hypermedia as the Engine of Application State*) – serwer przesyła linki do akcji, które mogą zostać wykonane dla danego stanu zasobu

Większość developerów pomija jednak implementowanie HATEOAS, ponieważ wymaga dużego nakładu pracy zarówno po stronie serwera jak i klienta i często nie jest potrzebne dla prostych zapytań typu CRUD.
Przykładowa odpowiedź na zapytanie `GET /accounts/12345` [3]:

```
{
    "account": {
        "account_number": 12345,
        "balance": {
            "currency": "usd",
            "value": 100.00
        },
        "links": {
            "deposit": "/accounts/12345/deposit",
            "withdraw": "/accounts/12345/withdraw",
            "transfer": "/accounts/12345/transfer",
            "close": "/accounts/12345/close"
        }
    }
}
```

W odpowiedzi zawarta jest lista `links` — są to akcje dostępne dla danego stanu konta. Jeśli stan konta byłby na minusie, część z tych akcji nie pojawiła by się w odpowiedzi. Takie działanie API wymaga jednak zaimplementowania maszyny stanów na serwerze.

**GraphQL** to język zapytań dla API oraz środowisko do wykonywania tych zapytań na istniejących danych [1]. Specyfikacja nie określa w jaki sposób ma być zbudowane Api ale oferuje następujące zasady projektowania[2]:

- hierarchiczna struktura zapytań — pola są zagnieżdżone, a zapytanie ma taki sam kształt jak dane które zwraca
- budowa GraphQL jest determinowana potrzebami aplikacji klienckiej
- silne typowanie — w schemacie każde pole ma zdefiniowany typ, który jest walidowany
- zapytania określane przez klienta — serwer GraphQL określa schemat wszystkich możliwych zapytań, a klient sam określa, które pola ma zwrócić
- introspekcyjny — język GraphQL umożliwia odpytywanie serwera o jego schemat i typy

Język zapytań GraphQL (ang. *Query Language*) znacznie różni się od SQL. Do pobierania danych zamiast *SELECT* używany jest typ *Query*, natomiast *INSERT*, *UPDATE* oraz *DELETE* zawierają się w jednym typie: *Mutatation*. Dodatkowo zdefiniowany jest typ *Subscription*, który używany jest do nasłuchiwania zmian przez gniazda (ang. *socket*). Przykładowe zapytanie GraphQL pokazano poniżej:

```
query getAlbum($id: ID!) {
  album(id: "7803c5ee938249daa7ed67574c13e389") {
    id
    title
    albumArtist {
      name
    }
  }
}
```

Odpowiedź z serwera na takie zapytanie ma analogiczną strukturę:

```
{
  "data": {
    "album": {
      "id": "7803c5ee938249daa7ed67574c13e389",
      "title": "Until the Quiet Comes",
      "albumArtist": {
        "name": "Flying Lotus"
      }
    }
  }
}
```

Aby wykonanie takiego zapytania było możliwe, serwer musi zadeklarować odpowiednie typy. Typ reprezentuje obiekt odpowiadający jakiejś funkcji aplikacji [5]. W kontekście strony muzycznej do typami byłyby między innymi albumy czy artyści. Typ składa się z pól, które odpowiadają danym obiektu. Każde pole zwraca określony typ danych.
Typy mogą być skalarne lub obiektowe: typ skalarny zawsze zwraca wartość, natomiast typ obiektowy złożony jest z kolejnych pól. Wbudowane typy skalarne to Int, Float, String, Boolean, ID. Możliwe jest tworzenie dodatkowych typów skalarnych np. DateTime, aby zapewnić odpowiednie formatowanie i walidację danych. Dodatkowo typy mogą zostać oznaczone wykrzyknikiem, co znaczy że zawsze zwrócą wartość (ang. *non-nullable*).
Przykładowy schemat, który umożliwi wykonanie powyższego zapytania wygląda następująco:

```
type Album {
  id: ID!
  title: String!
  albumArtist: Artist!
  releaseDate: DateTime
  averageRating: Float
}

type Artist {
  id: ID!
  albums: [Album!]!
  name: String!
}

type Query {
  album(id: ID!): Album
  artist(id: ID!): Artist
}
```

Wszystkie zapytania wywoływane są na jednym punkcie dostępowym (np. `/api/graphql`), przez co nie jest możliwe proste zapisywanie odpowiedzi HTTP serwera. Wymusza to implementację bardziej zaawansowanych mechanizmów cache po stronie klienta, np. poprzez identyfikacje częściowych obiektów po ID i uzupełnianie ich danych w pamięci podręcznej.

Żaden z tych protokołów nie jest najlepszy pod każdym aspektem i wszystkie wciąż mają swoje zastosowania.

RPC jest najczęściej używane w API nastawionych na akcje. W sytuacjach, gdzie często wywołanie metody nie wpływa na stan zasobu RPC jest wystarczające.

SOAP rozbudowuje RPC zapewniając bezpieczeństwo i transakcje. Istnieją jednak nowsze rozwiązania takie jak OData bazujące na REST, które mają znacznie lepszą wydajność i kompatybilność.

REST jest najbardziej zaawansowaną architekturą (jeśli zaimplementowana w pełni) i jednocześnie najczęściej spotykaną. Pozwala na szybkie operacje typu CRUD oraz zaawansowaną

nawigację przez HATEOAS. Dodatkowo dzięki unikalnym URI dla każdego zasobu umożliwia łatwe stworzenie skutecznego cache.

GraphQL umożliwia stworzenie uniwersalnego API dla wielu klientów zapewniając wysoką wydajność. Definiuje jednoznaczną specyfikację zapewniającą silne typowanie oraz introspekcję w przeciwieństwie do REST, do którego powstało mnóstwo różniących się implementacji.

### 1.3.2. Platformy do tworzenia SPA

**Angular**
**React**
**Vue**

### 1.3.3. Platformy do tworzenia API

**ASP.Net Core**
**Node.js**
**Django**

# Rozdział 2

# Analiza wymagań projektowych

## 2.1. Przegląd istniejących rozwiązań

Na rynku obecnych jest wiele stron oferujących podobną funkcjonalność:

- Rate Your Music
- AllMusic
- Discogs

## 2.2. Wymagania funkcjonalne

1. System zawiera katalog albumów
2. Każdy album można oceniać oraz dodawać recenzję
3. Istnieje możliwość dodawania nowych albumów
4. Wyszukiwanie albumów korzysta z bazy zewnętrznego serwisu
5. Możliwość rejestracji i logowania

## 2.3. Wymagania niefunkcjonalne

1. Aplikacja powinna być obsługiwana przez obecne wersje przeglądarek
2.

## 2.4. Założenia projektowe

Całość aplikacji zaprojektowana zostanie ze wsparciem platformy Docker. Każda odrębna część systemu zamknięta będzie we własnym wirtualnym kontenerze:

1. strona internetowa typu Single-Page Application:
   Architektura SPA pozwala tworzyć strony, które w swoim działaniu bardziej przypominają tradycyjne aplikacje komputerowe. Podczas interakcji użytkownika ze stroną fragmenty widoku są dynamicznie odświeżane zamiast przeładowywania całej strony. Dodatkowo strona powinna przechowywać w pamięci zapytania do serwera aby minimalizować czas oczekiwania na dane.
2. serwer uwierzytelniający użytkowników:
   Strona internetowa działa po stronie klienta, przez co możliwa jest znaczna ingerencja w dane. Aby minimalizować zagrożenia, zaimplementowane zostanie uwierzytelnianie

użytkowników wykorzystujące bezpieczny protokół. Wykorzystany standard powinien zapewnić zarówno uwierzytelnianie jak i autoryzację.

3. serwer dostępowy do danych aplikacji:
   Grafowy dostęp do bazy danych zaimplementowany zostanie za pomocą technologii GraphQL. Serwer umożliwiać będzie pobieranie danych stosując zapytania w języku GraphQL. Dzięki temu relacje między danymi przedstawione są w postaci grafu, co pozwala formować skompilowane zapytania bez potrzeby tworzenia dedykowanych kontrolerów i modeli DTO po stronie API.
4. baza danych:
   Baza danych powinna umożliwiać przechowywanie relacji między obiektami w taki sposób, aby możliwe było reprezentowanie danych w postaci grafu.
5. serwer dostarczający odwrócone proxy:
   Z racji tego, że użytkownicy będą przekierowywani między główną stroną, a stroną do uwierzytelniania, adresy serwerów zarejestrowane będą w odwróconym proxy. Będzie to również główny punkt dostępu do aplikacji, który będzie kierował ruchem. Dodatkowo, będzie obsługiwał szyfrowanie przesyłanych danych protokołem HTTPS.

# Rozdział 3

# Projekt aplikacji

## 3.1. Strona internetowa

Do stworzenia strony zdecydowano się użyć frameworku Angular 8. Serwowana będzie ze środowiska uruchomieniowego Node.js. Projekt zbudowany jest z osobnych modułów dla każdej funkcjonalności. Widoki składane są z komponentów — są to elementy zawierające szablon HTML, style CSS oraz logikę i dane w klasie napisanej w języku typescript. Dodatkowo komponenty mogą korzystać z serwisów. Serwisy to specjalne klasy, które są wstrzykiwane jako zależności (ang *Dependency injection*). Zawierają się w nich dane oraz logika dzielone między wieloma komponentami. Nawigacja pomiędzy poszczególnymi widokami odbywa się przy pomocy dedykowanego routera, który przechwytuje nawigację przeglądarki. Dzięki temu nawet nawigacja do tyłu nie powoduje przeładowania całej strony tylko poszczególnych zmienionych elementów.

Do budowy widoków użyta zostanie biblioteka Angular Material, która zawiera często wykorzystywane elementy zbudowane zgodnie z oficjalną specyfikacją *Material design*.

Taka modularna architektura pozwala na wielokrotne używanie tych samych komponentów oraz na zachowanie czytelnej struktury kodu.

## 3.2. Serwis API

Projekt podzielony zostanie na trzy warstwy:

**Api** warstwa obsługująca zapytania GraphQL

**Core** warstwa domenowa definiująca modele encji oraz interfejsy, z których korzysta warstwa Api

**Infrastructure** warstwa implementująca interfejsy oraz obsługująca dostęp do bazy danych

Podział ten znany jest jako „czysta architektura” (ang. *Clean architecture*). Taki układ umożliwia podział projektu na warstwy, które mają zdefiniowane odpowiedzialności. Dzięki zachowaniu zasady odwrócenia zależności możliwe jest testowanie każdej funkcjonalności z osobna.

Na platformie .Net Core dostępne są dwie aktywnie rozwijane biblioteki implementujące GraphQL: *GraphQL .NET* oraz *Hot Chocolate*. Mimo znacznie mniejszej popularności wybrana została biblioteka *Hot Chocolate*, ponieważ prezentuje wiele możliwości automatyzacji generowania schematu GraphQL oraz analizowania zapytań.

Z racji tego, że GraphQL jest protokołem służącym do przesyłania danych, jest on zaimplementowany w najwyższej warstwie serwisu – Api. W związku z tym nie zawęża sposobu implementacji zapisu danych w warstwie infrastruktury, a więc nic nie stoi na przeszkodzie, aby użyć relacyjnej bazy SQL. Takie rozwiązanie umożliwia wykorzystanie zalet relacyjnych

baz - integralność i niezależność danych, jednocześnie oferując grafowy odczyt danych poprzez GraphQL.

## 3.3. Baza danych

Baza danych stworzona zostanie w systemie zarządzania relacyjną bazą danych PostgreSQL. Jest to open source'owe oprogramowanie, obecnie jeden z najlepiej rozwiniętych RDBMS-ów.

## 3.4. Uwierzytelnianie

Jako protokół uwierzytelniania wybrano OpenID Connect. Standard ten rozszerza OAuth2 (służący do autoryzacji) o warstwę identyfikacji użytkowników. Jest obecnie jednym z najbezpieczniejszych standardów uwierzytelniania. Zaimplementowany zostanie przy pomocy biblioteki *IdentityServer4* na platformie *ASP.Net Core 3.0*. Użytkownik po zalogowaniu otrzyma token JWT, który będzie zawierał cyfrową sygnaturę. Dzięki temu jakakolwiek ingerencja w jego strukturę sprawi, iż nie jego walidacja zakończy się niepowodzeniem.

# Rozdział 4

# Implementacja

## 4.1. Strona internetowa

### 4.1.1. Zastosowane technologie

Zgodnie z projektem, strona stworzona została w Angularze. Oprócz podstawowych bibliotek Wykorzystane zostały następujące biblioteki:

**apollo** — wiodący klient GraphQL. Oprócz implementacji protokołu GraphQL, zapewnia również zarządzanie pamięcią podręczną (ang. *cache*) oraz stanem aplikacji (ang. *state management*). Dzięki temu wszystkie wszystkie odpowiedzi z serwisu Api są zapamiętywane, co pozwala na stworzenie aplikacji działającej szybko nawet przy słabym połączeniu z internetem.

**flex-layout** — biblioteka stworzona przez zespół tworzący Angulara, umożliwiająca stworzenie responsywnego interfejsu. Dostarcza API, które pozwala na definiowanie struktury elementów HTML zależnej od rozmiaru ekranu. Dzięki temu interfejs automatycznie dostosowuje się np. do ekranu telefonu komórkowego.

**oidc-client** — biblioteka implementująca protokół OpenID Connect oraz OAuth2. Zapewnia klasy obsługujące proces logowania oraz zarządzania tokenami.

**rxjs** — ReactiveX

### 4.1.2. Klient GraphQL

### 4.1.3. Połączenie z Last.fm

### 4.1.4. Responsive design

### 4.1.5. Przykładowy moduł

dodawanie recenzji

## 4.2. GraphQL API

### 4.2.1. Zastosowane technologie

**Hot Chocolate**
**Autofac**
**Entity Framework Core**
**Namotion.Reflection**
**Serilog**

### 4.2.2. Warstwa Api

### 4.2.3. Warstwa domenowa

### 4.2.4. Warstwa infrastrukturalna

## 4.3. Serwis uwierzytelniający

### 4.3.1. Zastosowane technologie

Oprócz opisanych wcześniej *Entity Framework Core* oraz *Serilog* wykorzystane zostały:

**IdentityServer4**
**AspNetCore.Identity**

### 4.3.2. Proces rejestracji i logowania

### 4.3.3. Autoryzacja dostępu do API

# Literatura


[1] GraphQL | A query language for your API. https://graphql.org. dostęp dnia 17 listopada 2019.

[2] GraphQL Spec, Czwerwiec 2018. https://graphql.github.io/graphql-spec/June2018. dostęp dnia 17 listopada 2019.

[3] HATEOAS - Wikipedia. https://en.wikipedia.org/wiki/HATEOAS. dostęp dnia 17 listopada 2019.

[4] XML Soap. https://www.w3schools.com/xml/xml_soap.asp. dostęp dnia 17 listopada 2019.

[5] E. Porcello, A. Banks. *Learning GraphQL*. O'Reilly Media, 2018.

[6] L. Richardson, M. Amundsen. *RESTful Web APIs*. O'Reilly Media, 2013.

[7] D. Tidwell, J. Snell, P. Kulchenko. *Programming Web Services with SOAP*. O'Reilly Media, 2001.

# Dodatek A

# Tytuł dodatku

Zasady przyznawania stopnia naukowego doktora i doktora habilitowanego w Polsce określa ustawa z dnia 14 marca 2003 r. o stopniach naukowych i tytule naukowym oraz o stopniach i tytule w zakresie sztuki (Dz.U. nr 65 z 2003 r., poz. 595 (Dz. U. z 2003 r. Nr 65, poz. 595). Poprzednie polskie uregulowania nie wymagały bezwzględnie posiadania przez kandydata tytułu zawodowego magistra lub równorzędnego (choć zasada ta zazwyczaj była przestrzegana) i zdarzały się nadzwyczajne przypadki nadawania stopnia naukowego doktora osobom bez studiów wyższych, np. słynnemu matematykowi lwowskiemu – późniejszemu profesorowi Stefanowi Banachowi.

W innych krajach również zazwyczaj do przyznania stopnia naukowego doktora potrzebny jest dyplom ukończenia uczelni wyższej, ale nie wszędzie.

# Dodatek B

# Opis załączonej płyty CD/DVD

Tutaj jest miejsce na zamieszczenie opisu zawartości załączonej płyty. Należy wymienić, co zawiera.