

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA (INF)
SPECJALNOŚĆ: SYSTEMY I SIECI KOMPUTEROWE (ISK)

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:

Spis treści

1. Wprowadzenie	5
1.1. Cel pracy	5
1.2. Zakres pracy	5
2. Przegląd technologii	6
2.1. Paradygmaty komunikacji z usługami sieciowymi	6
2.1.1. RPC	6
2.1.2. SOAP	6
2.1.3. REST	7
2.1.4. GraphQL	8
2.1.5. Porównanie	9
2.2. Platformy do tworzenia SPA	10
2.3. Platformy do tworzenia API	10
2.4. Docker	10
3. Analiza wymagań projektowych	11
3.1. Wymagania funkcjonalne	11
3.2. Wymagania niefunkcjonalne	11
3.3. Przypadki użycia	11
3.4. Założenia projektowe	12
4. Projekt aplikacji	14
4.1. Architektura fizyczna	14
4.2. Strona internetowa	14
4.3. Serwis API	16
4.4. Baza danych	17
4.5. Uwierzytelnianie i autoryzacja	17
5. Implementacja	19
5.1. Docker	19
5.1.1. Budowanie środowiska: docker-compose	19
5.1.2. Budowanie kontenera: Dockerfile	20
5.2. Strona internetowa	22
5.2.1. Zastosowane technologie	22
5.2.2. Klient GraphQL	22
5.2.3. Routing i nawigacja	25
5.2.4. Połączenie z Last.fm	28
5.2.5. Responsive design	28
5.2.6. Motyw aplikacji	29
5.3. GraphQL API	30

5.3.1. Warstwa Api	31
5.3.2. Warstwa domenowa	32
5.3.3. Warstwa infrastruktury	34
5.4. Serwis uwierzytelniający	36
5.4.1. Konfiguracja	37
5.4.2. Rejestracja	38
5.4.3. Logowanie	38
5.4.4. Autoryzacja dostępu do API	39
5.5. Baza danych	40
5.5.1. AuthServerDB	40
5.5.2. ReServerDB	41
5.6. Odwrócone proxy	41
6. Testy aplikacji	44
6.1. Testy jednostkowe	44
6.2. Testy integracyjne	45
6.3. Testy akceptacyjne	45
7. Prezentacja aplikacji	48
7.1. Widoki albumów	48
7.2. Dodawanie oceny	49
7.3. Rejestracja i logowanie	50
7.4. Jasny motyw	51
8. Podsumowanie	52
8.1. Wykonane prace	52
8.2. Wnioski	52
8.3. Możliwości dalszej rozbudowy	53
Literatura	54
A. Opis załączonej płyty CD/DVD	56

Skróty

IT (ang. *Information Technology*)

RPC (ang. *Remote Procedure Call*)

JSON (ang. *JavaScript Object Notation*)

SOAP (ang. *Simple Object Access Protocol*)

XML (ang. *Extensible Markup Language*)

WSDL (ang. *Web Services Description Language*)

REST (ang. *Representational state transfer*)

API (ang. *Application Programming Interface*)

CRUD (ang. *Create Read Update Delete*)

HTTP (ang. *HyperText Transfer Protocol*)

DTO (ang. *Data Transfer Object*)

HTML (ang. *HyperText Markup Language*)

CSS (ang. *Cascading Style Sheets*)

RDBMS (ang. *Relational Database Management System*)

JWT (ang. *JSON Web Token*)

URI (ang. *Uniform Resource Identifier*)

Rozdział 1

Wprowadzenie

1.1. Cel 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 został na stworzenie architektury stanowiącej solidną bazę do dalszego rozwijania aplikacji. Projekt wykorzystuje tylko oprogramowanie open source.

W ramach projektowania aplikacji szczególna uwaga zwrócona zostanie na użyteczność wykorzystania GraphQL do uzyskania grafowego dostępu do danych. GraphQL często uważane jest za rewolucyjne podejście do tworzenia API, co zostanie zbadane w tej pracy. Dużo uwagi poświęcone zostanie również kwestii bezpieczeństwa logowania do aplikacji.

1.2. Zakres pracy

W pierwszym rozdziale przedstawiono cel oraz zakres pracy. Kolejny rozdział zawiera informacje na temat technologii, które zostały wykorzystane w projekcie. Porównane zostały one do alternatywnych rozwiązań, które również spełniają wymagania projektu. Rozdział 3 opisuje wymagania projektowe: wymagania funkcjonalne i niefunkcjonalne, przypadki użycia oraz założenia projektowe. W kolejnym rozdziale przedstawiono projekt aplikacji. Zawarte w nim są projekty architektury poszczególnych warstw aplikacji oraz bazy danych. Rozdział 5 zawiera opis implementacji projektu. Podzielony został na podrozdziały, które opisują po kolei implementację każdego dockerowego kontenera aplikacji. Kolejny rozdział opisuje przeprowadzone testy aplikacji. W 7 rozdziale zaprezentowano zrzuty ekranu powstałej aplikacji. W ostatnim rozdziale zawarto podsumowanie pracy.

Rozdział 2

Przegląd technologii

2.1. Paradygmaty komunikacji z usługami sieciowymi

Powstało wiele sposobów komunikacji z usługami sieciowymi. W tym rozdziale opisane zostaną najpopularniejsze z nich w kolejności powstawania.

2.1.1. RPC

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  
}
```

2.1.2. SOAP

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 [11]. 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 [7]:

```
<?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>
```

2.1.3. REST

REST nie jest protokołem lecz stylem architektury. Nie narzuca konkretnej implementacji ani specyfikacji. Definiuje za to wymagania, które powinno spełnić API typu RESTful [10]:

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 do 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 przesyłać 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 [4]:

```
{
  "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.

2.1.4. GraphQL

GraphQL to język zapytań dla API oraz środowisko do wykonywania tych zapytań na istniejących danych [2]. Specyfikacja nie określa w jaki sposób ma być zbudowane API ale oferuje następujące zasady projektowania[3]:

- 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 określone 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: *Mutation*. Dodatkowo zdefiniowany jest typ *Subscription*, który używany jest do nasłuchiwanego 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 [9]. 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 zwrócią wartość (ang. *non-null*).

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.

2.1.5. Porównanie

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 zwykle 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 spotykana. 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.

2.2. Platformy do tworzenia SPA

Powstało mnóstwo bibliotek i frameworków do tworzenia stron internetowych i choć różnią się implementacją, to zasada działania pozostaje podobna. Oto trzy najpopularniejsze obecnie platformy:

Angular to framework stworzony przez Google, napisany w języku TypeScript. Zawiera w sobie pełen zestaw narzędzi i bibliotek do stworzenia strony wspierającej również urządzenia mobilne. Definiuje wzorce projektowe, które określają strukturę projektu i dobre praktyki.

React to biblioteka do tworzenia UI opracowana przez Facebooka. Do stworzenia pełnej, zaawansowanej strony wymaga użycia dodatkowych bibliotek do zarządzania stanem, trasowania czy interakcji z API. Nie narzuca żadnej struktury projektu, przez co każdy projekt może być zbudowany inaczej. Wymaga więc osoby doświadczonej, która kieruje projektem.

Vue nie jest zarządzany przez jedną korporację, lecz przez społeczność. Jest najbardziej przystępny, pozwala na stworzenie zarówno prostych stron, jak i złożonych poprzez rozszerzanie infrastruktury. Jest to osiągalne dzięki wysoce modularnej i elastycznej strukturze.

2.3. Platformy do tworzenia API

Serwer Api można stworzyć w większości istniejących języków programowania. Najczęściej wykorzystywane są frameworki zbudowane w PHP (Laravel), Java (Spring), Pythonie (Django), C# (ASP.Net Core), Ruby (Rails) i Node.js (Express). W tym projekcie zdecydowano się użyć ASP.Net Core 3.0 ze względu na wieloplatformowość (wsparcie dla kontenerów linuksowych), wbudowany system wstrzykiwania zależności (ang. *Dependency Injection*), nowoczesny język C#8 oraz wysoki stopień zaawansowania bibliotek implementujących GraphQL. Ważnym aspektem była również oficjalna biblioteka *Identity*, która zapewnia bezpieczną obsługę kont użytkowników.

2.4. Docker

Docker dostarcza narzędzi do zarządzania zaawansowanymi technologiami kernela linuksowego, między innymi: LXC (ang. *LinuX Containers*), cgroups (ang. *Control Groups*) i kopowanie przy zapisie (ang. *Copy-on-write*) [8]. LXC wykorzystuje metody na poziomie kernela do izolowania użytkowników, procesów i sieci. Cgroups implementują zarządzanie zasobami oraz pomiary wykorzystania tych zasobów przez poszczególne procesy wewnętrz kontenerów. Pozwalają na ograniczanie i rozdzielenie dostępu do pamięci, dysku i I/O. Ostatnią ważną częścią Dockera jest system plików copy-on-write. Dzięki temu kontenery wykorzystują wspólną część plików, a dopiero po zapisie tworzona jest kopia pliku unikalna dla danego kontenera. Techniki te pozwoliły na uzyskanie kontenerów, które zachowują się jak maszyny wirtualne przy znacznie lepszej wydajności. W przeciwieństwie do maszyn wirtualnych, kontenery wykorzystują tylko tyle zasobów, ile w danej chwili potrzebują.

Rozdział 3

Analiza wymagań projektowych

3.1. Wymagania funkcjonalne

Wymagania funkcjonalne określają wymagania dotyczące pożądanego zachowania systemu. Definiują, jakie usługi ma oferować i jak ma reagować na określone dane wejściowe. Wyszczególniono następujące wymagania funkcjonalne:

1. System zawiera katalog albumów
2. Albumy da się wyświetlać wraz ze szczegółami
3. Każdy album można oceniać oraz dodawać jego recenzję
4. Istnieje możliwość dodawania nowych albumów
5. Wyszukiwanie albumów korzysta z bazy zewnętrznego serwisu
6. Możliwość rejestracji i logowania
7. Wybór pomiędzy ciemnym i jasnym motywem aplikacji

3.2. Wymagania niefunkcjonalne

Wymagania te określają właściwości systemu, jego ograniczenia i standardy w jakich pracuje. Wymagania systemu spełniającego założenia projektowe są następujące:

1. Aplikacja powinna być obsługiwana przez obecne wersje przeglądarek
2. Do poprawnego działania wymagane jest połączenie z Internetem
3. Aplikacja powinna być napisana w sposób umożliwiający łatwe dodawanie nowej funkcjonalności
4. Dane użytkowników są przechowywane w bezpieczny sposób

3.3. Przypadki użycia

Na rysunku 3.1 przedstawiono diagram przypadków użycia użytkownika aplikacji. Zawarte zostały główne funkcjonalności skupiające się na przeglądaniu albumów i dodawaniu recenzji. W ten sposób wyszczególniono następujące przypadki użycia:

1. Dodanie oceny albumu
Przypadek użycia dotyczy akcji dodania oceny do albumu. Dodawanie opisowej recenzji jest opcjonalne, dlatego ujęte zostało jako rozszerzający przypadek użycia.
2. Wyszukanie albumu z bazy Last.fm
Aby dodać ocenę do albumu należy wyszukać odpowiedni album wpisując jego nazwę

lub wybrać z najpopularniejszych albumów podanego artysty. Wyszukiwarka korzysta z API Last.fm, jednak nie wymaga konta w tym serwisie.

3. Logowanie

Dodawanie ocen dostępne jest tylko dla zalogowanych użytkowników. Przeglądanie albumów i recenzji nie wymaga konta.

4. Rejestracja

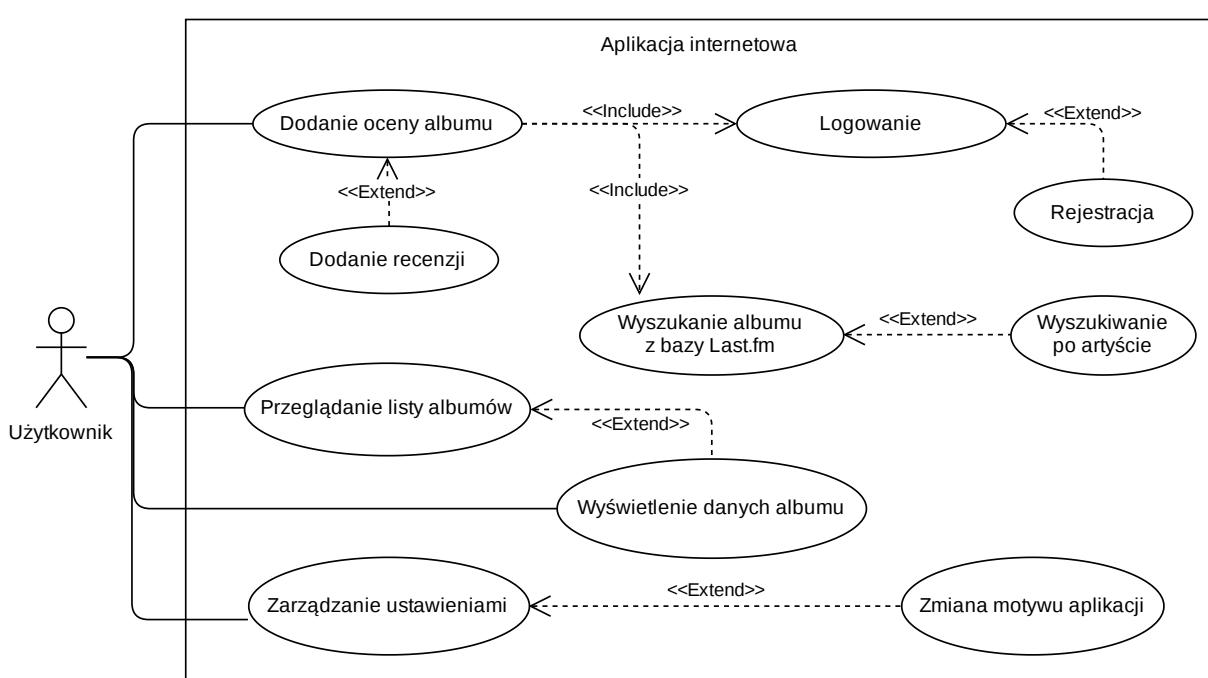
Jeśli użytkownik nie ma jeszcze utworzonego konta, a chce dodawać oceny to może się zarejestrować.

5. Przeglądanie listy albumów

Użytkownik może przeglądać wszystkie albumy, do których zostały dodane oceny. Po kliknięciu w album przenoszony zostaje do widoku wyświetlającego wszystkie dane albumu.

6. Zarządzanie ustawieniami

Przypadek użycia opisujący zmianę ustawień aplikacji, które są unikalne dla każdego użytkownika. Na chwilę obecną możliwa jest zmiana motywu głównego aplikacji (jasny/ciemny).



Rys. 3.1: Diagram przypadków użycia

3.4. 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 GraphQL. Serwer umożliwiać będzie pobieranie danych stosując zapytania w tym języku. Dzięki temu relacje między danymi przedstawione są w postaci grafu, co pozwala formować skomplikowane 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 4

Projekt aplikacji

4.1. Architektura fizyczna

Zgodnie z założeniami projektowymi, całość aplikacji serwowana będzie z platformy dockerowej. Podeczas tworzenia aplikacji uruchamiana ona będzie na systemie *Ubuntu 18.04* przy użyciu *Windows Subsystem For Linux*. Umożliwia to jednoczesne korzystanie z narzędzi dostępnych tylko pod systemem Windows (Visual Studio) i uruchamianie w docelowym systemie (Linux). Wersja produkcyjna aplikacji umieszczona zostanie na serwerze z systemem Linux.

Na schemacie 4.1 przedstawiono architekturę aplikacji z podziałem na poszczególne kontenery. Aby zapewnić najwyższe bezpieczeństwo, porty kontenerów nie są mapowane na zewnętrz dockera. Wyjątkiem jest kontener z odwróconym proxy (*revProxy*), który obsługuje całą komunikację ze światem zewnętrznym. Kieruje on odpowiednio ruchem w zależności od adresu zapytania. Na przykładzie lokalnego adresu, zapytania obsługiwane będą w następujący sposób:

localhost/ domyślnie cały ruch kierowany jest do strony internetowej serwowanej z kontenera *spa*

auth.localhost/ zapytania z subdomeną *auth* kierowane są do kontenera *authServer* obsługującego zapytania związane z OpenID

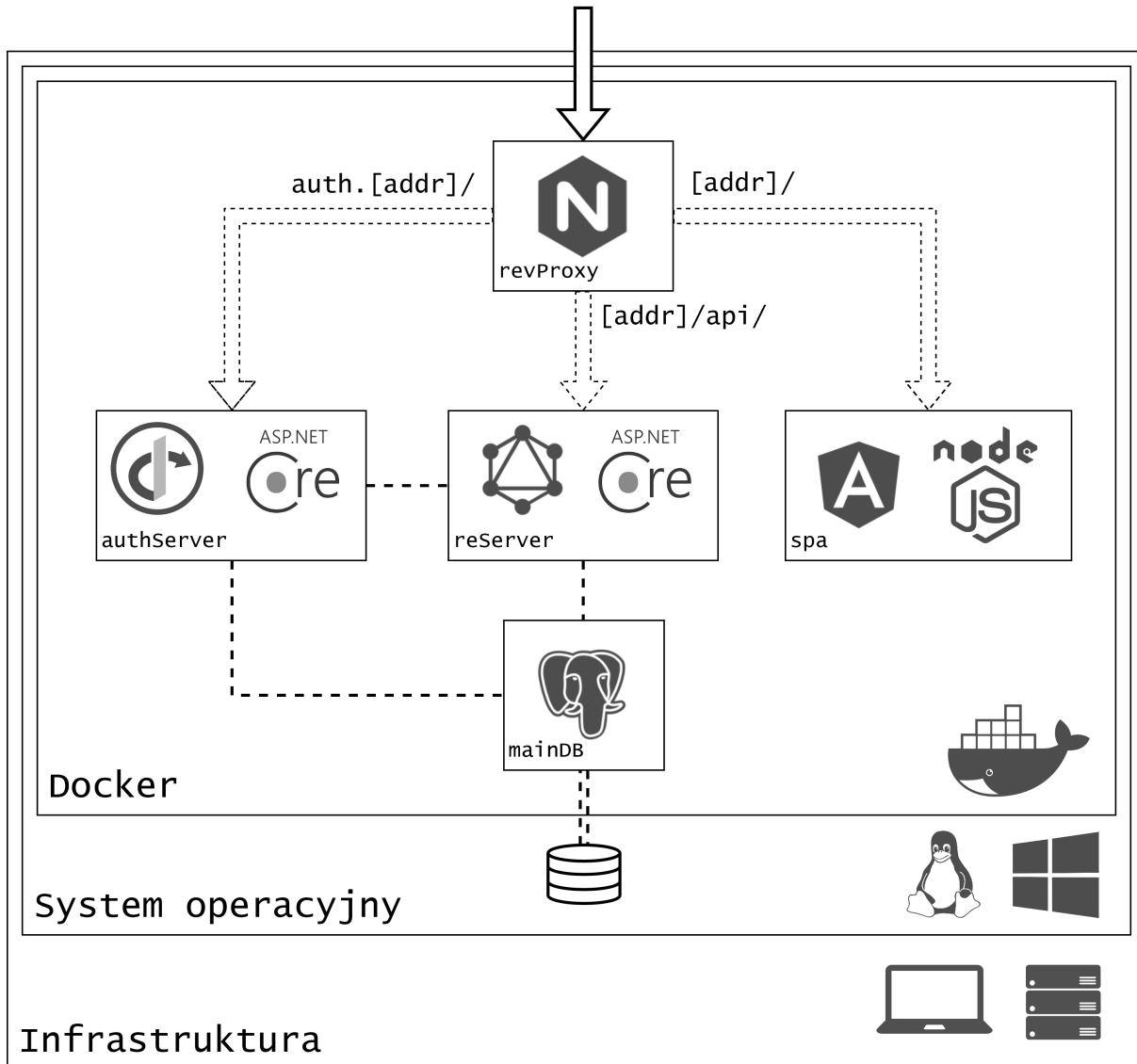
localhost/api/ jeśli zapytanie odnosi się do katalogu */api/* to zostaje obsłużone przez kontener *reServer*

Kontener obsługujący zapytania API musi być dostępny z zewnętrz, ponieważ strona internetowa jest typu SPA, a więc jest wszystkie zapytania będą wysyłane bezpośrednio z przeglądarki użytkownika. Z tego samego powodu kontener *spa* nie komunikuje się z resztą kontenerów, co ukazane zostało na schemacie. Pozostałe kontenery korzystają z wewnętrznej sieci do komunikacji z bazą danych. Dodatkowo, kontener *reServer* komunikuje się z kontenerem *authServer* do walidacji tokenów JWT oraz do pobierania danych zalogowanego użytkownika.

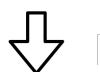
Dane nie mogą być zapisywane bezpośrednio w kontenerze, ponieważ mogły by zostać łatwo utracone jeśli istniała by potrzeba zresetowania kontenerów. Z tego powodu baza danych zapisuje dane do folderu zmapowanego na dysk systemu operacyjnego. Ogranicza to ryzyko utraty danych jednak w środowisku produkcyjnym baza powinna znajdować się w dedykowanym do tego serwerze.

4.2. Strona internetowa

Do stworzenia strony zdecydowano się użyć framework'u 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



Legenda:



Połączenie z zewnątrz



Połączenie korzystające z wewnętrznej sieci dockerowej



Komunikacja pomiędzy kontenerami



Połączenie pomiędzy dockerem a systemem operacyjnym

Rys. 4.1: Architektura fizyczna aplikacji

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.

4.3. Serwis API

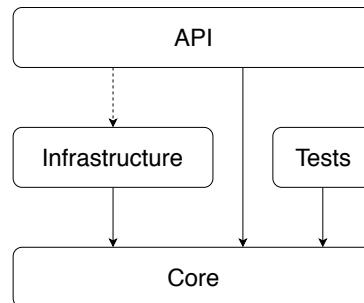
Projekt podzielony zostanie na trzy warstwy:

Api warstwa obsługująca zapytania GraphQL

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

Infrastructure warstwa implementująca interfejsy oraz obsługującą dostęp do bazy danych

Podział ten znany jest jako „czysta architektura” (ang. *Clean architecture*). Schemat ten przedstawiony został na rysunku 4.2, gdzie linie ciągłe oznaczają zależności na etapie budowania, natomiast linia przerywana oznacza zależność po uruchomieniu. Taki układ umożliwia umieszczenie logiki biznesowej oraz modelu aplikacji w centrum (Core). W ten sposób infrastruktura oraz szczegóły implementacji są zależne od Core, a nie na odwrót — odwrócenie zależności. Warstwa API pracuje na interfejsach zdefiniowanych w Core i nie zna ich implementacji w Infrastructure. Implementacje te są połączane dopiero po uruchomieniu poprzez wstrzykiwanie zależności. Istotną zaletą takiej architektury jest również możliwość testowania każdej funkcjonalności z osobna.



Rys. 4.2: Schemat czystej architektury użytej w projekcie

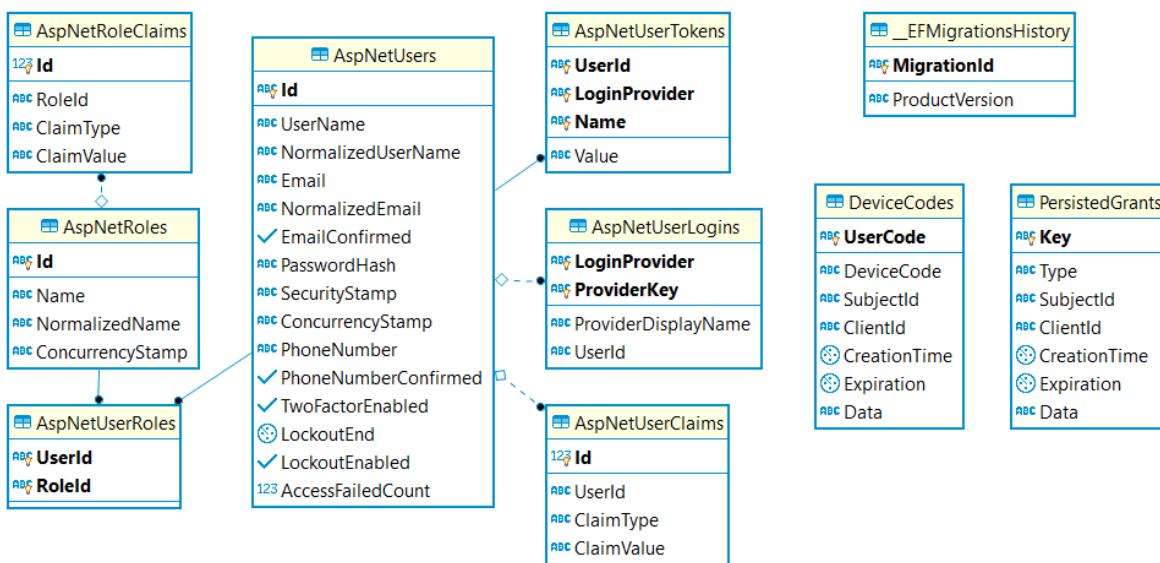
Na platformie .Net Core dostępne są dwie aktywnie rozwijane biblioteki implementujące GraphQL: *GraphQL .NET* oraz *Hot Chocolate*. Mimo znacznie mniejszej popularności wybrana została biblioteka *Hot Chocolate*, ponieważ prezentuje wiele możliwości automatyzacji generowania schematu GraphQL oraz analizowania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

4.4. 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.

Baza z kontami użytkowników (rys. 4.3) zostanie wygenerowana za pomocą platformy Identity. Tabele te przechowują podstawowe dane kont użytkowników takie jak login, hasz hasła czy email. Zawierają się tu również role użytkowników oraz ich upoważnienia. Oprócz danych użytkowników zapisywane są również dane związane z migracjami bazy zarządzane przez Entity Framework oraz dwie pomocnicze tabele biblioteki *IdentityServer4* (*DeviceCodes* i *PersistedGrants*).



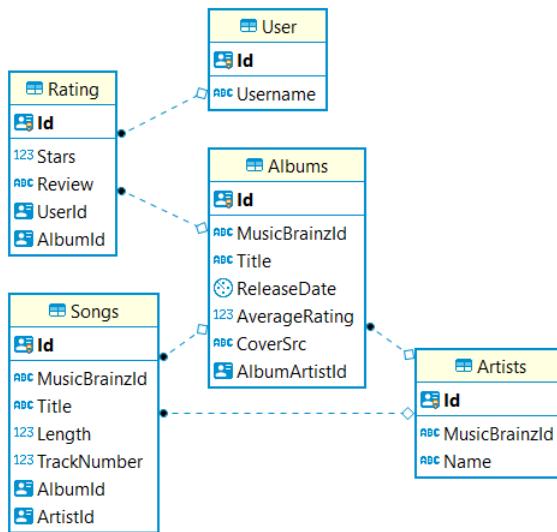
Rys. 4.3: Schemat ERD bazy z kontami użytkowników

Rysunek 4.4 przedstawia bazę przechowującą główne dane aplikacji. Przechowywana jest minimalna ilość danych wymagana do zaimplementowania wersji koncepcualnej aplikacji. Jeśli aplikacja miałaby zostać wdrożona, z konieczne byłoby dodanie większej ilości danych. Tabele połączone są relacjami umożliwiającymi grafową reprezentację. Baza ta również została wygenerowana metodą code first za pomocą Entity Framework.

4.5. Uwierzytelnianie i autoryzacja

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 jego walidacja zakończy się niepowodzeniem.

Dla zapewnienia najwyższej bezpieczeństwa zaimplementowany zostanie proces logowania z wykorzystaniem kodu PKCE (ang. *Proof Key for Code Exchange*), opisany w dokumencie RFC 7636 [6]. Stanowi on udoskonalenie przepływu wykorzystującego kod autoryzacji (ang. *Authorization Code Flow*). Zabezpiecza przed próbami przechwycenia kodu autoryzacji przez złośliwą aplikację, która mogła by za pomocą tego kodu uzyskać token dostępowy (ang. *Access Token*).

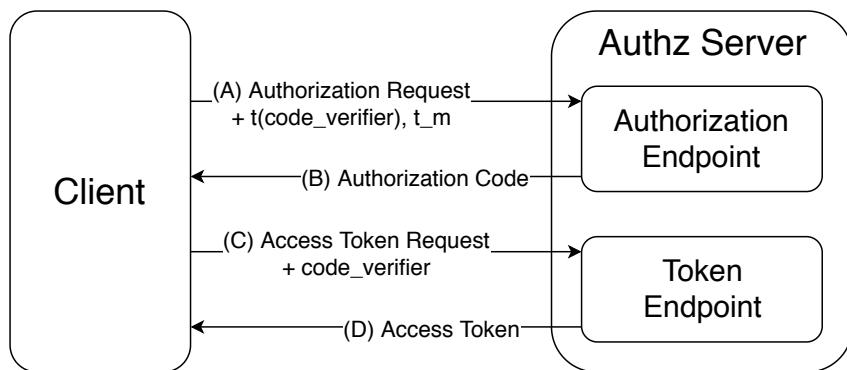


Rys. 4.4: Schemat ERD bazy z głównymi danymi

Proces uzyskania tokenu dostępowego, przedstawiony na schemacie 4.5, jest następujący:

- Klient tworzy losowy klucz `code_verifier` i przekształca go za pomocą metody transformacji `t_m`. Następnie przesyła przekształcony klucz wraz z metodą transformacji w żądaniu autoryzacji OAuth 2.0.
- Jeśli użytkownik zalogował się poprawnymi danymi, serwer zapisuje parametry weryfikacji i zwraca kod autoryzacji.
- Aby uzyskać token dostępowy, klient przesyła kod autoryzacji wraz z oryginalną wartością `code_verifier` wygenerowaną w punkcie (A).
- Serwer autoryzacji przekształca `code_verifier` za pomocą metody `t_m` i porównuje z wartością `t(code_verifier)` otrzymaną w punkcie (B). Jeśli są równe, serwer zwraca token dostępowy.

W ten sposób nawet jeśli kod autoryzacji w punkcie (B) zostanie przechwycony, nie będzie mógł zostać wykorzystany bez znajomości klucza `code_verifier`.



Rys. 4.5: Schemat procesu logowania z użyciem PKCE

Rozdział 5

Implementacja

5.1. Docker

Do zbudowania aplikacji użyto narzędzia *docker-compose*. Pozwala ono definiować kontenery, które uruchomione zostaną we wspólnym środowisku. Dla każdego kontenera zdefiniowane zostały następujące właściwości:

container_name nazwa kontenera

image obraz, z którego będzie zbudowany

build instrukcje do zbudowania obrazu

networks wewnętrzne sieci, do których podłączony zostanie kontener

volumes foldery hosta, które zostaną zmapowane do adresu wewnętrz kontenera

ports porty hosta, które zostaną zmapowane na porty kontenera

environment zmienne środowiskowe wewnętrz kontenera

depends_on kontenery, które powinny zostać zbudowane przed opisywanym kontenerem

Każdy kontener musi mieć zdefiniowany **image**, z którego zostanie zbudowany lub parametry **build** z adresem pliku Dockerfile zawierającym przepis na zbudowanie obrazu.

Pliki docker-compose są przyłączeniowe; podczas budowania środowiska można zdefiniować więcej niż jeden. Zaletę tę wykorzystano w projekcie i rozdzielono pliki docker-compose na wersje odpowiadające odpowiednim środowiskom. Proces budowania środowiska przedstawiony zostanie na przykładzie kontenera authServer.

5.1.1. Budowanie środowiska: docker-compose

We fragmencie kodu 5.1 przedstawiono ustawienia wspólne dla wszystkich środowisk. Znajduje się tu nazwa kontenera, nazwa wynikowego obrazu, sieć do której jest podłączony oraz kontenery, od których jest zależny. Sieć **mainNetwork** jest wspólna dla wszystkich kontenerów. Określona jest zależność od kontenera **mainDB**, ponieważ serwis korzysta z bazy od razu przy uruchomieniu serwera.

Listing 5.1: Wspólne ustawienia kontenera authServer

```
authserver:
  container_name: "authServer"
  image: ${DOCKER_REGISTRY}-authserver
  networks:
    - "mainNetwork"
  depends_on:
    - "maindb"
```

Natomiast fragment 5.2 zawiera ustawienia tego samego kontenera dla środowiska developerskiego. Zdefiniowane zostały następujące zmienne środowiskowe:

ENABLE_POLLING wykorzystywana w Dockerfile, określa czy kontener zostanie przebudowany po każdych zmianach w plikach źródłowych

ASPNETCORE_ENVIRONMENT wykorzystywana jest wewnątrz programu do określenia odpowiednich konfiguracji

ASPNETCORE_URLS adresy wykorzystywane przez serwer, adres HTTPS (443) nie jest wymieniony, ponieważ szyfrowaniem zajmuje się kontener z odwróconym proxy

ConnectionStrings__DefaultConnection ustawienia połączenia do bazy, wykorzystywany jest wewnętrzny adres kontenera `mainDB`

Kontener posiada osobne pliki Dockerfile dla każdego środowiska, ponieważ w środowisku developerskim zaimplementowano obserwowanie plików źródłowych. Dzięki temu, każda zmiana w kodzie skutkuje natychmiastowym przebudowaniem kontenera, co znacznie przyspiesza pracę. Treść pliku Dockerfile (listing 5.3) została dokładnie opisana w następnym rozdziale. W tym środowisku wszystkie kontenery mają również zmapowane porty na zewnątrz. Folder z kodem źródłowym został zmapowany na adres `/app`, co również podyktowane jest wykrywaniem zmian w plikach.

Listing 5.2: Ustawienia kontenera authServer dla środowiska developerskiego

```
authserver:
  environment:
    - ENABLE_POLLING=1
    - ASPNETCORE_ENVIRONMENT=Development
    - ASPNETCORE_URLS=http://+:80
    - ConnectionStrings__DefaultConnection=Server=mainDB;Port=5432; ...
  build:
    context: .
    dockerfile: AuthServer/Dockerfile
  ports:
    - "5550:80"
  volumes:
    - "./AuthServer/:/app"
```

5.1.2. Budowanie kontenera: Dockerfile

Często do utworzenia serwisu wystarczy gotowy obraz dostępny w Docker Hub. Jest to największy serwis typu Docker Registry: baza udostępnionych obrazów dockerowych przygotowanych przez użytkowników, grupy czy firmy. Takie rozwiązanie jest wystarczające w przypadku kontenera `revProxy`. Obraz `nginx` zawiera całą niezbędną funkcjonalność.

We wszystkich pozostałych przypadkach wymagane jest utworzenie pliku Dockerfile, który określa przepis na zbudowanie obrazu. Zwykle nie buduje się całego obrazu od zera, lecz określa się inny obraz jako bazę i dopisuje się brakującą funkcjonalność. Jest to możliwe dzięki warstwowej budowie obrazów. W każdym obrazie zapisane są tylko zmiany w stosunku do obrazu bazowego, co pozwala na przechowywanie ogromnej liczby obrazów w serwisach typu Docker Registry. Przykładowo, obraz używany w kontenerze `mainDB` rozszerza obraz `postgres` w wersji 11.5 o zaledwie dwie komendy:

```
FROM postgres:11.5
RUN echo "listen_addresses='*'" >> /var/lib/postgresql/postgresql.conf
EXPOSE 5432
```

komenda RUN wywołuje podaną komendę w środku kontenera, natomiast komenda EXPOSE aktywuje nasłuchiwanie na danym porcie.

Nieco bardziej skomplikowany jest Dockerfile budujący serwis authServer dla środowiska developerskiego, rozszerzający plik z projektu Dispersia/Dotnet-Watch-Docker-Example [1] o mechanizm wyłączania obserwowania z poziomu docker-compose:

Listing 5.3: Plik Dockerfile dla środowiska developerskiego

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.0

WORKDIR /vsdbg

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        unzip \
    && rm -rf /var/lib/apt/lists/* \
    && curl -sSL https://aka.ms/getvsdbgsh \
        | bash /dev/stdin -v latest -l /vsdbg

ENV DOTNET_USE_POLLING_FILE_WATCHER ${ENABLE_POLLING:-0}

WORKDIR /app

ENTRYPOINT dotnet ${ENABLE_POLLING:+watch} run --urls=http://+:80
```

W pierwszym kroku za pomocą WORKDIR ustawiany jest folder, względem którego wykonywane będą wszystkie komendy. Następnie instalowany jest debugger .NET Core, który umożliwia debugowanie z Visual Studio na systemie hosta (Windows). Zmienna DOTNET_USE_POLLING_FILE_WATCHER ustawiana jest na podstawie zmiennej ENABLE_POLLING skonfigurowanej w docker-compose. Po zmianie katalogu roboczego na /app ustawiana jest komenda uruchamiana po starcie kontenera. Ona również jest zależna od zmiennej ENABLE_POLLING: jeśli zmienna jest ustawiona na 1, uruchomiona zostanie komenda `dotnet watch run`. W przeciwnym wypadku będzie to komenda `dotnet run`.

Produkcyjny Dockerfile (listing 5.4) nie obsługuje debugowania i optymalizowany jest pod względem wydajności. Wykorzystuje w tym celu warstwową naturę obrazów dockerowych. Jako bazy wykorzystuje warianty `buster-slim` oraz `buster` obrazów, które stworzone zostały specjalnie w tym celu. Został opracowany przez grupę tworzącą .NET Core.

Listing 5.4: Plik Dockerfile budowany warstwowo

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.0-buster-slim AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/core/sdk:3.0-buster AS build
WORKDIR /src
COPY ["AuthServer/AuthServer.csproj", "AuthServer/"]
RUN dotnet restore "AuthServer/AuthServer.csproj"
COPY .
WORKDIR "/src/AuthServer"
RUN dotnet build "AuthServer.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "AuthServer.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "AuthServer.dll"]
```

5.2. Strona internetowa

5.2.1. Zastosowane technologie

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

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 wykorzystujące pod spodem media query, 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 — implementacja ReactiveX na JavaScript. Łączy w sobie najlepsze pomysły ze wzorców projektowych *Observer*, *Iterator* oraz z programowania funkcyjnego [5]. Umożliwia programowanie przy pomocy obserwowlanych obiektów aby ułatwić tworzenie asynchronicznych strumieni.

5.2.2. Klient GraphQL

Klient Apollo zbudowany został z opcjami przedstawionymi na listingu 5.5. Jako adres API GraphQL użyto względnego uri wykorzystującego adres zdefiniowany w odwróconym proxy: `uri = api/graphq1`. W domyślnych ustawieniach zdefiniowano nagłówek autoryzujący pobierający Access Token zalogowanego użytkownika z serwisu AuthService. Dzięki temu tożsamość użytkownika może zostać zweryfikowana po stronie serwera. W tym miejscu zapisywane są również domyślne ustawienia aplikacji w InMemoryCache. Uwarunkowane jest to tym, że ustawienia aplikacji nie są przetrzymywane w bazie danych. Możliwe jest to dzięki temu, że biblioteka apollo pozwala na definiowanie lokalnego schematu GraphQL, który rozszerza schemat wykorzystywanego API. W ten sposób uzyskano zarządzanie stanem aplikacji bez wykorzystywania dodatkowych bibliotek typu *NgRx* czy *Redux*.

Listing 5.5: Globalne ustawienia modułu Apollo

```
const auth = setContext(({_operation, _context}) => ({
  headers: {
    Authorization: authService.authorizationHeaderValue
  }
}));
const cache = new InMemoryCache();
const defaultSettings = {
  __typename: 'Settings',
  theme: 'dark-theme'
} as Settings;
cache.writeData({data: {settings: defaultSettings}});
return {
  link: ApolloLink.from([auth, httpLink.create({uri})]),
  cache,
  resolvers,
  typeDefs: {}
};
```

Wywoływanie zapytań

W projekcie wykorzystano narzędzia *graphql-cli* oraz *graphql-codegen*. Pierwsze pozwala na generowanie lokalnej kopii dostępnego schematu za pomocą komendy `graphql get-schema`. Drugi program generuje typy GraphQL jako interfejsy w języku TypeScript. Dodatkowo, konwertuje wszystkie zapytania oraz mutacje na serwisy angularowe, które mogą zostać wstrzyknięte do każdego komponentu. Dzięki temu tworząc zapytanie do API, wystarczy napisać zapytanie w języku GraphQL, a narzędzie wygeneruje gotowy serwis. Przykładowo, napisanie następującego zapytania dodającego nową ocenę albumu:

```
mutation addRating($rating: RatingInput!) {
  addRating(rating: $rating) {
    id
  }
}
```

wygeneruje następujący serwis:

```
export const AddRatingDocument = gql`
  mutation addRating($rating: RatingInput!) {
    addRating(rating: $rating) {
      id
    }
  }
`;

@Injectable({
  providedIn: 'root'
})
export class AddRatingGQL
  extends Apollo.Mutation<AddRatingMutation, AddRatingMutationVariables> {
  document = AddRatingDocument;
}
```

Klasa oznaczona jest angularowym dekoratorem `@Injectable`, dzięki czemu dostępna jest jako serwis w całej aplikacji. Uzyskano w ten sposób szybki i elastyczny proces korzystania z API jednocześnie zachowując pełną walidację typów. Serwis wystarczy zaimportować w komponencie i wywołać metodę `mutate` zwracającą typ `Observable`:

```
constructor(
  private addRating: AddRatingGQL
) {}

rating: RatingInput;

onSubmit() {
  this.addRating.mutate({rating: this.rating})
    .subscribe({
      error: err => console.log('mutation failed', err),
      complete: () => this.router.navigate(['/home'])
    });
}
```

W aplikacji przyjęto założenia, że obiektów zwracanych przez mutację nie używa się bezpośrednio. Zwrócony obiekt aktualizuje cache, co powoduje, że wszystkie zasubskrybowane obiekty asynchronicznie otrzymują najnowszą wersję. Aby było to możliwe, elementy widoków utrymują subskrypcję przez cały cykl życia komponentu. Subskrypcja odbywa się automatycznie dzięki zastosowaniu *async pipe*. *Pipe* to klasa, która pozwala na transformację zadanego wejścia i używa się ich bezpośrednio w pliku HTML.

Przykładowo, komponent wyświetlający dane albumu w pierwszej kolejności pobiera id albumu z parametrów ścieżki, a następnie pobiera dane albumu o tym id. Odbywa się to przy pomocy *rxjs pipe*, czyli metody pozwalającej łączyć wiele operatorów przetwarzających dane w strumieniu. W tym komponencie wykorzystano dwa operatory:

1. **switchMap**: operator, który przyjmuje dane i zwraca typ Observable. Służy do zmiany źródła strumienia danych poprzez subskrypcję pierwszego strumienia i wykorzystanie wyniku w tworzeniu drugiego strumienia. W tym przypadku przyjmuje mapę parametrów ścieżki, znajduje id albumu i zwraca strumień obserwujący zapytanie GraphQL `getAlbum.watch`.
2. **map**: operator służący do modyfikacji danych. Najczęściej wykorzystywany do zwracania nowych typów obiektów na podstawie danych wejściowych lub ich fragmentów. Użyty został w celu bezpośredniego zwrócenia zagnieżdżonych danych.

Powstały *pipe* nie pobiera od razu danych, lecz definiuje jedynie sposób ich przetwarzania:

```
album$: Observable<GetAlbumFullQuery['album']>;

constructor(
  private route: ActivatedRoute,
  private getAlbum: GetAlbumFullGQL
) {}

ngOnInit() {
  this.album$ = this.route.paramMap.pipe(
    switchMap((params: ParamMap) =>
      this.getAlbum.watch({id: params.get('id')}).valueChanges(),
      map(res => res.data.album)
    );
}
```

natomiast subskrypcja zarządzana jest w całości przez *async pipe*:

```
<div *ngIf="album$ | async as album">
  <h3>Title: {{album.title}}</h3>
</div>
```

Lokalny schemat GraphQL

Jak przedstawiono na listingu 5.6, aby rozszerzyć typ o nowe pola należy użyć komendy `extend`. W tym przypadku dodany został nowy typ `Settings`, który jest użyty w dodatkowym polu domyślnego `Query`.

Wczytywanie danych zdefiniowanych w lokalnym schemacie odbywa się tak samo jak przy zewnętrznym Api z jedną różnicą: podczas definiowania zapytania należy użyć dyrektywy `@client`, aby klient wiedział żeby nie odpytywać Api:

```
query getSettings {
  settings @client {
    theme
  }
}
```

Aby umożliwić zapisywanie danych do lokalnej bazy poprzez mutację `updateSettings`, trzeba stworzyć *resolver*, czyli funkcję, która określa działanie danego pola. *Resolver* dla tej mutacji przedstawiony został we fragmencie kodu 5.7. Tworzy on nowy obiekt typu `Settings`, który oprócz przekazanych ustawień zawiera pole `__typename`. Jest to pole identyfikujące obiekt, dzięki czemu cache nadpisze dane w odpowiednim miejscu.

Listing 5.6: Lokalny schemat GraphQL

```

extend type Query {
  settings: Settings
}

extend type Mutation {
  updateSettings(input: SettingsInput): Settings
}

type Settings {
  theme: String!
}

input SettingsInput {
  theme: String
}

```

Listing 5.7: Resolvers dla lokalnego cache

```

export const resolvers = {
  Mutation: {
    updateSettings: (_, {input}, {cache}) => {
      const settings = {
        __typename: 'Settings',
        ...input
      } as Settings;
      cache.writeData({data: {settings}});

      return null;
    }
  }
};

```

5.2.3. Routing i nawigacja

Nawigacja pomiędzy widokami wykorzystuje bibliotekę `angular/router`. Przechwytuje ona nawigację przeglądarki i zamiast przeładowywać całą stronę, ładuje jedynie odpowiednie komponenty. Każdy moduł posiada swoją tablicę tras odpowiadających komponentom, które mają zostać załadowane dla danego adresu. W routingu głównego modułu `app-routing.module` (Listing 5.8) zdefiniowane zostały trasy wywołań zwrotnych (ang. *callback*) serwisu uwierzytelniającego oraz trasy do odpowiednich modułów. Moduły ładowane są dynamicznie dopiero po przekierowaniu w odpowiedni adres za pomocą metody `loadChildren`. Podczas mapowania adresu router wyszukuje pierwsze dopasowanie. Bazowy adres (np. *localhost*) przekierowywany jest na świeżkę *home*. Następnie adres dopasowywany jest do któregoś z callback-ów lub modułów. Jeśli żaden nie pasuje, to znaczy że adres jest nieprawidłowy i jest łapany przez domyślną trasę (**), która przekierowuje na *home*.

Zabezpieczanie dostępu do komponentów przed nieautoryzowanymi użytkownikami odbywa się za pomocą dyrektywy `canActivate`:

```
{
  path: 'add',
  component: AddReviewComponent,
  canActivate: [AuthGuard],
}
```

AuthGuard to serwis, który deklaruje funkcję `canActivate` zwracającą typ boolean informującą o tym, czy użytkownik spełnia wymagania dostępu. Implementacjw w projekcie wykorzystuje serwis uwierzytelniający aby sprawdzić, czy użytkownik jest zalogowany. Jeśli jest zalogowany to zwraca `true`, w przeciwnym przypadku następuje przekierowanie do strony logowania:

```
canActivate(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot
): boolean {
  if (this.authService.isAuthenticated()) {
    return true;
  }
  this.router.navigate(['/login'], {
    queryParams: { redirect: state.url },
    replaceUrl: true
});
  return false;
}
```

Dane do komponentów można przekazywać bezpośrednio w adresie poprzez oznaczanie zmiennych dwukropkiem:

```
{
  path: ':id',
  component: AlbumComponent
}
```

Poprawna ścieżka to np. `/album/3`, gdzie `:id` jest równe 3. Możliwe jest również przekazanie danych do komponentu poprzez pole `data`, dzięki czemu można wykorzystywać ten sam komponent dla różnych zadań. Wykorzystano to przy callback-ach serwisu uwierzytelniającego, który wykorzystuje ten sam komponent dla wszystkich akcji. Dane te można odczytać w komponencie wykorzystując obiekt `ActivatedRoute`:

```
constructor(
  private authService: AuthService,
  private router: Router,
  private route: ActivatedRoute
) {}

async ngOnInit() {
  this.route.data.subscribe(async data => {
    switch (data.action) {
      case AuthAction.Login:
        await this.authService.completeAuthentication();
        this.router.navigate(['/home']);
        break;
      case ...
    }
  })
}
```

Po zarejestrowaniu komponentów należy jeszcze oznaćzyć na szablonie HTML miejsce, w którym router będzie wstawiać załadowane komponenty. W projekcie uczyniono to w głównym komponencie `app.component`, jak pokazane zostało na listingu 5.9. Ujście routera `<router-outlet>` (linia 16) zagnieżdżono w `mat-sidenav-content`, dzięki czemu nagłówki, menu boczne oraz stopka obecne są na każdym widoku. Zmienia się jedynie zawartość wstawiana przez router.

Listing 5.8: Trasy głównego modułu

```
const routes: Routes = [
  {
    path: '',
    component: AppComponent
  }
]
```

```

        redirectTo: 'home',
        pathMatch: 'full'
    },
    {
        path: 'login-callback',
        component: AuthCallbackComponent,
        data: {action: AuthAction.Login}
    },
    {
        path: 'silent-refresh',
        component: AuthCallbackComponent,
        data: {action: AuthAction.SilentRefresh}
    },
    {
        path: 'home',
        loadChildren: () => import('./home/home.module')
            .then(m => m.HomeModule)
    },
    {
        path: 'album',
        loadChildren: () => import('./modules/album/album.module')
            .then(m => m.AlbumModule)
    },
    [...]
    {path: '**', redirectTo: 'home', pathMatch: 'full'}
];

```

Listing 5.9: Główny szablon HTML aplikacji

```

1 <div [class]="'theme-wrapper mat-typography '+ (settings$ | async)?.theme">
2
3     <app-header class="toolbar" ( sidenavToggle)="mainSidenav.toggle()">
4         </app-header>
5
6         <mat-sidenav-container fxFlex fxFlex.xs="noshrink">
7             <mat-sidenav #mainSidenav role="navigation" mode="over"
8                 fixedInViewport="true" fixedTopGap="56">
9                 <app-sidenav (sidenavClose)="mainSidenav.close()"></app-sidenav>
10            </mat-sidenav>
11
12            <mat-sidenav-content>
13                <main role="main" class="wrapper" fxLayout="column">
14
15                    <div class="content" fxFlex="noshrink">
16                        <router-outlet></router-outlet>
17                    </div>
18
19                    <div class="footer" fxFlex="none">
20                        Marcin Kotas, 2019
21                    </div>
22
23            </main>
24        </mat-sidenav-content>
25    </mat-sidenav-container>
</div>

```

5.2.4. Połaczenie z Last.fm

W serwisie Last.fm zarejestrowano aplikację uzyskując dostęp do pełnego Api. Wykorzystywane jest tylko do pobierania publicznych danych, dlatego użytkownik aplikacji nie musi zakładać konta w serwisie. Z Api można korzystać zarówno w wersji XML, jak i JSON. W projekcie stworzono serwis wykorzystujący wersję JSON, definiując interfejsy dla każdego zapytania. Przykładowe zapytanie wykorzystujące metodę Api `Artist.getTopAlbums` przedstawiono na listingu 5.10.

Listing 5.10: Metoda wykorzystująca Api Last.fm

```
private readonly baseAddress = 'https://ws.audioscrobbler.com/2.0/';
private apiParams = new HttpParams({
  fromObject: {
    api_key: this.apiKey,
    format: 'json',
    limit: '30'
  }
});

topByArtistSearch(query: string): Observable<LastFmAlbum[]> {
  return this.http
    .get<LastFmApiQueryResults>(this.baseAddress, {
      params: this.apiParams
        .set('method', LastFmApiMethod.TopByArtist)
        .set('artist', query)
        .set('autocorrect', '1')
    })
    .pipe(
      map(res => res.topalbums.album)
    );
}
```

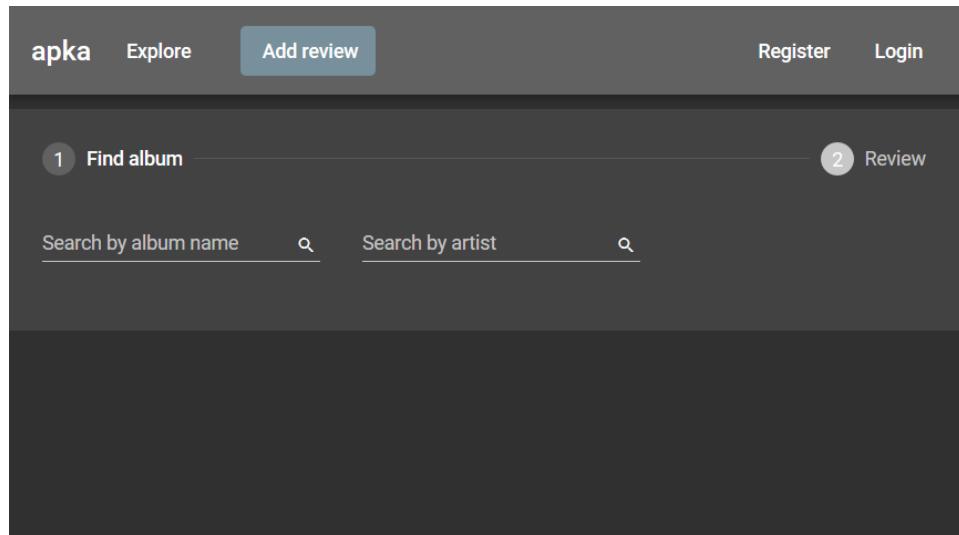
5.2.5. Responsive design

Widoki aplikacji testowane były zarówno na ekranie Full HD, jak i o rozmiarze telefonu komórkowego (414px na 736px — iPhone 7). Aby zapewnić wymaganą skalowalność interfejsu, elementy widoków zachowują się inaczej w zależności od szerokości ekranu. Atrybuty wykorzystujące responsywne API biblioteki *flex-layout* wykorzystane zostały przy budowie menu nawigacji. Atrybut `fxHide.gt-xs` chowa cały element jeśli rozmiar ekranu jest większy niż xs (599px). W ten sposób otrzymano pasek menu, który dla zwykłych ekranów zawiera przyciski nawigacji (Rys. 5.1), a dla mniejszych przełącza menu boczne (Rys. 5.2):

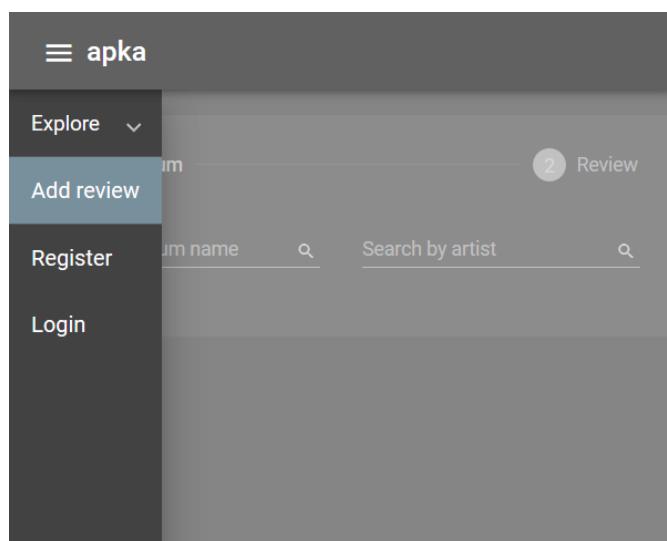
```
<div fxHide.gt-xs>
  <button mat-icon-button (click)="onToggleSidenav()">
    <mat-icon>menu</mat-icon>
  </button>
</div>

<h1>{{appName}}</h1>

<div fxFlex fxLayout="row" fxLayoutGap="10px" fxHide.xs>
  [...]
</div>
```



Rys. 5.1: Menu nawigacji dla zwykłych ekranów



Rys. 5.2: Wysunięte menu nawigacji dla małych ekranów

5.2.6. Motyw aplikacji

W aplikacji skorzystano z biblioteki *Anuglar Material*, która oprócz gotowych, często używanych komponentów, ułatwia korzystanie z globalnego motywu kolorystycznego. Definiowanie kolorystyki odbywa się za pomocą *mixin*, które określają kolory poszczególnych komponentów. Zaimplementowane zostały dwa motywy, choć dodanie kolejnego wiąże się jedynie ze zdefiniowaniem kolorystyki w jednym miejscu. Domyślny, ciemny motyw zdefiniowany został następująco:

```
$dark-primary: mat-palette($mat-grey, 700, 300, 900);
$dark-accent: mat-palette($mat-blue-grey, 400);
$dark-warn: mat-palette($mat-red, 500);

$dark-theme: mat-dark-theme($dark-primary, $dark-accent, $dark-warn);
```

Funkcja *mat-palette* zwraca paletę kolorów: pierwszy argument określa paletę koloru, a kolejne trzy odpowiednio domyślny, jasny i ciemniejszy odcień. Metoda *mat-dark-theme* oraz analogiczna *mat-light-theme* oprócz przekazanych palet kolorów definiują dodatkowo domyślne kolory tła. Na podstawie tych *mixin*-ów stworzone zostały dwa style css:

```

@mixin custom-components-theme($theme) {
  @include header-component-theme($theme);
  @include sidenav-component-theme($theme);
  @include album-grid-component-theme($theme);
  @include album-search-component-theme($theme);
}

.light-theme {
  @include angular-material-theme($light-theme);
  @include custom-components-theme($light-theme);
}

.dark-theme {
  @include angular-material-theme($dark-theme);
  @include custom-components-theme($dark-theme);
}

```

Aby wykorzystać kolory motywu we własnych komponentach (nie zawartych w paczce `angular-material-theme`), należy stworzyć nowy *mixin*, który może pobierać ustawione kolory za pomocą funkcji `map-get`. Przykładowy *mixin*, który wykorzystuje tą metodę jest następujący:

```

@mixin sidenav-component-theme($theme) {
  $primary: map-get($theme, primary);
  $accent: map-get($theme, accent);

  mat-sidenav {
    button, .mat-list-item {
      &.active {
        color: mat-color($accent, default-contrast);
        background-color: mat-color($accent);

        &:hover {
          background-color: mat-color($accent, darker);
        }
      }
    }
  }
}

```

Przełączanie pomiędzy globalnymi stylami odbywa się w głównym komponencie aplikacji `app.component`, przedstawionym na listingu 5.9, w pierwszej linijce. Odpowiednia klasa aktywowana jest na podstawie wartości `theme` ustawień użytkownika.

5.3. GraphQL API

Api stworzone zostało na platformie .Net Core 3.0. Wykorzystane zostały następujące technologie:

Hot Chocolate — platforma pozwalająca zbudować serwer GraphQL na infrastrukturze .NET.

Umożliwia definiowanie schematu GraphQL zarówno za pomocą języka SDL (ang. *Schema Definition Language*), jak i języka C# poprzez podejście code-first.

Autofac — biblioteka dostarczająca zaawansowany kontener służący do inwersji kontroli (IoC - *Inversion of Control*). Pozwala na szczegółowe lub automatyczne definiowanie interfejsów i ich implementacji.

Entity Framework Core — narzędzie służące jako ORM (ang. *Object-Relational Mapper*). Automatyzuje dostęp do bazy danych poprzez mapowanie encji bazodanowych na obiekty

POCO (ang. *Plain Old CLR Object*). Dodatkowo umożliwia generowanie bazy danych na podstawie modeli, tworząc migrację po każdej zmianie na istniejącej bazie.

5.3.1. Warstwa Api

W tej warstwie skonfigurowany został serwer GraphQL. Zdecydowano się definiować schemat metodą code-first, aby wykorzystać modele warstwy domenowej jako bazy typów.

Generowanie typów

Biblioteka *Hot Chocolate* umożliwia automatyczne generowanie typów z klas. Dodatkowo, w projekcie załączono opcjonalną funkcjonalność *Nullable reference types* wprowadzoną w C#8.0 i zaimplementowaną w .NET Core 3.0. Sprawia ona, że typy referencyjne domyślnie oznaczane są jako non-null, czyli nigdy nie będą równe null. Aby oznaczyć pole jako dopuszczające wartość null należy oznaczyć je znakiem zapytania. Jest to działanie analogiczne do typów wartościowych i pomaga zapobiegać nieobsłużonym wyjątkom *Null Reference Exception*. W użytej wersji *Hot Chocolate* (10.1.1), nie są one jednak interpretowane — wszystkie typy referencyjne oznaczane są jako obowiązkowe pola. Biblioteka ta pozwala jednak na rozszerzanie domyślnych konwencji, co postanowiono wykorzystać do zaimplementowania automatycznego oznaczania typów referencyjnych. Skorzystano przy tym z biblioteki *Namotion.Reflection*, która rozszerza refleksję o metody sprawdzające, czy w danym kontekście typ może przyjmować null, czy nie. W ten sposób powstała klasa *TypeInspector*, wprowadzająca następujące modyfikacje do automatycznego generowania typów:

- klucze obce w modelach nie są tłumaczone na pola typów graphql, ponieważ wykorzystywane są jedynie wirtualne pola nawigacyjne (ang. *navigation property*),
- pola z referencyjnymi typami są poprawnie oznaczane jako opcjonalne lub obowiązkowe.

Jeśli wymagana jest dodatkowa konfiguracja to tworzona jest funkcją *Configure*. Dla typu *AlbumType* nazwa pola *MusicBrainzId* ustawiana jest na *mbid*, gdyż pod taką nazwą pobierane jest z Api Last.fm:

```
public class AlbumType : ObjectType<Album>
{
    protected override void Configure(
        IObjectTypeDescriptor<Album> descriptor)
    {
        descriptor.Field(t => t.MusicBrainzId)
            .Type<StringType>()
            .Name("mbid");
    }
}
```

Generowanie odpowiedzi

Klasa *Query* zawiera metody, które zwracają odpowiednie wartości dla zapytań. Listing 5.11 przedstawia dwie takie metody. *GetAlbum* to prosta metoda zwracająca album o danym id z repozytorium, którego implementacja znajduje się w warstwie Infrastructure. *SearchArtists* wyszukuje artystów wykorzystując metodę *Search*, która jako argument przyjmuje obiekt typu *Expression<Func<T, bool>*. *Expression tree* w przeciwnieństwie do delegatów typu *Func<>* nie kompiluje się do IL (ang. *Intermediate Language*), dzięki czemu Entity Framework może przetłumaczyć wyrażenie na zapytanie SQL. C# automatycznie tłumaczy anonimowe funkcje lambda na wyrażenie, jednak nie wszystkie metody są tłumaczone. Z tego powodu podczas porównania

nazwy artysty z zapytaniem na obu argumentach wykonywana jest funkcja `ToLower()`, zamiast użycia `StringComparison.OrdinalIgnoreCase`.

Listing 5.11: Fragment klasy Query

```
private readonly IRepository _repository;

public Query(IRepository repository)
{
    _repository = repository;
}

public Album? GetAlbum(Guid id) => _repository.GetById<Album>(id);

public IEnumerable<Artist> SearchArtists(string? mbid, string name)
{
    List<Artist>? results = null;
    if (!string.IsNullOrEmpty(mbid))
        results = _repository.List<Artist>(artist =>
            artist.MusicBrainzId == mbid);

    return results?.Count > 0
        ? results
        : _repository.Search<Artist>(artist =>
            artist.Name.ToLower().Contains(name.ToLower()));
}
```

Middleware

Podczas przetwarzania niektórych zapytań wymagane są dodatkowe akcje. Wykorzystywane są do tego *middleware*. Przykładowym, wbudowanym middleware jest dyrektywa `authorize`. Sprawdza ona, czy zalogowany użytkownik posiada prawa do wywołania odpowiedniego pola wykorzystując wbudowany system Identity.

W projekcie stworzony został nowy middleware (Listing 5.12), który dodaje do bazy Api zalogowanego użytkownika, aby można było powiązać z nim dodawane oceny. Serwis pobiera id zalogowanego użytkownika z kontekstu, a następnie sprawdza, czy taki użytkownik już istnieje. Jeśli nie, to wykorzystuje `OpenIdService` (opisany w rozdziale 5.4.4), aby pobrać nazwę użytkownika z serwera uwierzytelniającego. Następnie dane te zapisuje w bazie za pomocą repozytorium. Na końcu przekazuje zapytanie do dalszego przetwarzania poprzez `_next(context)`. Tak napisaną klasę można zarejestrować jako middleware podczas konfiguracji dowolnego typu, np. mutacji:

```
descriptor.Field(t => t.AddRating(default!, default!))
    .Use<RecordUserMiddleware>()
    .Argument("rating", a => a.Type<NonNullType<RatingInputType>>())
    .Type<NonNullType<AlbumType>>();
```

5.3.2. Warstwa domenowa

Warstwa domenowa zawiera logikę biznesową aplikacji: encje, serwisy i interfejsy. Interfejsy definiują abstrakcje operacji, które wykonywane będą w warstwie infrastruktury. Są to dostęp do systemu plików, połączenie z bazą danych czy zapytania HTTP.

Modele encji opisane zostały w rozdziale 5.5.2. W projekcie zaimplementowano wzorzec projektowy *Repository*. Stworzono w tym celu generyczny interfejs definiujący repozytorium:

```

public interface IRepository
{
    T? GetById<T>(Guid id) where T : BaseEntity;
    List<T> List<T>() where T : BaseEntity;
    List<T> List<T>(Expression<Func<T, bool>> filter) where T : BaseEntity;
    T Add<T>(T entity) where T : BaseEntity;
    T Update<T>(T entity) where T : BaseEntity;
    void Delete<T>(T entity) where T : BaseEntity;
}

```

Przyjmowany typ T musi dziedziczyć po `BaseEntity`, co zapewnia obecność pola `Id` typu `Guid`. Implementacja repozytorium opisana została w następnym rozdziale.

Listing 5.12: Middleware zapisujące użytkowników

```

public class RecordUserMiddleware
{
    private readonly FieldDelegate _next;
    private readonly OpenIdService _openIdService;

    public RecordUserMiddleware(
        FieldDelegate next,
        OpenIdService openIdService)
    {
        _next = next;
        _openIdService = openIdService;
    }

    public async Task InvokeAsync(
        IMiddlewareContext context,
        IRepository repository,
        IHttpContextAccessor httpContextAccessor)
    {
        var id = httpContextAccessor.HttpContext.User
            .FindFirstValue(ClaimTypes.NameIdentifier);

        var user = repository.GetById<User>(new Guid(id));

        if (user == null)
        {
            var accessToken = await httpContextAccessor.HttpContext
                .GetTokenAsync("access_token");
            var userInfo = await _openIdService.GetUserInfo(accessToken);

            repository.Add(new User
            {
                Id = userInfo.Sub,
                Username = userInfo.Name
            });
        }

        await _next(context);
    }
}

```

5.3.3. Warstwa infrastruktury

W tej warstwie znajdują się implementacje dostępu do danych oraz rejestracja zależności aplikacji.

Listing 5.13: Implementacja repozytorium

```
public class EfRepository : IRepository
{
    private readonly ApplicationContext _dbContext;

    public EfRepository(ApplicationContext dbContext)
    {
        _dbContext = dbContext;
    }

    public T? GetById<T>(Guid id) where T : BaseEntity
    {
        return _dbContext.Set<T>().SingleOrDefault(e => e.Id == id);
    }

    public List<T> List<T>() where T : BaseEntity
    {
        return _dbContext.Set<T>().ToList();
    }

    public List<T> List<T>(Expression<Func<T, bool>> filter)
        where T : BaseEntity
    {
        return _dbContext.Set<T>().Where(filter).ToList();
    }

    public T Add<T>(T entity) where T : BaseEntity
    {
        _dbContext.Set<T>().Add(entity);
        _dbContext.SaveChanges();

        return entity;
    }

    public T Update<T>(T entity) where T : BaseEntity
    {
        _dbContext.Entry(entity).State = EntityState.Modified;
        _dbContext.SaveChanges();

        return entity;
    }

    public void Delete<T>(T entity) where T : BaseEntity
    {
        _dbContext.Set<T>().Remove(entity);
        _dbContext.SaveChanges();
    }
}
```

Dostęp do danych

Jest to jedyna warstwa aplikacji, która deklaruje jako zależność bibliotekę *Entity Framework Core*. Klasa `EfRepository` (Listing 5.13) implementuje interfejs zdefiniowany w warstwie Core wykorzystując EF Core. Ona również jest generyczna, ponieważ operacje na wszystkich encjach są analogiczne. Bardziej skomplikowane zapytania wymusiły by stworzenie nowego interfejsu, lub wyizolowanie funkcji `_dbContext.SaveChanges` jako osobnej metody. Wtedy możliwe było by przeprowadzenie wielu operacji w ramach tej samej transakcji. W obecnej

implementacji każde zapytanie modyfikujące bazę zamknie transakcję. W tej klasie umieszczone są również inne transformacje zapytań wykonywane za każdym razem, np. pomijanie wpisów, które oznaczone zostały jako nieaktywne (ang. *soft delete*).

Do tłumaczenia zapytań na język SQL kompatybilny z bazą PostgreSQL wykorzystano bibliotekę *Npgsql*. Przykładowo, wywołanie z listingu 5.11 wyszukujące po nazwie artysty zostanie przetłymaczone na zapytanie wykorzystujące metodę *STRPOS*, znajdującą pozycję ciągu znaków w tekście, oraz *LOWER*, zmieniającą znaki w tekście na małe litery:

```
Executing DbCommand [Parameters=[@__ToLower_0='name ']
SELECT a."Id", a."MusicBrainzId", a."Name"
FROM "Artists" AS a
WHERE STRPOS(LOWER(a."Name"), @__ToLower_0) > 0
```

Aby uniknąć dodawania referencji do bibliotek związanych z dostępem do bazy, rejestracja *DbContext* również odbywa się w tej warstwie. Stworzono w tym celu metodę rozszerzającą, która rozszerza klasę *IServiceCollection* o metodę *AddDbContext* przyjmującą connection string do bazy.

```
public static void AddDbContext(
    this IServiceCollection services, string connectionString) =>
    services.AddDbContext<AppDbContext>(options =>
    {
        options.UseLazyLoadingProxies();
        options.UseNpgsql(connectionString);
    });
}
```

Metoda ta rejestruje kontekst bazy wykorzystując funkcję *UseNpgsql* biblioteki *Npgsql*. Dodatkowo, w opcjach użyto metody *UseLazyLoadingProxies*. Jest to rozszerzenie zawarte w bibliotece *Microsoft.EntityFrameworkCore.Proxies*, które pozwala na późne ładowanie (ang. *lazy loading*) pokrewnych encji. Dzięki temu możliwe jest wykonywanie zapytań o dowolnej głębokości zagnieżdżenia bez jawnego deklaracji, które tablice mają zostać załadowane.

Należy jednak mieć na uwadze, że GraphQL sam w sobie nie stanowi rozwiązania problemu N+1. Jest to problem wynikający z ładowania pokrewnych wpisów. Dla jednego zapytania wykonywane jest kolejnych N zapytań, po jednym dla każdej relacji, stąd N+1. Przykładowo, dla zapytania zawierającego podwójne zagnieżdżenie, gdzie dla każdego albumu zwracane są piosenki:

```
query getart {
  artist(id: "a0000000-0000-0000-0000-000000000001") {
    id
    name
    albums {
      id
      title
      songs {
        id
        title
      }
    }
  }
}
```

wykonywane są następujące komendy SQL:

```
[Parameters=[@__id_0='a0000000-0000-0000-0000-000000000001']
SELECT a."Id", a."MusicBrainzId", a."Name"
FROM "Artists" AS a
WHERE (a."Id" = @__id_0) AND (@__id_0 IS NOT NULL)
LIMIT 2
```

```
[Parameters=[@__p_0='a0000000-0000-0000-0000-000000000001']]
SELECT a."Id", a."AlbumArtistId", a."AverageRating"
    a."CoverSrc", a."MusicBrainzId", a."ReleaseDate", a."Title"
FROM "Albums" AS a
WHERE (a."AlbumArtistId" = @_p_0) AND (@__p_0 IS NOT NULL)

[Parameters=[@__p_0='b0000000-0000-0000-0000-000000000000']]
SELECT s."Id", s."AlbumId", s."ArtistId", s."Length"
    s."MusicBrainzId", s."Title", s."TrackNumber"
FROM "Songs" AS s
WHERE (s."AlbumId" = @_p_0) AND (@__p_0 IS NOT NULL)

[...]

[Parameters=@__p_0='b0000000-0000-0000-0000-000000000009']
SELECT s."Id", s."AlbumId", s."ArtistId", s."Length"
    s."MusicBrainzId", s."Title", s."TrackNumber"
FROM "Songs" AS s
WHERE (s."AlbumId" = @_p_0) AND (@__p_0 IS NOT NULL)
```

W pierwszej kolejności wczytywane są dane artyści oraz jego albumów (pierwsze dwa zapytania). Następnie dla każdego z tych albumów ładowane są piosenki — kolejnych 10 zapytań do bazy. Jest to typowy problem ładowania danych, który powstaje podczas późnego ładowania powiązanych wpisów. Jednym z możliwych rozwiązań jest wcześnie ładowanie (ang. *eager loading*), czyli zadeklarowanie, które relacje mają zostać załadowane jeszcze przed wykonaniem zapytania do bazy. Jednak w związku z tym, że klient korzystający z GraphQL może zarządzać dowolną relację, nie ma możliwości statycznego zadeklarowania, które tablice mają zostać załadowane. Potrzebne jest zatem dynamiczne rozwiążanie, które będzie interpretować zapytania i dołączać odpowiednie relacje bazy danych. Można również zaimplementować *data loader* — narzędzie, które wysyła zapytania w partiiach oraz wprowadza dodatkową warstwę cache.

Rejestracja zależności

Rejestracja zależności odbywa się przy pomocy narzędzia *Autofac* (Listing 5.14). Metoda `InitializeApi`, wywoływana na końcu metody `ConfigureServices` klasy `Startup` warstwy `Api`. W pierwszej kolejności w kontenerze *Autofac* rejestrowane są wszystkie serwisy z projektów `Core` oraz `Infrastructure`. Funkcja `RegisterAssemblyTypes` skanuje całe assembly (skompilowany projekt) i wyszukuje serwisy utworzone zgodnie z przyjętymi konwencjami. Następnie do kontenera przekazywane są wszystkie dotychczas zarejestrowane serwisy, po czym skanowane jest assembly warstwy `Api`, przekazane podczas wywołania konfiguracji:

```
| ContainerSetup.InitializeApi(Assembly.GetExecutingAssembly(), services);
```

5.4. Serwis uwierzytelniający

Serwis również stworzony został na platformie Microsoftu. Do zaimplementowania głównej funkcjonalności wykorzystano następujące biblioteki:

IdentityServer4 biblioteka implementująca wszystkie protokoły związane z OpenID zgodnie ze specyfikacją (certyfikowana przez OpenID Foundation), dzięki czemu serwer akceptuje połączenia od każdego standardowego klienta

AspNetCore.Identity system do zarządzania kontami użytkowników

Zarówno rejestracja jak i logowanie wykorzystują widoki dostarczone przez Identity. Podczas wykonywania tych akcji użytkownik zostaje przekierowany na serwer Auth przy pomocy odwróconego proxy.

Listing 5.14: Rejestracja zależności wykorzystująca Autofac

```

public static IServiceProvider InitializeApi(
    Assembly webAssembly, IServiceCollection services) =>
    new AutofacServiceProvider(BaseAutofacInitialization(setupAction =>
    {
        setupAction.Populate(services);
        setupAction.RegisterAssemblyTypes(webAssembly)
            .AsImplementedInterfaces();
    }));
}

public static IContainer BaseAutofacInitialization(
    Action<ContainerBuilder>? setupAction)
{
    var builder = new ContainerBuilder();

    var coreAssembly = Assembly.GetAssembly(typeof(BaseEntity));
    var infrastructureAssembly = Assembly.GetAssembly(typeof(EfRepository));
    builder.RegisterAssemblyTypes(coreAssembly, infrastructureAssembly)
        .AsImplementedInterfaces();

    setupAction?.Invoke(builder);
    return builder.Build();
}

```

5.4.1. Konfiguracja

Na serwerze zarejestrowany został jeden klient, `angular_spa`, z którego korzystać będą użytkownicy. Poniżej opisane zostały najważniejsze opcje:

ClientId Id klienta musi być takie same zarówno po stronie serwera jak i klienta

AllowedGrantTypes dozwolone tryby uwierzytelniania. Dozwolono jedynie tryb wykorzystujący kod autoryzacji

RequirePkce wymagane użycie kodu PKCE

AllowedScopes dozwolone zakresy informacji o użytkowniku, o które może poprosić klient

RedirectUris adresy, na które może przekierować aplikacja

Listing 5.15: Konfiguracja OpenId po stronie serwera

```

ClientId = "angular_spa",
ClientName = "Angular SPA",
RequireConsent = false,
AllowedGrantTypes = GrantTypes.Code,
RequirePkce = true,
RequireClientSecret = false,
AllowedScopes = {"openid", "profile", "email"},
RedirectUris = {
    "http://localhost/login-callback",
    "http://localhost/register-callback",
    "http://localhost/silent-refresh"},
PostLogoutRedirectUris = {"http://localhost/"},
AllowedCorsOrigins = {"http://localhost"},
AllowAccessTokensViaBrowser = true,
AccessTokenLifetime = 3600

```

Listing 5.16: Konfiguracja OpenId po stronie Spa

```
authority: environment.authServerUri,
client_id: 'angular_spa',
response_type: 'code',
scope: 'openid profile email api.read',
filterProtocolClaims: true,
loadUserInfo: true,
automaticSilentRenew: true,
redirect_uri: '${environment.thisUri}/login-callback',
post_logout_redirect_uri: '${environment.thisUri}/',
silent_redirect_uri: '${environment.thisUri}/silent-refresh'
```

5.4.2. Rejestracja

Podczas rejestracji użytkownik podaje login i hasło. Na podstawie tych danych tworzony jest nowe konto i zapisywane jest w bazie danych przez pomocniczy obiekt klasy `UserManager`, jak ukazano na listingu 5.17. Następnie użytkownik zostaje przekierowany z powrotem na serwis SPA.

Listing 5.17: Fragment kodu rejestracji

```
var user = new IdentityUser {UserName = Input.Email, Email = Input.Email};
var result = await _userManager.CreateAsync(user, Input.Password);
if (result.Succeeded)
{
    await _signInManager.SignInAsync(user, isPersistent: false);
    return Redirect(returnUrl);
}
```

5.4.3. Logowanie

Logowanie odbywa się wykorzystując protokół OpenID z kodem autoryzacji i PKCE opisanym w rozdziale 4.5. W pierwszym kroku klient pobiera konfigurację serwera dostępną pod ustandaryzowanym adresem `/.well-known/openid-configuration`. Jest to plik typu JSON definiujący pełną konfigurację serwisu uwierzytelniającego oraz adresy punktów dostępnych. Na podstawie tej konfiguracji klient wykonuje zapytanie logowania pod adres `/connect/authorize` z następującymi parametrami:

```
client_id: angular_spa
redirect_uri: http://localhost/login-callback
response_type: code
scope: openid profile email api.read
state: 785f3544ba2a4df896290ccf5fedd51b
code_challenge: 9e5S4ass5foL7Y4ye6JQfl3Uuyd9HzmYli4Y732RLqys
code_challenge_method: S256
response_mode: query
```

Parametr `code_challenge` to wartość `code_verifier` zmodyfikowana przez funkcję S256. Funkcja ta zdefiniowana jest w punkcie 4.2 RFC 7636 [6]:

```
| code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))
```

Zapytanie to skutkuje przekierowaniem na stronę logowania `/Login`. Po podaniu prawidłowych danych użytkownik przekierowywany jest z powrotem na główną stronę, a w pasku adresu widoczne są następujące parametry:

```
code: stR57rgpV11RCd09AMn2PBuxKh0tb3CoW_V9ULEhChg
scope: openid profile email api.read
state: 785f3544ba2a4df896290ccf5fedd51b
session_state: [...]
```

Następnie w tle wykonywane jest zapytanie pod adresem /connect/token pobierające tokeny JWT:

```
client_id: angular_spa
code: stR57rgpV11RCd09AMn2PBuxKh0tb3CoW_V9ULEhChg
redirect_uri: http://localhost/login-callback
code_verifier: 5d797a22803645fc870b1f1d6c9e5dfc...
grant_type: authorization_code
```

Aplikacja precyzuje tryb uwierzytelniania na authorization_code i wysyła otrzymany przed chwilą kod autoryzacji. Przesyłany jest również code_verifier — klucz, który wcześniej został przesłany w zmodyfikowanej wersji. Serwer przekształca code_verifier sprecyzowaną wcześniej funkcją code_challenge_method i porównuje z code_challenge. Jeśli są sobie równe, w odpowiedzi aplikacja otrzymuje zarówno id_token, jak i access_token:

```
id_token: [...]
access_token: [...]
expires_in: 3600
token_type: Bearer
scope: openid profile email api.read
```

Uzyskany token zostaje użyty do pobrania informacji o użytkowniku. Służy do tego punkt połączeniowy /connect/userinfo. Przesyłany jest nie jako parametr, lecz jako nagłówek HTTP Authorization: Bearer [access_token]. W odpowiedzi serwer zwraca dane użytkownika identyfikującym się tym tokenem:

```
sub: 0d6dfb71-745e-47c2-81f0-c9649bf0a2b2
preferred_username: a@a.a
name: a@a.a
email: a@a.a
email_verified: false
```

Ten sam token przekazywany jest podczas zapytań do serwisu API.

5.4.4. Autoryzacja dostępu do API

Serwis Apollo po stronie SPA przekazuje w nagłówku autoryzacji token dostępowy, jak ukazano na listingu 5.5. Do walidacji tokena po stronie API użyta została wbudowana w .NET Core metoda AddJwtBearer:

```
services.AddAuthentication("Bearer").AddJwtBearer("Bearer", options =>
{
    options.Authority = "http://authServer";
    options.Audience = "reServer";
    options.RequireHttpsMetadata = false;
    options.TokenValidationParameters.ValidIssuer = "http://auth";
    options.SaveToken = true;
});
```

W opcjach podano parametry służące do walidacji tokena:

Authority to adres serwera, który wystawił token

Audience to identyfikator serwera zawierającego dane, do których dostęp daje access_token

RequireHttpsMetadata szyfrowane połączenie nie jest wymagane, ponieważ zapytania do serwisu Auth nie wychodzą poza wewnętrzną sieć dockerową

TokenValidationParameters.ValidIssuer ustawione na taką samą wartość jak po stronie serwisu Auth. Jest to wymagane, ponieważ serwis Auth przyjmuje zapytania zarówno z zewnątrz przez odwrócone proxy, jak i z wewnętrznej sieci dockerowej.

Tak ustwiony middleware rejestrowane jest w funkcji Configure, która definiuje kolejność obsługiwanego zapytań HTTP:

```
app.UseAuthentication();
app.UseAuthorization();
```

Dzięki temu dane uwierzytelnionego użytkownika dostępne są w `HttpContext.User` i mogą zostać wykorzystane w innych warstwach aplikacji, jak ukazano na listingu 5.12. Klasa ta wykorzystuje serwis `OpenIdService` — skonfigurowany klient HTTP, który łączy się z serwisem Auth (Listing 5.18). Dane pobierane są z adresu `/connect/userinfo` przesyłając token w nagłówku `Authorization`. Odpowiedź z serwera serializowana jest do zdefiniowanej klasy `UserInfo`.

Listing 5.18: Serwis HTTP pobierający dane użytkownika

```
public class OpenIdService
{
    public HttpClient Client { get; }

    public OpenIdService(HttpClient client)
    {
        client.BaseAddress = new Uri("http://authServer/connect/");
        Client = client;
    }

    public async Task<UserInfo> GetUserInfo(string accessToken)
    {
        Client.DefaultRequestHeaders.Authorization
            = new AuthenticationHeaderValue("Bearer", accessToken);
        var response = await Client.GetAsync("userinfo");
        response.EnsureSuccessStatusCode();

        using var responseStream = await response.Content.ReadAsStreamAsync();
        return await JsonSerializer.DeserializeAsync<UserInfo>(responseStream);
    }
}
```

5.5. Baza danych

W tym samym kontenerze `mainDB` utworzone zostały dwie bazy:

- `AuthServerDB`
- `ReServerDB`

Są to odpowiednio: baza z kontami użytkowników utworzona przez serwis Auth, oraz główna baza danych aplikacji wykorzystywana przez serwis API. Obie te bazy utworzone zostały metodą `code first` przy pomocy migracji Entity Framework Core.

5.5.1. AuthServerDB

W projekcie są dwa różne konteksty bazy danych:

ApplicationDbContext główny kontekst, który rozszerza **IdentityDbContext** definiujący bazę używaną przez Identity

PersistedGrantDbContext kontekst wygenerowany przez IdentityServer4, generuje tablice do przechowywania kodów i tokenów

Oba konteksty rejestrowane są standardowo w klasie **Startup**:

```
services.AddDbContext<ApplicationContext>(options =>
    options.UseNpgsql(
        Configuration.GetConnectionString("DefaultConnection")));

services.AddIdentityServer(options => options.IssuerUri = "http://auth")
    .AddOperationalStore(options =>
{
    options.ConfigureDbContext = builder => builder.UseNpgsql(
        Configuration.GetConnectionString("DefaultConnection")),
    options => options.MigrationsAssembly(
        Assembly.GetExecutingAssembly().GetName()));
    options.EnableTokenCleanup = true;
})
```

Oba połączenia wykorzystują ten sam *connection string*, ponieważ dane przechowywane są w jednej wspólnej bazie. Dwa osobne konteksty skutkują również tym, że generowane są osobne migracje. Obie jednak mogą zostać wykonane na tej samej bazie danych niezależnie, gdyż nie są ze sobą połączone żadnymi relacjami.

5.5.2. ReServerDB

Baza generowana jest z modeli znajdujących się w projekcie Core (opisanym w rozdziale 5.3.2). Wszystkie modele rozszerzają klasę **BaseEntity**, która definiuje pole **Id** typu Guid. Encje posiadają zadeklarowane zarówno klucze obce, jak i wirtualne ścieżki. Wirtualne ścieżki wykorzystywane są do zagnieżdżania zapytań graphql, natomiast klucze obce podczas dodawania nowych wpisów do bazy. Modele połączone są relacjami jeden do wielu. Aby określić relację między modelami *Album* i *Song*, w modelu *Album* tworzona jest lista typu **ICollection<Song>**, natomiast w modelu *Song* tworzona jest pole typu **Album**. Analogicznie zamodelowana została relacja między *Album*, a *Artist*:

```
public class Album : BaseEntity
{
    public virtual Artist AlbumArtist { get; set; }
    public virtual ICollection<Song> Songs { get; set; }
    public virtual ICollection<Rating> Ratings { get; set; }
}
```

Relacje te są wykrywane przez Entity Framework Core, który poprawnie tworzy ograniczenia (ang. *constraints*) na bazie. Kontekst bazy zawarty jest w warstwie infrastruktury i opisany został w rozdziale 5.3.3.

5.6. Odwrócone proxy

Odwrócone proxy zaimplementowane zostało przy pomocy narzędzia *nginx*. Utworzono dwa pliki konfiguracyjne:

main.conf przekierowanie na SPA oraz API

authserver.conf przekierowanie na Auth

Na listingu 5.19 przedstawiono konfigurację **main.conf**. Serwer nasłuchiwał zapytań kierowanych pod adresem **localhost:80**. Jest to więc konfiguracja dla lokalnego środowiska.

Na serwerze zdefiniowano dwie lokalacje: domyślna lokalizacja / kierująca połączenia na serwis spa, oraz lokalizacja /api/ kierująca zapytania do serwisu reserver.

Podobnie przedstawia się konfiguracja authserver.conf (listing 5.20), która nasłuchuje na adresie auth.localhost:80. Zdefiniowana jest tylko jedna, główna lokalizacja kierująca połączenia do serwisu auth. Dodatkowo, skonfigurowana została obsługa zapytań przełączających protokół komunikacji na websocket. Domyślnie nagłówki Upgrade oraz Connection nie są przekazywane przez proxy, dlatego musiały zostać ręcznie zmapowane.

Wszystkie lokalizacje importują tą samą konfigurację, przedstawioną na listingu 5.21. Ustawienia te przekazują podstawowe nagłówki zapytań, dzięki czemu dla serwisów połączenia wyglądają tak, jakby przyszły bezpośrednio od klienta.

Listing 5.19: Ustawienia proxy SPA i API

```
upstream spa {
    server           spa:4200;
}

upstream reserver {
    server           reServer;
}

server {
    listen          80;
    server_name     localhost;

    location / {
        proxy_pass  http://spa;
        include     conf.d/common_location.conf;
    }

    location /api/ {
        proxy_pass  http://reserver/;
        include     conf.d/common_location.conf;
    }
}
```

Listing 5.20: Ustawienia proxy Auth

```
upstream auth {
    server          authServer;
}

map $http_upgrade $connection_upgrade {
    default upgrade;
    ''      close;
}

server {
    listen        80;
    server_name   auth.localhost;

    location / {
        proxy_pass  http://auth;
        include     conf.d/common_location.conf;
        proxy_http_version 1.1;
        proxy_set_header Upgrade    $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
    }
}
```

Listing 5.21: Wspólna konfiguracja lokalizacji proxy

```
proxy_set_header X-Real-IP           $remote_addr;
proxy_set_header X-Forwarded-For     $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto   $scheme;
proxy_set_header Host               $host;
proxy_set_header X-Forwarded-Host   $server_name;
proxy_set_header X-Forwarded-Port   $server_port;
```

Rozdział 6

Testy aplikacji

6.1. Testy jednostkowe

Przeprowadzone testy jednostkowe sprawdzały poprawność logiki programu. Testowaną warstwą jest więc warstwa *Core*. Podczas testów wykorzystano bibliotekę *xunit*. Listing 6.1 przedstawia test sprawdzający poprawność dodawania oceny albumu. Po dodaniu oceny aktualizowana jest średnia ocena. Test przeprowadzono dla wartości zwracającej liczbę zmiennoprzecinkową, aby upewnić się że dzielenie wykonywane jest prawidłowo. Jako wartość początkową ustawiono album z dwoma ocenami dającymi średnią równą 2. Następnie dodano ocenę o wartości 4 i sprawdzono, czy nowa średnia jest równa 8/3.

Listing 6.1: Test jednostkowy dodawania oceny albumu

```
[Fact]
public void AddSubsequentRating()
{
    var ratings = new List<Rating>
    {
        new RatingBuilder().WithDefaultValues()
            .Stars(1).Build(),
        new RatingBuilder().WithDefaultValues()
            .Stars(3).Build()
    };

    var album = new AlbumBuilder().WithDefaultValues()
        .Ratings(ratings).Build();

    var newRating = new RatingBuilder().WithDefaultValues()
        .Stars(4).Build();

    album.AddRating(newRating);

    var expected = (float)(8.0 / 3);
    Assert.Equal(expected, album.AverageRating);
}
```

6.2. Testy integracyjne

Testy integracyjne sprawdzają zgodność komunikacji pomiędzy komponentami. Przeprowadzone zostały testy wykorzystujące wszystkie trzy warstwy serwisu API. Aby było to możliwe, zaimplementowano serwer testowy przy pomocy biblioteki *AspNetCore.Mvc.Testing* (Listing 6.2). Klasa *WebApplicationFactory* pozwala na stworzenie serwera, który obsługuje zapytania HTTP, bezpośrednio w pamięci podczas wykonywania testów. W serwerze skonfigurowano jedynie podstawowe wymagane serwisy:

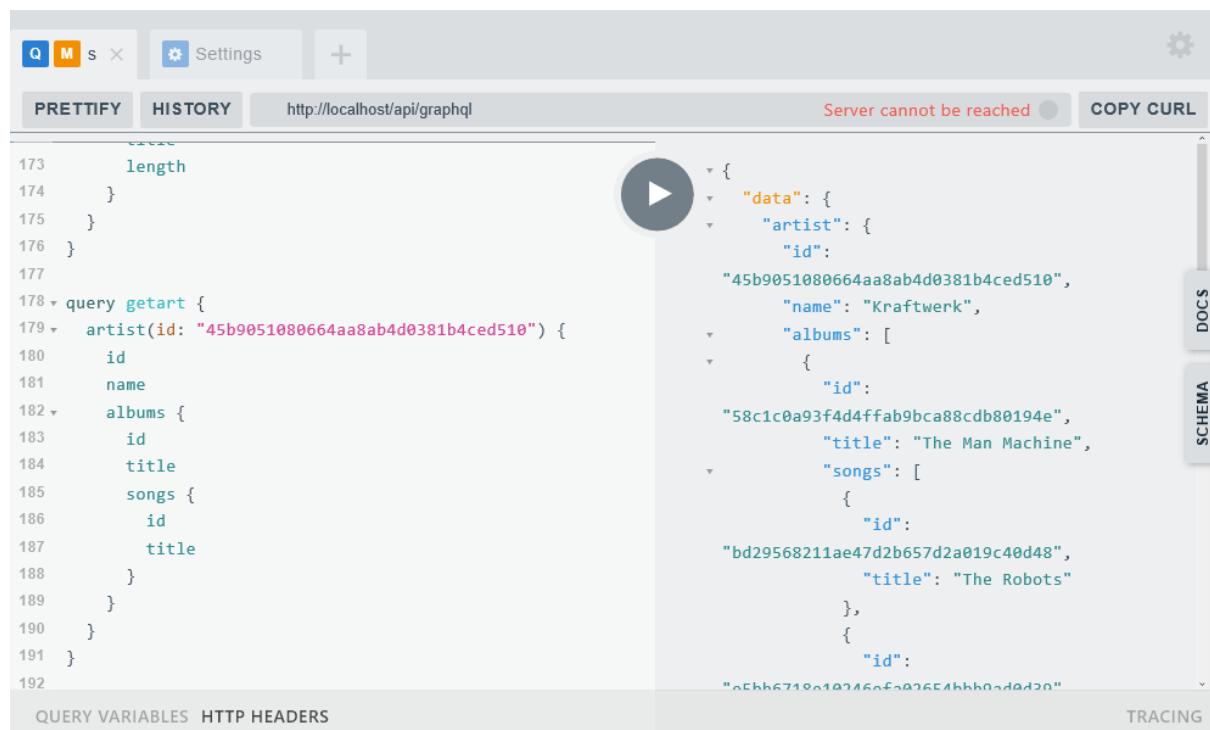
- GraphQL wykorzystujące te same typy, co serwis API,
- *DbContext* łączące się do bazy w pamięci
- *IRepository* i jego implementację z warstwy *Infrastructure*

Na sam koniec wywołano funkcję *SeedData.PopulateTestData*, dodającą dane testowe do bazy.

Na listingu 6.3 przedstawiono test wykorzystujący tą klasę. Sprawdzane jest, czy oprócz wczytywania i formatowania danych z bazy, poprawnie odbywa się ładowanie pokrewnych wpisów (*albumArtist*). Na początku definiowane jest zapytanie w języku GraphQL, które zwraca tytuł albumu oraz nazwę wykonawcy. Dane te są porównywane ze znanyymi, domyślnymi danymi w bazie.

6.3. Testy akceptacyjne

Przeprowadzone zostały testy akceptacyjne sprawdzające poprawność działania aplikacji. Aby przetestować API wykorzystano narzędzie *Playground* przedstawione na zrzucie ekranu 6.1. Pozwala ono na pisanie zapytań w języku GraphQL i wywoływać je na serwerze API. Dodatkowo, wczytuje schemat API dzięki czemu oferuje podpowiedzi podczas pisania zapytań.



The screenshot shows the GraphQL Playground interface. At the top, there are tabs for 'PRETTIFY', 'HISTORY', and the URL 'http://localhost/api/graphql'. To the right of the URL, it says 'Server cannot be reached'. Below the URL, there's a large text area containing a GraphQL query. The query is as follows:

```

173     length
174   }
175 }
176 }

177
178 + query getArtist {
179 +   artist(id: "45b9051080664aa8ab4d0381b4ced510") {
180     id
181     name
182     albums {
183       id
184       title
185       songs {
186         id
187         title
188       }
189     }
190   }
191 }
192

```

To the right of the query, the results are displayed in a tree-view format. The results show the artist's name ('Kraftwerk') and their albums. One album is shown in full, with its title ('The Man Machine') and songs ('The Robots'). The song titles are partially visible as '...'. The interface also includes buttons for 'PLAY' and 'COPY CURL', and tabs for 'DOCS' and 'SCHEMA' on the right side.

Rys. 6.1: Narzędzie do testowania API GraphQL

Listing 6.2: Serwer do testów

```

public class WebAppTestFactory<TStartup> : WebApplicationFactory<Startup>
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder
            .Configure(app => app.UseGraphQL("/"))
            .ConfigureServices(services =>
        {
            var serviceProvider = new ServiceCollection()
                .AddEntityFrameworkInMemoryDatabase()
                .AddEntityFrameworkProxies()
                .BuildServiceProvider();

            services.AddDbContext<AppDbContext>(options =>
            {
                options.UseInMemoryDatabase("TestDB");
                options.UseLazyLoadingProxies();
                options.UseInternalServiceProvider(serviceProvider);
            });
            services.AddScoped< IRepository , EfRepository>();

            services.AddGraphQL(sp => SchemaBuilder.New()
                .AddServices(sp)
                .AddQueryType<QueryType>()
                .AddMutationType<MutationType>()
                .BindClrType<Guid , IdType>()
                .Create());
        });

        var sp = services.BuildServiceProvider();

        using (var scope = sp.CreateScope())
        {
            var scopedServices = scope.ServiceProvider;
            var db = scopedServices.GetRequiredService<AppDbContext>();

            db.Database.EnsureCreated();
            SeedData.PopulateTestData(db);
        }
    });
}
}

```

Przeprowadzono również testy akceptacyjne procesu rejestracji i logowania użytkownika. Podczas tych testów upewniono się, że w konsoli przeglądarki nie można podejrzeć danych pozwalających na oszukanie systemu. Komunikacja była poprawnie zabezpieczona i zgodna z protokołem OpenID Connect z kodem autoryzacji i PKCE.

W głównym interfejsie sprawdzono poprawność dodawania ocen i recenzji albumów. Albumy są poprawnie zwracane z API Last.fm, zarówno wyszukując po nazwie albumu, jak i po artyście. Po wybraniu albumu można poprawnie dodać do niego ocenę, a album zostanie wyświetlony na głównej liście.

Listing 6.3: Test integracyjny zapytania GraphQL

```
public class GraphQLTests : IClassFixture<WebAppTestFixture<Startup>>
{
    private readonly HttpClient _client;

    public GraphQLTests(WebAppTestFixture<Startup> factory)
    {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task ReturnsAlbumWithArtist()
    {
        var request = new JObject();
        request.Add("query",
            "query getAlbum{ album(id: \"b0000000000000000000000000000001\")" +
            "{ title albumArtist {name}}}");

        var response = await _client.PostAsync("/", new StringContent(
            request.ToString(), Encoding.UTF8, "application/json"));

        response.EnsureSuccessStatusCode();

        var responseString = await response.Content.ReadAsStringAsync();
        dynamic actual = JObject.Parse(responseString);

        Assert.Equal("Album 1", Convert.ToString(actual.data.album.title));
        Assert.Equal("Artist 1",
            Convert.ToString(actual.data.album.albumArtist.name));
    }
}
```

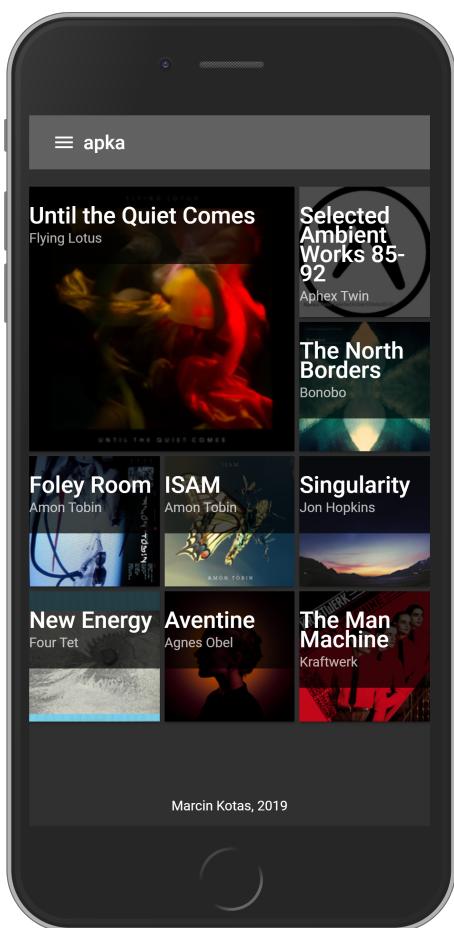
Rozdział 7

Prezentacja aplikacji

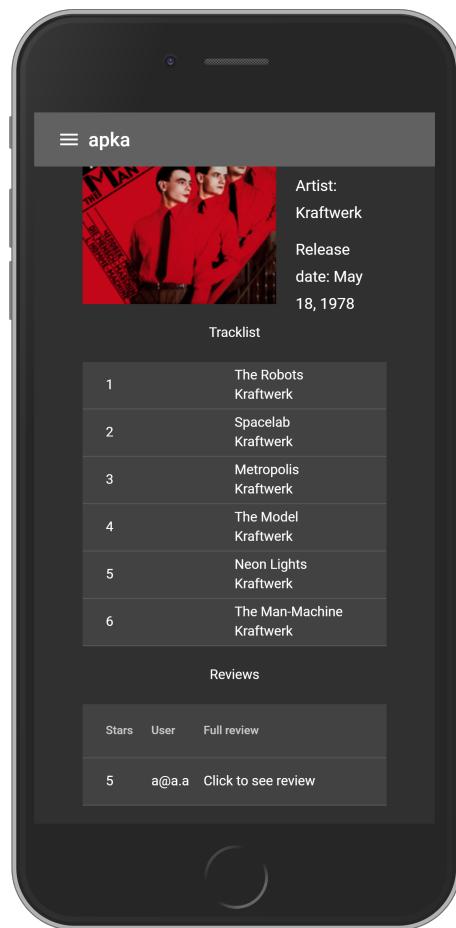
W projekcie zaimplementowane zostały podstawowe widoki wykorzystujące stworzoną architekturę. Zrzuty ekranu wykonane zostały w narzędziach developerskich przeglądarki Chrome, symulując ekran telefonu Iphone 7.

7.1. Widoki albumów

Po kliknięciu w przycisk menu „Explore” (widok menu ukazany został na rysunkach 5.2 oraz 5.1) wyświetlana jest lista albumów. Aby wyświetlić widok ze szczegółami (Rys. 7.2) należy nacisnąć kafelek z albumem.



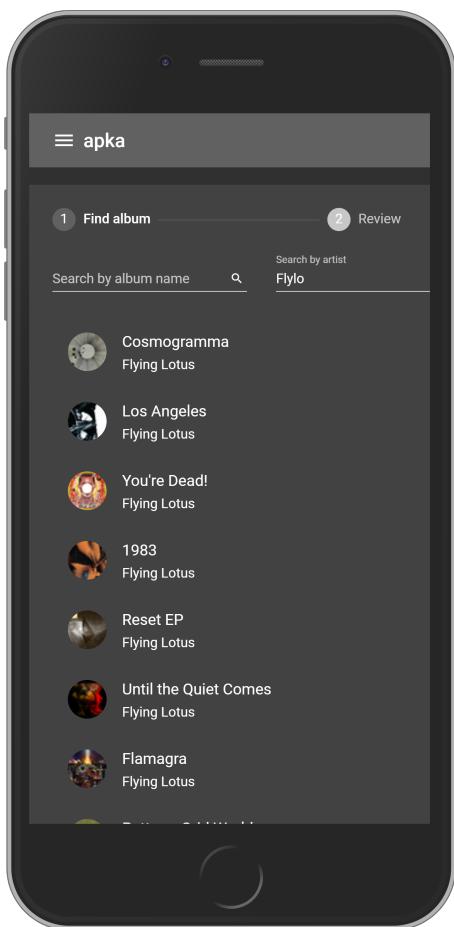
Rys. 7.1: Widok wyświetlający listę albumów



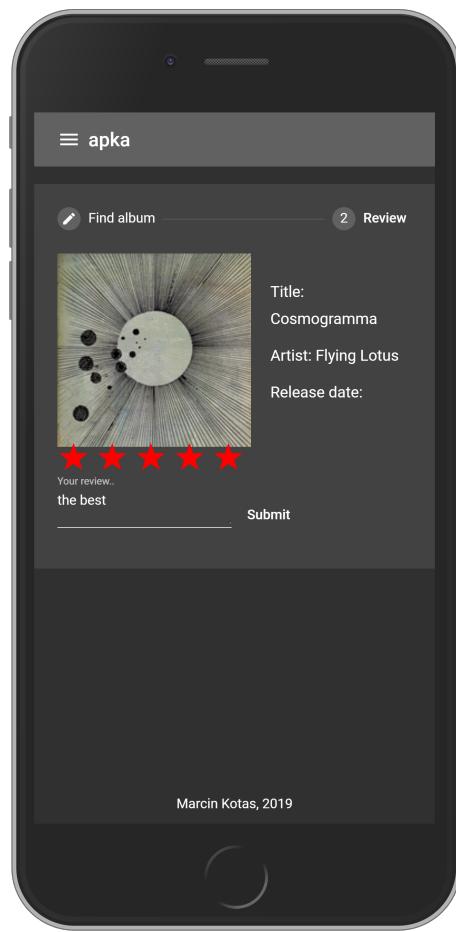
Rys. 7.2: Widok przedstawiający szczegóły albumu

7.2. Dodawanie oceny

Dodawanie oceny odbywa się w dwóch krokach. Najpierw użytkownik wyszukuje album wg nazwy albumu, lub wpisuje nazwę artysty. W drugim przypadku zwracane są najpopularniejsze albumy danego artysty. Następnie, po kliknięciu w wybrany album użytkownik ma możliwość dodania oceny w skali 1-5 oraz opcjonalnie recenzji tekstuowej (Rys. 7.4).



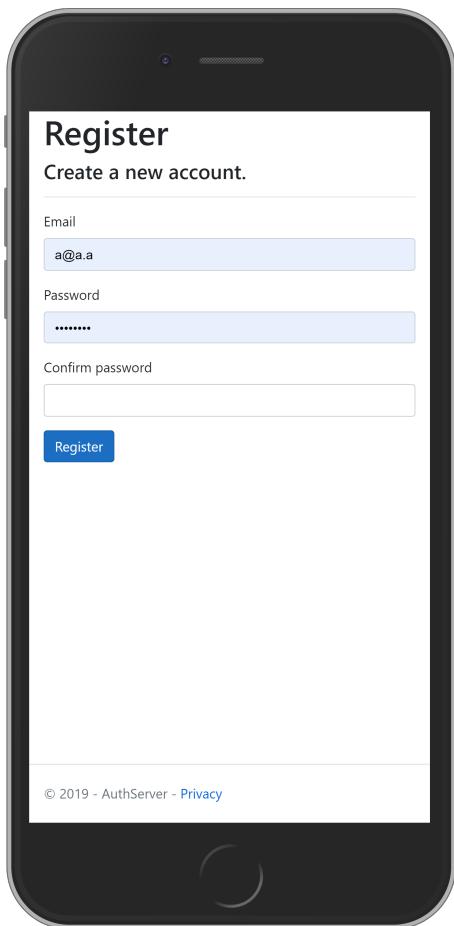
Rys. 7.3: Ekran wyszukiwania albumu



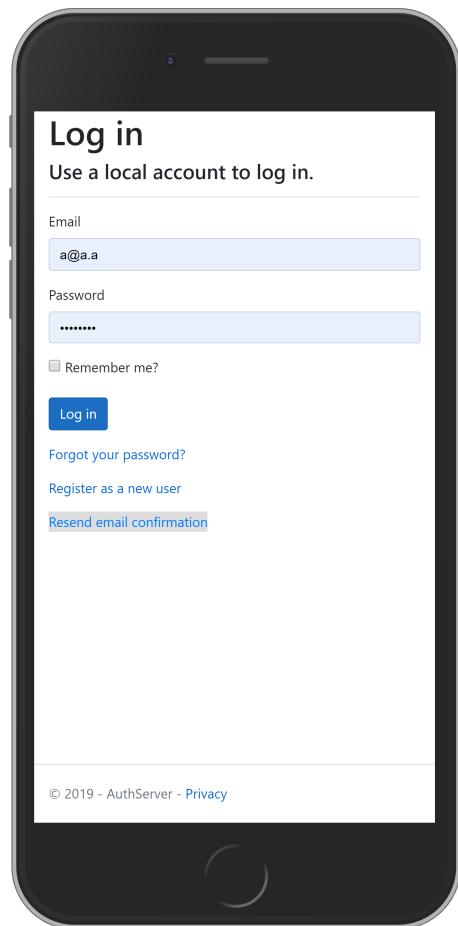
Rys. 7.4: Ekrany dodawania oceny

7.3. Rejestracja i logowanie

Zarówno rejestracja, jak i logowanie odbywa się po stronie serwisu Auth. Z tego powodu widoki te odstają stylistycznie od reszty aplikacji.



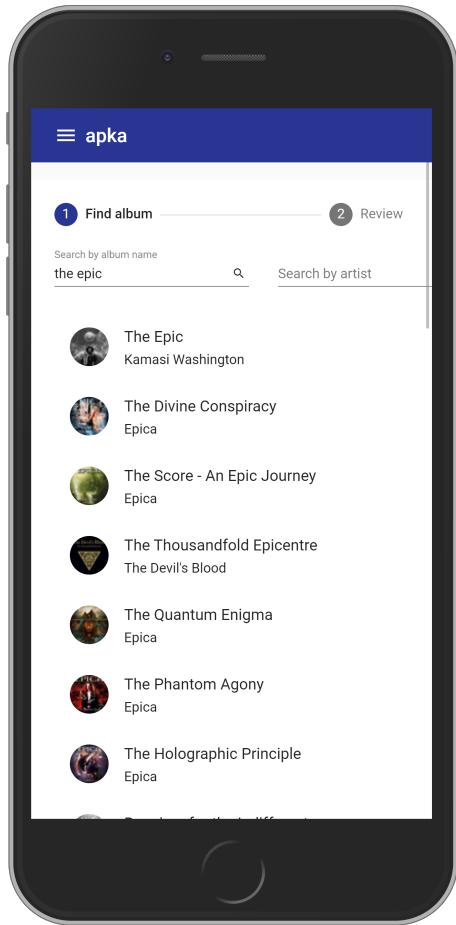
Rys. 7.5: Ekran rejestracji



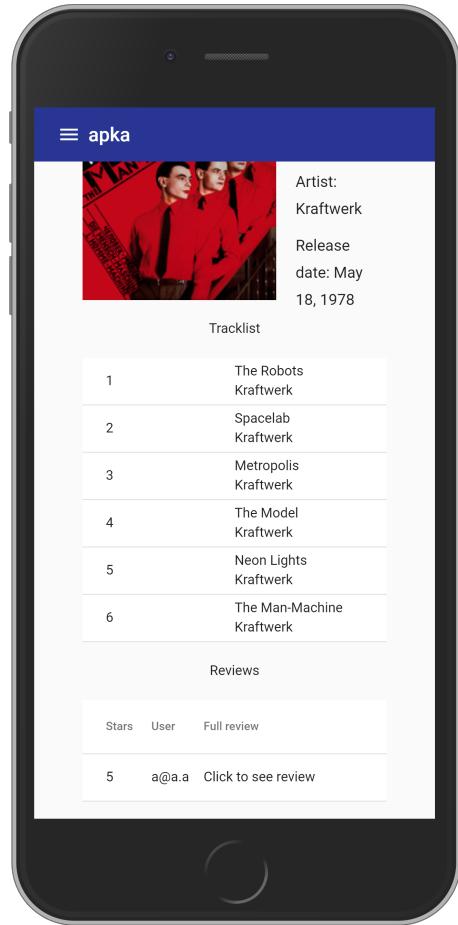
Rys. 7.6: Ekran logowania

7.4. Jasny motyw

Wszystkie widoki dostępne są również w jasnym motywie. Rysunek 7.7 przedstawia wyszukiwanie po nazwie albumu, natomiast na rysunku 7.8 przedstawiono widok 7.2 w jasnym motywie.



Rys. 7.7: Ekrany wyszukiwania albumu



Rys. 7.8: Widok przedstawiający szczegóły albumu

Rozdział 8

Podsumowanie

8.1. Wykonane prace

Stworzone zostało kompletne rozwiązanie pozwalające na sprawną rozbudowę zaawansowanej aplikacji. Zastosowano najnowsze rozwiązania z dziedziny projektowania stron internetowych, co pozwoliło na stworzenie uniwersalnej strony działającej zarówno na komputerach, jak i telefonach komórkowych. Jednocześnie dzięki zapytaniom wykorzystującym protokół GraphQL oraz zaawansowanemu systemowi cache, zużycie danych przesyłanych przez internet jest ograniczone do minimum. Dzięki temu GraphQL jest obecnie postrzegane jako przyszłość projektowania API dla stron internetowych typu SPA.

Zaimplementowano jeden z najbezpieczniejszych na chwilę obecną systemów uwierzytelniania i autoryzacji, OpenID Connect. Wykorzystany tryb uwierzytelniania z użyciem kodu autoryzacji i PKCE zapewnia największą ochronę użytkownika korzystającego ze strony w przeglądarce internetowej. Podczas tworzenia widoków utrzymano konwencję *Material design*, użyskując nowoczesny wygląd zgodny ze standardem na urządzeniach z systemem Android. Całość aplikacji stworzona została na platformie Docker, dzięki czemu może zostać bezproblemowo uruchomiona na każdej maszynie.

Zgodnie z założeniami, w projekcie wykorzystano tylko rozwiązania open source. Kod stworzonej aplikacji również udostępniony jest publicznie na platformie GitHub. Wykorzystanie systemu kontroli wersji Git umożliwia łatwe rozbudowanie aplikacji oraz ułatwia współpracę, jeśli osoby trzecie chciałyby rozpocząć współpracę przy rozwijaniu programu.

8.2. Wnioski

GraphQL pozwala na budowanie zaawansowanego API przy utrzymaniu zrozumiałej i czytelnej architektury API (brak powielanych endpoint-ów). Rozwiązuje dwa typowe problemy API: *underfetching* oraz *overfetching*, czyli odpowiednio pobieranie za mało lub za dużo danych z API. Wymaga jednak implementacji skomplikowanego cache i uniemożliwia stworzenie cache na poziomie sieci. Należy również pamiętać, że jest to jedynie protokół przesyłania danych i formatowania zapytań. Nie rozwiązuje problemów optymalizacyjnych występujących w API typu RESTful (np. problem N+1). W dalszym ciągu potrzebna jest dodatkowa logika, która optymalizować będzie zapytania do bazy.

Serwiisy stworzone w .NET Core sprawdzają się bardzo dobrze pod kątem bezpieczeństwa oraz stabilności. Stworzone przez Microsoft biblioteki pomocnicze pozwalają na stworzenie zaawansowanej funkcjonalności z zachowaniem pewności co do jakości kodu.

8.3. Możliwości dalszej rozbudowy

Całość projektu stanowi zaawansowany szkielet, na podstawie którego można stworzyć rozbudowaną aplikację.

Serwis API powinien zostać zoptymalizowany pod kątem zapytań do bazy. Dodatkowo, jeśli przewiduje się dużo danych, po stronie API należy zaimplementować mechanizm paginacji i sortowania. Zarówno Hot Chocolate, jak i Entity Framework Core wspierają operacje tego typu.

Zastosowany framework Angular oferuje wsparcie dla PWA (ang. *Progressive Web Application*) poprzez dodanie service worker-ów. Są to pomocnicze skrypty działające w tle, dzięki którym strona może działać bez połączenia z internetem. Aplikacja została zbudowana wykorzystując wiele procesów automatyzacji, dzięki czemu możliwe jest bardzo szybkie rozwijanie API o dodatkowe modele bazy i typy GraphQL. Użycie Entity Framework Core pozwala na proste modyfikacje nawet na istniejącej bazie poprzez zarządzanie migracjami.

Literatura

- [1] Dispersia/dotnet-watch-docker-example. <https://github.com/Dispersia/Dotnet-Watch-Docker-Example>. dostęp dnia 20 listopada 2019.
- [2] GraphQL | A query language for your API. <https://graphql.org>. dostęp dnia 17 listopada 2019.
- [3] GraphQL Spec, Czerwiec 2018. <https://graphql.github.io/graphql-spec/June2018>. dostęp dnia 17 listopada 2019.
- [4] HATEOAS - Wikipedia. <https://en.wikipedia.org/wiki/HATEOAS>. dostęp dnia 17 listopada 2019.
- [5] ReactiveX. <http://reactivex.io>. dostęp dnia 20 listopada 2019.
- [6] RFC 7636 - Proof Key for Code Exchange by OAuth Public Clients. <https://tools.ietf.org/html/rfc7636>. dostęp dnia 17 listopada 2019.
- [7] XML Soap. https://www.w3schools.com/xml/xml_soap.asp. dostęp dnia 17 listopada 2019.
- [8] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014.
- [9] E. Porcello, A. Banks. *Learning GraphQL*. O'Reilly Media, 2018.
- [10] L. Richardson, M. Amundsen. *RESTful Web APIs*. O'Reilly Media, 2013.
- [11] D. Tidwell, J. Snell, P. Kulchenko. *Programming Web Services with SOAP*. O'Reilly Media, 2001.

Spis rysunków

3.1. Diagram przypadków użycia	12
4.1. Architektura fizyczna aplikacji	15
4.2. Schemat czystej architektury użytej w projekcie	16
4.3. Schemat ERD bazy z kontami użytkowników	17
4.4. Schemat ERD bazy z głównymi danymi	18
4.5. Schemat procesu logowania z użyciem PKCE	18
5.1. Menu nawigacji dla zwykłych ekranów	29
5.2. Wysunięte menu nawigacji dla małych ekranów	29
6.1. Narzędzie do testowania API GraphQL	45
7.1. Widok wyświetlający listę albumów	48
7.2. Widok przedstawiający szczegóły albumu	48
7.3. Ekran wyszukiwania albumu	49
7.4. Ekrany dodawania oceny	49
7.5. Ekran rejestracji	50
7.6. Ekran logowania	50
7.7. Ekrany wyszukiwania albumu	51
7.8. Widok przedstawiający szczegóły albumu	51

Dodatek A

Opis załączonej płyty CD/DVD

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