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:

WROCŁAW, 2019

Spis treści

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 była implementacja strony internetowej pozwalającej dodawać recenzje albumów muzycznych, wykorzystując najnowsze technologie z tej dziedziny zgodne z obecnymi trendami. Główny nacisk nałożony zostanie na stworzenie architektury stanowiącej solidną bazę do dalszego rozwijania aplikacji. Projekt wykorzystywać będzie tylko oprogramowanie open source.

1.2. Zakres pracy

W pierwszym rozdziale przedstawiono cel oraz zakres pracy. Kolejny rozdział zawiera informacje na temat technologii, które zostały wykorzystanie 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 następnym 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.

dokończyć strukturę pracy jak będą wszystkie rozdziały

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.

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 [12]. 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 [8]:

```
<?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 [11]:

1. Klient-Serwer: podział ról i odpowiedzialności – klient i serwer mogą być rozwijane niezależnie, o ile interfejs pozostanie bez zmian.
2. Bezstanowość: wszystkie informacje potrzebne to wykonania żądania zawarte są w zapytaniu – jego wykonanie jest niezależne od poprzedniego stanu serwera.
3. Pamięć podręczna: klient jest w stanie zapisać odpowiedź serwera w pamięci podręcznej aby ograniczać zapytania do serwera.
4. System warstwowy: klient nie musi mieć bezpośredniego połączenia z serwerem – pomiędzy może być zaimplementowany load-balancer, proxy lub inny serwis, który jest niewidoczny ani dla serwera ani klienta.
5. Kod na żądanie (opcjonalne): Serwer jest w stanie przesłać wykonywalny kod na żądanie klienta, np. skrypty w języku JavaScript.
6. Jednolity interfejs: określa wymagania dotyczące interfejsu między klientem a serwerem:
   - zasoby są identyfikowane unikalnym URI
   - serwer opisuje dane w postaci reprezentacji, klient modyfikuje dane poprzez przesłanie reprezentacji
   - każda wiadomość zawiera informacje, w jaki sposób przetworzyć dane
   - HATEOAS (ang. *Hypermedia as the Engine of Application State*) – serwer przesyła linki do akcji, które mogą zostać wykonane dla danego stanu zasobu

Większość developerów pomija jednak implementowanie HATEOAS, ponieważ wymaga dużego nakładu pracy zarówno po stronie serwera jak i klienta i często nie jest potrzebne dla prostych zapytań typu CRUD.

Przykładowa odpowiedź na zapytanie `GET /accounts/12345` [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 określane przez klienta — serwer GraphQL określa schemat wszystkich możliwych zapytań, a klient sam określa, które pola ma zwrócić
- introspekcyjny — język GraphQL umożliwia odpytywanie serwera o jego schemat i typy

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

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

Odpowiedź z serwera na takie zapytanie ma analogiczną strukturę:

```
{
  "data": {
```

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

Aby wykonanie takiego zapytania było możliwe, serwer musi zadeklarować odpowiednie typy. Typ reprezentuje obiekt odpowiadający jakiejś funkcji aplikacji [10]. W kontekście strony muzycznej do typami byłyby między innymi albumy czy artyści. Typ składa się z pól, które odpowiadają danym obiektu. Każde pole zwraca określony typ danych.

Typy mogą być skalarne lub obiektowe: typ skalarny zawsze zwraca wartość, natomiast typ obiektowy złożony jest z kolejnych pól. Wbudowane typy skalarne to Int, Float, String, Boolean, ID. Możliwe jest tworzenie dodatkowych typów skalarnych np. DateTime, aby zapewnić odpowiednie formatowanie i walidację danych. Dodatkowo typy mogą zostać oznaczone wykrzyknikiem, co znaczy że zawsze zwrócą wartość (ang. *non-nullable*).

Przykładowy schemat, który umożliwi wykonanie powyższego zapytania wygląda następująco:

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

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

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

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

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 często wywołanie metody nie wpływa na stan zasobu RPC jest wystarczające.

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

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

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

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

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), Javie (Spring), Pytonie (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 kopiowanie przy zapisie (ang. *Copy-on-write*) [9]. LXC wykorzystuje metody na poziomie kernela do izolowania użytkowników, procesów i sieci. Cgroups implementują zarządzanie zasobami oraz pomiary wykorzystania tych zasobów przez poszczególne procesy wewnątrz kontenerów. Pozwalają na ograniczanie i rozdzielanie 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 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

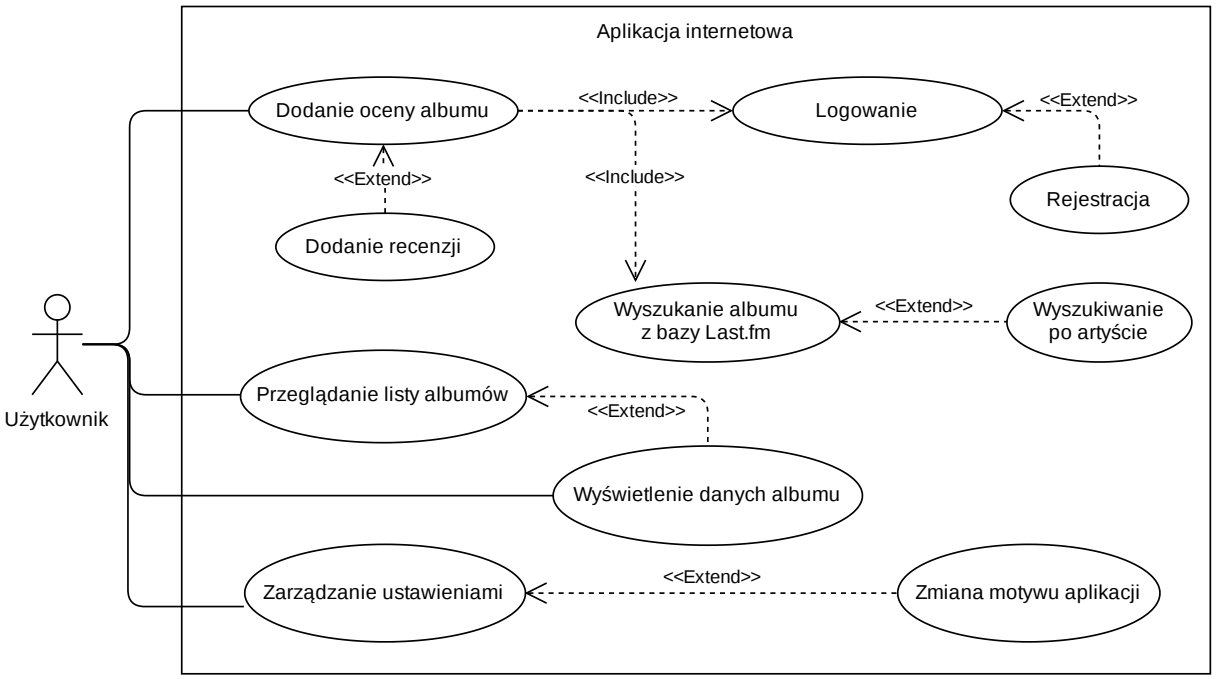
3.3. Przypadki użycia

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

Rozdział 4

Projekt aplikacji

4.1. Architektura fizyczna

Zgodnie z założeniami projektowymi, całość aplikacji serwowana będzie z platformy dockerowej. Podczas 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 zewnątrz 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

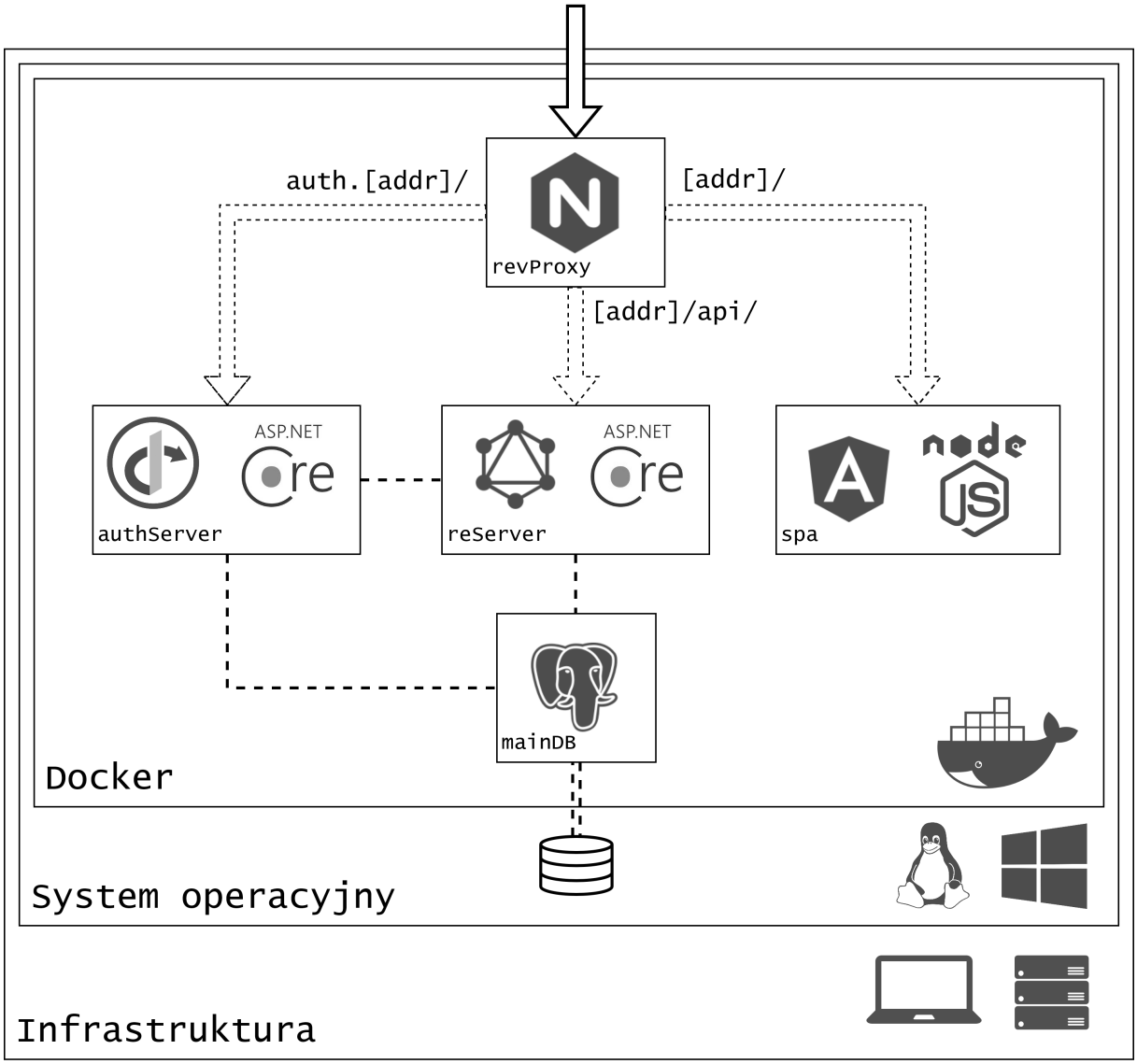
localhost/api/ jeśli zapytanie odnosi się do katalogu */api/* to zostaje obsłużone przez kontener *reServer*

Kontener obsługujący zapytania API musi być dostępny z zewnątrz, 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 frameworku Angular 8. Serwowana będzie ze środowiska uruchomieniowego Node.js. Projekt zbudowany jest z osobnych modułów dla każdej funkcjonalności. Widoki składane są z komponentów — są to elementy zawierające szablon



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

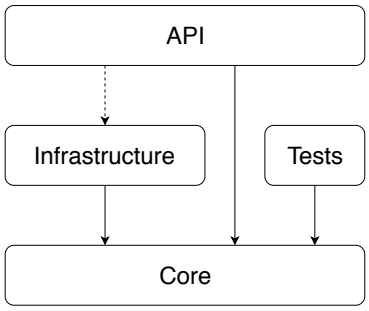
Projekt podzielony zostanie na trzy warstwy:

Api warstwa obsługująca zapytania GraphQL

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

Infrastructure warstwa implementująca interfejsy oraz obsługująca 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 podłączane 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.

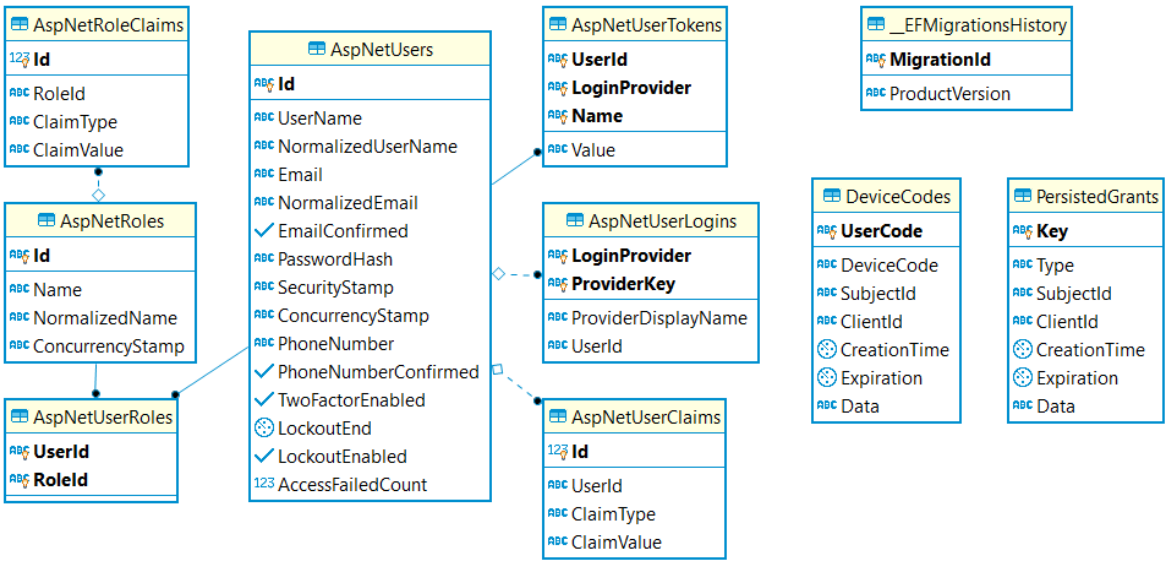
Z racji tego, że GraphQL jest protokołem służącym do przesyłania danych, jest on zaimplementowany w najwyższej warstwie serwisu – Api. W związku z tym nie zawęża sposobu implementacji zapisu danych w warstwie infrastruktury, a więc nic nie stoi na przeszkodzie, aby użyć relacyjnej bazy SQL. Takie rozwiązanie 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 oraz ich upoważnienia (opisane szerzej w rozdziale). Oprócz danych użytkowników zapisywane są również dane związane z migracjami bazy zarządzane przez *Entity Framework* oraz dwie pomocnicze tabele biblioteki *IdentityServer4* (*DeviceCodes* i *PersistedGrants*).

Dodać ref do rozdziału o Identity



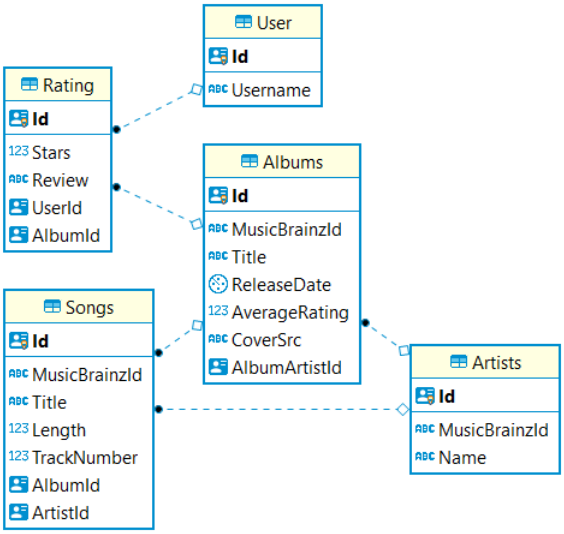
Rys. 4.3: Schemat ERD bazy z kontami użytkowników

Rysunek 4.4 przedstawia bazę przechowującą główne dane aplikacji. Przechowywana jest minimalna ilość danych wymagana do zaimplementowania wersji konceptualnej 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 najwyższego standardu bezpieczeństwa zaimplementowany zostanie proces logowania z wykorzystaniem kodu PKCE (ang. *Proof Key for Code Exchange*), opisany w dokumencie RFC 7636 [7]. 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*).

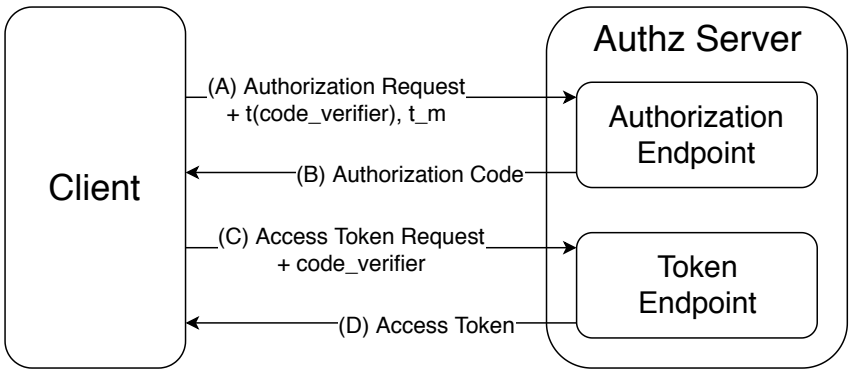


Rys. 4.4: Schemat ERD bazy z głównymi danymi

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

A. 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.
B. Jeśli użytkownik zalogował się poprawnymi danymi, serwer zapisuje parametry weryfikacji i zwraca kod autoryzacji.
C. Aby uzyskać token dostępowy, klient przesyła kod autoryzacji wraz z oryginalną wartością `code_verifier` wygenerowaną w punkcie (A).
D. 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 wewnątrz kontenera
ports porty hosta, które zostaną zmapowane na porty kontenera
environment zmienne środowiskowe wewnątrz 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 potrzebną 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 obserwowalnych 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. 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`, cache przetrzymywanym w pamięci przeglądarki. 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 bilbiotek typu *NgRx* czy *Redux*.

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 dodawającego nową ocenę albumu:

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

wygeneruje następujący serwis:

```
export const AddRatingDocument = gql`
```

Listing 5.5: Globalne ustawienia modułu Apollo

```
const uri = 'api/graphql';

@NgModule({
  exports: [ApolloModule, HttpLinkModule],
  providers: [
    {
      provide: APOLLO_OPTIONS,
      useFactory: (httpLink: HttpLink, authService: AuthService) => {
        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: {}
        };
      },
      deps: [HttpLink, AuthService]
    }
  ]
})
export class GraphQLModule {}
```

```
  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 żywotności komponentu. Subskrypcja odbywa się automatycznie dzięki zasosowaniu *async pipe*. *Pipe* to klasa, która pozwalaja na transformację zadanego wejścia i używa się ich bezpośrednio w pliku HTML. Przykładowo, komponent wyświetlający dane albumu operuje na strumieniu zwracanym przez funkcję `watch`:

```
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`.

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
}
```

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.

5.2.3. Routing

Nawigacja pomiędzy widokami wykorzystuje bibliotekę `angular/router`. Dzięki niej każdy moduł aplikaji posiada zdefiniowany ro

5.2.4. Połączenie 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 5.8.

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;
    }
  }
};
```

Listing 5.8: Metoda wykorzystujaca 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 (360px na 740px). 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 nawigacji przedstawionej na listingu 5.9. Atrybut `fxHide.gt-xs` chowa cały element jeśli rozmiar ekranu jest większy niż `xs`. W ten sposób otrzymano toolbar, który dla zwykłych ekranów zawiera przyciski nawigacji (Rys. 5.1), a dla mniejszych przełącza menu boczne (Rys. 5.2). Znaczenie aliasów wykorzystywanych do definiowania rozmiarów zamieszczono w tabeli 5.1.

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

Listing 5.9: Menu nawigacji zależne od rozmiaru ekranu

```
<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>
```

Tab. 5.1: Zestawienie aliasów Api flex-layout z odpowiadającymi komendami mediaQuery [6]

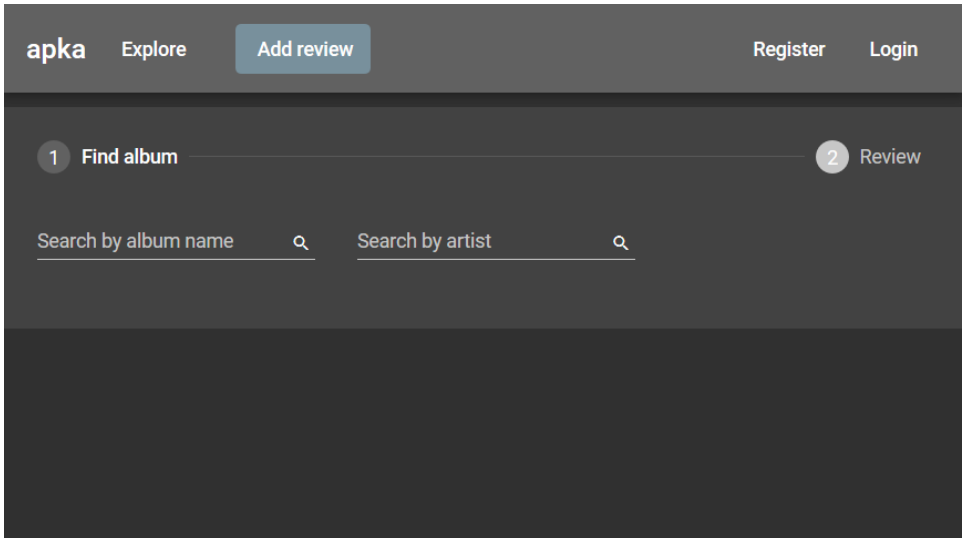
Alias	mediaQuery
xs	'screen and (max-width: 599px)'
sm	'screen and (min-width: 600px) and (max-width: 959px)'
md	'screen and (min-width: 960px) and (max-width: 1279px)'
lg	'screen and (min-width: 1280px) and (max-width: 1919px)'
xl	'screen and (min-width: 1920px) and (max-width: 5000px)'
lt-sm	'screen and (max-width: 599px)'
lt-md	'screen and (max-width: 959px)'
lt-lg	'screen and (max-width: 1279px)'
lt-xl	'screen and (max-width: 1919px)'
gt-xs	'screen and (min-width: 600px)'
gt-sm	'screen and (min-width: 960px)'
gt-md	'screen and (min-width: 1280px)'
gt-lg	'screen and (min-width: 1920px)'

kolorystyki odbywa się za pomoca *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:

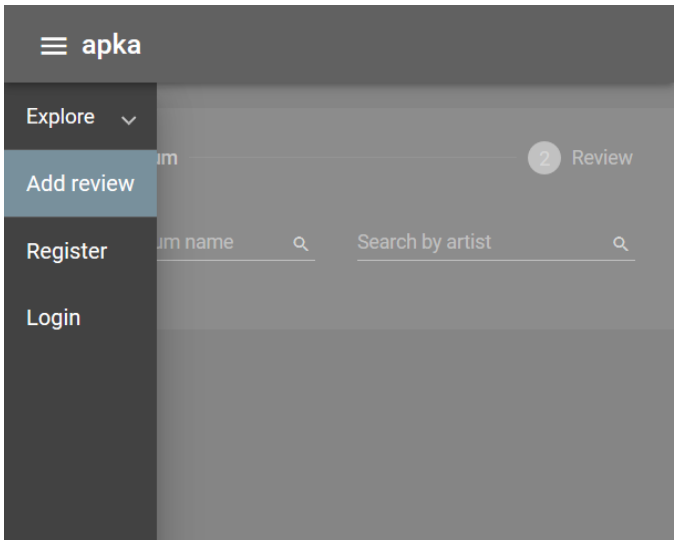
```
$dark-primary: mat-palette($mat-grey, 700, 300, 900);
$dark-accent: mat-palette($mat-blue-grey, 400);
$dark-warn: mat-palette($mat-red, 500);

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

Funkcja `mat-palette` zwraca paletę kolorów: pierwszy argument określa bazową 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);
}
```



Rys. 5.1: Menu nawigacji dla zwykłych ekranów



Rys. 5.2: Wysunięte manu nawigacji dla małych ekranów

```
.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 `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);
```

Listing 5.10: Główny szablon HTML aplikacji

```
<div [class]="'theme-wrapper mat-typography ' + (settings$ | async)?.theme">

  <app-header class="toolbar" (sidenavToggle)="mainSidenav.toggle()">
  </app-header>

  <mat-sidenav-container fxFlex fxFlex.xs="noshrink">
    <mat-sidenav #mainSidenav role="navigation" mode="over"
      fixedInViewport="true" fixedTopGap="56">
      <app-sidenav (sidenavClose)="mainSidenav.close()"></app-sidenav>
    </mat-sidenav>

    <mat-sidenav-content>
      <main role="main" class="wrapper" fxLayout="column">

        <div class="content" fxFlex="noshrink">
          <router-outlet></router-outlet>
        </div>

        <div class="footer" fxFlex="none">
          Marcin Kotas, 2019
        </div>

      </main>
    </mat-sidenav-content>
  </mat-sidenav-container>
</div>
```

```
  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 5.10. Odpowiednia klasa aktyowowana jest na podstawie ustawień użytkownika.

5.3. GraphQL API

5.3.1. Zastosowane technologie

Hot Chocolate
Autofac
Entity Framework Core
Namotion.Reflection

Serilog

5.3.2. Warstwa Api

5.3.3. Warstwa domenowa

5.3.4. Warstwa infrastrukturalna

5.4. Serwis uwierzytelniający

5.4.1. Zastosowane technologie

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

IdentityServer4
AspNetCore.Identity

5.4.2. Proces rejestracji i logowania

5.4.3. Autoryzacja dostępu do API

5.5. Baza danych

5.6. Odwrócone proxy

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, Czwerwiec 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] Responsive API · angular/flex-layout Wiki. https://github.com/angular/flex-layout/wiki/Responsive-API. dostęp dnia 20 listopada 2019.

[7] RFC 7636 - Proof Key for Code Exchange by OAuth Public Clients. https://tools.ietf.org/html/rfc7636. dostęp dnia 17 listopada 2019.

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

[9] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014.

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

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

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

Spis rysunków

Dodatek A

Opis załączonej płyty CD/DVD

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