**Dokumentacja projektu**

**Zaawansowane Techniki Programistyczne**



**MusicPlayer**

**Skład zespołu:**

* Magdalena Kalisz
* Adam Bajguz
* Michał Kierzkowski

**Data oddania:** 22.01.2019 r.

**Prowadzący:** mgr inż. Daniel Reska

# Opis projektu

**Odtwarzacz muzyczny na komputer PC z systemem Windows dla jednego użytkownika działający jako aplikacja Universal Windows Platform:**

* obsługa podstawowych formatów plików dźwiękowych (przynajmniej WAV i MP3);
* playlisty odtwarzanych plików z możliwością edycji (dodawanie, usuwanie, zmiana kolejności);
* zapis playlist i ich eksport (do formatu XML i/lub JSON);
* biblioteka utworów z prezentacją w widokach względem nazwy artysty, jego albumów i ścieżek w albumie;
* możliwość sortowania widoków biblioteki przynajmniej po nazwie, roku wydania, długości ścieżki;
* grupowanie utworów w albumy (jeden utwór może znajdować się w kilku albumach);
* każdy utwór ma mieć przypisany dokładnie jeden gatunek (lista gatunków jest ustalana przez użytkownika, tzn. użytkownik dodaje, usuwa i edytuje dostępne gatunki);
* każdy utwór ma mieć możliwość dodania własnej grafiki, jeśli jej nie ma wyświetlana jest domyślna grafika zapisana w aplikacji lub wyświetlana jest okładka albumu o ile istnieje;
* każdy utwór w albumie ma przypisany numer ścieżki;
* przypisywanie albumu do artysty (artysta może posiadać wiele albumów);
* przypisywanie artysty do zespołu (artysta może być tylko w jednym zespole);
* album, utwór oraz artysta mogą posiadać dokładnie jedno zdjęcie/okładkę;
* oprócz playlist powinna istnieć również kolejka odtwarzania zawierająca wszystkie utwory do odtworzenia;
* użytkownik ma mieć możliwość dodania albumu/playlisty lub pojedynczego utworu do kolejki odtwarzania.

# Wykorzystane technologie

|  |  |
| --- | --- |
|  |  |
|  | Znalezione obrazy dla zapytania c# |

# Diagramy klas

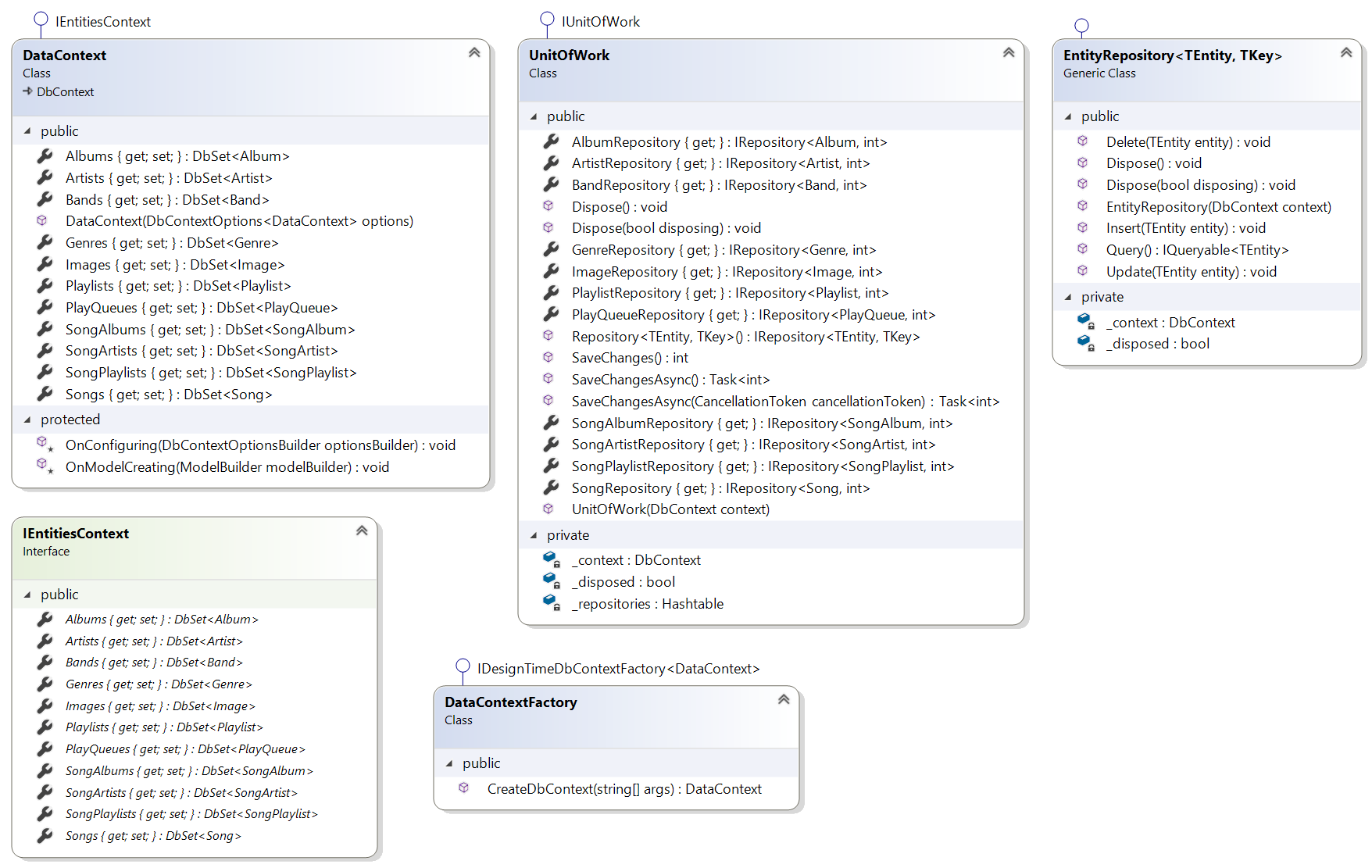
W skład kodu aplikacji (rozwiązania – solution) wchodzą nastęujące projekty:

* **MusicPlayer.Core** – projekt tworzący bibliotekę MusicPlayer.Core.dll
* **MusicPlayer.Data** – projekt tworzący bibliotekę MusicPlayer.Data.dll
* **MusicPlayer.Migrations.Startup** – projekt uruchamiania migracji
* **MusicPlayer.UWP** – projekt tworzący aplikację MusicPlayer.UWP.exe

## MusicPlayer.Core



## MusicPlayer.Data



## MusicPlayer.UWP

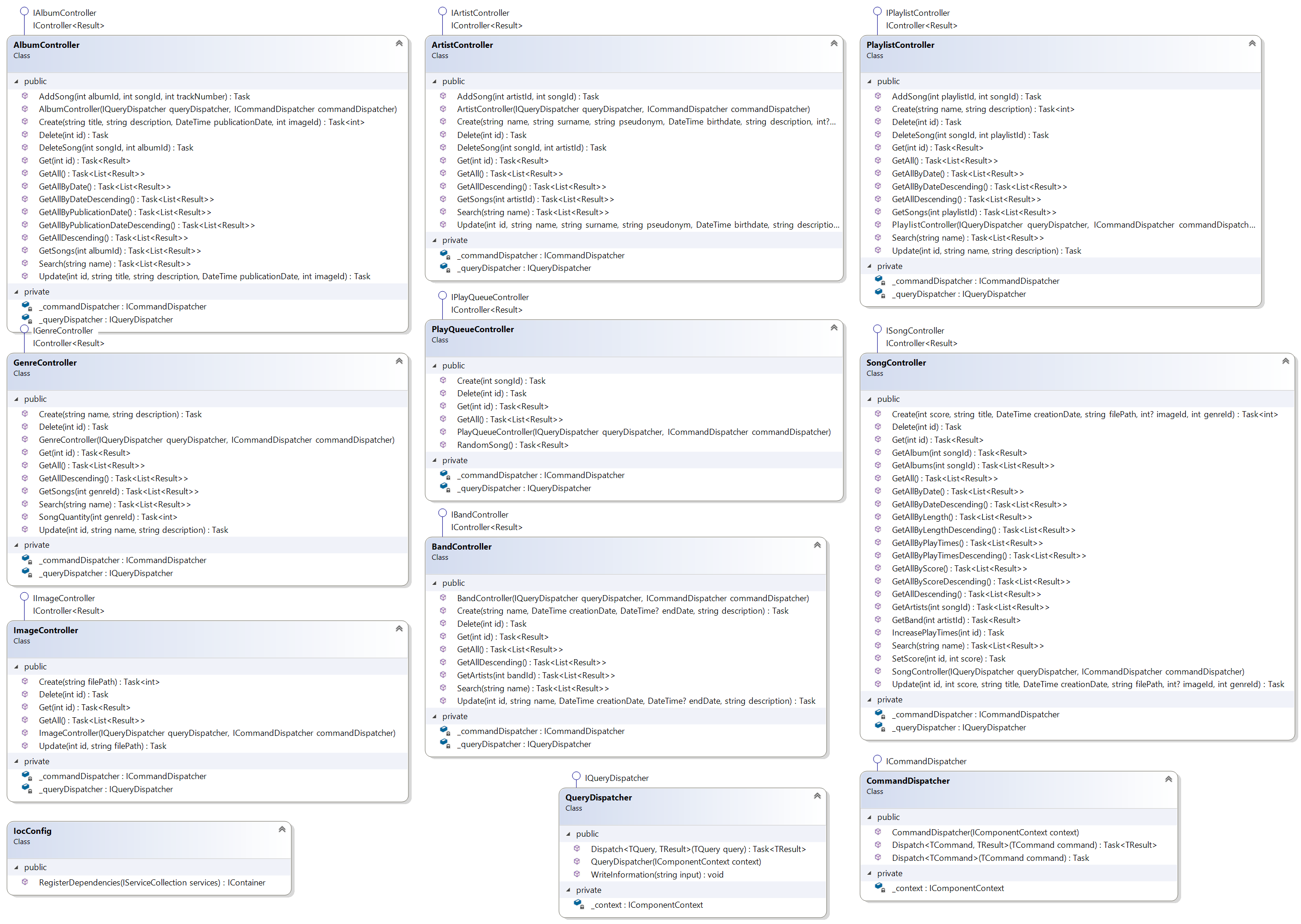
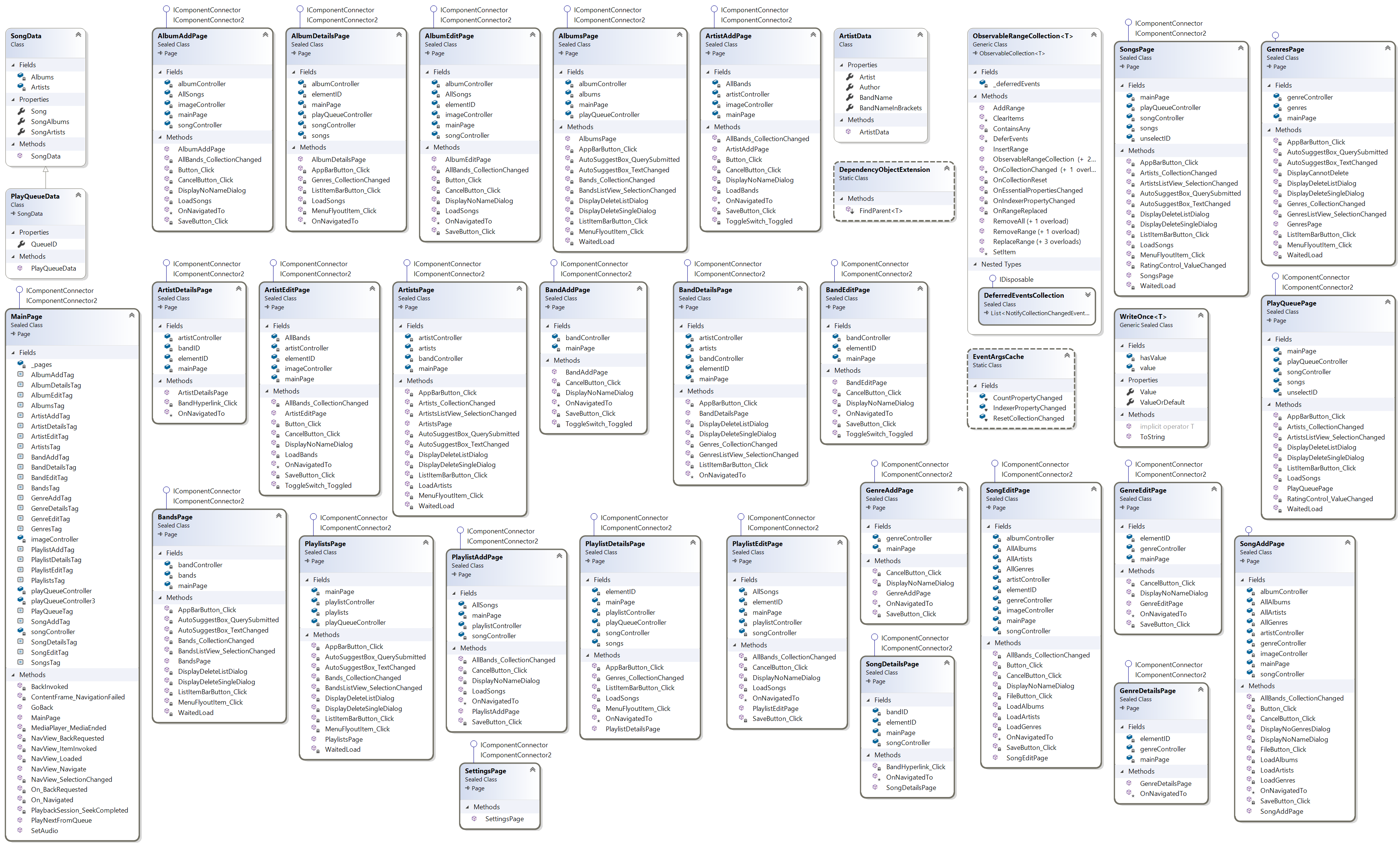
****



Diagram przestawia interfejsy i Klasy związane z prezenterem Albumu zawarte w MusicPlayer.UWP. Pozostałym modelom bazy odpowiadają zbliżone (mniej lub bardziej rozbudowane) prezentery.



# Opisy wzorców

## Wzorzec #1 (kreacyjny): Singleton

### Cel użycia

Ograniczenie możliwości tworzenia obiektów klasy NLogLogger do jednej instancji.

NLogLogger jest klasą służącą do rejestrowania w bazie zachowań, operacji i ich rezultatów. Używana była podczas tworzenia apliakcji.

### Przyporządkowanie klas do ról wzorca

MusicPlayer.Core -> Logging -> NLogLogger.cs

***Szczegóły w rozdziale:*** [***Wzorzec #1 (kreacyjny): Singleton***](#_Wzorzec_#1_(kreacyjny):)

### Lokalizacja wzorca w kodzie

MusicPlayer.Core -> Logging -> NLogLogger.cs

***Szczegóły w rozdziale:*** [***Wzorzec #1 (kreacyjny): Singleton***](#_Wzorzec_#1_(kreacyjny):)

### Wektor zmian wzorca

## Wzorzec #2 (kreacyjny): Dependency injection

### Cel użycia

Wzorzec zastosowany w projekcie w celu usunięcia bezpośrednich zależności pomiędzy komponentami na rzecz architektury typu plug-in. Wzorzec ten stanowi alternatywę do podejścia, gdzie obiekty tworzą instancję obiektów, z których korzystają np. we własnym konstruktorze. Dzięki takiemu podejściu kod aplikacji opartej na Entitiy Framework jest prostszy, bardziej zrozumiały i łatwiejszy do testowania.

### Przyporządkowanie klas do ról wzorca

MusicPlayer.UWP -> AppStart -> IocConfig.cs

***Szczegóły w rozdziale:*** [***Wzorzec #2 (kreacyjny): Dependency injection***](#_Wzorzec_#2_(kreacyjny):)

### Lokalizacja wzorca w kodzie

MusicPlayer.UWP -> AppStart -> IocConfig.cs

***Szczegóły w rozdziale:*** [***Wzorzec #2 (kreacyjny): Dependency injection***](#_Wzorzec_#2_(kreacyjny):)

### Wektor zmian wzorca

## Wzorzec #3 (kreacyjny): Factory

### Cel użycia

Niektóre z poleceń narzędzia EF Core Tools (na przykład polecenia migracji) wymagają utworzenia pochodnej instancji DbContext w czasie projektowania w celu zebrania szczegółowych informacji o typach encji aplikacji i sposobie ich mapowania do schematu bazy danych. Pożądane jest, aby utworzony w ten sposób DbContext był skonfigurowany w podobny sposób, w jaki byłby skonfigurowany w czasie wykonywania (klasa DataContext).

Narzędzie EF Core Tools można poinformować o sposobie tworzenia usługi DbContext implementując interfejs IDesignTimeDbContextFactory<TContext>. Wzorze Factory (Fabryki) został użyty w celu realizacji połączenia z bazą podczas tworzenia migracji.

### Przyporządkowanie klas do ról wzorca

MusicPlayer.Data -> DataContextfactory.cs

***Szczegóły w rozdziale:*** [***Wzorzec #3 (kreacyjny): Factory***](#_Wzorzec_#3_(kreacyjny):)

### Lokalizacja wzorca w kodzie

MusicPlayer.Data -> DataContextfactory.cs

***Szczegóły w rozdziale:*** [***Wzorzec #3 (kreacyjny): Factory***](#_Wzorzec_#3_(kreacyjny):)

### Wektor zmian wzorca

## Wzorzec #4 (strukturalny): Extension Object

### Cel użycia

Wzorzec Extension Object został użyty w celu rozszerzenia istniejących klas, np. Object o dodatkowe funkcjonalności (metody). Zwykle klasa jest rozszerzana poprzez podklasę i dodawanie metod do klasy pochodnej. Obiekt rozszerzenia zapewnia rozszerzalność bez podklas.

### Przyporządkowanie klas do ról wzorca

Roszerzenie klas:

MusicPlayer.Core -> Extensions -> \*

MusicPlayer.UWP -> Pages -> DependecyObjectExtension.cs

***Szczegóły w rozdziale:*** [***Wzorzec #4 (strukturalny): Extension Object***](#_Wzorzec_#4_(strukturalny):)

### Lokalizacja wzorca w kodzie

MusicPlayer.Core -> Extensions -> \*

MusicPlayer.UWP -> Pages -> DependecyObjectExtension.cs

***Szczegóły w rozdziale:*** [***Wzorzec #4 (strukturalny): Extension Object***](#_Wzorzec_#4_(strukturalny):)

### Wektor zmian wzorca

## Wzorzec #5 (strukturalny): Repository

### Cel użycia

Repozytoria to klasy lub komponenty, które w projekcie zawierają logikę wymaganą do uzyskania dostępu do źródeł danych. Centralizują one wspólne funkcjonalności dostępu do danych, zapewniając lepszą edytowalność i oddzielenie infrastruktury lub technologii wykorzystywanej do uzyskiwania dostępu do baz danych z warstwy modelu domeny. Z uwagi na zastosowanie Mapowania Obiektowego Relacji (ORM), jakim jest Entity Framework, kod, który został zaimplementowany, jest uproszczony, dzięki LINQ i silnemu typowaniu. Pozwoliło to skoncentrować się na logice, trwałości danych, a nie na dostępie do danych.

### Przyporządkowanie klas do ról wzorca

MusicPlayer.Data -> EntityRepository.cs

MusicPlayer.Data -> DataContext.cs

***Szczegóły w rozdziale:*** [***Wzorzec #5 (strukturalny): Repository***](#_Wzorzec_#5_(strukturalny):)

### Lokalizacja wzorca w kodzie

MusicPlayer.Data -> EntityRepository.cs

MucisPlayer.Data -> UnitOfWork.cs

***Szczegóły w rozdziale:*** [***Wzorzec #5 (strukturalny): Repository***](#_Wzorzec_#5_(strukturalny):)

### Wektor zmian wzorca

## Wzorzec #6 (strukturalny): Private class data

### Cel użycia

Wzorzec Private class data rozwiązuje w projekcie problem dotyczący ochrony stanu obiektu, w którym nie można zadeklarować finału. Działanie tego wzorca polega na usunięciu ekspozycji danych przez zabezpieczenie jej w klasie utrzymującej stan danych. W rezultacie oddzielono dane od tych metod, które go używają, a tym samym stworzono kolejną warstwę separacji od prezenterów, które mają relacje danymi tworzonego obiektu. W skrócie, wzorzec umożliwia w prezenterach inicjowanie danych należących do klas modeli.

### Przyporządkowanie klas do ról wzorca

Enkapsulacja danych: MusicPlayer.UWP -> Controllers -> *presenter name* -> Result.cs

***Szczegóły w rozdziale:*** [***Wzorzec #6 (strukturalny): Private class data***](#_Wzorzec_#6_(strukturalny):)

### Lokalizacja wzorca w kodzie

MusicPlayer.UWP -> Controllers -> *presenter name* -> Result.cs

***Szczegóły w rozdziale:*** [***Wzorzec #6 (strukturalny): Private class data***](#_Wzorzec_#6_(strukturalny):)

### Wektor zmian wzorca

## Wzorzec #7 (czynnościowy): Null Object

### Cel użycia

Null Object (Pusty obiekt) to czynnościowy wzorzec projektowy, użyty w celu realizacji braku obiektu – Zdjecia poprzez dostarczenie alternatywy, która oferuje domyślnie działanie puste, czyli niewykonujące żadnych operacji.

### Przyporządkowanie klas do ról wzorca

MusicPlayer.Core -> NullObjects -> ImageNullObject.cs

***Szczegóły w rozdziale:*** [***Wzorzec #7 (czynnościowy): Null Object***](#_Wzorzec_#7_(czynnościowy):)

### Lokalizacja wzorca w kodzie

MusicPlayer.Core -> NullObjects -> ImageNullObject.cs

***Szczegóły w rozdziale:*** [***Wzorzec #7 (czynnościowy): Null Object***](#_Wzorzec_#7_(czynnościowy):)

### Wektor zmian wzorca

## Wzorzec #8 (czynnościowy): CQRS

### Cel użycia

W tradycyjnych architekturach sam model danych jest używany do wysyłania zapytań do bazy danych i aktualizowania jej. Jest to proste i dobrze się sprawdza w przypadku podstawowych operacji CRUD. Jednak w naszej aplikacji, która jest bardziej złożona metoda została uznana za niewygodną i nieczytelną. Wówczas, mapowanie obiektu stało by się skomplikowane. Po stronie zapisu model może wdrażać złożoną walidację i logikę biznesową. W efekcie można uzyskać zbyt skomplikowany model, który wykonuje zbyt dużo działań.

Podejście CQRS rozwiązuje wymienione wyżej problemy, rozdzielając odczyty i zapisy na osobne modele przy użyciu **poleceń** do aktualizacji danych i **zapytań** do odczytu danych.

* Polecenia będą oparte na zadaniach, a nie skoncentrowane na danych. Dzięki temu będą one być umieszczane w kolejce do przetworzenia asynchronicznego, a nie przetwarzane synchronicznie.
* Zapytania nigdy nie modyfikują bazy danych. Zapytanie zwraca obiekt DTO, który nie hermetyzuje żadnej wiedzy domeny.

### Przyporządkowanie klas do ról wzorca

Polecenia: MusicPlayer.UWP -> CQRS -> CommandDispatcher.cs

Zaptytania: MusicPlayer.UWP -> CQRS -> QueryDispatcher.cs

***Szczegóły w rozdziale:*** [***Wzorzec #8 (czynnościowy): CQRS***](#_Wzorzec_#8_(czynnościowy):)

### Lokalizacja wzorca w kodzie

MusicPlayer.UWP -> CQRS -> \*

***Szczegóły w rozdziale:*** [***Wzorzec #8 (czynnościowy): CQRS***](#_Wzorzec_#8_(czynnościowy):)

### Wektor zmian wzorca

## Wzorzec #9: Unit of Work

### Cel użycia

Unit of Work jest określana jako pojedyncza transakcja, która obejmuje wiele operacji wstawiania, aktualizowania lub usuwania. Mówiąc najprościej, oznacza to, że dla określonego działania użytkownika, takiego jak odtwarzenie utworu, wszystkie operacje wstawiania, aktualizacji i usuwania są obsługiwane w ramach pojedynczej transakcji. Jest to bardziej wydajne niż obsługa wielu transakcji baz danych w sposób oparty na sieci.

Te wielokrotne operacje utrwalania wykonywane są później w pojedynczej akcji, gdy kod z warstwy aplikacji zarządza nimi wydając polecenia. Decyzja o wprowadzeniu zmian w pamięci do rzeczywistej bazy danych jest zwykle oparta na schemacie Unit of Work.

### Przyporządkowanie klas do ról wzorca

MusicPlayer.Data -> UnitOfWork.cs

MusicPlayer.Core -> Data -> UnitOfWork.cs

***Szczegóły w rozdziale:*** [***Wzorzec #9: Unit of Work***](#_Wzorzec_#9:_Unit)

### Lokalizacja wzorca w kodzie

MusicPlayer.Data -> UnitOfWork.cs

MusicPlayer.Core -> Data -> UnitOfWork.cs

***Szczegóły w rozdziale:*** [***Wzorzec #9: Unit of Work***](#_Wzorzec_#9:_Unit)

### Wektor zmian wzorca

## Wzorzec #10 MVP (architekturalny)

### Cel użycia

Model-View-Presenter to wzorzec powstały na bazie wzorca MVC (Model-View-Controller). We wzorcu MVP prezenter jest tym samym, czym kontroler we wzorcu MVC z jedną mała różnicą, w prezenterze zawiera się logika biznesowa. Wzorzec ten został użyty w celu oddzielenia danych, widoku i obsługi bazy danych.

### Przyporządkowanie klas do ról wzorca

Modele: MusicPlayer.Core -> Entities -> \*

Widoki: MusicPlayer.UWP -> Pages -> \*

Prezentery: MusicPlayer.UWP -> Controllers -> \*

***Szczegóły w rozdziale:*** [***Wzorzec #10 (architekturalny)***](#_Wzorzec_#10_(architekturalny))

### Lokalizacja wzorca w kodzie

MusicPlayer.Core -> Entities -> \*

MusicPlayer.UWP -> Pages -> \*

MusicPlayer.UWP -> Controllers -> \*

***Szczegóły w rozdziale:*** [***Wzorzec #10 (architekturalny)***](#_Wzorzec_#10_(architekturalny))

### Wektor zmian wzorca

# Opis rozwiązań specyficznych dla użytych technologii

W aplikacji wykorzystano Entity Framework oparty na .NET Core. Jest to framework używany zazwyczaj przy technologii ASP.NET MVC. W projekcie wykorzystano go do implementacji obsługi bazy oraz wzorca Model-View-Presenter. Głównymi elementami są modele, widoki i prezentery. MVP to wzorzec powstały na bazie wzorca MVC (Model-View-Controller). We wzorcu MVP prezenter jest tym samym, czym kontroler we wzorcu MVC z jedną mała różnicą, w prezenterze zawiera się logika biznesowa. Dane nie są przekazywane bezpośrednio z modelu do widoku jak to ma miejsce w MVC. Prezenter wysyła zapytanie do modelu, model zwraca dane do prezentera, prezenter przetwarza otrzymane dane i przekazuje do widoku. Widokiem w aplikacji są strony budujący interfejs aplikacji Universal Windows Platform, która w przeciwieństwie do technologii Windows Presentation Foundation buduje aplikację działającą w jednym oknie i bazującą na stornach. Jako rozszerzenie wzorca MVP, zastosowano wzorzec CQRS który pozwolił na rozdzielnie zapytań do bazy i poleceń, dzięki czemu uzyskano lepszą hermetyzację kodu, a także czytelniejszy kod.

<https://msdn.microsoft.com/pl-pl/library/ff714342.aspx>

<https://msdn.microsoft.com/en-us/magazine/mt788619.aspx>

<https://docs.microsoft.com/pl-pl/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/cqrs-microservice-reads>

# Mapy kodu według wzorców projektowych

## Legenda oznaczeń na diagramach

Obraz zawierający monitor, ekran, telefon komórkowy, telefon

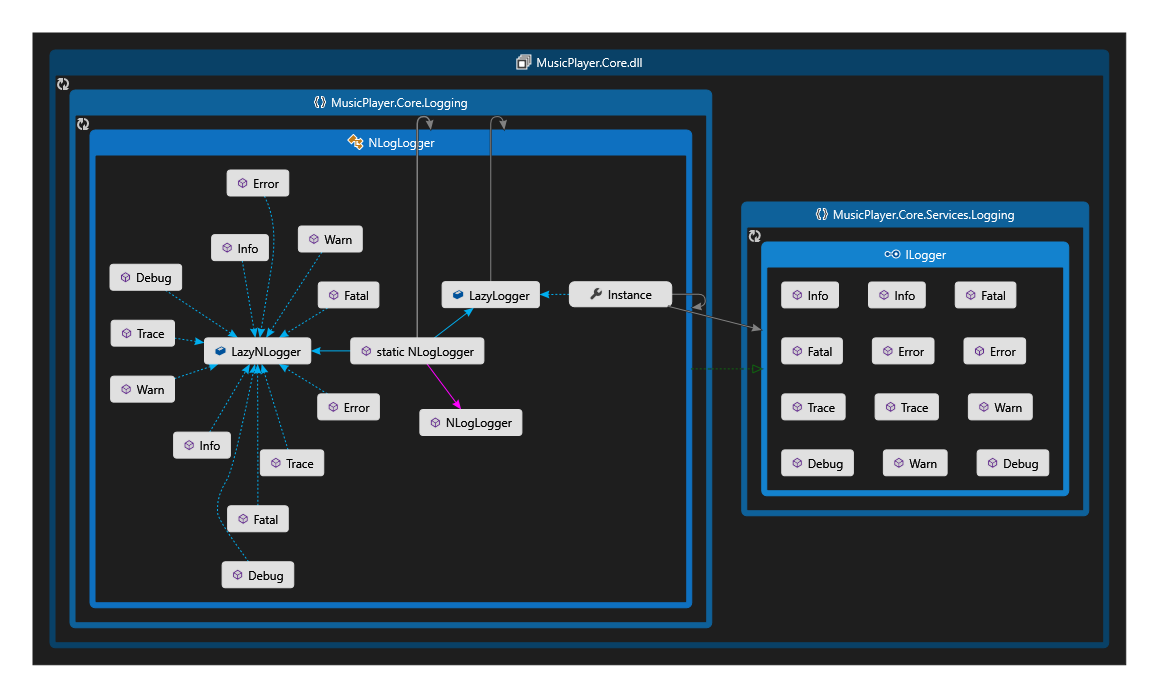
Opis wygenerowany automatycznie

## Zależności stworzonych bibliotek w projekcie

Obraz zawierający zrzut ekranu, monitor, ściana, ekran

Opis wygenerowany automatycznie

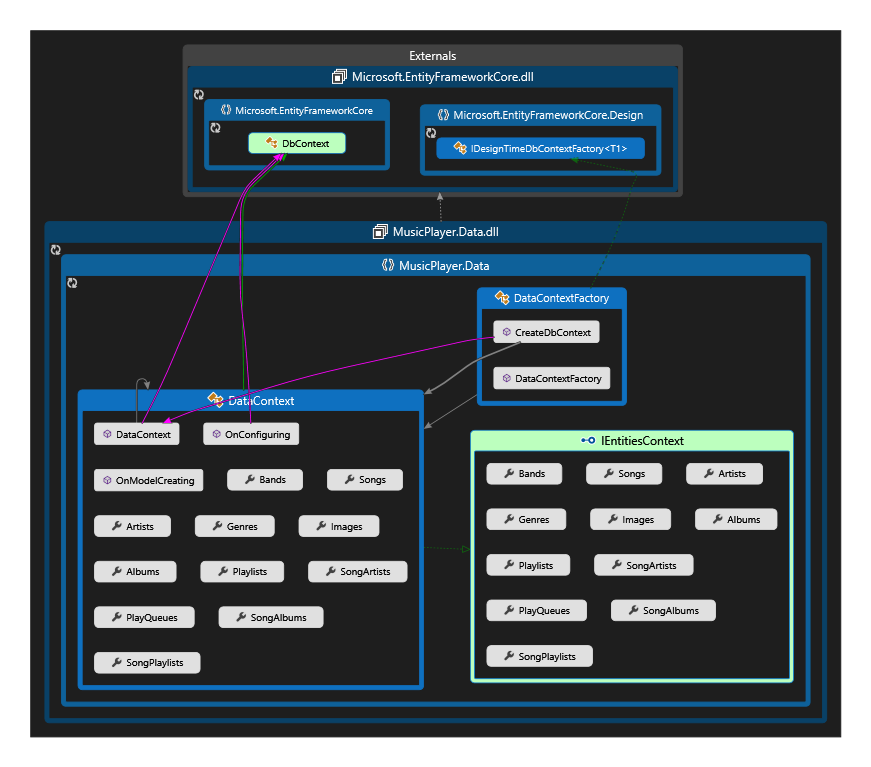
## Wzorzec #1 (kreacyjny): Singleton



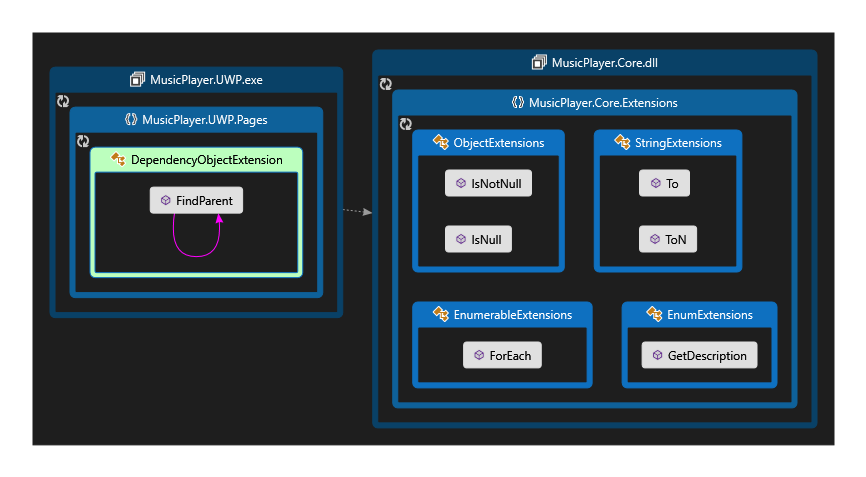
## Wzorzec #2 (kreacyjny): Dependency injection



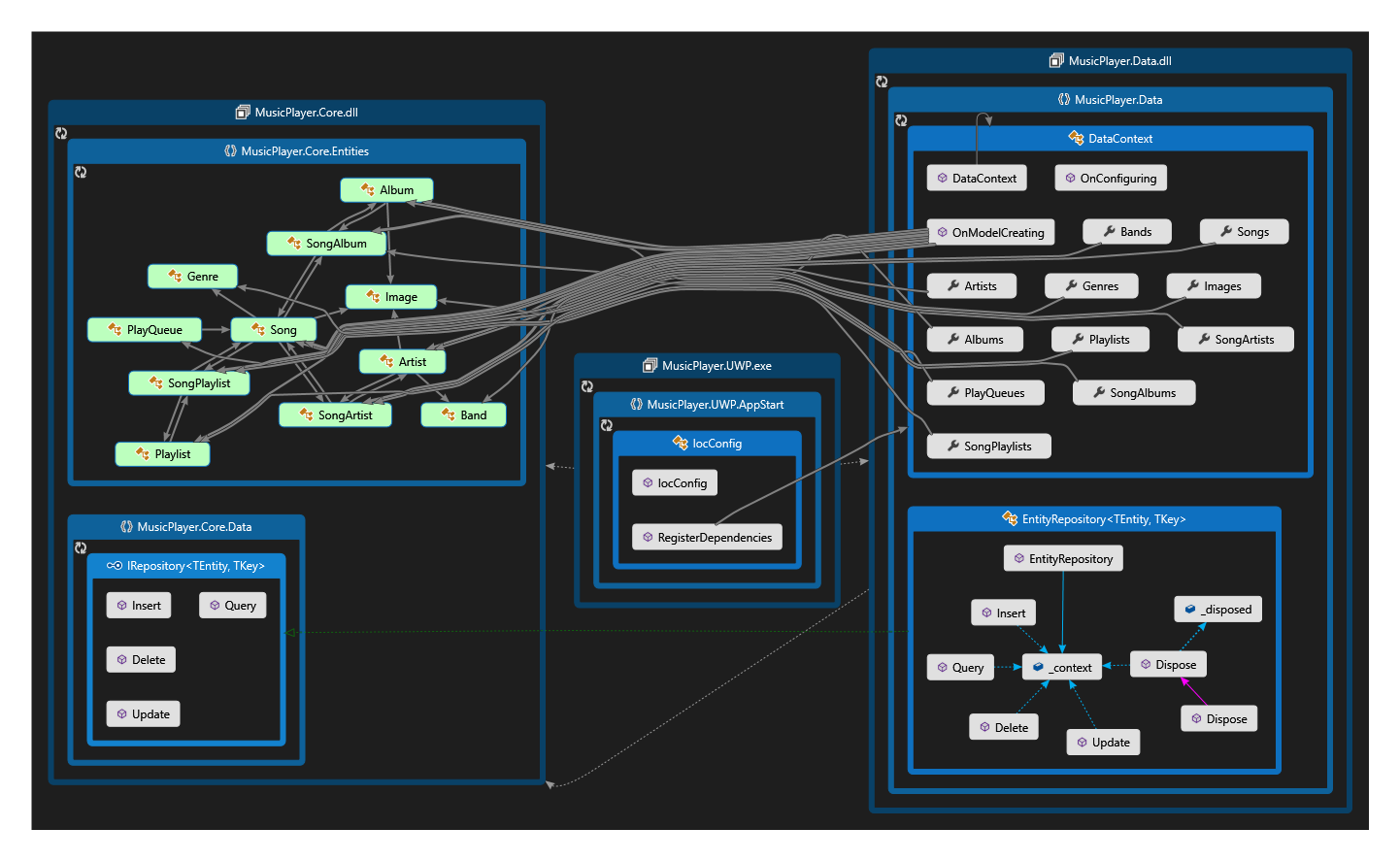
## Wzorzec #3 (kreacyjny): Factory



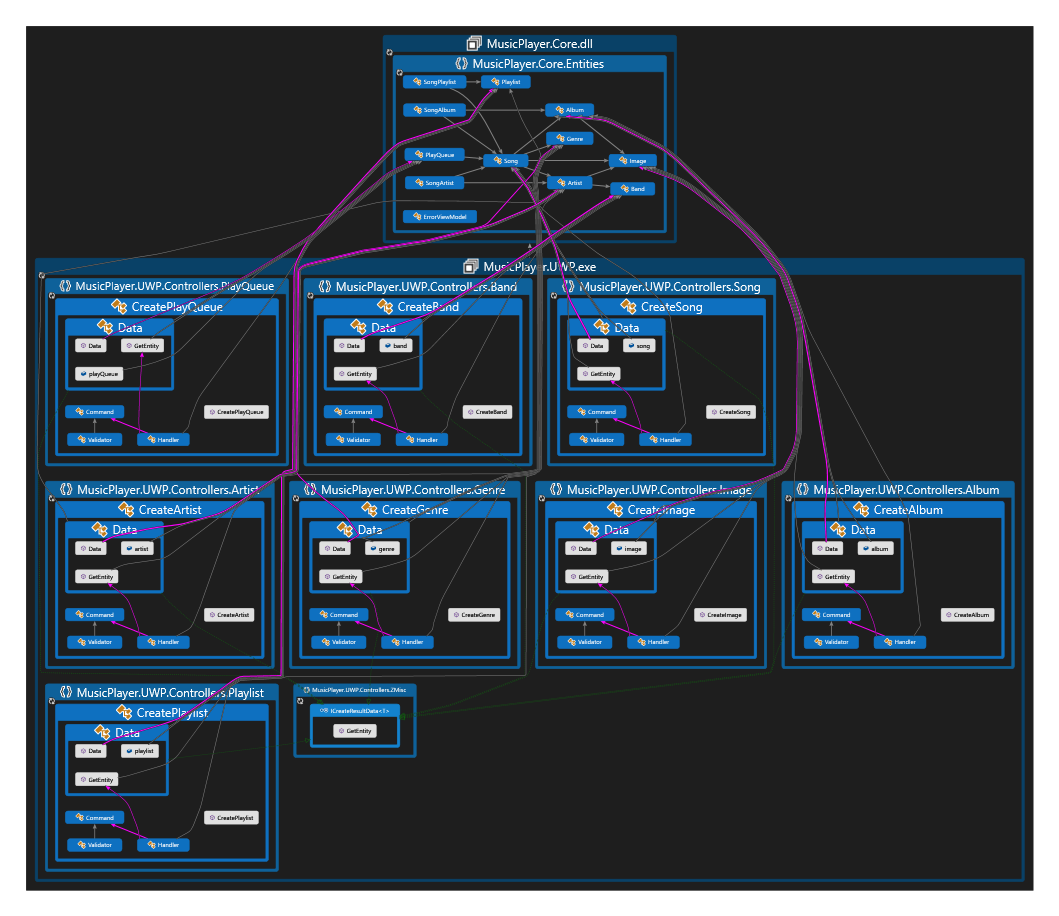
## Wzorzec #4 (strukturalny): Extension Object



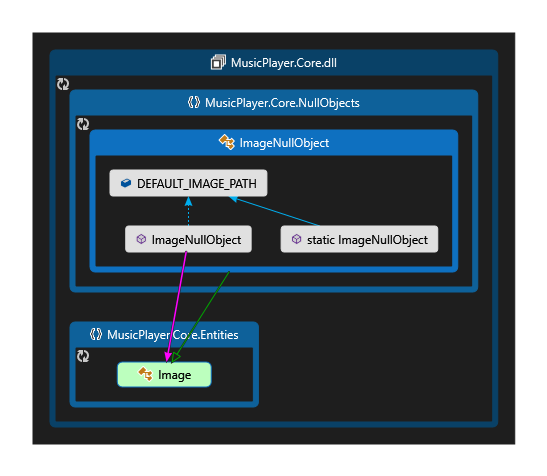
## Wzorzec #5 (strukturalny): Repository



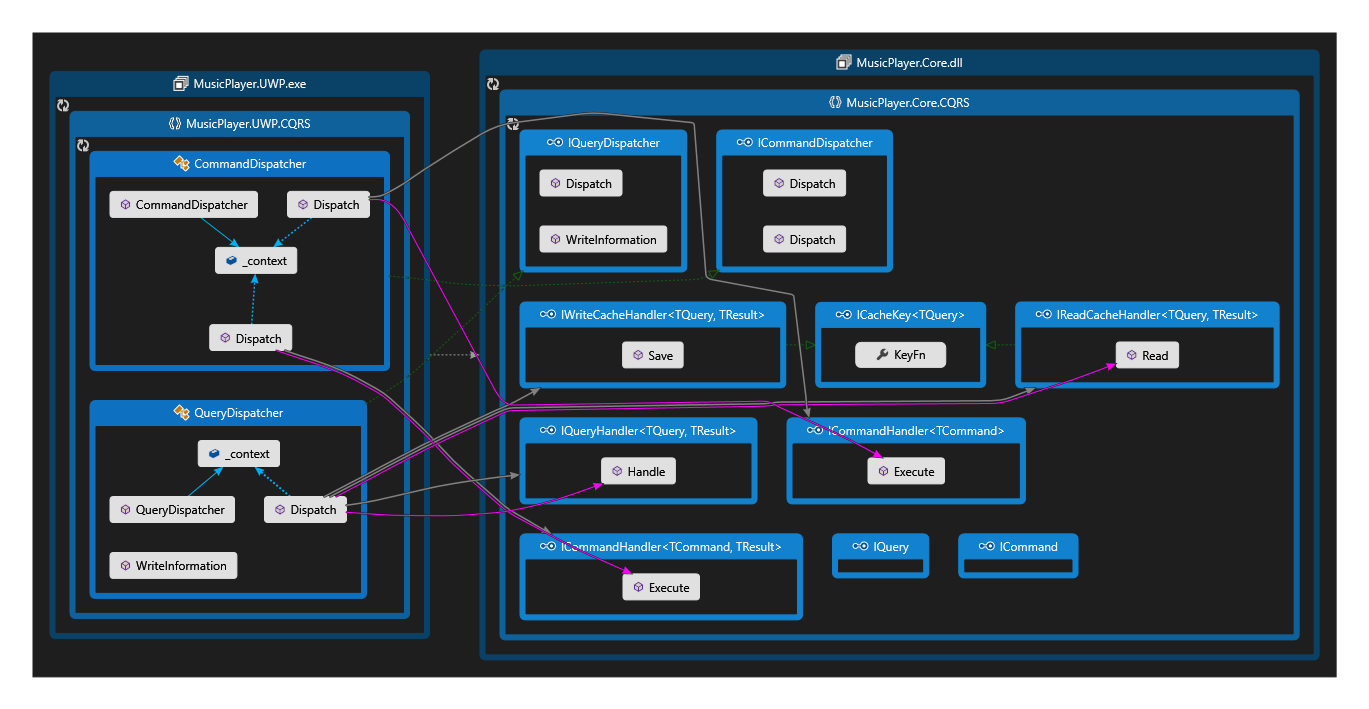
## Wzorzec #6 (strukturalny): Private class data



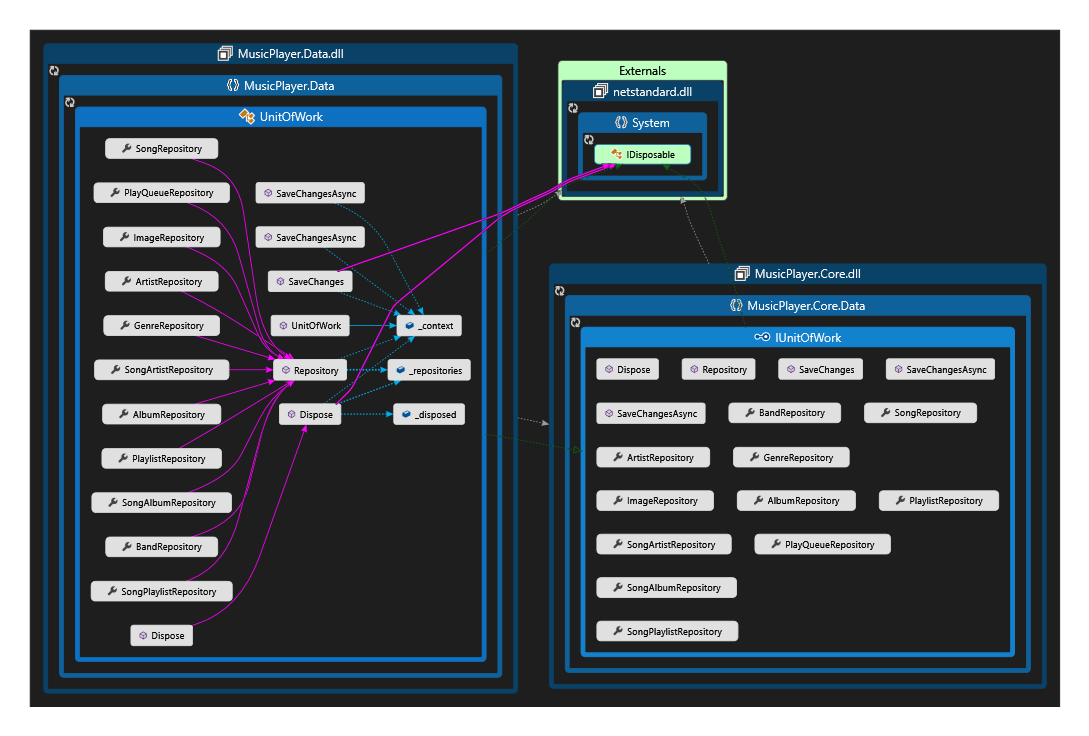
## Wzorzec #7 (czynnościowy): Null Object



## Wzorzec #8 (czynnościowy): CQRS

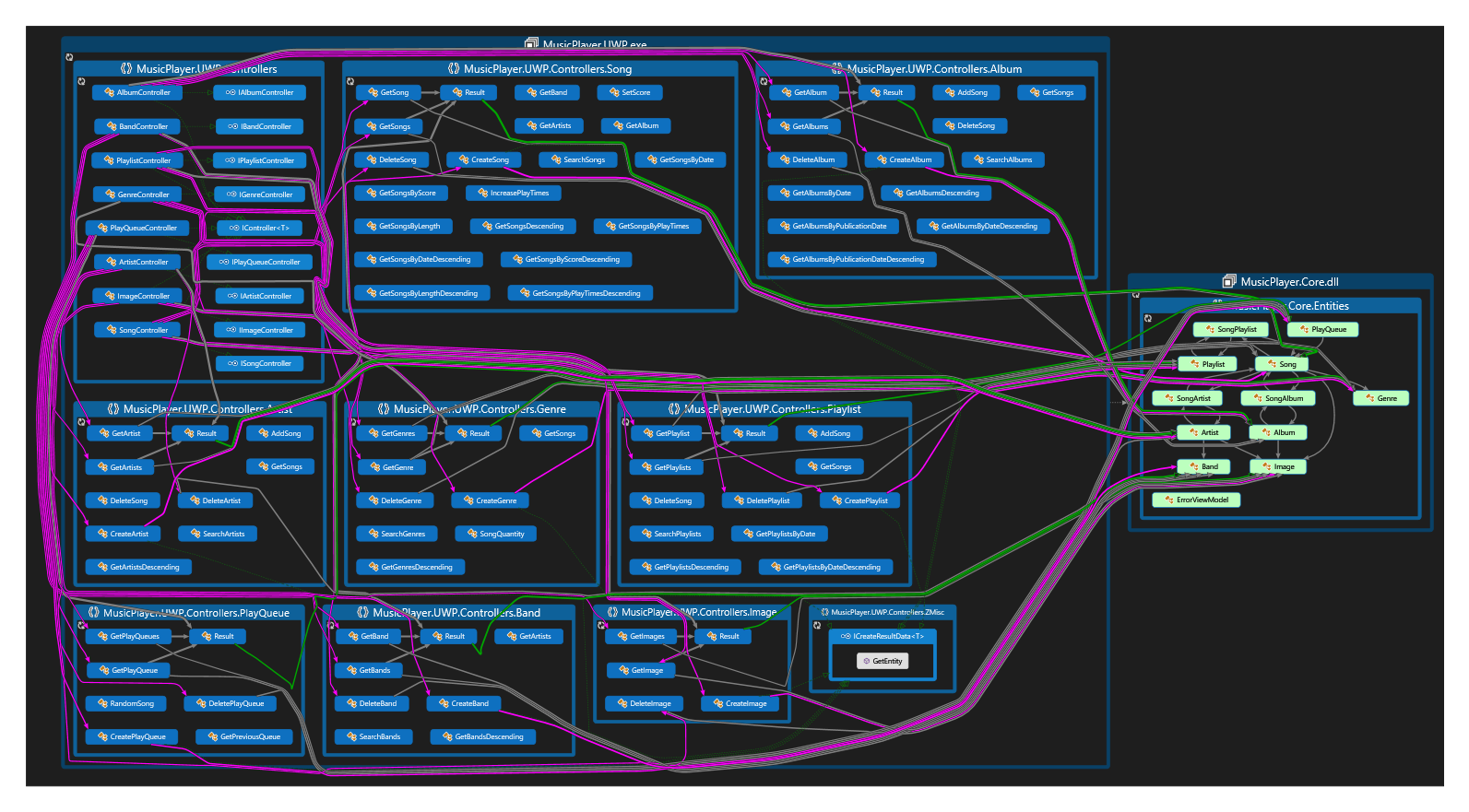


## Wzorzec #9: Unit of Work

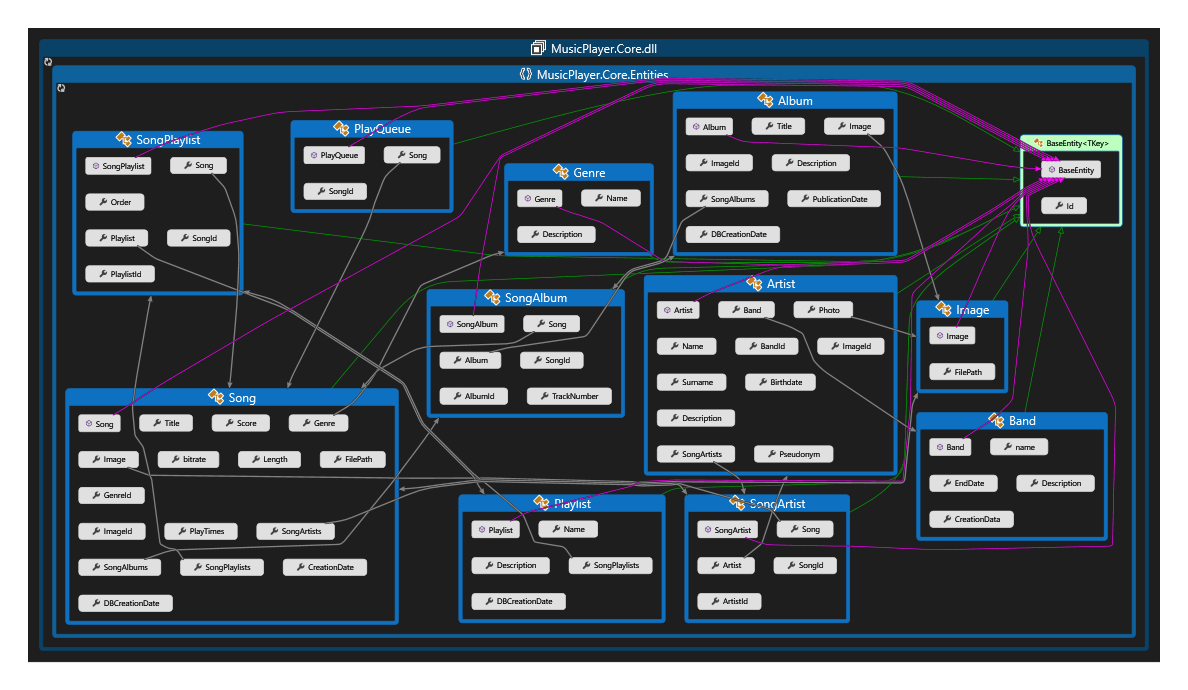


## Wzorzec #10 (architekturalny)

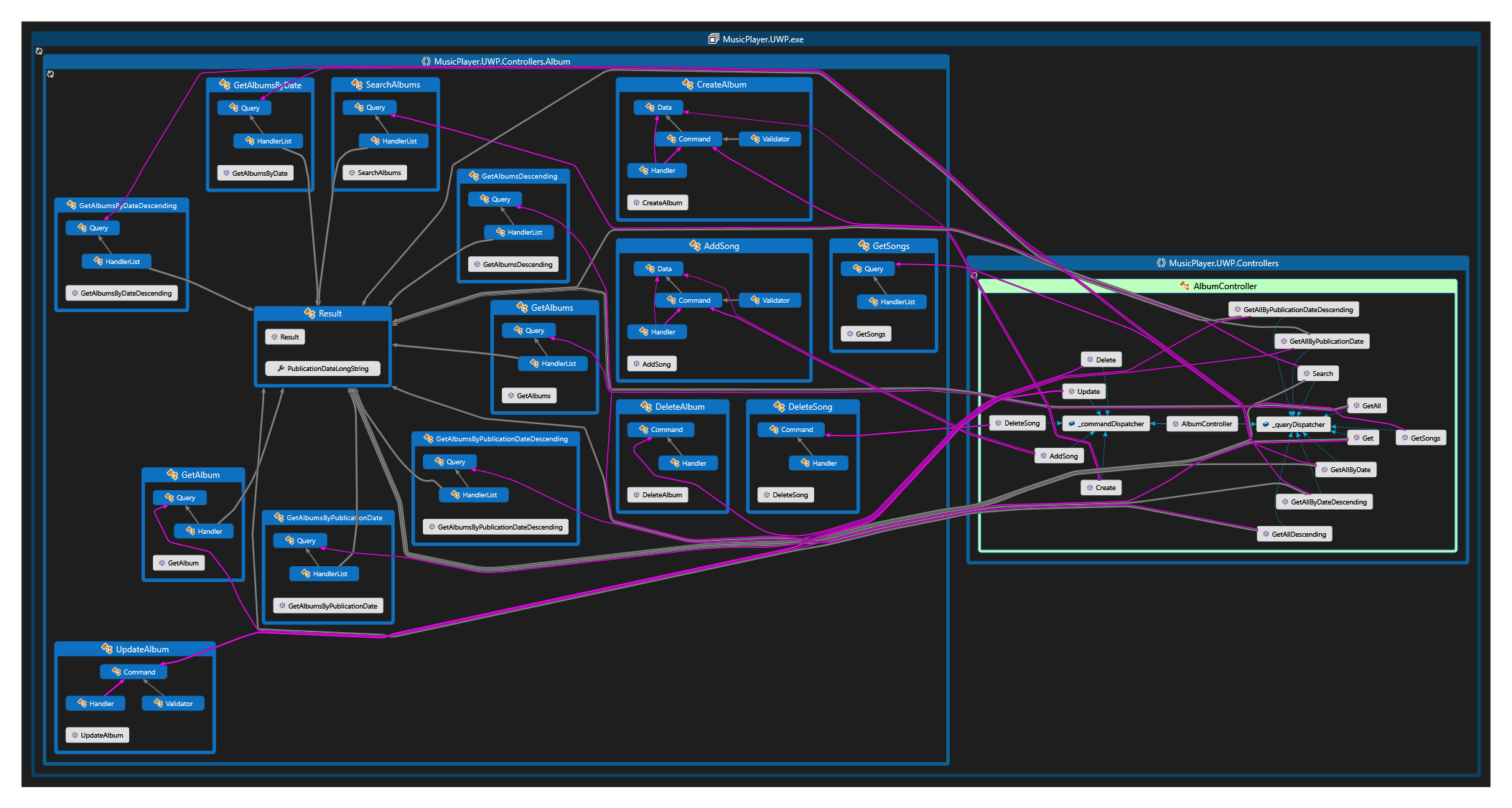
### MVP – struktura



### MVP – Modele



### MVP – Przykładowy prezenter Albumu



# Podział pracy w zespole

| **Czynność/Zadanie** | **Osoba** | | |
| --- | --- | --- | --- |
| **Magdalena Kalisz** | **Adam Bajguz** | **Michał Kierzkowski** |
| Zaprojektowanie bazy danych |  | + |  |
| Utworzenie modeli bazy danych | + | + |  |
| Utworzenie i skonfigurowanie rozwiązania (projektów w aplikacji) | + |  | + |
| Utworzenie prezentacji wzorców |  |  | + |
| Utworzenie końcowej dokumentacji |  | + | + |
| Utworzenie ikony aplikacji |  |  | + |
| Utworzenie map kodu |  | + |  |
| Stworzenie diagramów klas |  |  | + |
| **BACKEND** | | | |
| Utworzenie prezenterów | + |  |  |
| Pomoc przy tworzeniu prezenterów |  | + |  |
| Implementacja dostępu do bazy danych | + |  |  |
| Implementacja repozytoriów, CQRS tworzących dodatkową warstwę abstrakcji w MVP | + |  |  |
| **FRONTEND (strony w aplikacji, widok szczegółów, edycji, dodawania itp.)** | | | |
| Utworzenie nawigacji |  |  | + |
| Utworzenie stron utworu |  | + | + |
| Utworzenie stron albumu |  | + |  |
| Utworzenie stron zespołu |  | + |  |
| Utworzenie stron playlisty |  | + |  |
| Utworzenie stron gatunku |  |  | + |
| Utworzenie stron artysty |  | + |  |
| Utworzenie strony kolejki odtwarzania |  |  | + |
| Utworzenie odtwarzacza i jego integracja z bazą | + |  |  |
| **TESTOWANIE APLIKACJI** | | | |
| Testowanie backendu i bazy danych | + |  |  |
| Testowanie aplikacji (działania stron) |  |  | + |

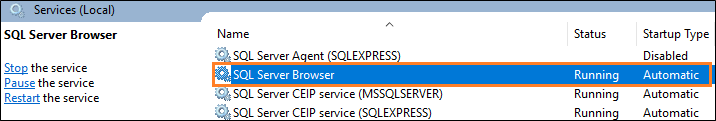
**Podsumowanie:** podział pracy przy aplikacji był zrównoważony. W przedstawionej tabeli widać wyraźny podział zespołu na backend i frontend. Należy pamiętać że część backendowa pomimo mniejszej ilości czynności/zadań w tabeli była tak samo rozbudowana jak cześć frontendowa. Ponadto stopień skomplikowania projektu (20494 linii kodu, 386 klasy i 24 interfejsy) powoduje, że jeszcze bardziej szczegółowe wypisanie czynności/zadań zrealizowanych przez poszczególne osoby jest utrudnione.

# Instrukcja obsługi aplikacji

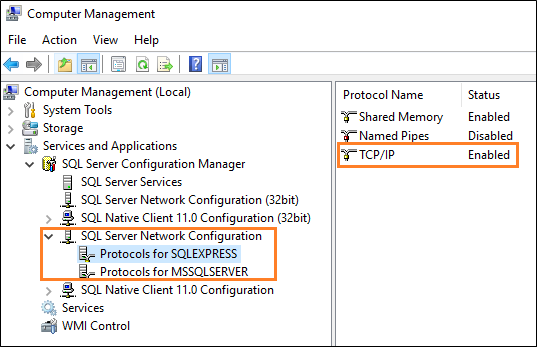
Aplikacja umożliwia:

# Instrukcja instalacji

1. Zainstalować Microsoft SQL Server 2014 zgodnie z poradnikiem: <https://www.sqlshack.com/how-to-install-sql-server-2014-management-studio/?fbclid=IwAR1Pa-VjREPi0VUOOo8WMnBHKED8d-axSk4DQnjz95rkx9w9_J0_8MzuObE>
2. Zainstalować Microsoft Visual Studio 2017 najlepiej w wersji Enterprise wraz z:
   1. Entitiy Framework
   2. .NET Core
   3. Windows 10 SDK (10.0.17763.0)
   4. .NET Native
   5. .NET Framework (4.5 – 4.7.2)
3. Skonfigurować SQL Server:
   1. Uruchomić wszystkie usługi SQL Server’a



* 1. Włączyć protokół TCP/IP:



1. Utworzyć bazę danych poleceniem Update-database w konsoli NuGet w programie Microsoft Visual Studio używając projektu MusicPlayer.Migrations.Sturtup
2. Wdrożyć rozwiązanie (deploy) przy projekcie MusicPlayer.UWP ustawionym jako startowy.

# Analiza możliwości rozbudowy programu

* Zastosowane wzorce projektowe takie jak: CQRS, Repozytoria, MVP, Private Class Data pozwalają na łatwe dodanie kolejnych funkcjonalności do programu np. kont użytkownika.
* Zastosowania wzorca MVP pozwala na łatwą rozbudowę prezenterów w celu dodania kolejnych funkcjonalności do stron.
* Zastosowanie wzorca CQRS pozwala na rozszerzenie aplikacji o bardziej złożoną walidację podczas dodawania i edycji danych.