MusicPlayer

Skład zespołu:

- Magdalena Kalisz
- Adam Bajguz
- Michał Kierzkowski

Opis projektu

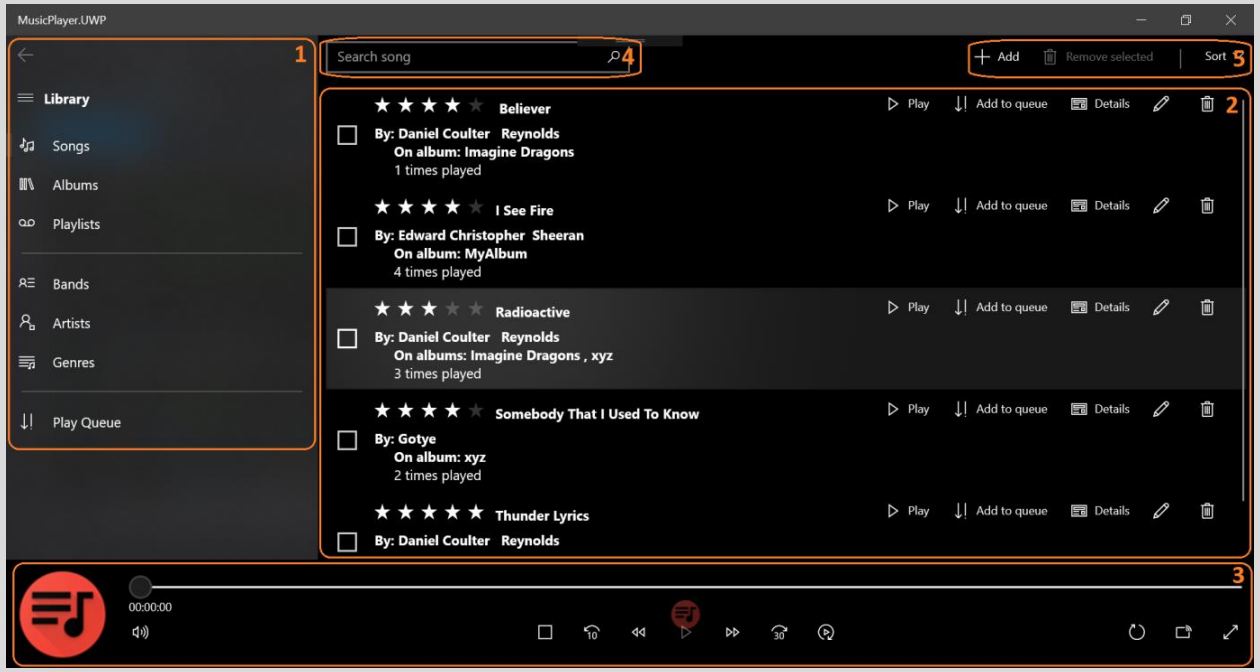
Odtwarzacz muzyczny na komputer PC z systemem Windows dla jednego użytkownika działający jako aplikacja Universal Windows Platform:

- 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.

Profil użytkonika końcowego

Odtwarzacz muzyczny został zaprojektowany dla użytkowników komputerów PC z systemem Windows. Działa on jako aplikacja Universal Windows Platform, czyli aplikacja ze sklepu Microsoft Store. Przystosowana jest dla jednego użytkownika.

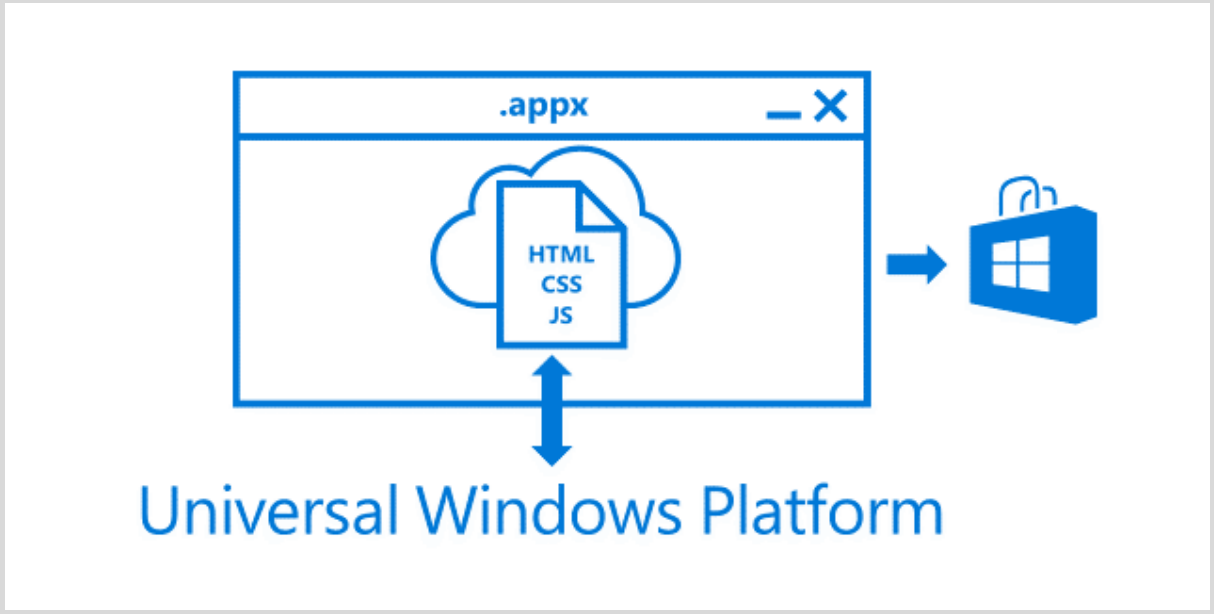
Opis interfejsu



OBJAŚNIENIA:

1. Panel nawigacji po widokach
2. Lista elementów danego typu
3. Kontroler odtwarzania
4. Pole wyszukiwania z autouzupełnianiem
5. Panel akcji

Wykorzystane technologie



Wykorzystane narzędzia

- Microsoft Visual Studio 2017 Enterprise
- Microsoft SQL Server 2014

Najważniejsze informacje o stworzonej aplikacji

- Aplikacja posiada nowoczesny interfejs oparty o Universal Windows Platform (UWP) zgodny ze standardami aplikacji Microsoft Store
- Aplikacja jest dość rozbudowana (20494 linii kodu, 386 klasy i 24 interfejsy), ale nadal umożliwia łatwe rozszerzanie o dodatkowe funkcjonalności
- Aplikacja obsługuje dwa style kolorystyczne: Jasny i Ciemny wbudowane w system Windows i przełączane z poziomu ustawień systemu
- Aplikacja jest wyposażona w rozbudowany odtwarzacz muzyczny (np.: przewijanie, powtarzanie, niezależna regulacja głośności, przesyłanie strumieniowe)

Diagramy klas

W skład kodu aplikacji (rozwiązania – solution) wchodzą następujące 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

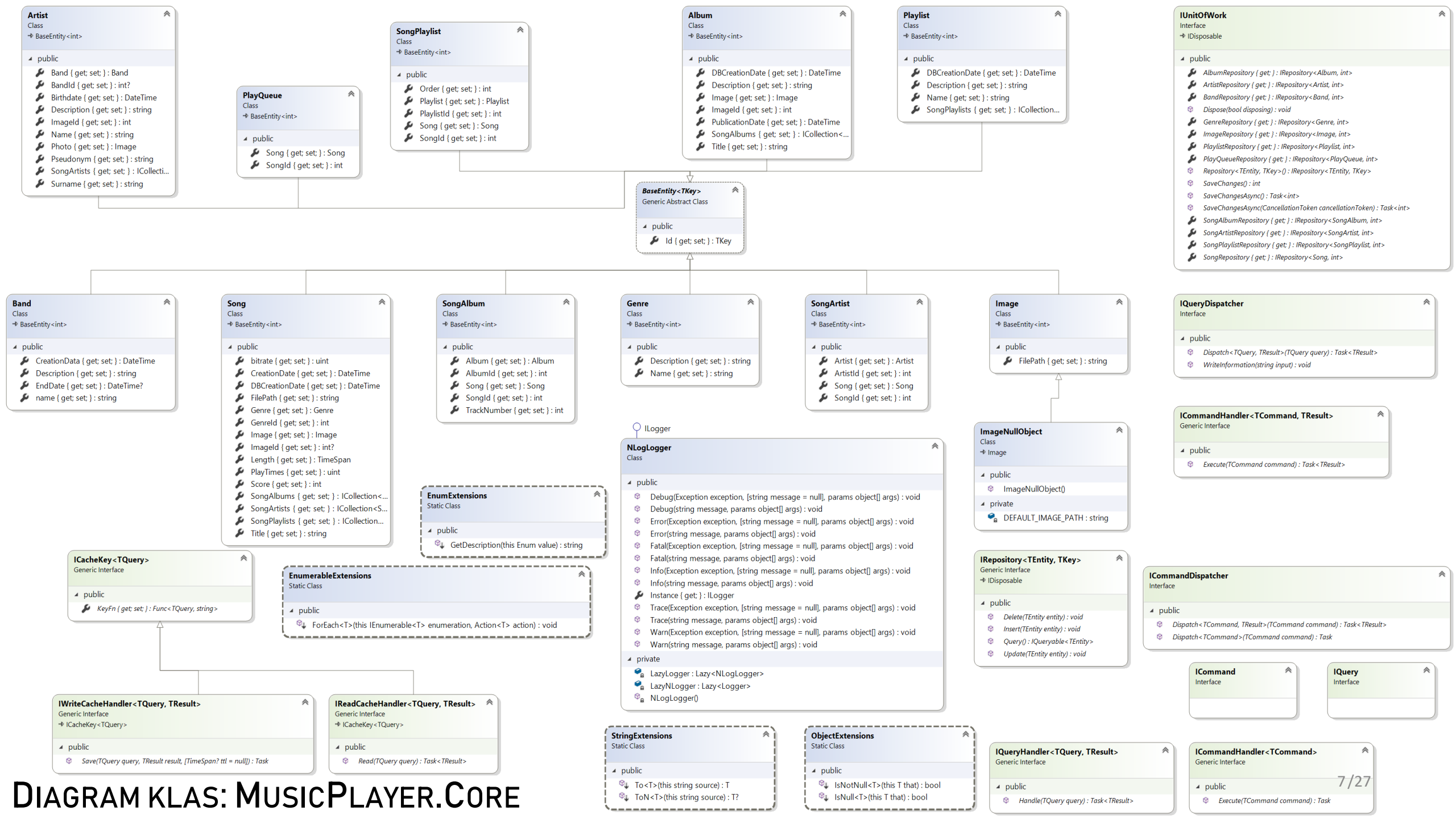


DIAGRAM KLAS: MUSICPLAYER.CORE

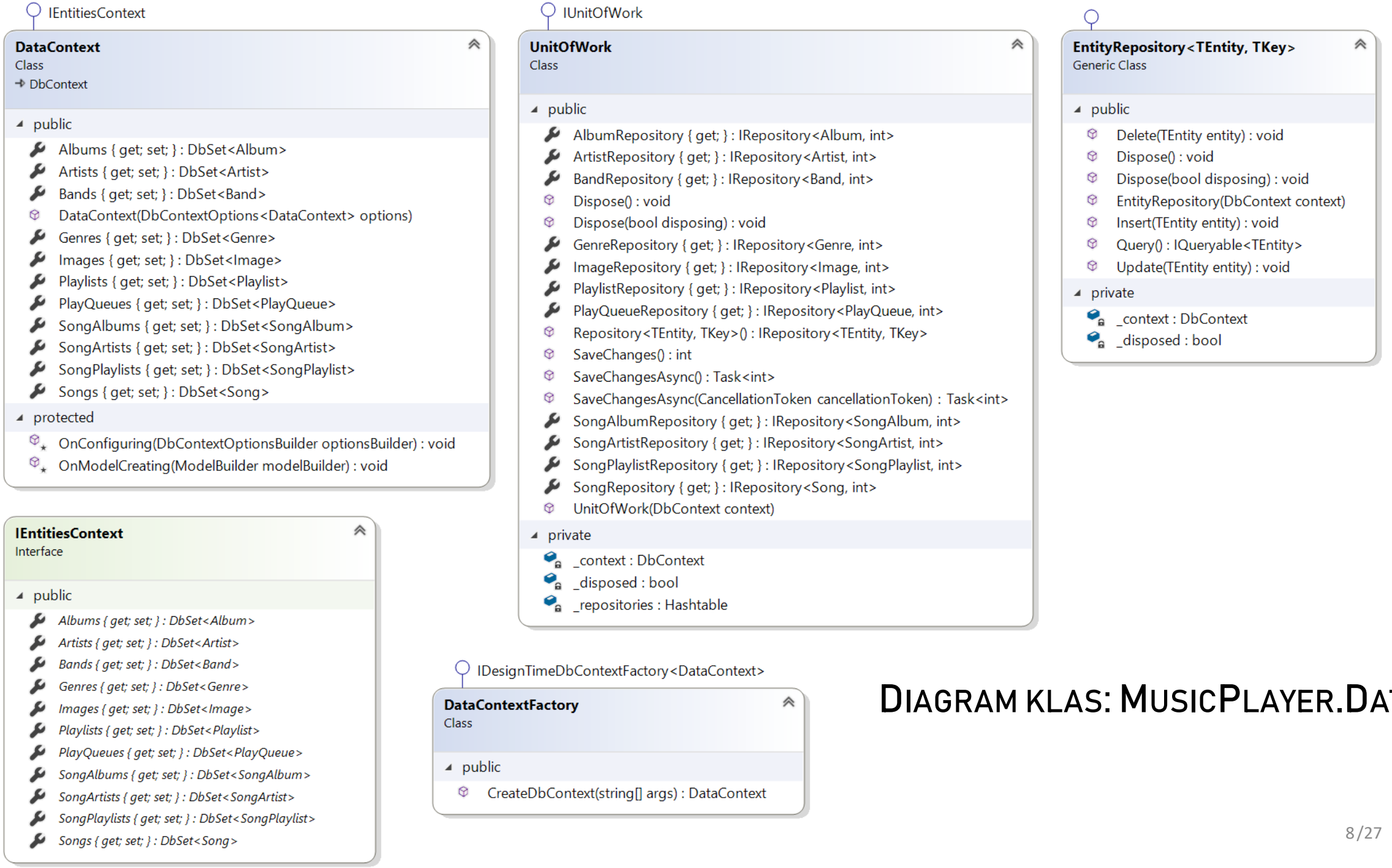


Diagram klas: MusicPlayer.Data

IAlbumController
IController<Result>

AlbumController
Class

- public
  - AddSong(int albumId, int songId, int trackNumber) : Task
  - AlbumController(IQueryDispatcher queryDispatcher, ICommandDispatcher commandDispatcher)
  - Create(string title, string description, DateTime publicationDate, int imageId) : Task<int>
  - Delete(int id) : Task
  - DeleteSong(int songId, int albumId) : Task
  - Get(int id) : Task<Result>
  - GetAll() : Task<List<Result>>
  - GetAllByDate() : Task<List<Result>>
  - GetAllByDateDescending() : Task<List<Result>>
  - GetAllByPublicationDate() : Task<List<Result>>
  - GetAllByPublicationDateDescending() : Task<List<Result>>
  - GetAllDescending() : Task<List<Result>>
  - GetSongs(int albumId) : Task<List<Result>>
  - Search(string name) : Task<List<Result>>
  - Update(int id, string title, string description, DateTime publicationDate, int imageId) : Task
- private
  - _commandDispatcher : ICommandDispatcher
  - _queryDispatcher : IQueryDispatcher

IGenreController
IController<Result>

GenreController
Class

- public
  - Create(string name, string description) : Task
  - Delete(int id) : Task
  - GenreController(IQueryDispatcher queryDispatcher, ICommandDispatcher commandDispatcher)
  - Get(int id) : Task<Result>
  - GetAll() : Task<List<Result>>
  - GetAllDescending() : Task<List<Result>>
  - GetSongs(int genreId) : Task<List<Result>>
  - Search(string name) : Task<List<Result>>
  - SongQuantity(int genreId) : Task<int>
  - Update(int id, string name, string description) : Task
- private
  - _commandDispatcher : ICommandDispatcher
  - _queryDispatcher : IQueryDispatcher

IImageController
IController<Result>

ImageController
Class

- public
  - Create(string filePath) : Task<int>
  - Delete(int id) : Task
  - Get(int id) : Task<Result>
  - GetAll() : Task<List<Result>>
  - ImageController(IQueryDispatcher queryDispatcher, ICommandDispatcher commandDispatcher)
  - Update(int id, string filePath) : Task
- private
  - _commandDispatcher : ICommandDispatcher
  - _queryDispatcher : IQueryDispatcher

IocConfig
Class

- public
  - RegisterDependencies(IServiceCollection services) : IContainer

IArtistController
IController<Result>

ArtistController
Class

- public
  - AddSong(int artistId, int songId) : Task
  - ArtistController(IQueryDispatcher queryDispatcher, ICommandDispatcher commandDispatcher)
  - Create(string name, string surname, string pseudonym, DateTime birthdate, string description, int?...
  - Delete(int id) : Task
  - DeleteSong(int songId, int artistId) : Task
  - Get(int id) : Task<Result>
  - GetAll() : Task<List<Result>>
  - GetAllDescending() : Task<List<Result>>
  - GetSongs(int artistId) : Task<List<Result>>
  - Search(string name) : Task<List<Result>>
  - Update(int id, string name, string surname, string pseudonym, DateTime birthdate, string descriptio...
- private
  - _commandDispatcher : ICommandDispatcher
  - _queryDispatcher : IQueryDispatcher

IPlayQueueController
IController<Result>

PlayQueueController
Class

- public
  - Create(int songId) : Task
  - Delete(int id) : Task
  - Get(int id) : Task<Result>
  - GetAll() : Task<List<Result>>
  - PlayQueueController(IQueryDispatcher queryDispatcher, ICommandDispatcher commandDispatcher)
  - RandomSong() : Task<Result>
- private
  - _commandDispatcher : ICommandDispatcher
  - _queryDispatcher : IQueryDispatcher

IBandController
IController<Result>

BandController
Class

- public
  - BandController(IQueryDispatcher queryDispatcher, ICommandDispatcher commandDispatcher)
  - Create(string name, DateTime creationDate, DateTime? endDate, string description) : Task
  - Delete(int id) : Task
  - Get(int id) : Task<Result>
  - GetAll() : Task<List<Result>>
  - GetAllDescending() : Task<List<Result>>
  - GetArtists(int bandId) : Task<List<Result>>
  - Search(string name) : Task<List<Result>>
  - Update(int id, string name, DateTime creationDate, DateTime? endDate, string description) : Task
- private
  - _commandDispatcher : ICommandDispatcher
  - _queryDispatcher : IQueryDispatcher

IQueryDispatcher

QueryDispatcher
Class

- public
  - Dispatch<TQuery, TResult>(TQuery query) : Task<TResult>
  - QueryDispatcher(IComponentContext context)
  - WriteInformation(string input) : void
- private
  - _context : IComponentContext

IPlaylistController
IController<Result>

PlaylistController
Class

- public
  - AddSong(int playlistId, int songId) : Task
  - Create(string name, string description) : Task<int>
  - Delete(int id) : Task
  - DeleteSong(int songId, int playlistId) : Task
  - Get(int id) : Task<Result>
  - GetAll() : Task<List<Result>>
  - GetAllByDate() : Task<List<Result>>
  - GetAllByDateDescending() : Task<List<Result>>
  - GetAllDescending() : Task<List<Result>>
  - GetSongs(int playlistId) : Task<List<Result>>
  - PlaylistController(IQueryDispatcher queryDispatcher, ICommandDispatcher commandDispatch...
  - Search(string name) : Task<List<Result>>
  - Update(int id, string name, string description) : Task
- private
  - _commandDispatcher : ICommandDispatcher
  - _queryDispatcher : IQueryDispatcher

ISongController
IController<Result>

SongController
Class

- public
  - Create(int score, string title, DateTime creationDate, string filePath, int? imageId, int genreId) : Task<int>
  - Delete(int id) : Task
  - Get(int id) : Task<Result>
  - GetAlbum(int songId) : Task<Result>
  - GetAlbums(int songId) : Task<List<Result>>
  - GetAll() : Task<List<Result>>
  - GetAllByDate() : Task<List<Result>>
  - GetAllByDateDescending() : Task<List<Result>>
  - GetAllByLength() : Task<List<Result>>
  - GetAllByLengthDescending() : Task<List<Result>>
  - GetAllByPlayTimes() : Task<List<Result>>
  - GetAllByPlayTimesDescending() : Task<List<Result>>
  - GetAllByScore() : Task<List<Result>>
  - GetAllByScoreDescending() : Task<List<Result>>
  - GetAllDescending() : Task<List<Result>>
  - GetArtists(int songId) : Task<List<Result>>
  - GetBand(int artistId) : Task<Result>
  - IncreasePlayTimes(int id) : Task
  - Search(string name) : Task<List<Result>>
  - SetScore(int id, int score) : Task
  - SongController(IQueryDispatcher queryDispatcher, ICommandDispatcher commandDispatcher)
  - Update(int id, int score, string title, DateTime creationDate, string filePath, int? imageId, int genreId) : Task
- private
  - _commandDispatcher : ICommandDispatcher
  - _queryDispatcher : IQueryDispatcher

ICommandDispatcher

CommandDispatcher
Class

- public
  - CommandDispatcher(IComponentContext context)
  - Dispatch<TCommand, TResult>(TCommand command) : Task<TResult>
  - Dispatch<TCommand>(TCommand command) : Task
- private
  - _context : IComponentContext

DIAGRAM KLAS: MUSICPLAYER.UWP CZĘŚĆ 1

DIAGRAM KLAS: MUSICPLAYER.UWP CZEŚĆ 2

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

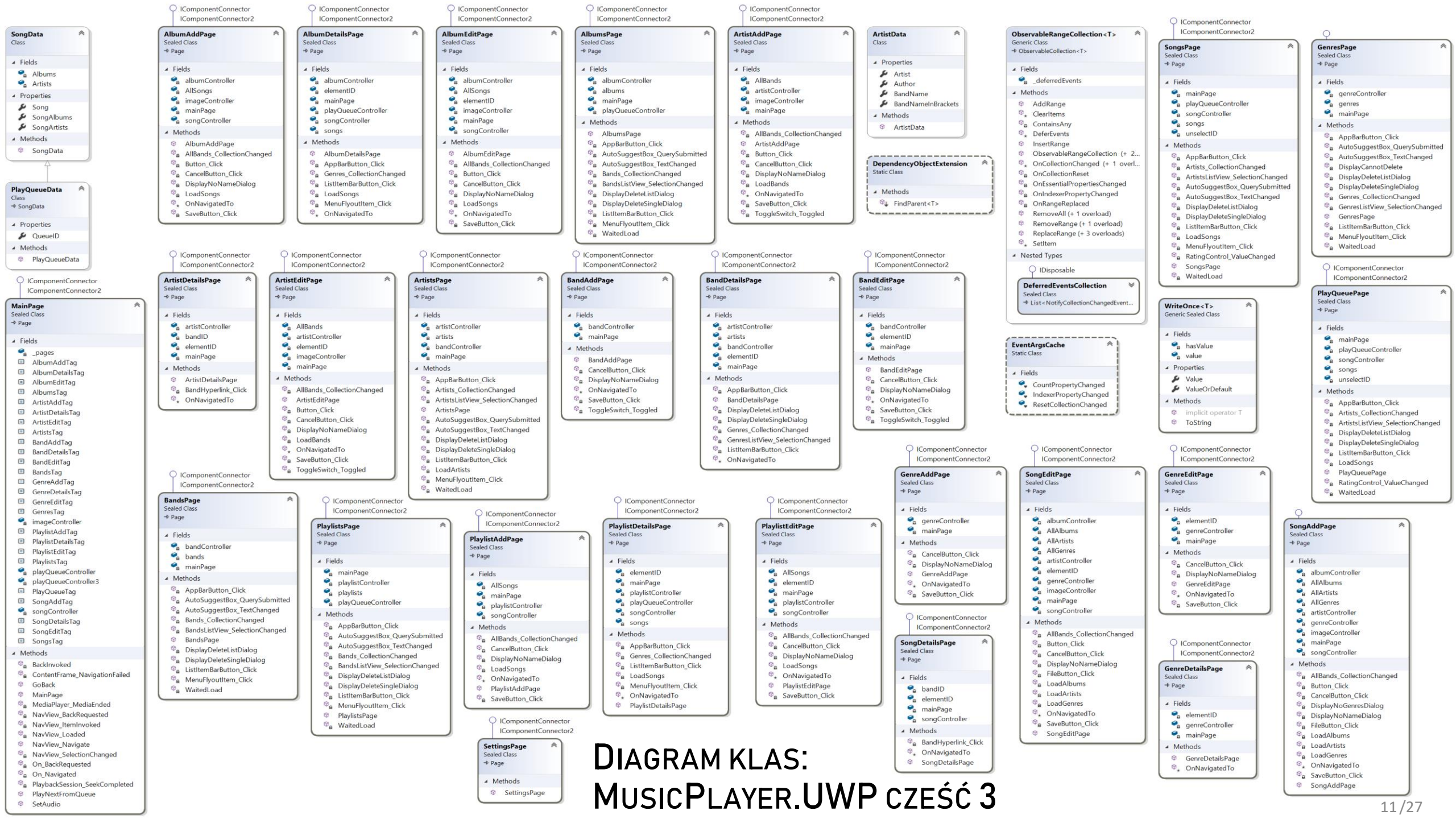


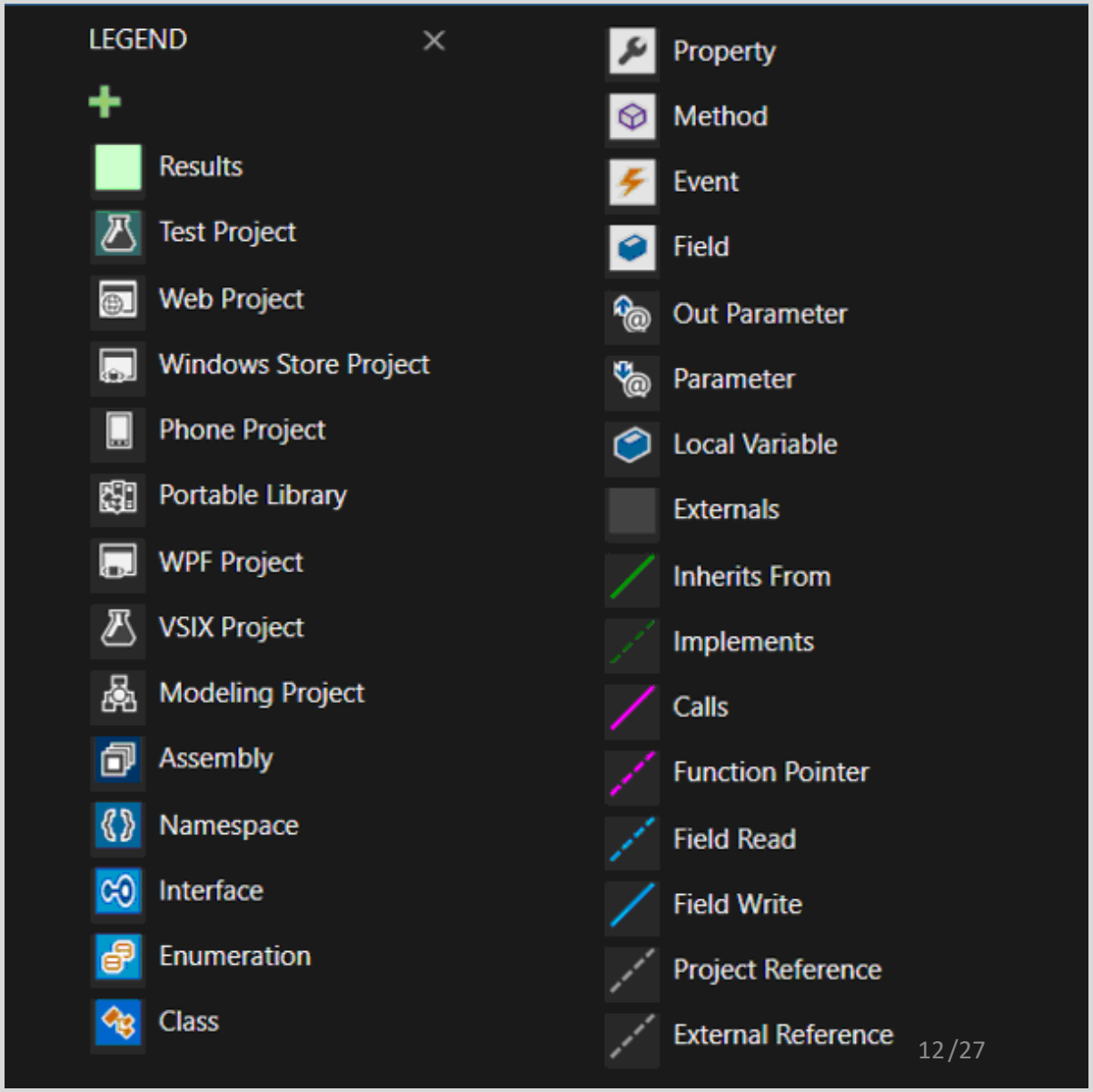
DIAGRAM KLAS: MUSICPLAYER.UWP CZEŚĆ 3

Najciekawsze rozwiązania

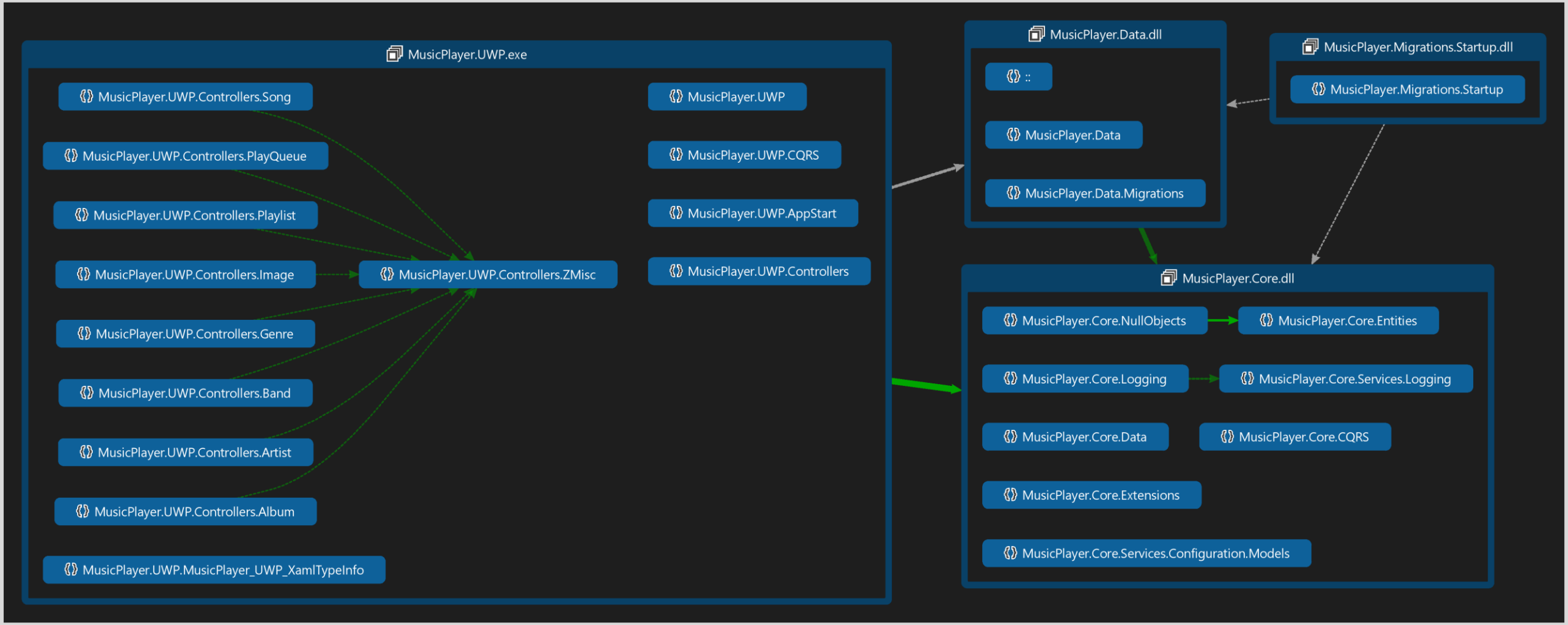
Zastosowanie wzorców projektowych:

1. Dependency injection
2. Repository
3. CQRS
4. Unit of Work
5. Model-View-Presenter (MVP)

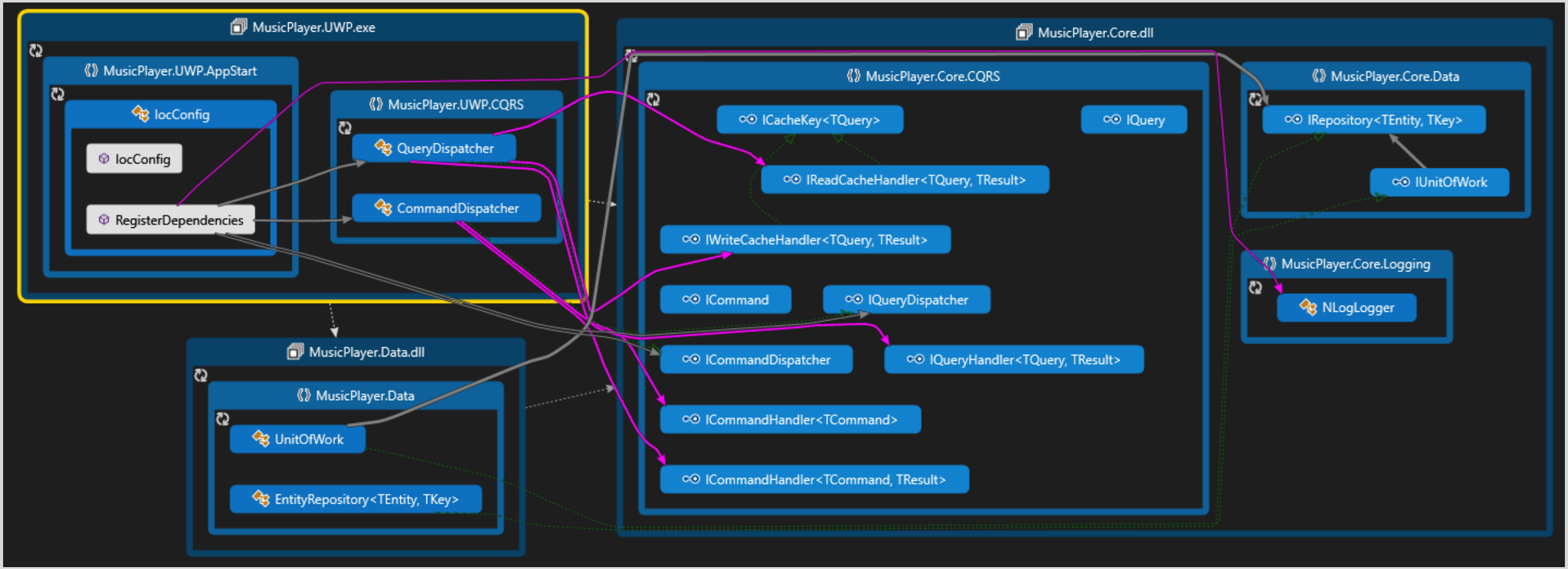
Legenda oznaczeń na diagramach



Zależności stworzonych bibliotek w projekcie



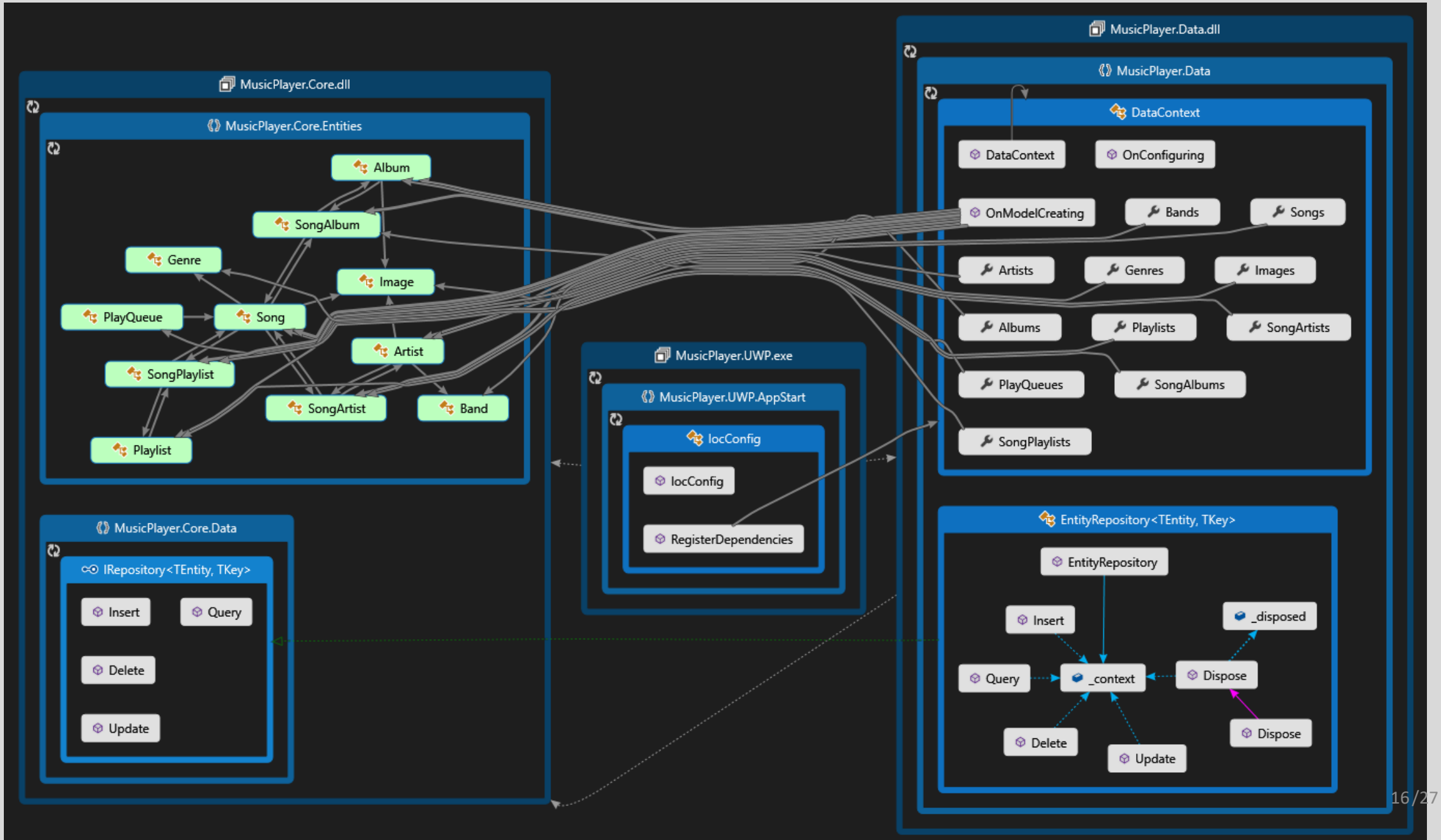
Wzorzec #1 (kreacyjny): Dependency injection



Wzorzec #1 (kreacyjny): Dependency injection

Dependency Injection (Wstrzykiwanie zależności) – wzorzec projektowy i wzorzec architektury oprogramowania zastosowany w projekcie w celu usunięcia bezpośrednich zależności pomiędzy komponentami na rzecz architektury typu plug-in. Polegać on na przekazywaniu utworzonych instancji obiektów udostępniających swoje metody i właściwości obiektom, które z nich korzystają (np. jako parametry konstruktora). 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.

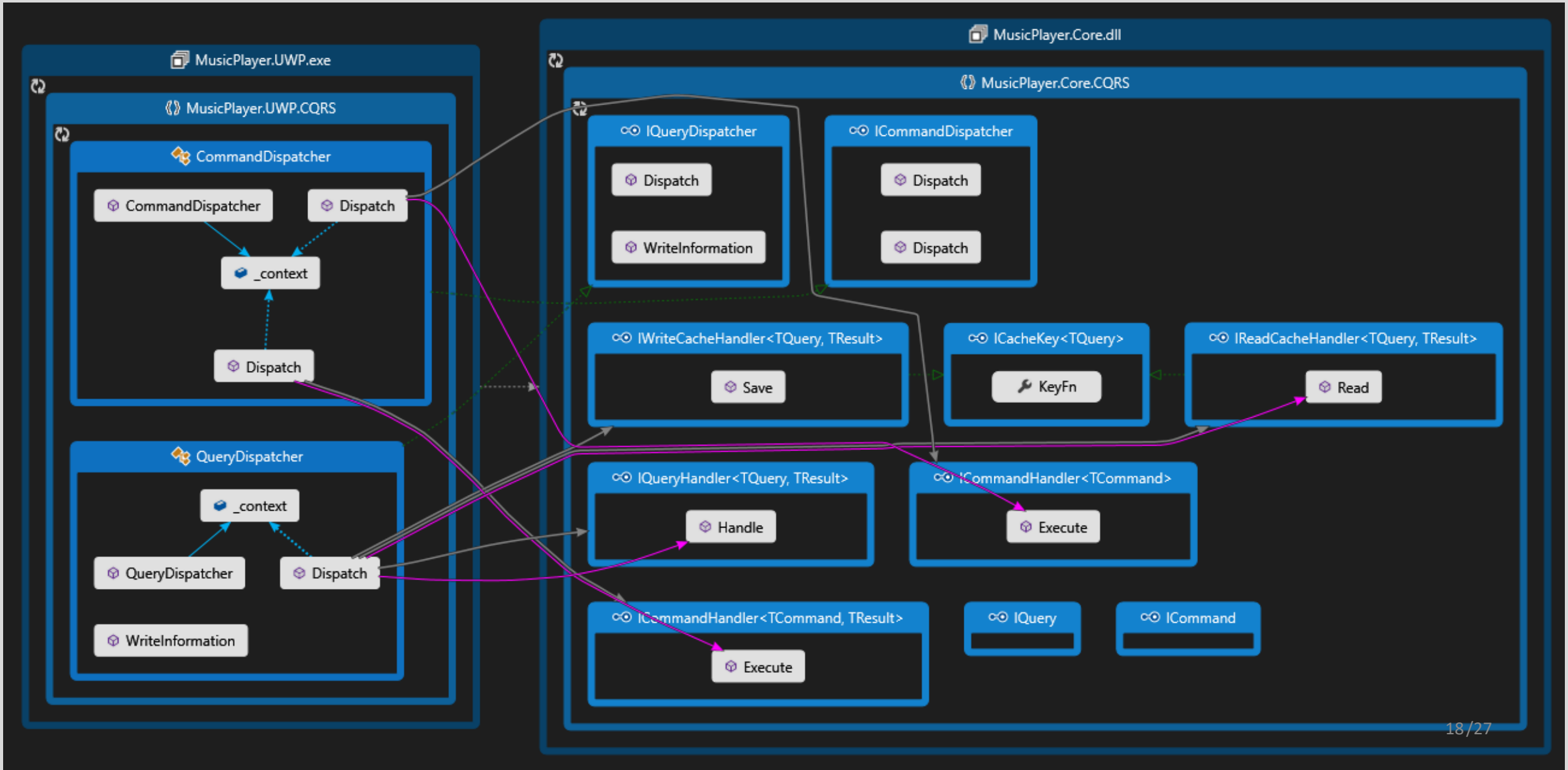
Wzorzec #2 (strukturalny): Repository



Wzorzec #2 (strukturalny): Repository

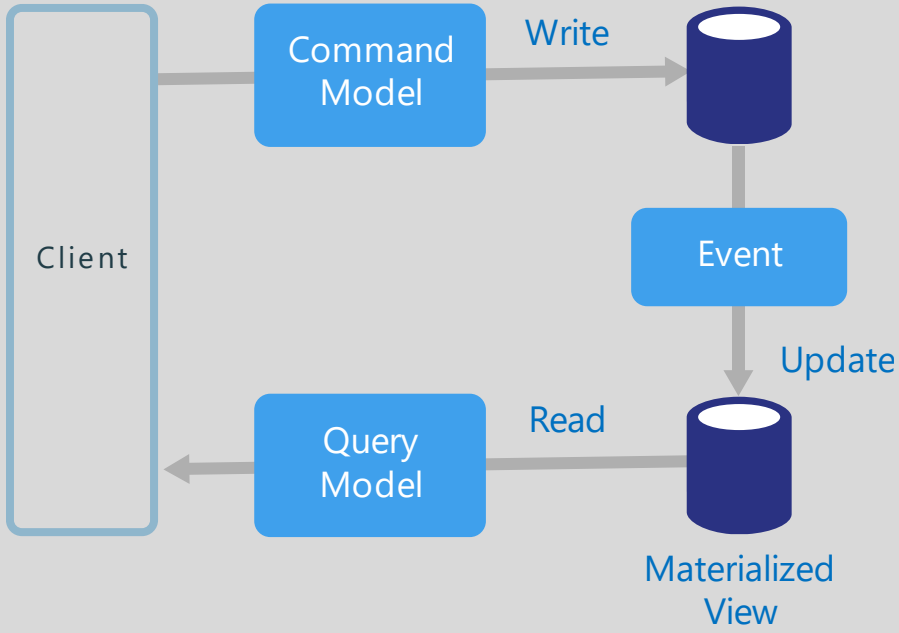
Repozytoria to klasy lub komponenty, które zawierają logikę wymaganą do uzyskania dostępu do źródeł danych. Centralizują one wspólną funkcjonalność dostępu do danych, zapewniając lepszą edytowalność i oddzielenie infrastruktury lub technologii wykorzystywanej do uzyskiwania dostępu do baz danych z warstwy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

Wzorzec #3 (czynnościowy): CQRS



Wzorzec #3 (czynnościowy): CQRS

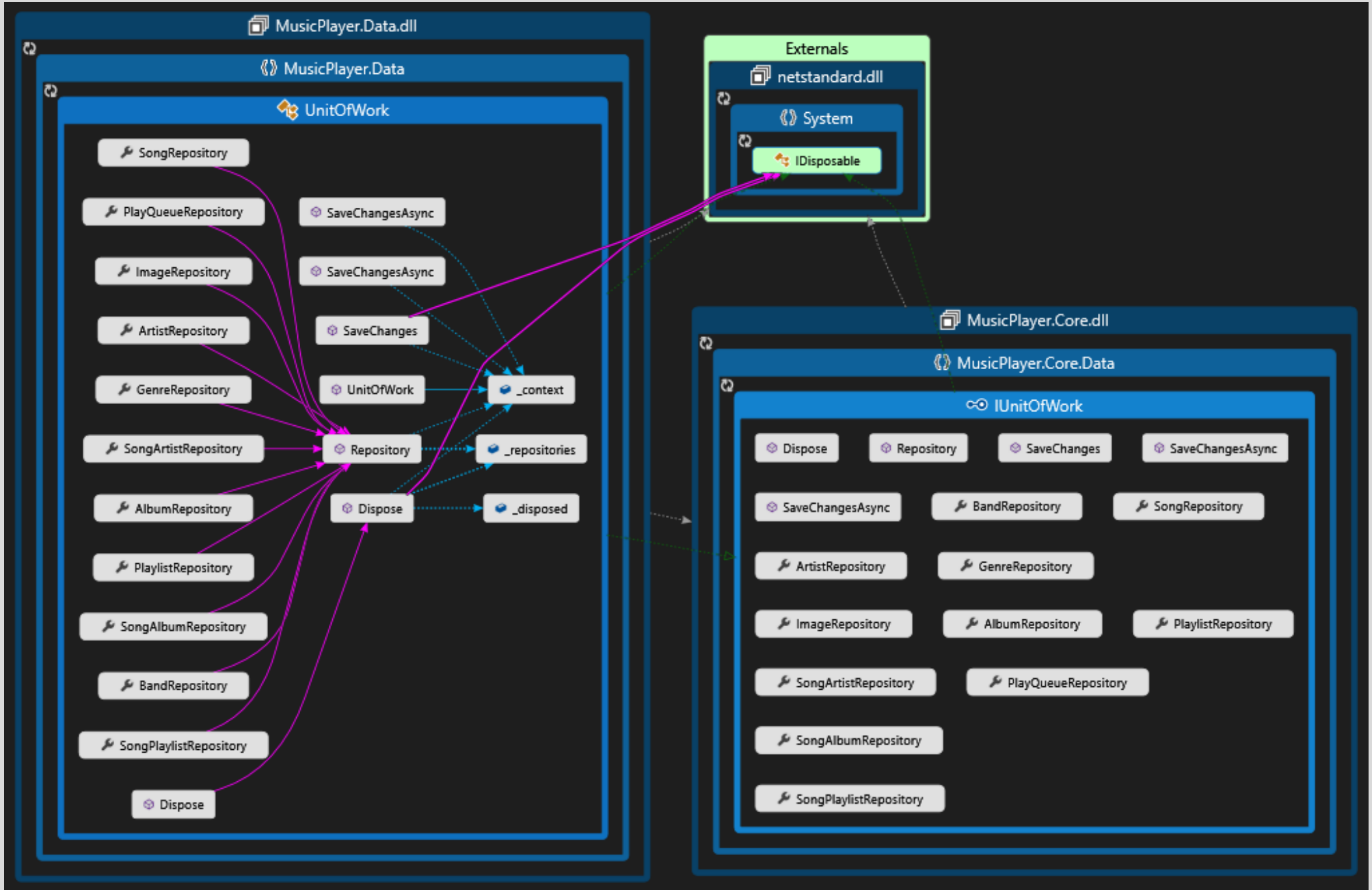
W tradycyjnych architekturach ten sam model danych jest używany do wysyłania zapytań do bazy danych i aktualizowania jej. Jest to proste i dobrze się sprawdza w przypadku podstawowych operacji CRUD. Jednak w naszej aplikacji, która jest bardziej złożona metoda została uznana za niewygodną i nieczytelną. 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 są oparte na zadaniach, a nie skoncentrowane na danych. Dzięki temu są one 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.

Wzorzec #4: Unit of Work

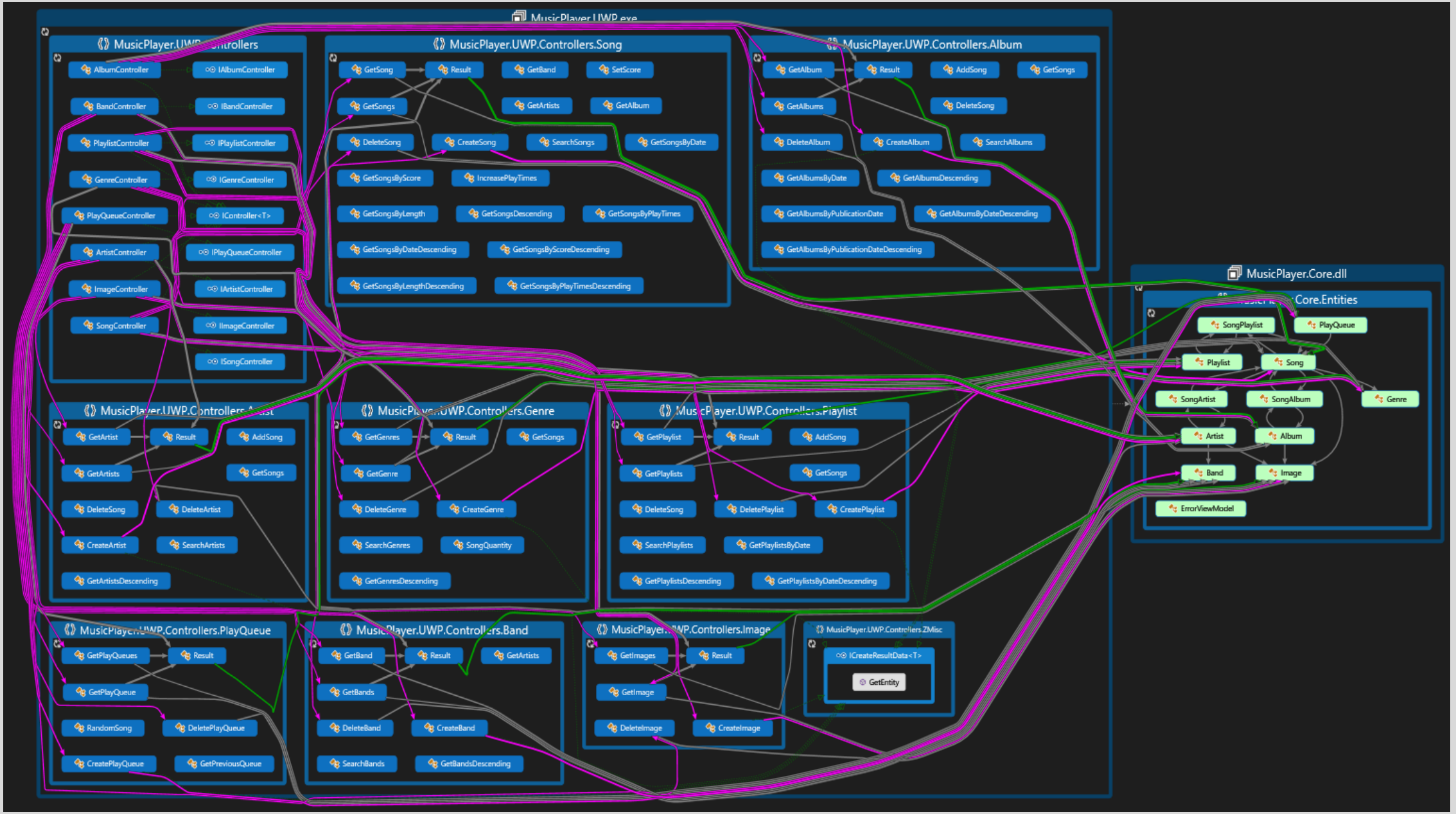


Wzorzec #4: Unit of Work

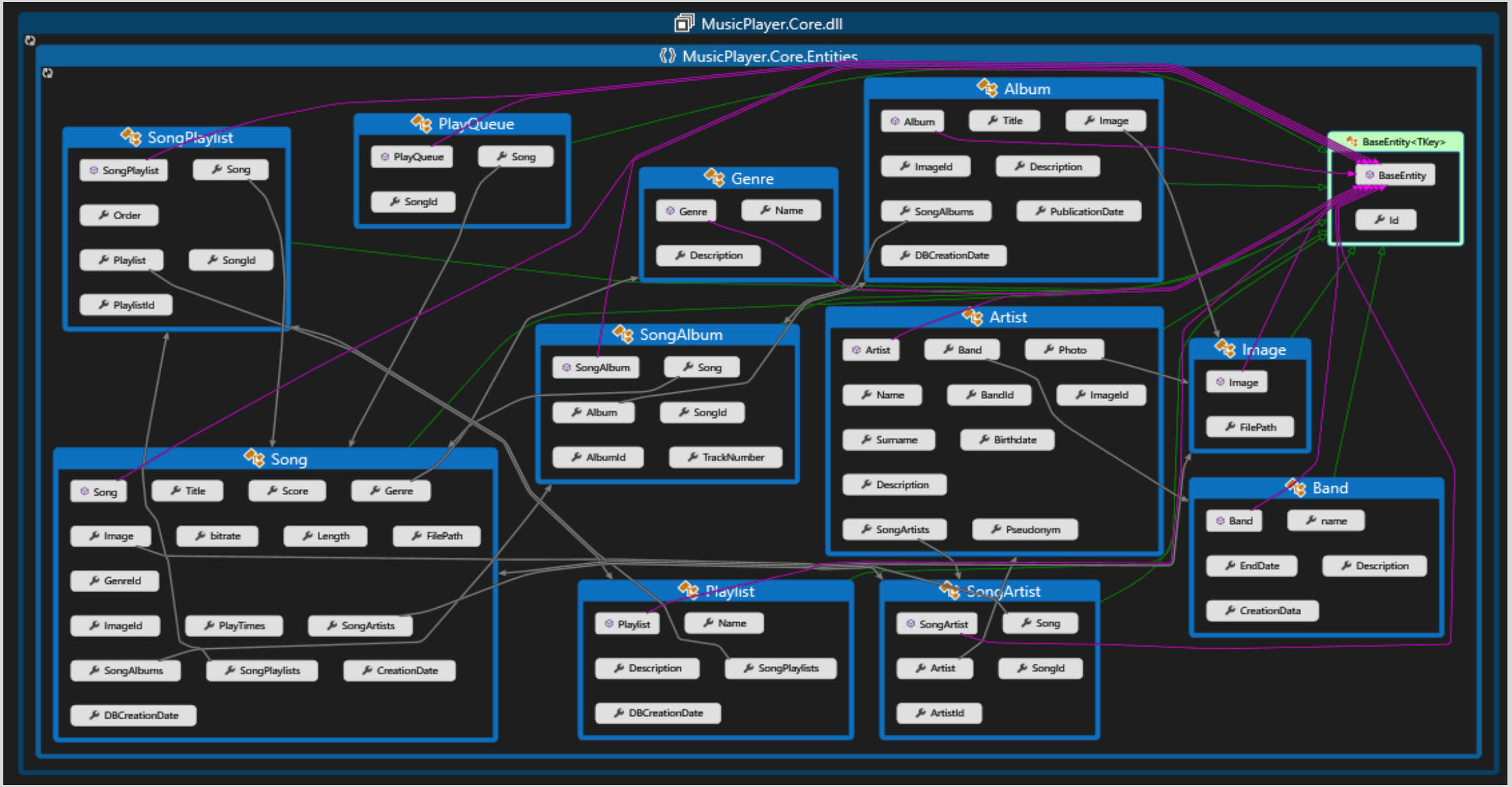
Unit of Work jest określana jako pojedyncza transakcja, która obejmuje wiele operacji wstawiania, aktualizowania lub usuwania. Mówiąc najprościej, oznacza to, że dla określonego działania użytkownika, takiego jak rejestracja na stronie internetowej, wszystkie operacje wstawiania, aktualizacji i usuwania 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.

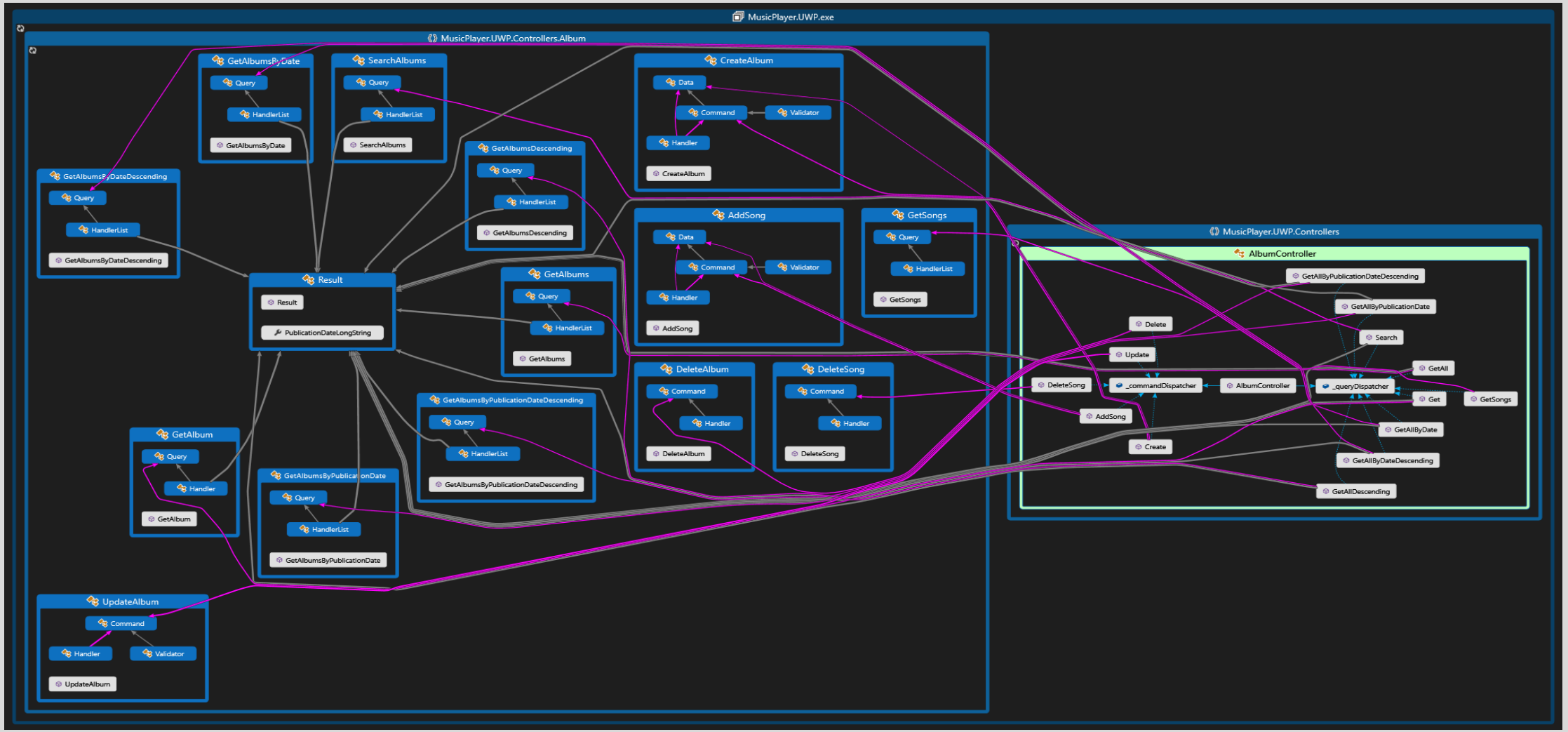
Wzorzec #5 (architekturalny): MVP - struktura



Wzorzec # 5 (architekturalny): MVP - Modele

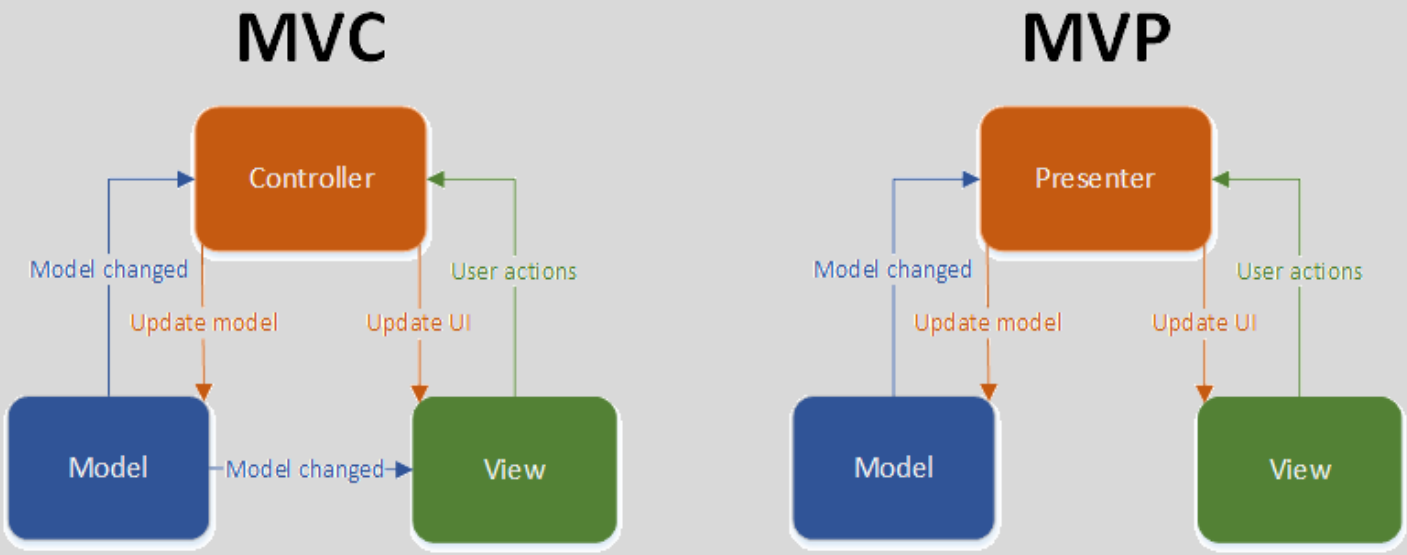


Wzorzec # 5 (architekturalny): MVP – Przykładowy prezenter Albumu



Wzorzec #5 (architekturalny): MVP

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.



Dane nie są przekazywane bezpośrednio z modelu do widoku jak to ma miejsce w MVC. Prezener wysyła zapytanie do modelu, model zwraca dane do prezentera, prezenter przetwarza otrzymane dane i przekazuje do widoku.

W projekcie zastosowano nazewnictwo Prezentera jako Kontroler.

Podział pracy nad projektem

Czynność/Zadanie	Osoba		
	Magdalena Kalisz	Adam Bajguz	Michał Kierzkowski
OGÓLNE			
Zaprojektowanie 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		+	
Utworzenie modeli bazy danych	+	+	
Implementacja dostępu do bazy danych	+		
Implementacja repozytoriów, CQRS tworzących dodatkową warstwę abstrakcji w MVP	+		
Stworzenie pozostałych części backendu	+		
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

- 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.
- Sposób implementacji połączenia z bazą i jej obsługi umożliwia łatwą zmianę typu używanej bazy, a także uruchomienie i połączenie ze zdalną bazą danych działająca np. na platformie Azure lub własnym serwerze innym niż lokalny.
- Obecna implementacja loggera (NLogLogger) pozwala na rozszerzenie jego funkcjonalności w taki sposób aby różnego typu logi były zapisywane w bazie danych, zamiast obecnego przeznaczenia loggera jako pomocy przy pisaniu programu, co ułatwiło by naprawianie błędów w programie pojawiających się podczas użytkowania przez użytkowników.
- Zastosowanie wzorca Dependency Injection pozwala na odseparowanie funkcjonalności (abstrakcji) usług wykorzystywanych przez kontrolery od ich implementacji. Dzięki temu można je dowolnie podmieniać.