Hurtownia danych

# Projekt: Proces ETL

# Wstęp

W celu współdzielenia kodu pomiędzy członków zespołu, stworzone zostało repozytorium w systemie kontroli wesji GIT, który umożliwia korzystanie z repozytoriów na portalu GitHub.

Zdalne repozytorium projektu dostępne znajduję się pod adresem: <https://github.com/Cenarius1/HurtowniaDanych> Projekt został wykonany w języku programowania Microsoft Visual C# z wykorzystaniem platformy .NET Framework w wersji 4.7.2. Na potrzeby projektu wybrano platformę bazodanową MS SQL Server w wersji 13.0.

# Technologie

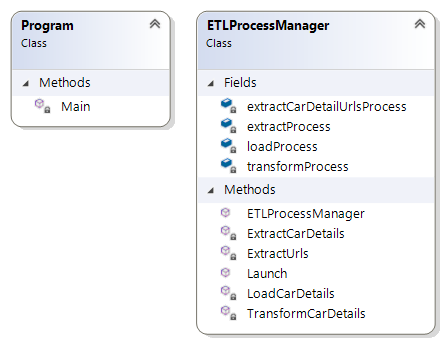
Sss

Całościowe rozwiązanie nosi nazwę *HurtowniaDanych* i składa się z pięciu projektów:

* DWH – *TBC*
* DWH.Domain – *TBC*
* DWH.ETL – *TBC*
* DWH.GUI – *TBC*
* DWH.Storage – *TBC*

# Projekt DWH

## Diagram klas



W skład projektu DWH wchodzą dwie klasy, *Program* oraz *ETLProcessManager*.

## Class Program

W klasie tej wyróżniamy główną metodę statyczną – *Main()*, która na wejściu przyjmuje argumenty podawane z linii poleceń. Następnie zostaje utworzony obiekt *etlManager* typu *ETLProcessManager*, który jest głównym elementem aplikacji. Na tak utworzonym obiekcie wywołujemy metodę *Launch()*.

## Class ETLProcessManager

W klasie zadeklarowane są cztery pola: *extractCarDetailUrlsProcess*, *extractProcess*, *transformProcess* oraz *loadProcess*. Nadanie im wartości następuje poprzez wywołanie konstruktora klasy, metody *ETLProcessManager()*.

Metoda *Launch()* na podstawie ustalonego adresu URL w zmiennej ‘baseUrl’, wywołuje metodę *ExtractUrls()* która przyjmuje ‘baseUrl’ jako parametr i zwraca listę adresów URL portalu OtoMoto. Kolejnym krokiem w procesie jest wyekstrahowanie danych z każdego ogłoszenia poprzez wywołanie metody *ExtractCarDetails()*. Na powstałej kolekcji obiektów, dokonujemy transformacji do docelowego modelu danych, który odpowiada schematowi bazy danych – metoda *TransformCarDetails()*. Finalnym etapem jest wykonanie metody *LoadCarDetails()*, która jak nazwa sugeruje służy do wprowadzenia danych do bazy.

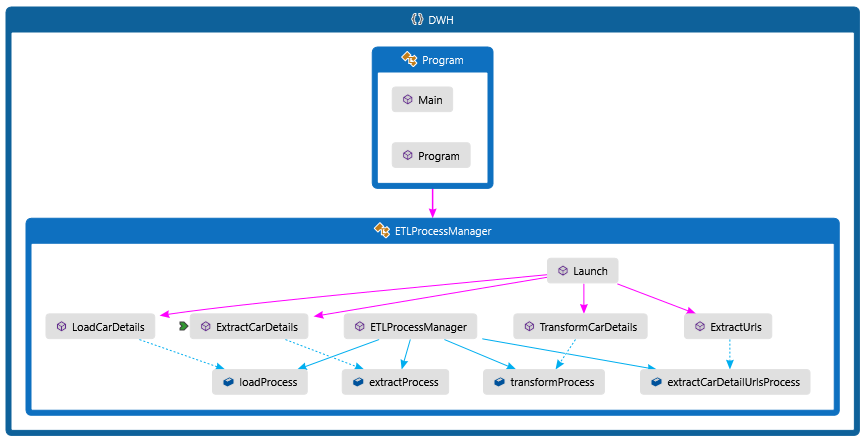
Metoda *ExtractUrls()* otrzymuję parametr wejściowy typu *string*. Poprzez wykorzystanie uprzednio zainicjowanego pola *extractCarDetailUrlsProcess,* wykonuje metodę *Process()* której wynikiem jest kolekcja ogłoszeń.

Metoda *ExtractCarDetails()* przyjmuje na wejściu listę adresów URL w postaci listy. Wewnątrz metody tworzona jest osobna lista, która zawiera listę obiektów typu *ExtractCarDetail* i jest wynikiem równoległego przetwarzania każdego pojedynczego adresu URL na ustalony model.

Metoda *TransformCarDetails()* otrzymuje jako parameter wejściowy listę obiektów typu *ExtractCarDetail*. Każdy pojedynczy obiekt odpowiada pojedynczemu ogłoszeniu portalu OtoMoto. Zainicjowana zostaje wynikowa lista obiektów typu *LoadCarDetail*, która zostaje wypełniona podczas procesu mapowania odpowiednich wartości do ustalonego modelu przechowywania danych.

Metoda *LoadCarDetails()* akceptuje na wejściu listę obiektów typu *LoadCarDetail*, a następnie wywołuje metodę *Process()* pola *loadProcess.* W przypadku poprawnego załadowania danych zostaje zwrócony komunikat treści ‘ETL Process finished succesfully’, w scenariuszu negatywnym komunikat ‘ETC Process encountered problems, check console logs for details.’

## Mapa zależności projektu DWH



# Projekt DWH.Domain

## Abstract Class Entity

W klasie abstrakcyjnej chodzi o wspólną implementację dla dziedziczonych klas. Zdefiniowane właściwości tworzą unikatowe parametry odróżnienia danych, tj:

* Id – numer ogłoszenia nadawany w procesie extract
* RowGuid – identyfikator rekordu tabeli typu GUID, wartość automatycznie ustawiana
* InsertedDttm – znacznik czasu użyty w procesie load

## Class LoadCarDetail

Klasa dziedziczy klasę abstrakcyjną *Entity*, w której skład wchodzą właściwości charakteryzujące pojedyncze ogłoszenie serwisu OtoMoto. Jest to kompletna lista elementów zawierająca podstawowe informacje ogłoszenia takie jak tytuł (‘*’Title‘’*), markę pojazdu (*‘’Mark’*’) czy cenę (*‘’PriceRaw’’*), jak również pełną listę wyposażenia pojazdu, tj. asystent parkowania (*‘’ParkAssist’’*), złącze IsoFix (*‘’Isofix’’*) – łącznie 93 właściwości.

## Class ExtractCarDetails

Klasa głównie składa się z skróconej deklaracji właściwości, które zapewniają dostęp do pól klasy posługując się akcesoriami get i set. Celem tej klasy jest mapowanie parsowanego łańcucha JSON do obiektu, który zostanie poddany kolejno transformacji. Mapowanie odbywa się przez ustawienie atrybutu *JsonProperty* dla każdej właściwości, np.

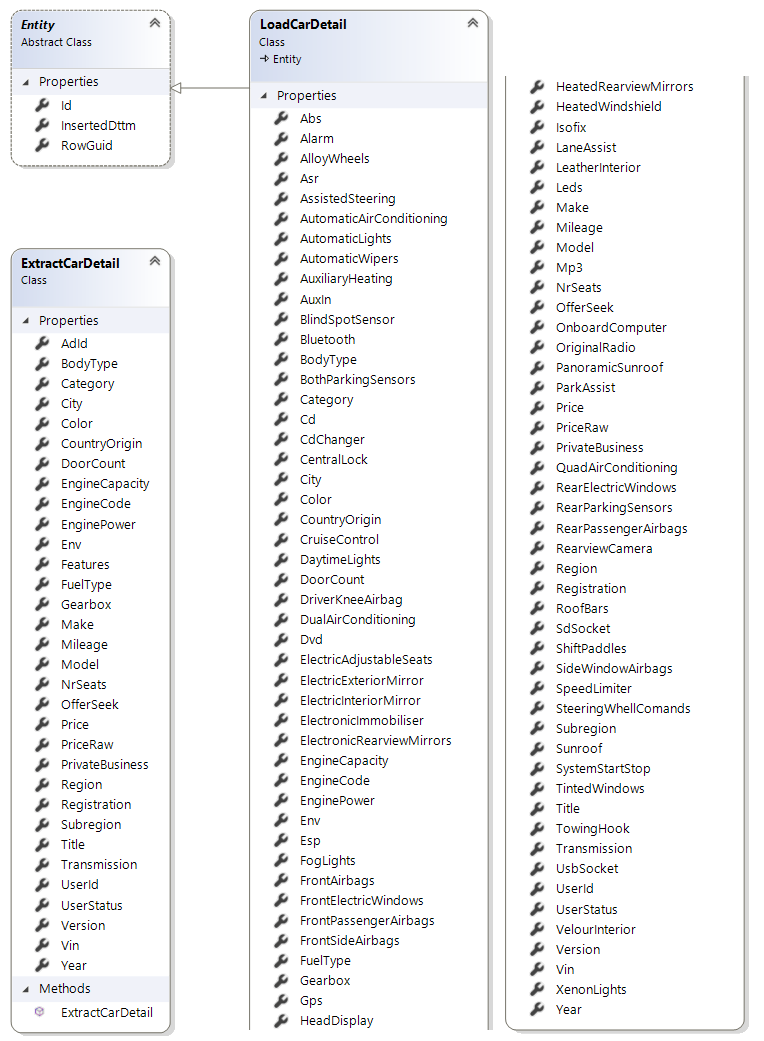
*JsonProperty(‘’title’’) 🡪 string Title*

*JsonProperty(‘’engine\_code’’) 🡪 string EngineCode*

W celu identyfikacji każdego utworzonego obiektu, użyta została właściwość *AdId* – jest to unikalny numer pojedynczego ogłoszenia serwisu.

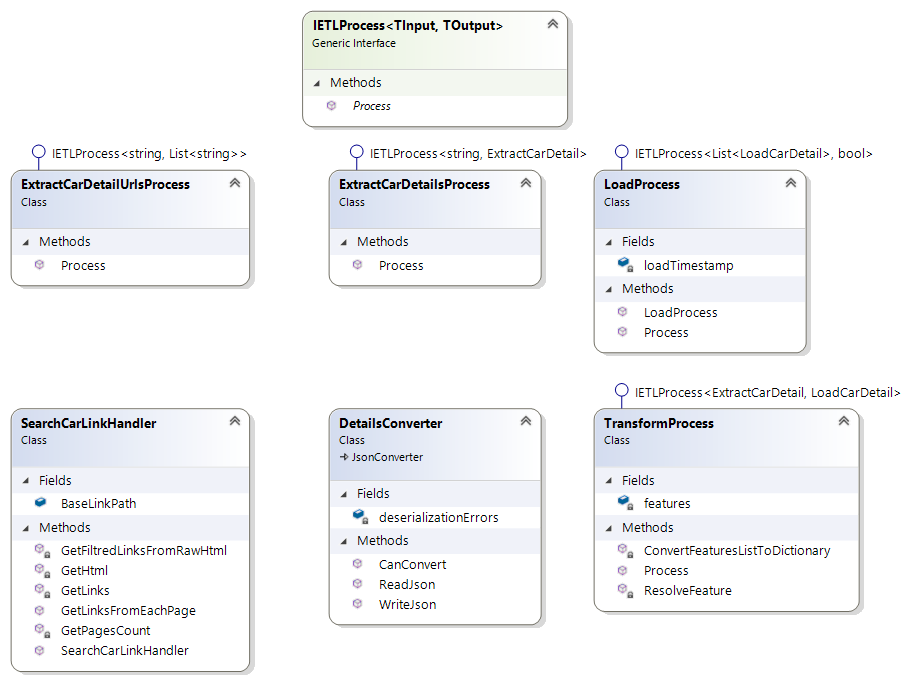
Użyty konstruktor ExtractCarDetails inicjują pustą kolekcję *Features*, która zostaje wypełniona elementami wyposażenia auta w trakcie parsowania.

## Diagram klas



# Projekt DWH.ETL

## Diagram klas



## Interface IETLProcess<TInput, TOutput>

*Opis…*

## Class ExtractCarDetailUrlsProcess

*Opis…*

## Class ExtractCarDetailsProcess

*Opis…*

## Class LoadProcess

*Opis…*

## Class TranformProcess

*Opis…*

## Class SearchCarLinkHandler

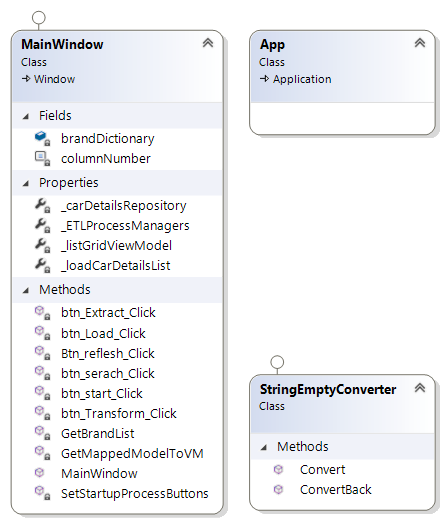
*Opis…*

## Class DetailsConverter

*Opis…*

# Projekt DWH.GUI

## Diagram klas



## Class App (Xaml)

Klasa pozwala na uruchomienie głównego okna *MainWindow.xaml*. Elementem root tego pliku jest element ‘Application’ – początek aplikacji. Użyty atrybut ‘StartupUri’ określa, które okno ma być otwarte w dalszym kroku. Domyślnie jest to plik *MainWindow.xaml.*

## Class MainWindow (Xaml)

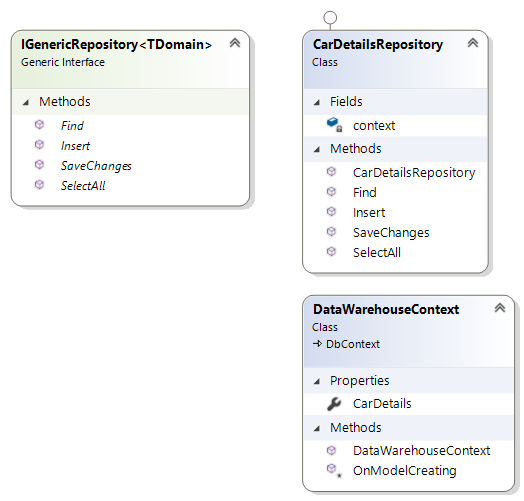
Klasa dziedziczy po klasie *System.Windows.Window*. Elementem root tego pliku jest element ‘Window’. Wewnątrz znajduje się element ‘Grid’, który tworzy zawartość okna poprzez zagnieżdżenie odpowiednich kontrolek UI tj. przyciski, tabela, pole tekstowe.

## Class StringEmptyConverter

Klasa implementuje interfejs *IValueConverter* i jest klasą pomocniczą. Zaimplementowana metoda *Covert()* pozwala na przypisanie w miejscu pustej wartości napisu ‘N/A’ (*not available*) oraz transformację występującej wartości do ‘Title Case’ notacji.

# Projekt DWH.Storage

## Diagram klas



## Interface IGenericRepository<TDomain>

Deklaruje składowe metody użyte do współpracy z funkcjami które udostępnia Entity Framework.

## Class CarDetailsRepository

Klasa implementuje interfejs *IGenericRepository* typu *LoadCarDetail*. Konstruktor *CarDetailsRepository* inicjuje pole *context*, które jest obiektem bazy.

Metoda *Insert()* jest odpowiedzialna za dodanie nowego ogłoszenia do bazy MS SQL. Przed dodaniem nowego wpisu, weryfikowana jest obecność ogłoszenia w bazie. W przypadku pojawienia się duplikatu informacji, ogłoszenie zostaje zingnorowane.

Metoda *SelectAll()* służy do pobrania wszystkich rekordów tabeli ‘CarDetails’. Otrzymane dane tworzą kolekcję typu *LoadCarDetail*, co pozwala w łatwy sposób na iterację wszystkich ogłoszeń.

Metoda *Find()* pozwala za pomocą parametru wejściowego *prediction* na zwrócenie podzbioru ogłoszeń według określonego kryterium.

Metoda *SaveChanges()* zapisuje zmiany które zostały dokonane na obiekcie bazy – *context*.

## Class DataWarehouseContext

Klasa dziedziczy po klasie *DBContext* i utworzone w niej obiekty odpowiadają fizycznej bazie danych. Konstruktor *DataWarehouseContext()*, zawiera w sobie właściwość z klasy bazowej. Parametr *base* w konstruktorze klasy nawiązuje połączenie z bazą danych o nazwie ETL\_DWH. Ciąg połączenia jest ustawiony w pliku *App.config* projektu DWH.

Właściwość *CarDetails* typu DbSet<LoadCarDetail> reprezentuje tabelę ‘dbo.CarDetails’ w bazie. Pozwala na wykonanie wszystkich operacji CRUD na zdefiniowanych modelach.

Projekt ‘DWH.Storage’ zawiera również katalog Migrations. Jego zawartość jest automatycznie generowana po uprzednim włączeniu migracji komendą „Enable-Migrations”. Następnie pozostaje wpisać „Update-Database” aby utworzyć bazę danych o nazwie ETL\_DWH w lokalnej instancji SQL Server.