Spis treści

[1. Jak uruchomić na przykładzie 3](#_Toc518315269)

[1.1 Dodanie template 3](#_Toc518315270)

[1.2 Przygotowanie projektu web 3](#_Toc518315271)

[1.3 Uruchomienie projektu 3](#_Toc518315272)

[1.4 Dodanie encji 4](#_Toc518315273)

[1.4.1 Atrybuty z generatora 4](#_Toc518315274)

[1.4.1.1 Atrybuty ogólne 4](#_Toc518315275)

[1.4.1.2 Atrybuty do rejestrów agregacji 6](#_Toc518315276)

[1.4.2 Inne wykorzystywane atrybuty 7](#_Toc518315277)

[1.4.3 Przykład 7](#_Toc518315278)

[1.5 Dodanie i wgranie migracji 10](#_Toc518315279)

[1.6 Automatyczne generowanie aplikacji 10](#_Toc518315280)

[1.7 Generowanie aplikacji samemu 10](#_Toc518315281)

[1.7.1 Generowanie dla zwykłej, podstawowej encji (nie będącej enumem, intersekcją ani edytowaną przez inną encję): 11](#_Toc518315282)

[1.7.2 Generowanie dla encji które są tabelą intersekcji 13](#_Toc518315283)

[1.7.3 Generowanie dla encji która jest edytowana przez inna encje 13](#_Toc518315284)

[1.7.4 Generowanie dla enumów 14](#_Toc518315285)

[1.8 Zmiany w projekcie po generacji 15](#_Toc518315286)

[1.8.1 Projekty C# 15](#_Toc518315287)

[1.8.2 Angular 16](#_Toc518315288)

[2. Opis generatora 18](#_Toc518315289)

[2.1 Header 19](#_Toc518315290)

[2.2 Autogenerate 20](#_Toc518315291)

[2.3 TemplateRunner 21](#_Toc518315292)

[2.4 Class generator 22](#_Toc518315293)

[3. Funkcjonalności template’u projektu 22](#_Toc518315294)

[3.1 InjectableService 22](#_Toc518315295)

[3.2 Logger 23](#_Toc518315296)

[3.2.1 LoggingLevel 23](#_Toc518315297)

[3.2.2 ILoggerDataProvider 24](#_Toc518315298)

[3.3 Emailing 24](#_Toc518315299)

[3.3.1 DataLayer.MailDbContext 24](#_Toc518315300)

[3.3.2 Web.appsettings.json 24](#_Toc518315301)

[3.3.3 DomainClasses.Mail 24](#_Toc518315302)

[3.3.4 DTO.EmailMessageModel 25](#_Toc518315303)

[3.3.5 EmailingService 26](#_Toc518315304)

[3.3.6 EmailingConfig 26](#_Toc518315305)

[3.3.7 EmailingConfigResolver 26](#_Toc518315306)

[3.3.8 EmailSender 26](#_Toc518315307)

[3.3.9 Przykład użycia 26](#_Toc518315308)

[3.4 Notyfikacje 27](#_Toc518315309)

[3.4.1 INotificationService 27](#_Toc518315310)

[3.4.2 ApiResult 27](#_Toc518315311)

[3.5 Eventy 28](#_Toc518315312)

[3.5.1 IEventPublisher 28](#_Toc518315313)

[3.5.2 Rodzaje eventów 28](#_Toc518315314)

[3.5.3 Przykład 28](#_Toc518315315)

[3.6 ScheduledTask 29](#_Toc518315316)

[3.6.1 IScheduledTask 29](#_Toc518315317)

[3.6.2 Przykład 29](#_Toc518315318)

[3.7 Rejestry agregacji 30](#_Toc518315319)

# Jak uruchomić na przykładzie

Wymagania:

1. Zainstalować npm (node.js wersja 8 lub wyższa)
2. Zainstalować angular-cli. (npm install -g @angular/cli). Należy upewnić się, że mamy zainstalowaną odpowiednia wersję. Testowane na wersji 6.0.8

Uwagi:

Jeżeli nie masz uprawnień admina to polecenia z angular-cli mogą nie przechodzić. W takim przypadku należy je puszczać z node.js command prompt a nie zwykłego cmd.

## Dodanie template

1. W repo z generatorem kodu w folderze template znajduje się template w pliku zip. Należy go przenieść do Documents\visual studio 2017\Templates\ProjectTemplates **bez wypakowywania.**
2. Jeżeli znajduje się tam inna wersja template’a, to należy ją usunąć i zrestartować visuala

## Przygotowanie projektu web

1. Edycja appsettings.json

Zmienić connection string w ConnectionStrings -> Default

Zmienić ustawienia JWT(JSON Web Token)

1. Należy wygenerować migracje dla DbContextów komendą *add-migration NAZWAMIGRACJI -context NAZWAKONTEKSTU*
2. Wgrać migracje dla DbContextów *update-database -context NAZWAKONTEKSTU*

## Uruchomienie projektu

W czasie budowania projektu Web, Visual Studio podejmie próbę doinstalowania niezbędnych pakietów. W razie błędów warto spróbować usunąć plik package-lock.json i katalog *node\_modules*  w katalogu ClientApp projektu Web i spróbować ponownie. Gdyby to nie pomogło, należy znów dokonać usunięcia katalogu i użyć komendy *npm install* w *node.js command prompt* z poziomu katalogu projektu Web.

Po uruchomieniou projektu i odczekaniu do 3 minut, najprawdopodobniej pojawi się poniższy widok błędu:

HTTP Error 502.3 - Bad Gateway

The specified CGI application encountered an error and the server terminated the process.

Aplikacja powinna działać poprawnie po odświeżeniu strony.

W razie występowania innych błędów jak np. **Cannot Get /**,**/**, błędy zostaną wypisane w oknie output w visual studio. Ewentualnie można użyć komendy *ng serve* lub *ng build*  w *node.js command prompt* w celu otrzymania informacji o błędach. Dla testów środowiska produkcyjnego uruchamiać z konsoli poleceniem ng build –prod.

## Dodanie encji

Każda encja musi dziedziczyć z BaseEntity.

Każda encja która ma mieć queryService i procesingService musi mieć atrybut Name i atrybut SearchField

Wszystkie relacje muszą się znajdować po obu stronach. Jeżeli gdzieś jest kolekcja to po drugiej stronie musi być encja (do klucza obcego).

Każda encja która ma jako property drugą encje musi mieć również property z kluczem tej encji.

### Atrybuty z generatora

### Atrybuty ogólne

**BridgeTable** – atrybut nadawany klasie która jest tabelą intersekcji

Np.

[BridgeTable]

public class MovieCategory : BaseEntity {}

**Name** – atrybut nadawany klasie oznaczający tekst jaki jest wyświetlany w lookupach. W tekście jest wyszukiwany parametr @@Entity@@(musi być z wielkiej litery) który odnosi się do danego obiektu. Na przykład @@Entity@@.Name spowoduje że w lookupach będą wyświetlane wartości pola Name danych encji. Bardziej skoplikowany przykład wykorzystania atrybutu:

[Name(Query = "@@Entity@@." + nameof(FirstName) + "+\" \"+" + "@@Entity@@." + nameof(LastName))]

Wyświetloną wyświetlone pola FirstName i LastName oddzielone spacją

**ListMetadata** – atrybut nadawany klasie który jako argument przyjmuje inna klasę. W podawanej klasie są zdefiniowane dodatkowe pola które będą wyświetlane na liście z encją. Klasa wskazywana musi być namespace DomainClasses.

Np.

[ListMetadata(typeof(MovieListMetadata))]

public class Movie : BaseEntity {}

public class MovieListMetadata {}

**SearchField** – atrybut nadawany property które ma zostać wykorzystane do wyszukiwania w lookupach.

Np.

[SearchField]

public string Name { get; set; }

**ExpressionField** – atrybut nadawany polu oznaczający że będzie one pobierane na podstawie wprowadzonego zapytania. Zapytanie jest podawane jako string analogicznie jak w NameAttribute

Np. Pobranie pierwszej kategorii dla każdego filmu

public class MovieListMetadata

{

[SearchField]

[ExpressionField("@@Entity@@.MovieCategories.FirstOrDefault().Category.Name")]

public string CateogryName { get; set; }

}

**MasterDetailEditor** – atrybut nadawany property których typem jest kolekcja innych encji (details). Spowoduje on że na formularzu edycji danej encji znajdzie się tabelka z edycją encji z kolekcji (details).

public class Document : BaseEntity

{

[MasterDetailEditor]

public ICollection<DocumentRow> DocumentRows { get; set; } = new HashSet<Cast>();

}

**Dokument z wpisami**

**MasterEntity** – atrybut nadawany property którego typem jest inna encja. Jest to uzupełnienie MasterDetailEditorAttribute oznacza on encje która jest właścicielem/rodzicem (master) aktualnej encji. Zawsze po jednej stronie powinien być MasterDetailEditorAttribute a po drugiej MasterEntityAttribute.

public class DocumentRow : BaseEntity

{

[MasterEntity]

public Document Document { get; set; }

[ForeignKey(nameof(Document))]

public Guid DocumentId { get; set;

}

**ListFilter** – atrybut który generuje filtr dla danego pola do tabeli.

Np.

[ListFilter]

public string Name { get; set; }

**ListIgnore** – atrybut oznacza pole które nie będzie się pojawiało na liście ani w modelach od listy.

Np.

[ListIgnore]

public string Description { get; set; }

**FormIgnore** – atrybut oznacza pole które nie będzie się pojawiało na formularzu ani w modelach od formularza.

Np.

[FormIgnore]

public string PasswordHash { get; set; }

### Atrybuty do rejestrów agregacji

Większy opis i przykłady użycia są w rozdziale o agregacji.

**AggregateTableAttribute** – atrybut oznacza encje która jest rejestrem agregacji z wymiarami. Parmametry atrybutu:

Turnover – wskazuję encję która przechowuje operacje na danym rejestrze (Atrybut AggregateTurnoverTableAttribute)

Balance – wskazuje klasę która zawiera jedynie wymiary i pole balance. Patrz rozdział o agregacji.

Precision – precision w polu typu decimal, domyślnie 18

Scale – scale w polu typu decimal, domyślnie 4

**BalanceAttribute** – atrybut oznacza pole które jest aktualnym stanem w danym rejestrze agregacji. Powinien być tylko w klasach oznaczonych adnotacją AggregateTableAttribute.

**DimentionAttribute** – atrybut oznaczający wymiar w rejestrze agregacji. Powinien być tylko w klasach oznaczonych adnotacją AggregateTableAttribute.

**AggregateTurnoverTableAttribute** – atrybut oznacza encje która zawiera operacje wykonane na konkretnym rejestrze agregacji. Parametry atrybutu:

AggregateKeyPropName – wskazuje na klucz obcy do tabeli z agregacjami(encji z atrbyutem AggregateTableAttribute)

**ExpenseBalanceAttribute** – atrybut oznaczający pole rozchodu na tabeli z operacjami w rejestrze agregacji. Powinien być tylko w klasach oznaczonych adnotacją AggregateTurnoverTableAttribute.

**ProfitBalanceAttribute** – atrybut oznaczający pole przychodu na tabeli z operacjami w rejestrze agregacji. Powinien być tylko w klasach oznaczonych adnotacją AggregateTurnoverTableAttribute.

**RecordAttribute** – atrybut oznaczający pole z rekordem powiązanym z daną operacją w rejestrze agregacji. Powinien być tylko w klasach oznaczonych adnotacją AggregateTurnoverTableAttribute.

**TurnoverDateAttribute** – atrybut oznaczający pole z datą operacji w rejestrze agregacji. Powinien być tylko w klasach oznaczonych adnotacją AggregateTurnoverTableAttribute.

**ValueAttribute** – atrybut oznaczający pole z wartością operacji operacjami w rejestrze agregacji. Powinien być tylko w klasach oznaczonych adnotacją AggregateTurnoverTableAttribute.

### Inne wykorzystywane atrybuty

**Required** – dodaje że pole ma być wymagane i jako wiadomość błędu wyświetla zawartość ErrorMessage

**MaxLength** – dodaje walidacje dotyczącą długości i jako wiadomość błędu wyświetla zawartość ErrorMessage

**Display** – Jako nazwe pola wyświetla wartość property Name.

**Description** – Jako nazwę dla wartości enum.

### Przykład

Aplikacja do edycji bazy o filmach. Każdy film może mieć wiele kategorii i z poziomu filmu powinna być opcja edycji jego obsady.

Klasa film, zawiera enum z kategorią wiekową, kolekcje z intersekcją z kategoriami oraz kolekcje z obsadą. Obsada jest oznaczona atrybutem MasterDetailEditor żeby dało się ją edytować z poziomu filmu. Ponieważ pole Description nie ma ustawionej maksymalnej długości to będzie wyświetlane jako textarea. Klasa jest oznaczona atybutem ListMetadata który wskazuje definicje dodatkowych pól które mają się pojawić na liście.

[Name("@@Entity@@.Name")]

[ListMetadata(typeof(MovieListMetadata))]

public class Movie : BaseEntity

{

[Display(Name = "Nazwa")]

[MaxLength(64, ErrorMessage = "Maksymalna długość nazwy to 64 znaków")]

[Required(ErrorMessage = "Nazwa jest wymagana")]

[SearchField]

[ListFilter]

public string Name { get; set; }

[Display(Name = "Długośc")]

public int Length { get; set; }

[Display(Name = "Opis")]

[ListIgnore]

[SearchField]

public string Description { get; set; }

[ListFilter]

public AgeRating AgeRating { get; set; }

public ICollection<MovieCategory> MovieCategories { get; set; } = new HashSet<MovieCategory>();

[MasterDetailEditor]

public ICollection<Cast> Cast { get; set; } = new HashSet<Cast>();

}

Klasa z dodatkowym polem które będzie wyświetlane na liście – nazwa pierwszej kategorii danego filmu.

public class MovieListMetadata

{

[SearchField]

[ExpressionField("@@Entity@@.MovieCategories.FirstOrDefault().Category.Name")]

public string CateogryName { get; set; }

}

Enum z kategorią wiekową, każda wartość ma atrybut z nazwą żeby było z czego wygenerować tekst do lookupów.

public enum AgeRating

{

[Description("General Audience")]

G,

[Description("Parental Guidance Suggested")]

PG,

[Description("Parents Strongly Cautioned")]

PG13,

[Description("Restricted")]

R,

[Description("Adults Only")]

NC17

}

Kategoria filmu zawierająca kolekcje intersekcji z filmami.

[Name("@@Entity@@.Name")]

public class Category : BaseEntity

{

[Display(Name = "Nazwa")]

[MaxLength(64, ErrorMessage = "Maksymalna długość nazwy to 64 znaków")]

[Required(ErrorMessage = "Nazwa jest wymagana")]

[SearchField]

public string Name { get; set; }

public ICollection<MovieCategory> CategoryMovies { get; set; } = new HashSet<MovieCategory>();

}

Klasa będąca intersekcją filmu i kategorii. Posiada atrybut BridgeTable

[BridgeTable]

public class MovieCategory : BaseEntity

{

public Movie Movie { get; set; }

[ForeignKey(nameof(Movie))]

public Guid MovieId { get; set; }

public Category Category { get; set; }

[ForeignKey(nameof(Category))]

public Guid CategoryId { get; set; }

}

Klasa z aktorem

[Name(Query = "@@Entity@@." + nameof(FirstName) + "+\" \"+" + "@@Entity@@." + nameof(LastName))]

public class Actor : BaseEntity

{

[Display( Name = "Imie")]

[MaxLength(64, ErrorMessage = "Maksymalna długość imienia to 64 znaki")]

[Required(ErrorMessage = "Imie jest wymagane")]

[SearchField]

public string FirstName { get; set; }

[Display( Name = "Nazwisko")]

[MaxLength(64, ErrorMessage = "Maksymalna długość nazwiska to 64 znaki")]

[Required(ErrorMessage = "Nazwisko jest wymagane")]

[SearchField]

public string LastName { get; set; }

[Display( Name = "Data urodzin")]

public DateTime DateOfBirth { get; set; }

}

Klasa z obsadą, pole z filmem jest oznaczone jako nadrzędne (master) ponieważ z jego poziomu można edytować obsadę.

[Name(Query = "@@Entity@@." + nameof(Name))]

public class Cast : BaseEntity

{

[Display(Name = "Nazwa postaci")]

[MaxLength(64, ErrorMessage = "Maksymalna długość nazwy postaci to 64 znaków")]

[Required(ErrorMessage = "Nazwa postaci jest wymagana")]

[SearchField]

public string Name { get; set; }

public Actor Actor { get; set; }

[Display(Name = "Aktor")]

[ForeignKey(nameof(Actor))]

public Guid ActorId { get; set; }

[MasterEntity]

public Movie Movie { get; set; }

[ForeignKey(nameof(Movie))]

public Guid MovieId { get; set; }

}

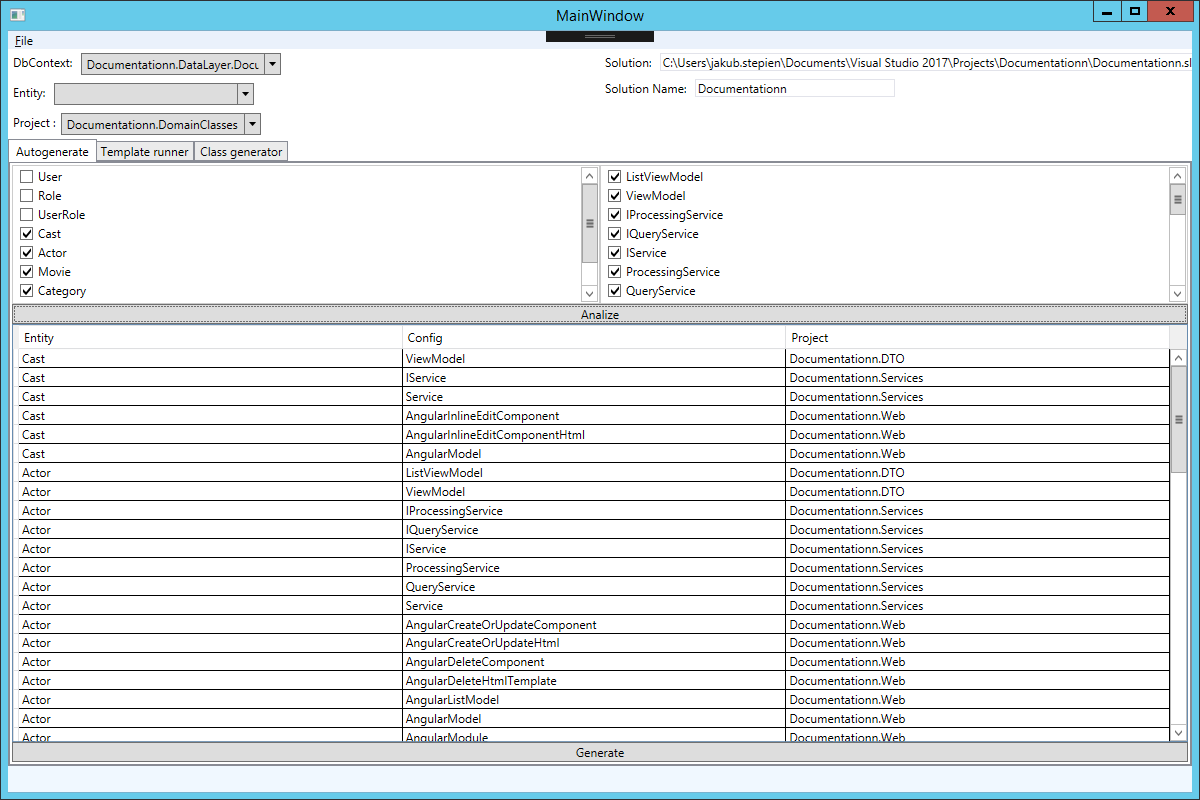
## Dodanie i wgranie migracji

Dodać wszystkie encje do DbContextu.

Z Package Manager Console wybrać jako projekt **DataLayer** i wpisać add-migration NAZWA-MIGRACJI, po wygenerowniu migracji można ją edytować a później wgrać przez polecenie update-database

## Automatyczne generowanie aplikacji

Uruchomić CodeGenerator, z menu wybrać file i OpenSolution. Wskazać plik z solucją. Należy chwilę poczekać aż pojawi się okno z wybraniem kontesktu, wybrać kontekst i wcisnąć Open. Upewnić się że w comboxie **Project** jest wybrany projekt z klasami bazowymi.. W zakładce autogenerate w pozostawić zaznaczone wszystkie encje poza User, Role i UserRole i wszystkie configi. Następnie kliknąć Analize pojawi się lista elementów do wygenerowania. Następnie wcisnąć przycisk Generate by wygenerować wszystko



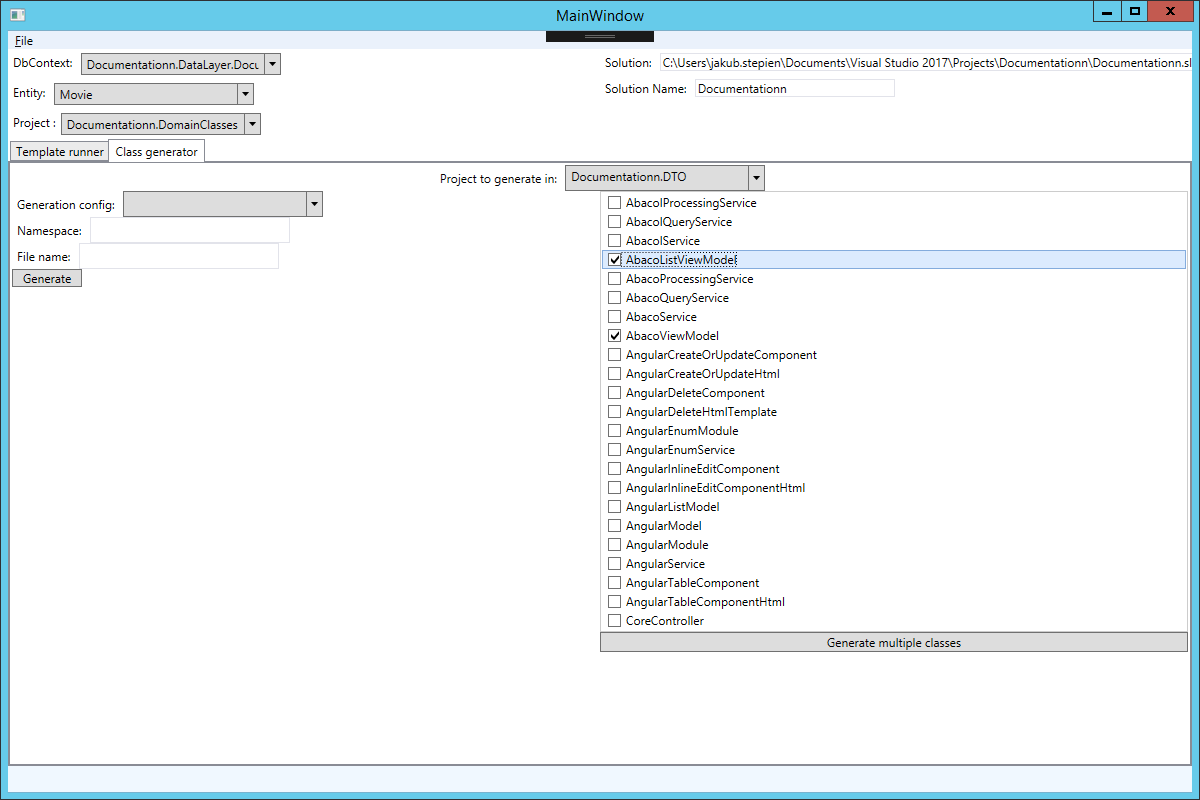
## Generowanie aplikacji samemu

Uruchomić CodeGenerator, z menu wybrać file i OpenSolution. Wskazać plik z solucją. Po wczytaniu z solucją pojawi się okno z wybraniem kontesktu, wybrać kontekst i wcisnąć Open. Następnie w polu entity powinny się pojawić do wyboru encje i enumy w aplikacji.

### Generowanie dla zwykłej, podstawowej encji (nie będącej enumem, intersekcją ani edytowaną przez inną encję):

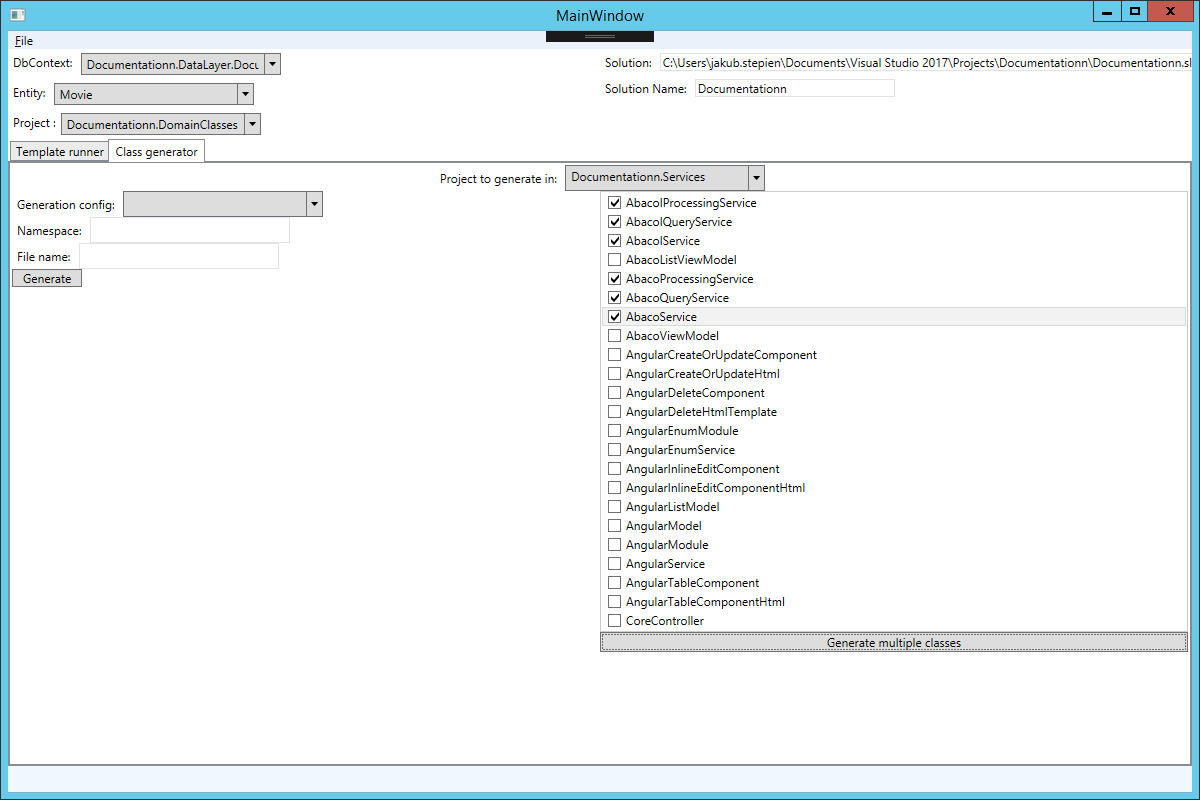
Wybieramy encje z comboxa, autoatycznie „**Project”** powinien się zmienić na **DomainClasses**. Jeżeli encja rzeczywiście deklarowana w danym projekcie to zostawiamy. W przeciwnym razie zmieniamy na projekt w którym była ona deklarowana.

Następnie przełączamy zakładkę na **Class generator**. Z dropdown „**Project to generate in**” wybieramy projekt **DTO**. Z listy wybieramy **AbacoViewModel** i **AbacoListViewModel**.

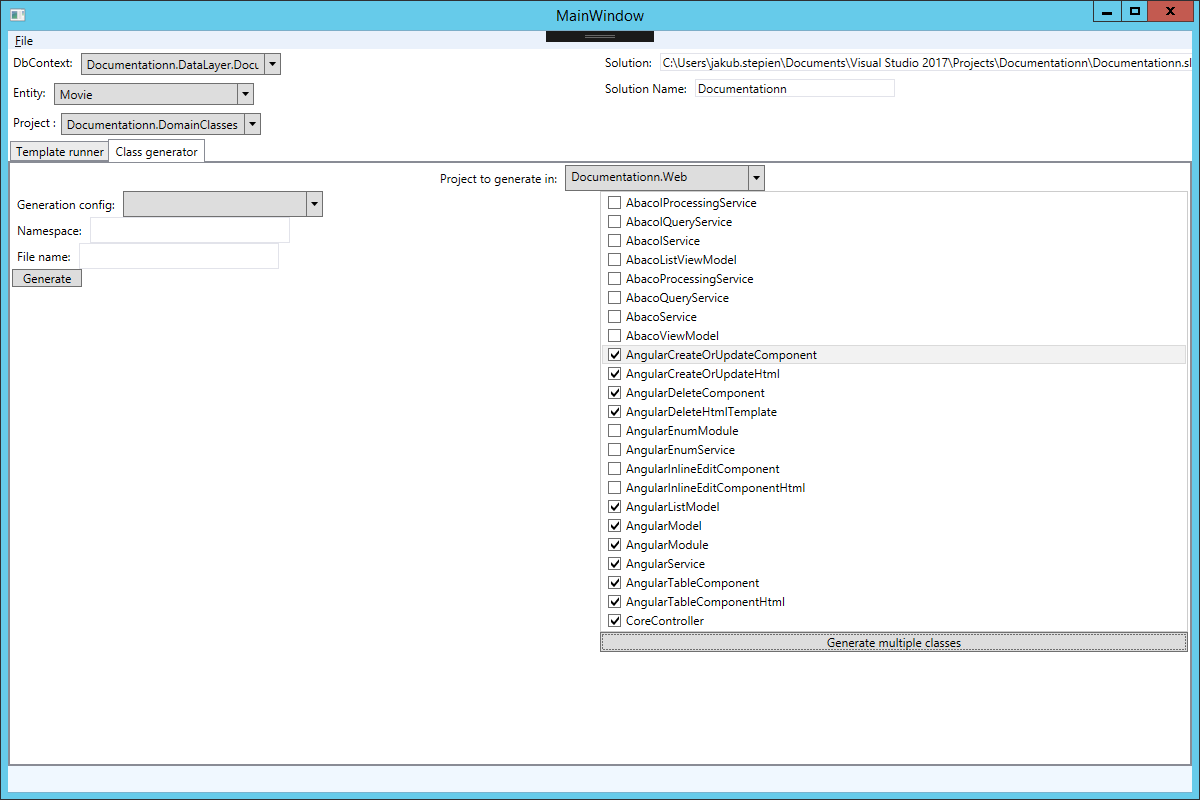


Klikamy „**gnerate multiple classes”**.W dolnym panelu pojawi się komunikat o generowaniu klas. Kiedy komunikat zniknie można generować kolejne elementy.

Zmieniamy „**Project to generate in**” na **Services** i zaznaczamy implementacje i interfejsy do serwisów **QueryService**, **ProcesingService**, **Service**. I znowu klikamy „**generate multiple classes”**.



Następnie zmieniamy projekt na **WEB** i wbieramy **CoreController** orazwszystko z angulara, poza elementami dotyczącymi ~~enumów~~ i ~~InlineEdit~~. I znowu generujemy.

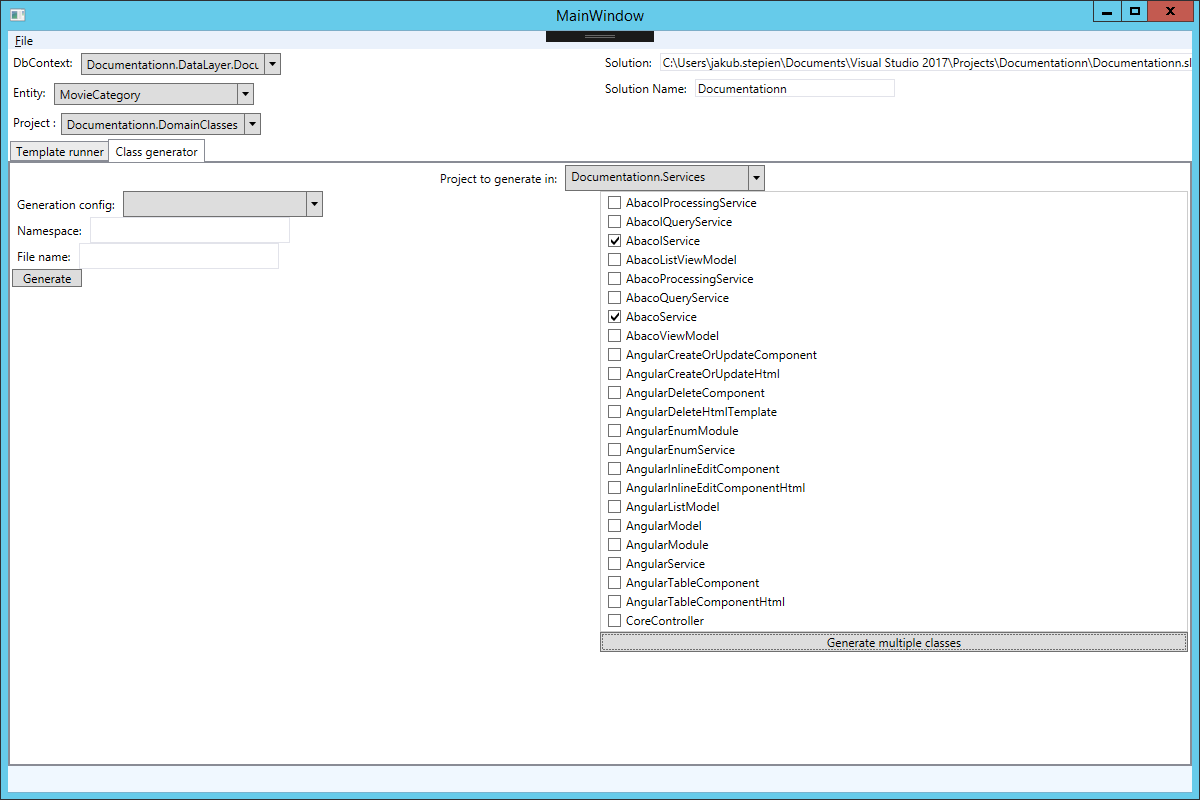


Po wygenerowaniu pojawi się informacja że wygenerowane elementy mogły nie zostać dodane do solucji. Należy wcisnąć ok i sprawdzić czy wygenerowane elementyangulara zostały dodane do solucji (powinny być).

Powyżej opisane kroki generacji nie muszą być wykonywane po kolei. Jeżeli mamy wiele encji to najprościej jest wygenerować **DTO** dla zdla wielu encji tego samego „typu” po kolei. Po wygenerowaniu wszystkich **DTO** zmienić projekt i checkboxy na serwisy i znowu po kolej przejść po wszystkich encjach. Ostatecznie zrobić to samo dla projektu **Web**

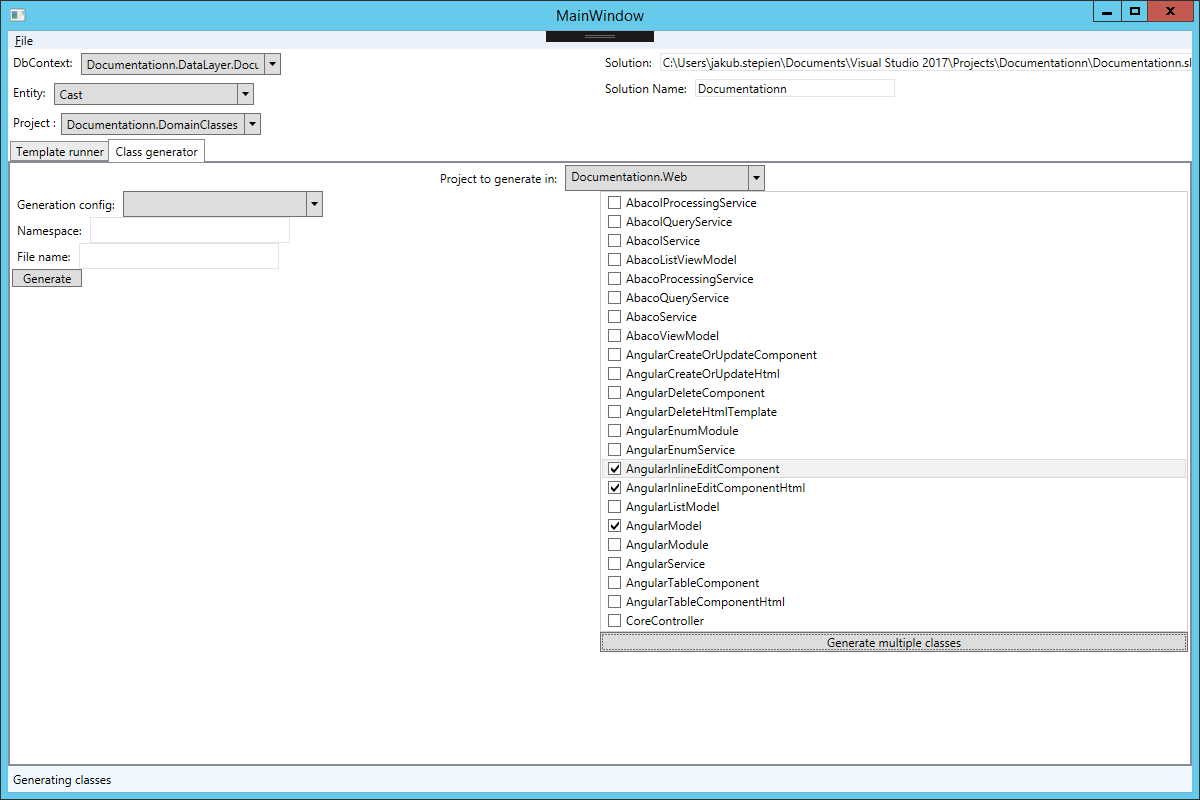
### Generowanie dla encji które są tabelą intersekcji

Generowanie przebiega analogicznie jak dla zwykłych encji tylko w tym przypadku generowane są jedynie podstawowe serwisy (elementy **Servie** i **IService**), bez DTO i bez obsługi na web.



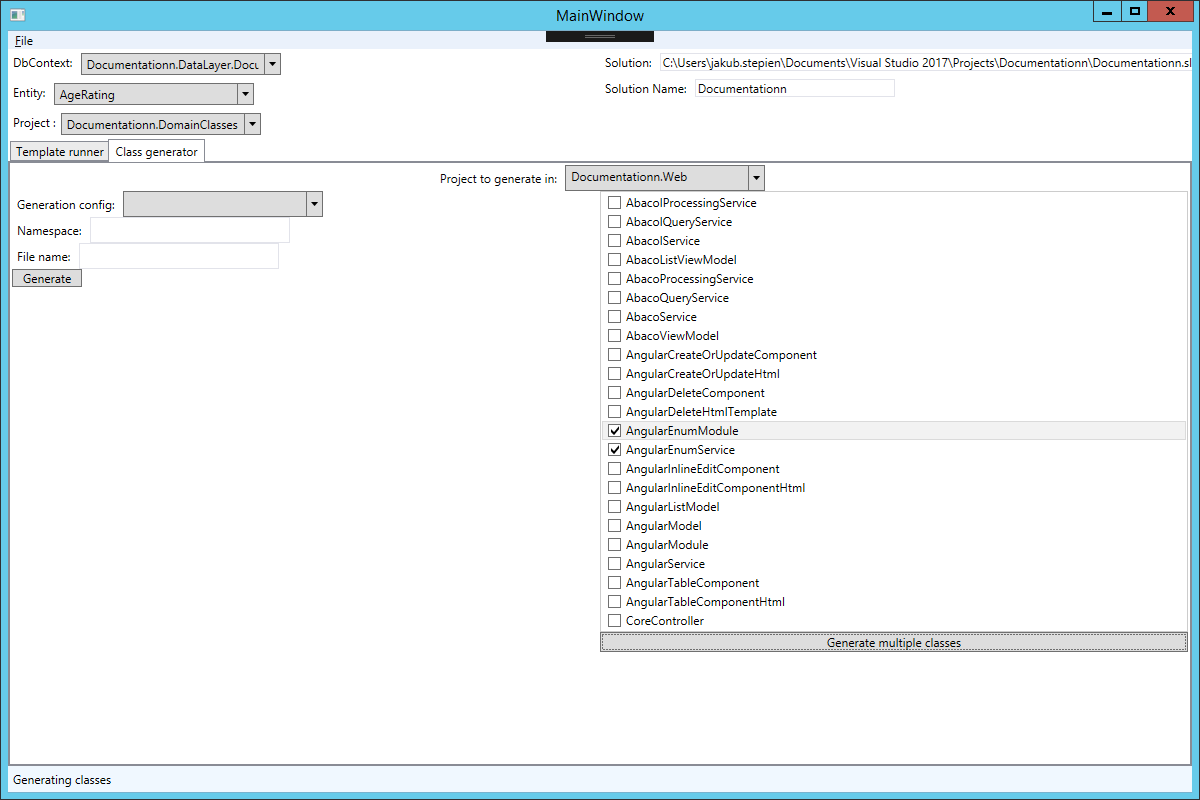
### Generowanie dla encji która jest edytowana przez inna encje

Dla tej encji z dto trzeba wygenerować jedynie zwykły **ViewModel**. Z serwisów jedynie podstawowe **Service** i **IService**. W projekcie **Web** wyłącznie **AngularModel** i **InlineEditComponent** i **InlineEditComponentHtml**.



### Generowanie dla enumów

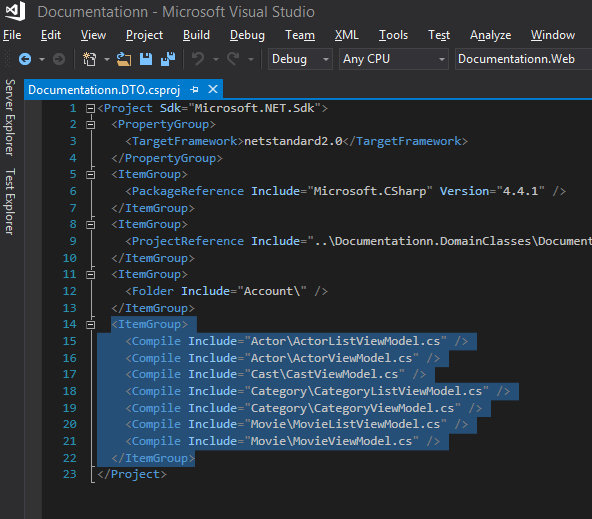
Dla enumów generowany jest jedynie **AngularEnumModule** i **AngularEnumService** w projekcie **Web**.



## Zmiany w projekcie po generacji

### Projekty C#

Po wygenerowaniu klas c# zostają one automatycznie dodane do pliku csproj projektu przez co nie chce się on kompilować. Trzeba dla każdego projektu w którym generowaliśmy otworzyć plik csproj i usunąć wygenerowane elementy



### Angular

Dla encji dla których generowaliśmy moduly należy dodać je do routingu aplikacji. W Web/ClientApp/src/app/app.routing.ts dopisać do tablicy children:

{path: ‘’, loadChildern: ‘ŚCIEŻKA-DOMODUŁU#NAZWA-KLASY’}

import { Routes } from '@angular/router'

import { AdminLayoutComponent } from './layouts/admin/admin-layout.component';

import { AuthGuard } from './authentication/guards/auth-guard';

export const appRoutes : Routes = [

{

path: "",

component: AdminLayoutComponent,

canActivate: [AuthGuard],

children: [

{path: '', loadChildren: "./role/role.module#RoleModule"},

{path: '', loadChildren: "./user/user.module#UserModule"},

{path: '', loadChildren: "./movie/movie.module#MovieModule"},

{path: '', loadChildren: "./actor/actor.module#ActorModule"},

{path: '', loadChildren: "./category/category.module#CategoryModule"},

]

}

]

Następnie trzeba je dodać do linków z menu. W Web/ClientApp/src/app/common/menu-tiems/menu-items.ts.

const MENUITEMS = [

{

state: "user",

name: "User",

type: "link",

icon: 'basic-sheet-txt'

},

{

state: "role",

name: "Role",

type: "link",

icon: 'basic-sheet-txt'

} ,

{

state: "actor",

name: "Actor",

type: "link",

icon: 'basic-sheet-txt'

} ,{

state: "movie",

name: "Movie",

type: "link",

icon: 'basic-sheet-txt'

} ,{

state: "category",

name: "Category",

type: "link",

icon: 'basic-sheet-txt'

}

];

Gdzie state to domyślny route a name to tekst w menu.

Wszystkie angularowe serwisy (poza enumami), które zostały wygenerowane, należy dodać do Web/ClientApp/src/app/shared.module.ts i zaimportować je na górze pliku, oraz dodać do tablicy providers w obiekcie zwracanym przez statyczne forRoot().

import { NgModule } from "@angular/core";

import { RoleService } from "./role/role.service";

import { UserService } from './user/user.service'

import { ActorService } from './actor/actor.service';

import { MovieService } from './movie/movie.service';

import { CategoryService } from './category/category.service';

@NgModule({

})

export class SharedModule {

static forRoot() {

return {

ngModule: SharedModule,

providers: [UserService, RoleService,ActorService,CategoryService, MovieService ]

};

}

}

Dla wszystkich wygenerowanych enumów, trzeba je zaimportować i dodać do imports w Web/ClientApp/src/app/app.module.ts @NgModule -> imports

...

import { AgeRatingModule} from './ageRating/ageRating.module';

export function createTranslateLoader(http: HttpClient) {

return new TranslateHttpLoader(http, './assets/i18n/', '.json');

}

@NgModule({

declarations: [

AppComponent,

AdminLayoutComponent

],

imports: [

BrowserModule,

CommonAppModule,

RouterModule.forRoot(appRoutes),

BrowserAnimationsModule,

HttpClientModule,

NgbModule.forRoot(),

SharedModule.forRoot(),

AuthenticationModule.forRoot(),

TranslateModule.forRoot({

loader: {

provide: TranslateLoader,

useFactory: (createTranslateLoader),

deps: [HttpClient]

}

}),

SidebarModule.forRoot(),

AgeRatingModule,

],

providers: [DatePipe,

{ provide: MAT\_DATE\_LOCALE, useValue: 'pl-PL' },

{ provide: DateAdapter, useClass: LocalDateAdapter, deps: [MAT\_DATE\_LOCALE] },

{ provide: MAT\_DATE\_FORMATS, useValue: MAT\_MOMENT\_DATE\_FORMATS },

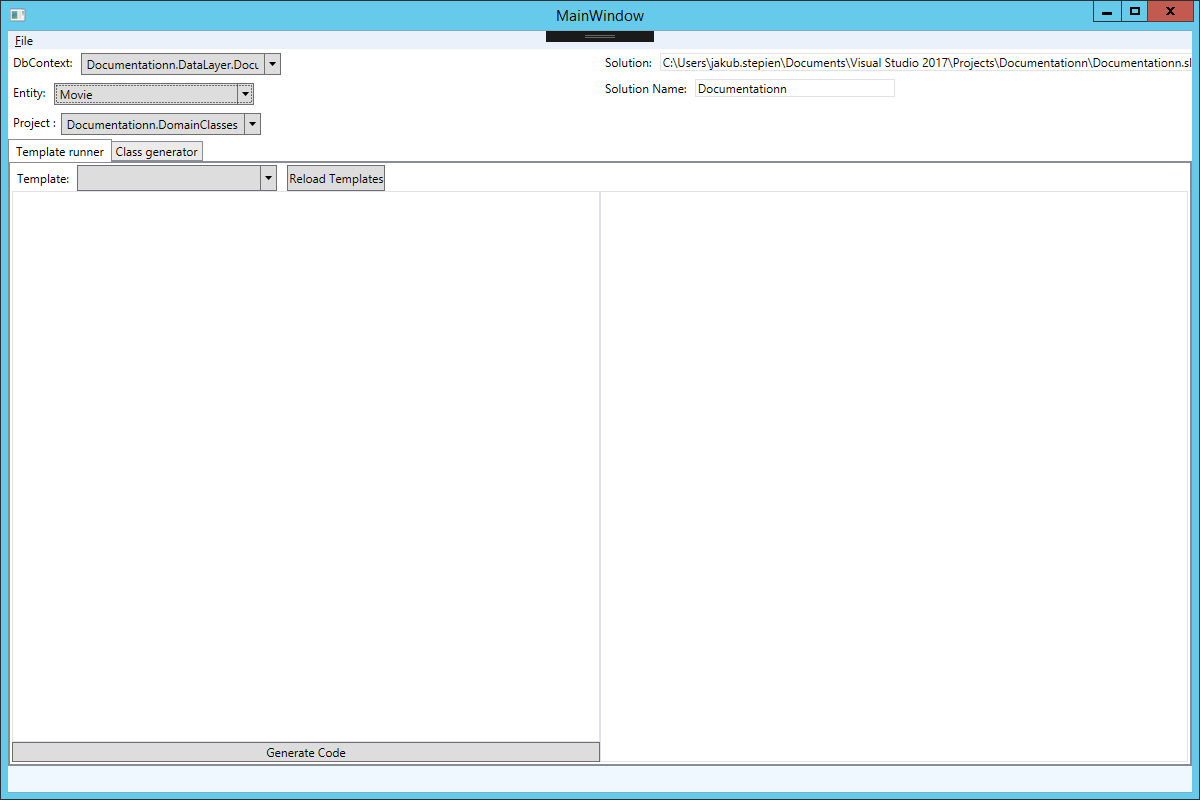
],

bootstrap: [AppComponent]

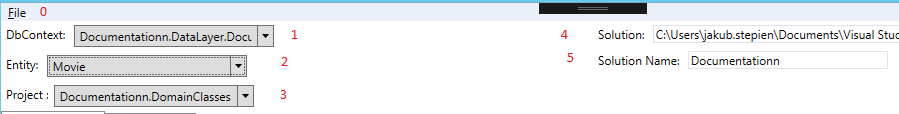
})

export class AppModule { }

# Opis generatora



## Header



0: Menu File z jedną opcją do otwarcia solucji. Po wybraniu solucji aplikacja próbuje znaleźć wszystkie DbContexty. Jeżeli się nie uda to wyświetla okno z wyborem DbContextu

1: Dropdown z wyborem DbContextu

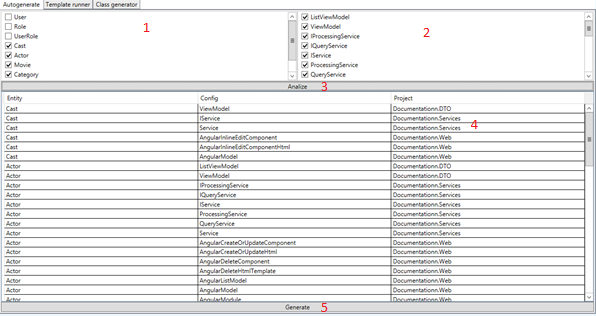
2: Aktualnie wybrana encja lub enum

3: Projekt w którym zadeklarowana jest encja lub enum

4: Ścieżka do solucji jedynie do odczytu

5: Nazwa solucji, domyślnie brana z nazwy pliku. Powinna ona odpowiadać domyślnemu namespace aplikacji czyli solution name + .Services powinno wskazywać domyśln namespace da serwisów i tak samo dla dto, web i domainClasses.

## Autogenerate



1. Lista klas dla których elementy będą generowane

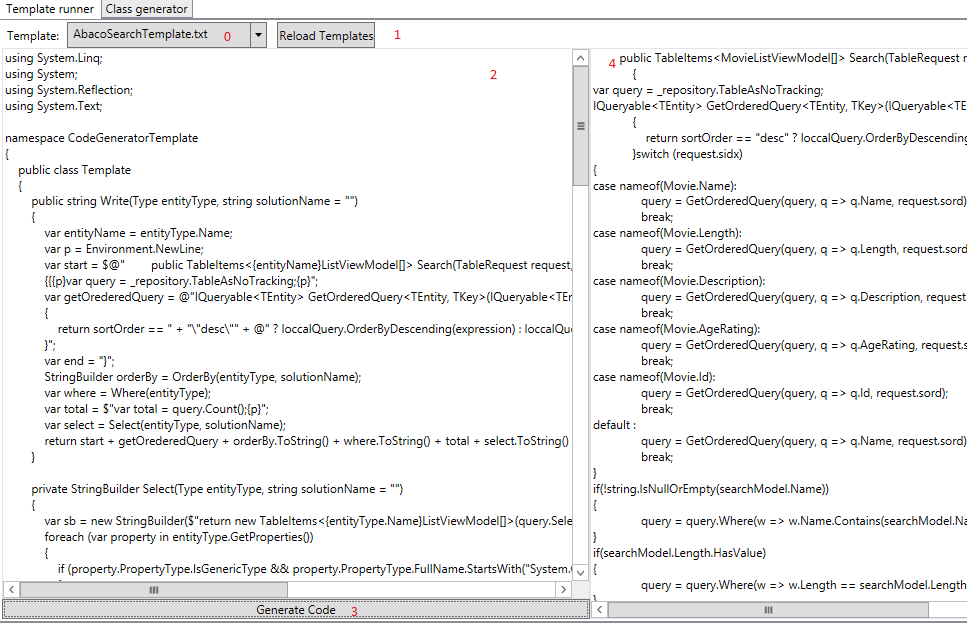
2. Lista configów które mają być brane pod uwagę

3. Przycisk który po kliknięciu określi listę elementów do wygenerowanie

4. Przycisk który po kliknięciu wygeneruje wszystko.

## TemplateRunner

Służy do uruchomienia pojedynczego template bez generowania żadnych klas.



0: Wybór aktywnego template

1: Odświerzenie listy templateów

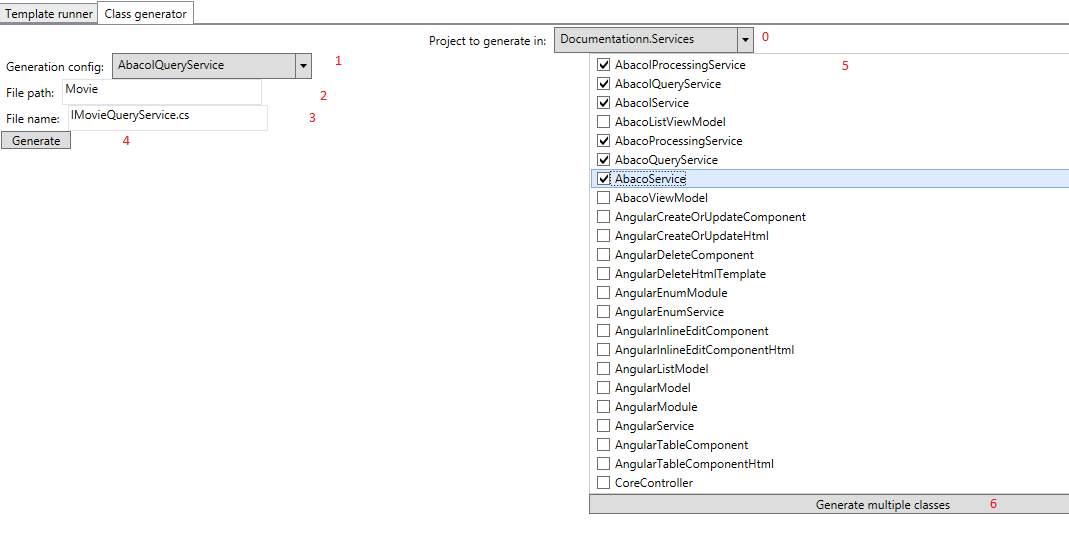
2: Kod aktywnego template, edycja jego nie zmienia template na dysku

3: Przycisk do uruchomienia danego template

4: Otrzymany kod po uruchomieniu template dla encji.

## Class generator

Class generator ma dwie części po lewej stronie jest generowanie pojedynczych klas które pozwala na konfiguracje ścieżki i nazwie pliku. Po prawej stronie jest generowanie wielu klas przy domyślnych ustawieniach.



0: Wybór projektu do którego będą generowane klasy, dotyczy obu części generatora.

1: Wybór konfiguracji

2: Ścieżka wewnątrz projektu gdzie będzie wygenerowany element

3: Nazwa wygenerowanego pliku

4: Przycisk do generowania

5: Klasy do wygenerowania

6: Przycisk do generowania wielu klas.

# Funkcjonalności template’u projektu

## InjectableService

Do dodawania service’ów do dependency injection można wykorzystać atrybut InjectableService który przyjmuje argument z typem pod który ma zostać zarejestrowany. Typ zostaje rejestrowany w trybie scoped.

[InjectableService(typeof(IPasswordService))]

public class MD5PasswordService : IPasswordService {}

## Logger

Do logowania należy korzystać z service’u ILogger w projekcie z serwisami. Kożysta on z innego DbContextu LoggingDbContext który domyślnie korzysta z innego scheme „log”.

Przed uruchomieniem aplikacji należy dodać connection string w pliku appsettings.json w Sekcji ConnectionStrings, może on być taki sam jak Default.

### LoggingLevel

LoggingLevel definiuje stopień od którego zapisywać informacje

// Summary:

// Logs that contain the most detailed messages. These messages may contain sensitive

// application data. These messages are disabled by default and should never be

// enabled in a production environment.

Trace = 0,

//

// Summary:

// Logs that are used for interactive investigation during development. These logs

// should primarily contain information useful for debugging and have no long-term

// value.

Debug = 1,

//

// Summary:

// Logs that track the general flow of the application. These logs should have long-term

// value.

Information = 2,

//

// Summary:

// Logs that highlight an abnormal or unexpected event in the application flow,

// but do not otherwise cause the application execution to stop.

Warning = 3,

//

// Summary:

// Logs that highlight when the current flow of execution is stopped due to a failure.

// These should indicate a failure in the current activity, not an application-wide

// failure.

Error = 4,

//

// Summary:

// Logs that describe an unrecoverable application or system crash, or a catastrophic

// failure that requires immediate attention.

Critical = 5,

Stopień logowania ustala się w appsettings.json w sekcji Logging. Podsekcje LogLevel i IncludeScopes są domyślnymi ustawieniami logowania w asp.core. Podsekcja DbLogger jest konfiguracją DbLoggera z template’a projektu, jest ona deserializowana do obiektu LoggerOptions.

{

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Warning"

},

"DbLogger": {

"LogLevel": "Warning"

}

},

…

}

### ILoggerDataProvider

Implementując interfejs ILoggerDataProvider można dodawać dodatkowe elementy do obiektu data zapisywanego do logu. Ma on jedną metodę GetData która zwraca tablice par klucz – wartość które zostaną dodane do jsona data w logu. Interfejs należy dodać do dependecy injection.

## Emailing

Funkcjonalność emailingu to wysyłanie maili podanych przez programistę lub wczytanych z bazy danych. Wysyłanie maili oparte jest o bibliotekę MailKit.

### DataLayer.MailDbContext

klasa kontekstu dla bazy danych. Domyślny schema to „sys”.

### Web.appsettings.json

W sekcji "ConnectionStrings” znajduje się pole "Emailing": "" , które należy wypełnić odpowiednim connection stringiem.

Domyślnie w appsettings.json znajduje się również sekcja z konfiguracją emailingu.

"EmailingConfig": {

"SMTP": {

"ServerName": "", // Adres serwera (np. smtp.gmail.com)

"Sender": "", // Adres skrzynki z którego wysyłane są wiadomości (np. jan.kowalski@gmail.com)

"Port": "465", // Domyślny port dla SMTP to 465

"User": "", // Nazwa użytkownika służąca do logowania (np. jan.kowalski)

"Password": "", // Hasło używane do zalogowania na skrzynkę

"SSL": false // Flaga określająca czy używane jest SSL.

}

}

Elementy w środku sekcji można dowolnie edytować. Domyślnie znajduje się w nim tylko podsekcja „SMTP”, jednak można zmienić jej nazwę oraz zawartość oraz dodać kolejne. Domyślnie zaimplementowana jest jedynie funkcja wysyłania przez SMTP, jednak można dodać konfiguracje dla różnych serwerów SMTP, czy nawet innych protokołów (IMAP, POP3)

### DomainClasses.Mail

Plik zawierający model danych do bazy danych I niezbędne element.

public enum MailStatus // Enum ze statusem.

{

//Waiting for set time

Pending, // Maile oczekujące na ustawiony czas wysłania

//Will be sent ASAP

Queued, // Maile do wysłania w najbliższym czasie

//Already sent

Sent, // Wysłane

//Something has gone wrong

Failed, // Błąd wysyłania

}

public enum MailPriority // Priorytet wiadomości.

{

Critical,

Important,

Preferred,

Normal

}

public class Mail : BaseEntity

{

[Required]

public MailPriority Priority { get; set; }

public MailStatus Status { get; set; }

[Required]

public DateTime CreatedOn { get; set; }

public DateTime ToBeSentOn { get; set; }

public DateTime SentOn { get; set; }

//Multiple adresses possible. Use ; to divide them

[Required]

public string From { get; set; }

//Multiple adresses possible. Use ; to divide them

[Required]

public string Receivers { get; set; }

//Multiple adresses possible. Use ; to divide them

public string CarbonCopyReceivers { get; set; }

//Multiple adresses possible. Use ; to divide them

public string BlindCarponCopyReceivers { get; set; }

[Required]

[MaxLength(78)] //Safe limit for single-line subject - according to RFC 2822

public string Subject { get; set; }

[Required]

public string Content { get; set; }

}

### DTO.EmailMessageModel

DTO dla wiadomości mailowych.

public class EmailMessageModel

{

public string SenderAddress { get; set; } // Adres wysyłającego (skrzynki/serwera)

public List<EmailAdress> From { get; set; } // Adres i nazwa nadawcy

public List<EmailAdress> Recipients { get; set; } // Adresy i nazwy odbiorcw

public List<EmailAdress> Cc { get; set; } // Adresy i nazwy DW

public List<EmailAdress> Bcc { get; set; } // Adresy i nazwy ukrytych DW

public string Subject { get; set; } // Temat

public string Content { get; set; } // Zawartość w postaci tekstowej

}

W tym samym pliku znajduje się również klasa pomocnicza:

public class EmailAdress

{

public string Name { get; set; }

public string Address { get; set; }

}

W przypadku użycia gmail nazwa nadawcy lub odbiorcy będzie w postaci np. **Twitter** <[info@twitter.com](mailto:info@twitter.com)> , gdzie „Twitter” odpowiada polu Name, a adres w nawiasach trójkątnych polu Address.

### EmailingService

Klasa ta zawiera podstawowy CRUD niezbędny do korzystania z bazy danych w przypadku kolejkowania maili.

### EmailingConfig

Klasa mapująca config znajdujący się w appsettings.json. Struktura i nazewnictwo w tej klasie musi dokładnie odpowiadać strukturze sekcji EmailingConfig w appsettings.json.

### EmailingConfigResolver

Klasa pomocnicza do wczytywania odpowiedniej konfiguracji (subsekcji).

### EmailSender

Klasa służąca do wysyłania maili. Obecnie posiada 2 metody Send przyjmujące parametr w postaci modelu bazy danych lub DTO. Obecnie zmiana statusu maila należy do obowiązków metody wywołującej. Klasa posiada atrybut [InjectableService(typeof(IEmailSender))] co oznacza że zostaje wstrzykiwana (Dependency Injection) przy uruchomieniu aplikacji. Wystarczy dodać do konstruktora wybranej klasy parametr typu IEmailSender.

### Przykład użycia

W konstruktorze klasy należy podać odpowiedni parametr IEmailSender emailSender

Wysłanie maila może odbywać się poprzez stworzenie wiadomości w kodzie. Przykładowe utworzenie i wykorzystanie wiadomości w kodzie:

var mail = new EmailMessageModel

{

Content = "Doggy doggy whaaaaaaaaat?",

Subject = "Bo for GoJack!",

SenderAddress = "jan.kowalski@gmail.pl",

From = new List<EmailAdress>()

{

new EmailAdress

{

Address = "jan.kowalski@gmail.com",

Name = "Janek"

}

},

Recipients = new List<EmailAdress>()

{

new EmailAdress

{

Address = "grazyna.kowalska@gmail.com",

Name = "Grazynka"

}

},

};

\_emailSender.Send(MailConfigType.SMTP, mail);

## Notyfikacje

### INotificationService

Podczas requestu można w dowolnym miejscu wstrzyknąć INotificationService który pozwala na dodanie do zwracanej odpowiedzi powiadomienia które zostaną automatycznie wyświetlone po stronie angulara. Stopnie powiadomień to: Success, Info, Warning, Alert, Error.

Opcje dotyczące wyświetlania powiadomień w angularze można konfigurować w Web/ClientApp/src/app/common/notifications/notification.ts w polu options.

### ApiResult

Żeby notyfikacje zostały dodane do odpowiedzi z serwera należy zwracać obiekty pakując je w klasę ApiResult w kontrolerze.

Np.

[HttpGet("{id}")]

public async Task<IActionResult> GetById(Guid id)

{

var model = await \_userQueryService.PrepareViewModelAsync(id);

return new ApiResult(model);

}

ApiResult opakowuję każdą odpowiedź w:

public class ApiResultJson

{

public object Result { get; set; }

public bool Success { get; set; }

public Notification[] Notifications { get; set; }

}

ApiResult ma statyczne metody:

* Empty która zwraca success bez danych
* Error która automatycznie zwraca błędną odpowiedź z opcjonalnym powiadomieniem o błędzie.

Np.

[HttpDelete("{id}")]

public async Task<IActionResult> Delete(Guid id)

{

try

{

await \_userProcessingService.DeleteAsync(id);

return ApiResult.Empty;

}

catch (Exception e)

{

\_logger.Error("Błąd usunięcia użytkownika.", e, id);

return ApiResult.Error("Wystąpił błąd usunięcia użytkownika");

}

}

## Eventy

### IEventPublisher

Zdarzenia można publikować przy pomocy interfejsu IEventPublisher, ma on tylko jedną metodę która rozsyła zdarzenie do odbiorców i zwraca czy zdarzenie nie zostało przerwane. Zdarzenie musi implementować interfejs IEvent, dodana jest również generyczna implementacja tego interfejsu Event<T>. Obsługa zdarzeń jest realizowane przez klasy implementujące IEventSubscriber<TEvent> where TEvent : IEvent. Te klasy **są automatycznie rejestrowane do dependecy injection**, dodanie samemu do DI, ręcznie lub przez atrybut spowoduję dodanie dwa raz. Subskrybenci są informowani w kolejności na podstawie property Order. Jeżeli IsCancelled na zdarzeniu będzie true to przekazywanie eventu zostaje przerwane.

### Rodzaje eventów

W template projektu jest 12 generycznych eventów które są wykorzystywane w wygenerowanych serwisach. Te eventy to:

BeforeEntityInsert – Przesyła viewmodel przed insertem

BeforeEntityInsertTransaction – Przesyła encje w rozpoczętej transakcji dodania

AfterEntityInsert – Przesyła encje po zakończeniu transakcji

AfterEntityInsertTransaction – Przesyła encje po zapisaniu jej w transakcji ale przed commitem transakcji.

BeforeEntityUpdate – Przesyła viewmodel przed update

BeforeEntityUpdateTransaction – Przesyła encje w rozpoczętej transakcji dodania

AfterEntityUpdate – Przesyła encje po zakończeniu transakcji

AfterEntityUpdateTransaction – Przesyła encje po zapisaniu jej w transakcji ale przed commitem transakcji.

AfterEntityDelete – Przesyła encje encji do usunięcia

BeforeEntityDelete – Przesyła encje po usunięciu

### Przykład

Walidacja filmu gdzie odrzucamy filmy które mają mniej niż 30 minut. W obsłudze eventu można normalnie korzystać z dependency injection. W ProcessingService IsCancelled zdarzenia jest sprawdzane i jeżeli zostanie ono ustawione na true to operacja zostaje przerwana.

public class MovieLengthValidator : IEventSubscriber<BeforeEntityInsert<MovieViewModel>>

{

INotificationService notificationService;

public MovieLengthValidator(INotificationService notificationService)

{

this.notificationService = notificationService;

}

public int Order => 0;

public void HandleEvent(BeforeEntityInsert<MovieViewModel> eventMessage)

{

var movie = eventMessage.Entity;

if(movie.Length < 30)

{

notificationService.Error("Film musi mieć conajmniej 30 minut");

eventMessage.IsCancelled = true;

}

}

}

## ScheduledTask

### IScheduledTask

W solucji można definiować zadania które będą wykonywane co określony przedział czasowy. Do tego należy stworzyć klasę implementującą interfejs IScheduledTask. Posiada on dwa property:

IntervalBetweenRuns - określa co ile sekund zadanie powinno być wykonywane

ErrorThreshold - określa po ilu błędnych wykonanych przestać ponownie wykonywać zadanie

Metoda RunAsync jest automatycznie wykonywana co określony czas i zwraca czy zadnie zostało wykonane poprawnie. Chociaż jest ona oznaczona jako async to może wykonywać się synchronicznie tylko wtedy trzeba zwrócić wartość przez Task.FromResult(true).

Każda implementacja IScheduledTask jest wykonywana w oddzielnym tasku i specjalnie do każdego uruchomienia jest tworzony nowy scope od dependecy injection. Z tym scopem nie jest powiązany żaden request dlatego wstrzykiwanie elementów takich jak IActionContextAccessor nie będzie działać. Wstrzykiwanie innych elementów jest takie samo jak w serwisach.

**Klasy implementujące** IScheduledTask **są automatycznie rejestrowane w dependecy injection.**

W przypadku debugowania takiego tasku warto zajrzeć do klasy generycznej TaskScheduler<T> która odpowiada za uruchamianie zadań.

### Przykład

Task który co godzinę ma zapisać do pliku aktora który wystąpił w największej liczbie filmów

public class MostPopularActorSavingTask : IScheduledTask

{

IRepository<Cast> repository;

public MostPopularActorSavingTask(IRepository<Cast> repository)

{

this.repository = repository;

}

public int IntervalBetweenRuns => 3600;

public int ErrorThreshold => 3;

public async Task<bool> RunAsync(CancellationToken cancellationToken)

{

try

{

var actor = await repository.TableAsNoTracking

.GroupBy(d => d.Actor)

.Select(s => new { Actor = s.Key, Roles = s.Count() })

.OrderByDescending(o => o.Roles)

.FirstOrDefaultAsync();

var path = AppContext.BaseDirectory + "task-example.txt.";

File.WriteAllText(path, $"{actor.Actor.FirstName} {actor.Actor.LastName}: {actor.Roles}");

return true;

}

catch (Exception e)

{

return false;

}

}

}

## Rejestry agregacji

Aplikacja może wygenerować repozytorium do tabeli agregującej przychody i rozchody według podanych wymiarów. Potrzebne do tego są trzy klasy jedna z wymiarami, jedna z operacjami i jedna pomocnicza do zapytań. Klasa z wymiarami musi być oznaczona atrybutem AggregateTableAttribute i jako argumenty musi mieć podane typy dwóch pozostałych klas. Dodatkowo można w tym atrybucie zdefiniować dokładność pola z wartościami, uwaga: to jest wykorzystywane jedynie do zapytań dokładność pola na bazie trzeba samemu ustawić. W samej encji należy oznaczyć pola z wymiarami przy użyciu atrybutu Dimention oraz pole z aktualną wartością z atrybutem Balance. Id tej encji powinno być int lub long.

Przykład przedstawia agregacje sprzedaży biletów na filmy. Interesuje nas sprzedaż na konkretny film w zależności od tego czy wybrano seans 2d czy 3d. W tym przypadku potrzebne są tylko dwa wymiary jeden na film i drugi na wybrany typ seansu. Część filmów może nie mieć obu typów seansów wtedy typ będzie nullem.

[AggregateTable(typeof(MovieTicketTurnover), typeof(MovieTicketSaleBalance), precision: 18, scale: 3)]

public class MovieTicketSaleTransaction

{

[Key]

public long Id { get; set; }

[Dimention]

public Guid MovieId { get; set; }

[ForeignKey(nameof(MovieId))]

public virtual Movie Movie { get; set; }

[Dimention]

public TicketType? TicketType { get; set; }

[Balance]

public decimal Balance { get; set; }

public virtual ICollection<MovieTicketTurnover> Turnovers { get; set; }

}

Encja z operacjami musi być oznaczona atrybutem AggregateTurnoverTable do którego trzeba podać nazwę pola które jest kluczem obcym do tabeli z agregacjami. Encja musi mieć pole Id typu long lub int z atrybutem [DatabaseGenerated(DatabaseGeneratedOption.Identity)]. Jednak pole Id nie jest oznaczone jako klucz główny. Klucz główny powinien być złożony z pola klucza obcego do tabeli z agregacjami i datą operacji. Potrzebna jest również kilka innych atrybutów:

TurnoverDate – pole z datą operacji

Record – pole z Guidem do encji powiązanej z operacją

Value – pole z wartością danej operacji

ProfitBalance – pole z wartością przychodu

ExpenseBalance – pole z wartością rozchodu

Przykład:

W tym przypadku nie będzie raczej rozchodu ale i tak musimy dodać te pola. Jako Record id przyjmujemy użytkownika który kupił bilet.

[AggregateTurnoverTable(nameof(DimensionId))]

public class MovieTicketTurnover

{

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public long Id { get; set; }

[Key, Column(Order = 0)]

public long DimensionId { get; set; }

[Key, Column(Order = 1)]

[TurnoverDate]

public DateTime SaleDate { get; set; }

[Record]

public Guid UserId { get; set; }

[Value]

public decimal Value { get; set; }

[ForeignKey(nameof(UserId))]

public User User { get; set; }

[ProfitBalance]

public decimal Profit { get; set; }

[ExpenseBalance]

public decimal Expense { get; set; }

[ForeignKey(nameof(DimensionId))]

public virtual MovieTicketSaleTransaction TicketTransaction { get; set; }

}

Trzecią klasą jaką należy zrobić jest MovieTicketSaleBalance. W niej muszą być pola z wymiarami z głównej encji tylko wszystkie jako nullable i pole z balance. Wszystkie pola muszą nazywać się tak samo jak w głównej encji. Przykład:

Id filmu staje się nullable Guid, rodzaj biletu już był nullable więc zostaje bez zmian.

public class MovieTicketSaleBalance

{

public Guid? MovieId { get; set; }

public TicketType? TicketType { get; set; }

public decimal Balance { get; set; }

}

Po wygenerowaniu repozytorium trzeba go dodać do Dependecy injection. W Web w pliku startup.cs w metodzie RegisterDependencies na końcu dodać services.AddScoped<Iterfejs,Implementacja>() np. services.AddScoped<IMovieTicketSaleTransactionRepository, MovieTicketSaleTransactionRepository>();