

Aplikacje WWW **(studia niestacjonarne)** **Laboratorium 2**

Entity Framework, DAL (Data Access Layer)

1. Wprowadzenie

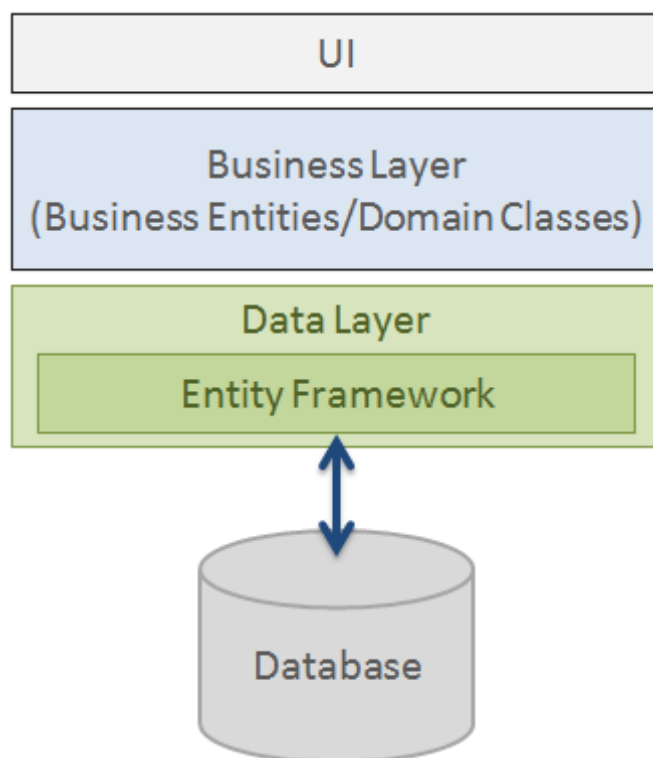
Celem dzisiejszych laboratoriów jest stworzenie schematu bazy danych na podstawie utworzonego poprzednio *data modelu* (Lab 1) oraz poprzez wykorzystanie jednego z tzw. ORM-ów (Object-Relational Mapper). W tym celu będziemy wykorzystywać ORM o nazwie Entity Framework Core (EF). Należy więc wspomnieć czym jest EF oraz jakie kroki należy wykonać aby poprawnie zbudować schemat bazy danych.

1.1. Entity Framework Core

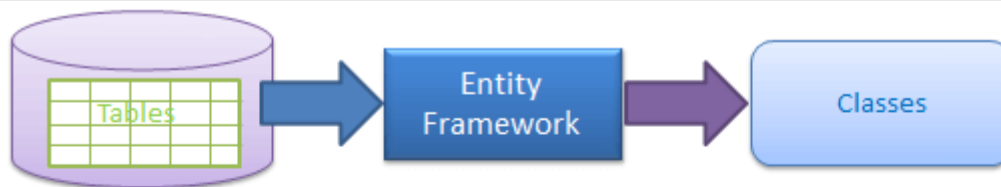
Entity Framework – jest mapperem realacyjno–obiekowym (ORM) pozwalającym odwzorowywać stan obiektów (wysokopoziomowy / biznesowy) na dane relacyjne (niskopoziomowe / tabelaryczne). Innymi słowy EF realizuje dostęp do danych w aplikacjach obiektowych, które to obiekty mappuje na relacje w relacyjnych bazach danych. EF Core umożliwia dostęp do wielu różnych DBMS (systemów baz danych), między innymi: MS SQL Server, MySQL, SQL Lite, PostgreSQL, DB2, oraz wiele innych. Podczas laboratoriów będziemy wykorzystywać jedynie MS SQL Server Express w wersji *localDb* lub w całości (dla użytkowników MacOS lub Linux).

Wyróżniamy dwa podstawowe podejścia w EF:

- Code first (obecnie najpopularniejszy) – najpierw tworzony jest data-model w postaci klas (encji), następnie na ich podstawie tworzony jest schemat bazy danych (DDL).
- Database first – w pierwszym kroku budujemy schemat bazy danych (DDL, *Create*, *Alter*, *Drop*). Następnie na jego podstawie budowane są klasy data modelu.



<http://www.entityframeworktutorial.net/what-is-entityframework.aspx>



Generate Data Access Classes for Existing Database



Create Database from the Domain Classes

<https://www.c-sharpcorner.com/article/getting-started-with-entity-framework/>

Jednym z najważniejszych elementów w EF jest *DbContext*. Jest to predefiniowana klasa reprezentująca bazę danych po stronie aplikacji. Przechowuje wszelkie informacje dotyczące mapowania. Umożliwia pobieranie danych w postaci obiektów z bazy danych, wraz z ich relacjami, jak również modyfikację danych poprzez zmianę obiektów w kolekcjach. Inaczej mówiąc, możliwe jest wykorzystanie baz relacyjnych opartych o SQL bez znajomości samego SQL, manipulując jedynie obiektami. EF sam generuje oraz wykonuje polecenia SQL. Ponadto, posiadając dane w postaci kolekcji w tym przypadku typu *DbSet<T>*. Możemy skorzystać z jednej z największych dobrodziejstw języka C# - wyrażeń lambda (lambda expressions). Więcej na temat samych wyrażeń można znaleźć [tutaj](#). W trakcie trwania laboratoriów nie będziemy wchodzić w szczegóły tej tematyki, zakładając że była ona przyswojona podczas zajęć z poprzednich przedmiotów. Natomiast będziemy często korzystali z wyrażeń lambda przy pobieraniu oraz modyfikacji danych.

Sama klasa *DbContext* jest jedynie klasą bazową dla naszej klasy kontekstu którą będziemy tworzyć. Najczęściej, jeżeli wykorzystujemy ASP.NET Identity Core istnieje jeszcze kilka klas pośrednich w hierarchii dziedziczenia pomiędzy naszą klasą, a klasą bazową *DbContext*. Natomiast zwykle w naszej aplikacji powinniśmy mieć jedną klasę kontekstu (Wyjątkiem są przypadki, gdy korzystamy z wielu baz danych). Poniżej przedstawiono przykładową klasę kontekstu o nazwie *ApplicationDbContext*.

```

public class ApplicationDbContext : IdentityDbContext<User, Role, int>
{
    // Table properties e.g
    // public virtual DbSet<Entity> TableName { get; set; }
    public virtual DbSet<Post> Posts { get; set; } // example table property
    public virtual DbSet<Comment> Comments { get; set; }

    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        base.OnConfiguring(optionsBuilder);
        //configuration commands
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
        // Fluent API commands
        modelBuilder.Entity<User>() // defining a one to many relation in fluent API.
            .HasMany(c => c.Comments)
            .WithOne(a => a.Author)
            .IsRequired()
            .onDelete(DeleteBehavior.Restrict);
    }
}
  
```

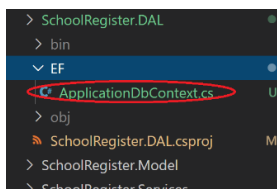
Przeanalizujemy szczegółowo poniższą klasę. Dziedziczy ona po predefiniowanej klasie *IdentityDbContext<User, Role, int>*. Jest to klasa posiadająca trzy parametry generyczne. Dwa z nich są to klasy które stworzyliśmy na poprzednich zajęciach (*User*, *Role*). Natomiast ostatni parametr, definiuje nam jakiego typu będą klucze główne tabel frameworka Identity. W klasie istnieją również dwie właściwości (properties)

są to *Posts* oraz *Comments*. Są one typu *DbSet* wraz z odpowiednim parametrem generycznym. Stanowią one odpowiednik tabel w bazie danych, są to kolekcje, które będą przechowywały obiekty z bazy danych. Następną składową klasy jest konstruktor. Przyjmuje on jeden parametr, którego wartość jest z kolei jest przekazywana przy wywołaniu konstruktora klasy bazowej. Kolejną składową to metoda *OnConfiguring* to w niej mogą być zdefiniowane są wszystkie ustawienia dotyczące konfiguracji połączenia z bazą danych. Ostatnią metodą jest *OnModelCreating*. Umożliwia ona zdefiniowanie tzw. poleceń *fluent API*, które to z kolei pozwalają zdefiniowanie wielu skomplikowanych aspektów dotyczących data modelu (np. klucze główne złożone, zdefiniowanie hierarchii dziedziczenia [TPH](#), oraz wiele innych). Należy pamiętać że tzw. *DataAnnotations* są pomijane jeśli stanowią sprzeczność z *fluent API*.

2. Instalacja i konfiguracja Entity Framework Core

Aby poprawnie zainstalować i skonfigurować Entity Framework należy postępować według poniższych kroków:

- 1) Proszę dodać folder **EF** w projekcie **SchoolRegister.DAL**
- 2) W kolejnym kroku proszę stworzyć nową klasę o nazwie **ApplicationDbContext** w projekcie **SchoolRegister.DAL** w folderze **EF**.



- 3) Klasa **ApplicationDbContext** powinna wyglądać następująco:

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using SchoolRegister.Model.DataModels;

namespace SchoolRegister.DAL.EF
{
    public class ApplicationDbContext : IdentityDbContext<User, Role, int>
    {
        // Table properties e.g
        // public virtual DbSet<Entity> TableName { get; set; }
        public virtual DbSet<Grade> Grades { get; set; }
        // more properties need to added...

        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            base.OnConfiguring(optionsBuilder);
            //configuration commands
            optionsBuilder.UseLazyLoadingProxies(); //enable lazy loading proxies
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            // Fluent API commands
            modelBuilder.Entity<User>()
                .ToTable("AspNetUsers")
                .HasDiscriminator<int>("UserType")
                .HasValue<User>((int)RoleValue.User)
                .HasValue<Student>((int)RoleValue.Student)
                .HasValue<Parent>((int)RoleValue.Parent)
                .HasValue<Teacher>((int)RoleValue.Teacher);
        }
    }
}
```

Jak można zauważyć w metodzie *OnModelCreating* w predefiniowanej tabeli *AspNetUsers* ustalamy tzw. *discriminator* jest to kolumna definiująca jakiego typu będzie dany użytkownik (*Parent*, *Teacher*, *Student*). **Proszę nie mylić tego z rolą. Role określają dostęp do poszczególnych funkcjonalności aplikacji dziedziczenie pozwala rozszerzyć encję o dodatkowe dane.** Przykładem wykorzystania dziedziczenia w EF jest TPH (table per hierarchy) – czyli jedna tabela dla całej hierarchii dziedziczenia. Więcej na ten temat można znaleźć [tutaj](#).

WAŻNE: Klasa *ApplicationDbContext* NIE jest kompletna. Podczas wykonywania zadania należy do niej dodać pozostałe *table properties* dla pozostałych klas. NIE DODAJEMY *table properties* dla klas: Student, Teacher, User, Parent oraz Role. Ze względu na to że korzystamy z podejścia TPH i *table property* dla klasy *User* została dodana w klasie bazowej *IdentityDbContext*. Proszę rozważyć dodawanie *table properties*.

4) Aby klasa *ApplicationDbContext* działała poprawnie należy zainstalować dodatkowo z NuGet-a poniższe pakiety dla projektu *SchoolRegister.DAL*:

- `dotnet add ..\SchoolRegister.DAL\SchoolRegister.DAL.csproj package Microsoft.EntityFrameworkCore --version 5.0.3`
- `dotnet add ..\SchoolRegister.DAL\SchoolRegister.DAL.csproj package Microsoft.AspNetCore.Identity.EntityFrameworkCore --version 5.0.3`
- `dotnet add ..\SchoolRegister.DAL\SchoolRegister.DAL.csproj package Microsoft.EntityFrameworkCore.Proxies --version 5.0.3`
- `dotnet add ..\SchoolRegister.DAL\SchoolRegister.DAL.csproj package Microsoft.EntityFrameworkCore.Design --version 5.0.3`
- `dotnet add ..\SchoolRegister.DAL\SchoolRegister.DAL.csproj package Microsoft.EntityFrameworkCore.SqlServer --version 5.0.3`

Plik *SchoolRegister.DAL.csproj*:

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <ProjectReference Include="..\SchoolRegister.Model\SchoolRegister.Model.csproj" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="5.0.3" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="5.0.3" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="5.0.3">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Proxies" Version="5.0.3" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="5.0.3" />
  </ItemGroup>
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Po poprawnej instalacji powinniśmy poprawnie skompilować solucję:

```
PS B:\Projects\UCZELNIA\Studenci\Aplikacje_MWW_S> dotnet build
Microsoft (R) Build Engine 16.8.3+39993bd9d dla platformy .NET
Copyright (C) Microsoft Corporation. Wszelkie prawa zastrzeżone.

Trwa określanie projektów do przywrócenia...
Przywrócono element B:\Projects\UCZELNIA\Studenci\Aplikacje_MWW_S\SchoolRegister.Web\SchoolRegister.Web.csproj (w 406 ms).
Projekty aktualne na potrzeby przywrócenia: 3 z 5.
SchoolRegister.Model -> B:\Projects\UCZELNIA\Studenci\Aplikacje_MWW_S\SchoolRegister.Model\bin\Debug\net5.0\SchoolRegister.Model.dll
SchoolRegister.DAL -> B:\Projects\UCZELNIA\Studenci\Aplikacje_MWW_S\SchoolRegister.DAL\bin\Debug\net5.0\SchoolRegister.DAL.dll
SchoolRegister.Services -> B:\Projects\UCZELNIA\Studenci\Aplikacje_MWW_S\SchoolRegister.Services\bin\Debug\net5.0\SchoolRegister.Services.dll
SchoolRegister.Web -> B:\Projects\UCZELNIA\Studenci\Aplikacje_MWW_S\SchoolRegister.Web\bin\Debug\net5.0\SchoolRegister.Web.dll
SchoolRegister.Web -> B:\Projects\UCZELNIA\Studenci\Aplikacje_MWW_S\SchoolRegister.Web\bin\Debug\net5.0\SchoolRegister.Web.Views.dll

Kompilacja powiodła się.
Ostrzeżenia: 0
Liczba błędów: 0
```

1.2. Ładowanie powiązanych danych

W każdym istniejącym frameworku ORM występuje problem ładowania powiązanych danych. Aby dokonać połączenia tabel na poziomie bazy danych możemy wykorzystać klauzulę SQL JOIN oraz jej modyfikacje (LEFT, RIGHT, OUTER, INNER). Jednakże takie złączenie jest wykonywane na poziomie bazy danych i w jego efekcie otrzymujemy złączoną tabelę. Relacja ta nie jest jednak tak prosta od odwzorowania na poziomie obiektywnym, czyli na poziomie aplikacji. Przechowywanie dużej ilości danych wiąże się z załadowaniem ich do pamięci. Nie jest to problem jeżeli danych jest mało i struktura bazy danych jest prosta. Złączenia tabel nawet na poziomie SQL często są problematyczne. Wykonywanie

skomplikowanych zapytań na produkcyjnej bazie danych może znacznie spowolnić jej wydajność, a nawet doprowadzić do zawieszenia działania usługi bazy danych. Oczywiście istnieją mechanizmy które mają temu przeciwdziałać (np. load balancing), lecz nawet na poziomie SQL złączenia często prowadzą do różnych przykrych problemów. Proszę wyobrazić sobie sytuację, gdzie mamy duży serwis społecznościowy, posiadający wiele milionów użytkowników. Każdy użytkownik z kolei posiada wiele postów w serwisie, natomiast post posiada wiele komentarzy. Złączenie tych trzech tabel (users, posts, comments) może doprowadzić do sytuacji w której takie zapytanie będzie trwało minuty (np. 5min.). Teraz proszę sobie wyobrazić że takie zapytanie jest umieszczone w aplikacji webowej, do której jednocześnie ma dostęp ma wspomniane kilka milionów użytkowników. W ten sposób możemy doprowadzić zawieszenia działania usługi bazy danych, np. ze względu na brak zasobów. Nawet wykorzystując mechanizm ograniczenia zasobów czy load balancing. Czas kolejkowania będzie tak duży, że nie jest możliwe korzystanie z aplikacji. Co gorsza, problem ten nie jest praktycznie w ogóle widoczny na początku działania aplikacji, gdyż zasoby serwera SQL wystarczają na obsługę małej ilości danych. Problem zaczyna się pojawiać wraz z liniowym wzrostem ilości danych.

Jako że EF buduje zapytania dynamicznie, należy zwrócić **szczególną uwagę** na technikę, którą chcemy wykorzystać do ładowania danych powiązanych. EF jak każdy ORM posiada kilka technik które należy stosować w odpowiednich sytuacjach:

- **Eager Loading** – technika ładowania danych powiązanych w jednym zapytaniu, wszystkie dane zostaną załadowane po wykonaniu jednego zapytania na bazie danych. W tym celu należy stosować metody *Include*, oraz *ThenInclude*. Jednakże należy wybierać tę technikę jedynie w przypadku, gdy mamy pewność że ilość pobranych danych nie będzie zbyt duża oraz relacje nie są zbyt złożone. Jest to również dobra technika do redukcji zbyt wielu dodatkowych zapytań do serwera SQL. Przykład użycia Eager Loading:

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .ThenInclude(author => author.Photo)
        .Include(blog => blog.Owner)
        .ThenInclude(owner => owner.Photo)
        .ToList();
}
```

- **Explicit Loading** – technika ładowania danych powiązanych poprzez jawne ładowanie. Ładowanie danych powiązanych w ten sposób następuje po załadowaniu głównej encji. Technika ta jest o tyle bezpieczna, że ładowanie nie następuje dla całego zbioru danych, a jedynie dla uzyskanych wcześniej encji głównych. W tej technice najpierw wyszukujemy wybraną encję (główną), a następnie przy pomocy metody **Load()** doładowujemy wybrane encje powiązane. Zarówno zaletą i jak i wadą (w zależności od przypadku użycia) jest fakt że wykonywane jest kilka zapytań do bazy danych. Przykład poniżej:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    context.Entry(blog)
        .Collection(b => b.Posts)
        .Load();

    context.Entry(blog)
        .Reference(b => b.Owner)
        .Load();
}
```

- **Lazy loading** – metoda ładowania danych powiązanych w momencie pierwszego pobrania wybranej właściwości nawigacyjnej. Inaczej ujmując, dane ładowane są w momencie użycia. Przykład:

```
User usr = dbContext.Users.FirstOrDefault(a => a.UserId == userId); // pobranie tylko użytkownika
UserDetails ud = usr.UserDetails; // doładowanie danych powiązanych
```

Jak można zauważyć technika ta jest bardzo użyteczna i rozwiązuje wiele problemów występujących w poprzednich metodach. Jednakże nie jest panaceum na wszystkie problemy i w wielu przypadkach lepszym rozwiązaniem jest zastosowanie jednej z pozostałych technik. Aby włączyć lazy loading należy:

- Wywołać metodę **UseLazyLoadingProxies** w metodzie **OnConfiguring**. Dzięki temu włączamy lazy loading globalnie w aplikacji

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    base.OnConfiguring(optionsBuilder);
    optionsBuilder.UseLazyLoadingProxies(); //enable lazy loading proxies
}
```

- Każda właściwość nawigacyjna (navigation property) musi być posiadać modyfikator **virtual**.

Przykładowa klasa **Student**:

```
public class Student : User
{
    public virtual Group Group { get; set; }
    [ForeignKey("Group")]
    public int? GroupId { get; set; }
    public virtual IList<Grade> Grades { get; set; }
    public virtual Parent Parent { get; set; }

    [ForeignKey("Parent")]
    public int? ParentId { get; set; }

    [NotMapped]
    public double AverageGrade => Grades == null || Grades.Count == 0 ? 0.0d : Math.Round(Grades.Average(g => (int)g.GradeValue), 1);

    [NotMapped]
    public IDictionary<string, double> AverageGradePerSubject => Grades == null ? new Dictionary<string, double>() :
        Grades.GroupBy(g => g.Subject.Name)
        .Select(g => new { SubjectName = g.Key, AvgGrade = Math.Round(g.Average(avg => (int) avg.GradeValue), 1)})
        .ToDictionary(avg => avg.SubjectName, avg => avg.AvgGrade);

    [NotMapped]
    public IDictionary<string, List<GradeScale>> GradesPerSubject => Grades == null ? new Dictionary<string, List<GradeScale>>() : Grades
        .GroupBy(g => g.Subject.Name)
        .Select(g => new { SubjectName = g.Key, GradeList = g.Select(x=>x.GradeValue).ToList() })
        .ToDictionary(x=>x.SubjectName, x=>x.GradeList);
}
```

- 5) Aby poprawnie zdefiniować relacje pomiędzy klasami a później tabelami w bazie danych możemy te relacje zdefiniować za pomocą **DataAnnotations**. Aby utworzyć relację **one-to-many** należy w klasie będącej **many** zdefiniować trzy elementy:

- **Navigation property** – jest to właściwość przechowująca referencję do obiektu w relacji **one**. W poniższym przykładzie jest to `public virtual Group Group { get; set; }`
- **Foreign key property** – jest to właściwość przechowująca id klucza obcego do referowanej tabeli. Np. `public int? GroupId { get; set; }`
- **Foreign key attribute** – jest to atrybut (nie mylić z parametrem, czy argumentem) definiujący klucz obcy oraz wspomnianą **navigation property** przekazaną jako parametr atrybutu. Przykład: `[ForeignKey("Group")]`

WAŻNE: Aby relacja działała poprawnie należy pamiętać aby nazwa **navigation property** została podana w atrybucie jako parametr.


```

public class Student : User
{
    0 references
    public virtual Group Group { get; set; }
    [ForeignKey("Group")]
    0 references
    public int? GroupId { get; set; }
    0 references

```

6) Następnie w klasie **one** należy zdefiniować listę. Przykład poniżej:

```

public class Group
{
    [Key]
    0 references
    public int Id { get; set; }
    [Required]
    1 reference
    public string Name { get; set; }
    0 references
    public virtual IList<Student> Students { get; set; }
    1 reference

```

7) Możliwe jest również osiągnięcie tego samego poprzez wykorzystanie *fluent Api*:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    // Fluent API commands
    modelBuilder.Entity<Group>()
        .HasMany(g => g.Students)
        .WithOne(s => s.Group)
        .HasForeignKey(x => x.GroupId)
        .IsRequired();
}

```

8) Aby ustawić klucz główny tabeli należy umieścić atrybut **[Key]** nad wybraną właściwością:

```

public class Subject
{
    [Key]
    0 references
    public int Id { get; set; }
    [Required]
    2 references
    public string Name { get; set; }

```

9) Aby pominąć którąś z właściwości w procesie tworzenia tabeli, należy skorzystać z atrybutu **[NotMapped]**

```

[NotMapped]
0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
public double AverageGrade => Math.Round(Grades.Average(g => (int)g.GradeValue), 1);

```

10) Wielokrotnie podczas tworzenia kluczy obcych lub głównych przy pomocy atrybutów **[Key]** lub **[ForeignKey]** mogą wystąpić poniższe błędy.

The type or namespace name 'RequiredAttribute' could not be found (are you missing a using directive or an assembly reference?)

The type or namespace name 'ForeignKey' could not be found (are you missing a using directive or an assembly reference?)

Należy wtedy odpowiednio dodać poniższe komendy **using**, w plikach w których występują błędy.

```
using System.ComponentModel.DataAnnotations;  
using System.ComponentModel.DataAnnotations.Schema;
```

- 11) EF analizuje encje i tworzy relacje w oparciu o typy danych w nich zawarte. Jeżeli korzystamy z relacji w której wartość klucza obcego ma być niepusta to wystarczy posłużyć się typem **int**. Wtedy w momencie tworzenia obiektu wartość klucza obcego będzie wymagana.

```
public virtual Subject Subject { get; set; }  
[ForeignKey("Subject")]  
1 reference  
public int SubjectId { get; set; }
```

Jeżeli natomiast chcemy umożliwić wstawienie wartości pustych do wartości kluczy obcych, należy wtedy użyć typu umożliwiającego wstawienie null. Jest to typ [Nullable<int>](#) lub **int?**. Oba zapisy są poprawne. Taki typ pozwala na wstawienie wartości pustej (null), co powoduje odwzorowanie tego zachowania przez EF na poziomie bazy danych.

```
public virtual Group Group { get; set; }  
[ForeignKey("Group")]  
0 references  
public int? GroupId { get; set; }
```

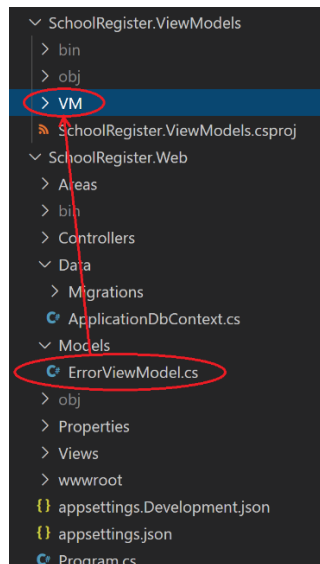
Bardzo podobnie EF zachowuje się w przypadku typu **string**. Warto podkreślić że w C# string jest **niemodyfikowalnym (immutable) typem referencyjnym**. Co oznacza że jakkolwiek zmiana wartości zmiennych tego typu powoduje stworzenie nowego obiektu. Co może powodować wrażenie że jest to typ wartościowy, jednakże tak nie jest i należy pamiętać o tej właściwości typu **string**. Dzięki temu że string jest typem referencyjnym nie jest potrzebne użycie **Nullable**, aby umożliwić tworzenie kolumn posiadających pustą wartość. EF domyślnie analizuje encje i wszędzie tam gdzie występuje typ string stworzy odpowiadające mu kolumny, w których wierszach będzie można wstawić wartość null. Ta zasada dotyczy także kluczy obcych. Aby wymusić wstawienie wartości dla typu string, należy umieścić atrybut **[Required]** nad właściwością posiadającą typ string.

```
[Required]  
1 reference  
public string Name { get; set; }
```

Warto również dodać że analogiczne ograniczenie można dodać na poziomie **Fluent Api**. Oba podejścia są poprawne.

```
modelBuilder.Entity<Group>()  
    .Property(g=>g.Name)  
    .IsRequired();
```

- 12) Kolejno proszę w projekcie **SchoolRegister.ViewModels** stworzyć nowy folder o nazwie **VM**
- 13) Następnie proszę przenieść klasę **ErrorViewModel.cs** do folderu **VM** w projekcie **SchoolRegister.ViewModels**



14) Kolejno proszę zmienić przestrzeń nazw w klasie ***ErrorViewModel*** na ***SchoolRegister.ViewModels.VM***

```
using System;

namespace SchoolRegister.ViewModels.VM
{
    0 references
    public class ErrorViewModel
    {
        1 reference
        public string RequestId { get; set; }

        0 references
        public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);
    }
}
```

15) W kolejnym kroku proszę w pliku ***Views->Shared->Error.cshtml*** proszę dodać nazwę przestrzeni nazw

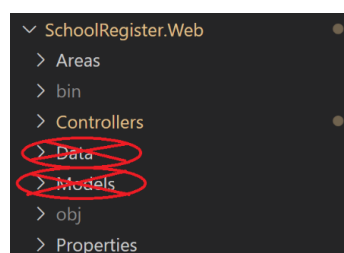
```
@using SchoolRegister.ViewModels.VM
```

```
SchoolRegister.Web > Views > Shared > Error.cshtml
1 | @using SchoolRegister.ViewModels.VM
2 | @model ErrorViewModel
3 | @{
4 |     ViewData["Title"] = "Error";
5 | }
```

Następnie proszę w pliku ***Views -> _ViewImports.cshtml*** podmienić przestrzeń nazw tak aby plik wyglądał następująco:

```
SchoolRegister.Web > Views > _ViewImports.cshtml
1 | @using SchoolRegister.Web
2 | @using SchoolRegister.ViewModels.VM
3 | @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Proszę w projekcie ***SchoolRegister.Web*** usunąć foldery ***Data*** oraz ***Models***



- 16) Aby wygenerować schemat bazy danych należy również dokonać zmian w pliku **SchoolRegister.Web -> Startup.cs**

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using SchoolRegister.DAL.EF;
using SchoolRegister.Model.DataModels;

namespace SchoolRegister.Web
{
    public class Startup
    {
        public IConfiguration Configuration { get; }
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

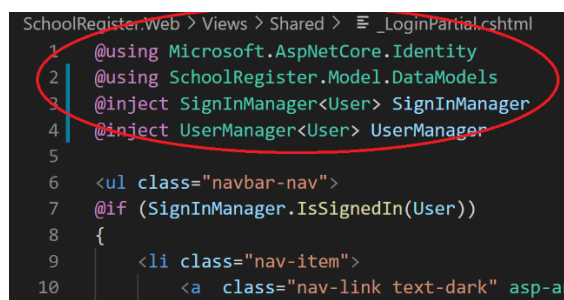
        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")) //here you can define a database type.
            );
            services.AddDatabaseDeveloperPageExceptionFilter();
            services.AddDefaultIdentity<User>(options => options.SignIn.RequireConfirmedAccount = false)
                .AddRoles<Role>()
                .AddRoleManager<RoleManager<Role>>()
                .AddUserManager<UserManager<User>>()
                .AddEntityFrameworkStores<ApplicationDbContext>();
            services.AddTransient(typeof(ILogger), typeof(Logger<Startup>));
            services.AddControllersWithViews();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
                app.UseMigrationsEndPoint();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseAuthentication();
            app.UseAuthorization();
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
                endpoints.MapRazorPages();
            });
        }
    }
}
```

- 17) Proszę również w pliku **Views->Shared->_LoginPartial.cshtml** dodać dyrektywę using oraz podmienić klasę IdentityUser na **User**:

```
@using Microsoft.AspNetCore.Identity
@using SchoolRegister.Model.DataModels
@inject SignInManager<User> SignInManager
@inject UserManager<User> UserManager
```

Efekt finalny powinien wyglądać następująco:



```
SchoolRegister.Web > Views > Shared > _LoginPartial.cshtml
1  @using Microsoft.AspNetCore.Identity
2  @using SchoolRegister.Model.DataModels
3  @inject SignInManager<User> SignInManager
4  @inject UserManager<User> UserManager
5
6  <ul class="navbar-nav">
7  @if (SignInManager.IsSignedIn(User))
8  {
9      <li class="nav-item">
10         <a class="nav-link text-dark" asp-ar
```

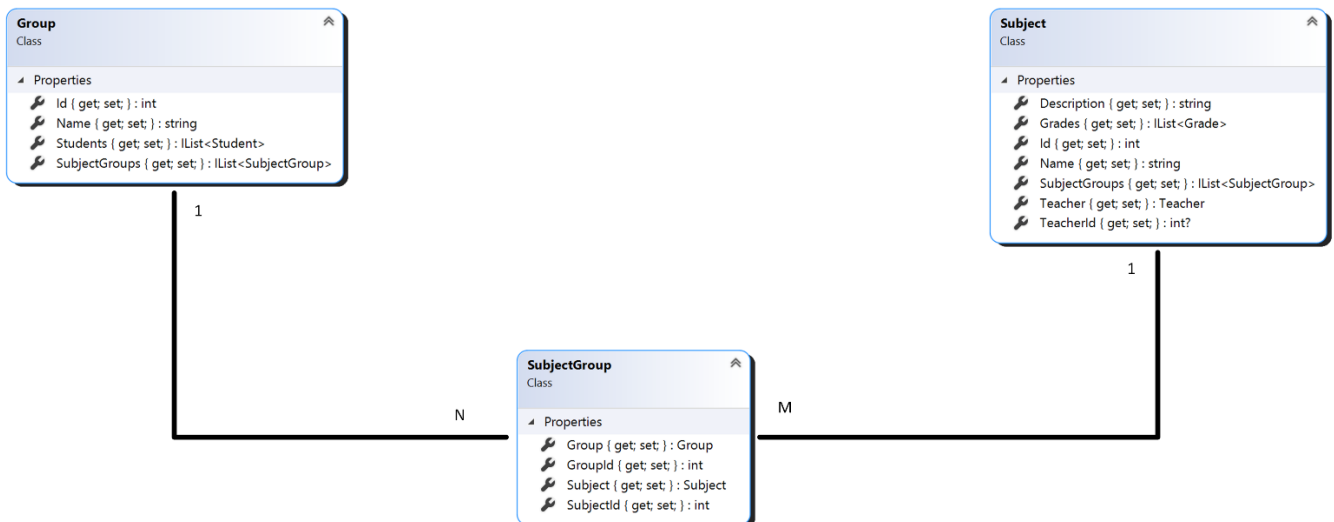
18) Następnie należy otworzyć plik **appsettings.json** znajdujący w SchoolRegister.Web. Jest o plik *json* przechowujący wszystkie statyczne ustawienia (niezmienne w czasie pracy aplikacji).

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Data-
base={Imie_Nazwisko}AppDb;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

19) **WAŻNE: Proszę wpisać swoje imię i nazwisko w connection string zamiast {Imie_Nazwisko}. Proszę USUNĄĆ nawiasy {} w connection stringu.**

20) Kolejnym problemem z którym możecie się Państwo spotkać jest realizacja relacji *many to many (N:M)*

Many to many relationship (N:M)



21) Po stworzeniu encji *SubjectGroup* należy:

- Ustanowić klucz główny złożony dla encji *SubjectGroup* dla kolumn *SubjectId* oraz *GroupId*.
- Ustanowić klucz obcy (przy pomocy fluent Api) z *SubjectGroup* (*GroupId*) do tabeli *Groups* (*Id*)
- Ustanowić klucz obcy (przy pomocy fluent Api) z *SubjectGroup* (*SubjectId*) do tabeli *Subjects* (*Id*)
- Następnie należy zdefiniować reakcję na usunięcie encji *OnDelete* – *Restrict*.

22) Powyższe kroki można wykonać w metodzie **OnModelCreating** w klasie **ApplicationDbContext**:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    // Fluent API commands
    modelBuilder.Entity<User>()
        .ToTable("AspNetUsers")
        .HasDiscriminator<int>("UserType")
        .HasValue<User>((int)RoleValue.User)
        .HasValue<Student>((int)RoleValue.Student)
        .HasValue<Parent>((int)RoleValue.Parent)
        .HasValue<Teacher>((int)RoleValue.Teacher);

    modelBuilder.Entity<SubjectGroup>()
        .HasKey(sg => new {sg.GroupId, sg.SubjectId});

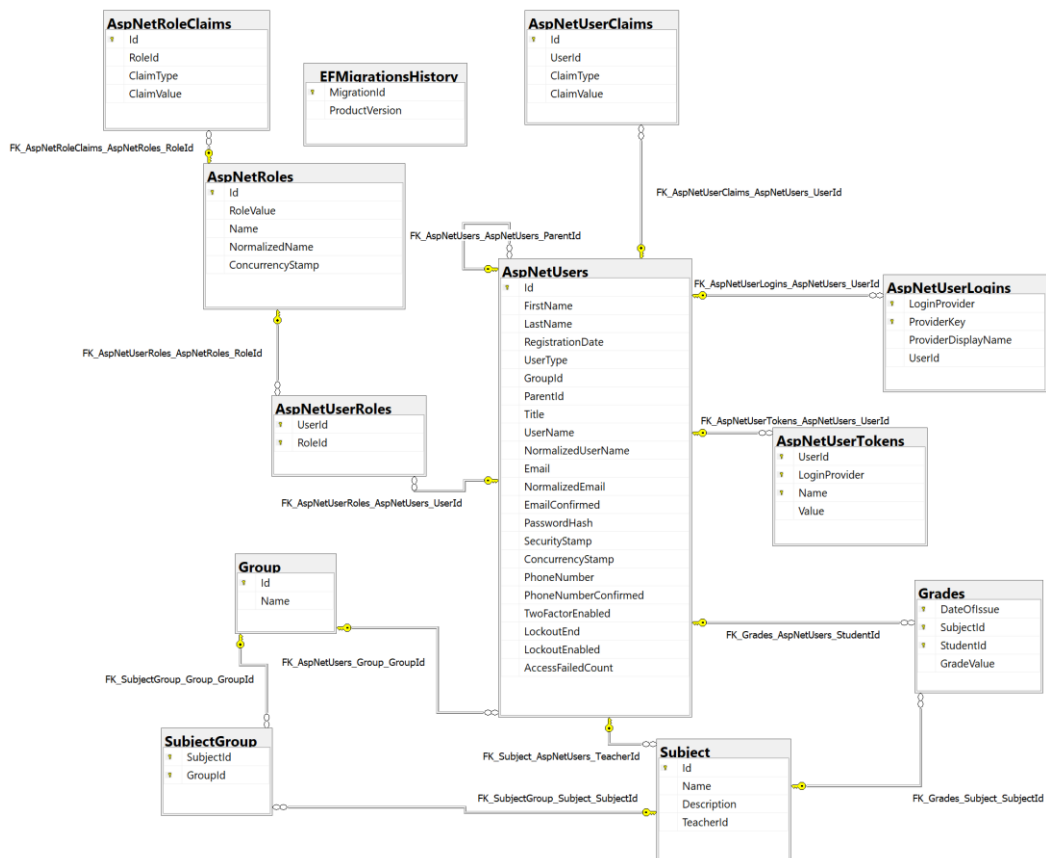
    modelBuilder.Entity<SubjectGroup>()
        .HasOne(g => g.Group)
        .WithMany(sg => sg.SubjectGroups)
        .HasForeignKey(g => g.GroupId);

    modelBuilder.Entity<SubjectGroup>()
        .HasOne(s => s.Subject)
        .WithMany(sg => sg.SubjectGroups)
        .HasForeignKey(s => s.SubjectId)
        .OnDelete(DeleteBehavior.Restrict);
}
```

23) Proszę skompilować całą Solucję. Jeżeli kompilacja zakończyła się sukcesem możemy przejść do wykonania zadania.

3. Zadanie

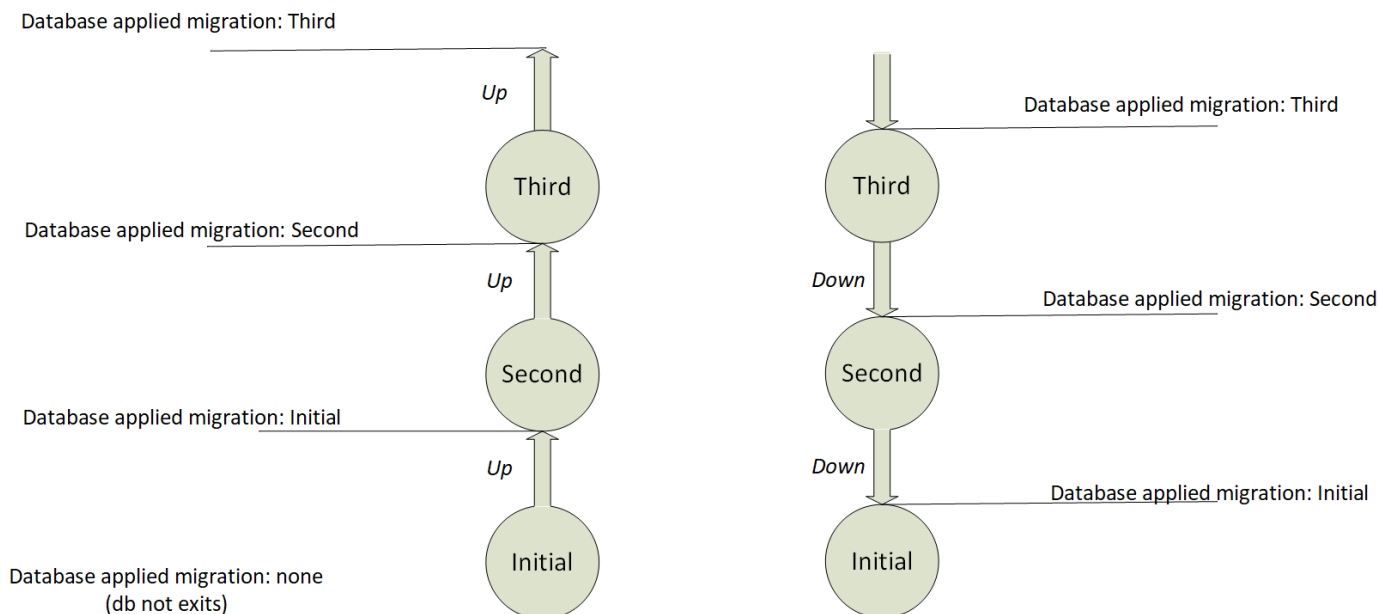
Proszę zdefiniować relacje pomiędzy encjami (w klasach), przy pomocy *Data Annotations* oraz/lub poprzez *Fluent Api*, w oparciu o załączony poniżej diagram encji. Następnie proszę przejść do etapu stworzenia oraz zastosowania migracji opisanego w punkcie 4. Po zakończonej pracy proszę zatwierdzić i wysłać (Commit & Push) swój kod na serwer github.com. **Proszę pamiętać aby dokonywać zmian na swojej gałęzi.** Poniżej diagram encji – wynikowy.



Wynikowy diagram encji (tabel)

4. Migracje (Punkty migracyjne)

Migracje zwane również punktami migracyjnymi pozwalają zapisywać poszczególne wersje schematu bazy danych. Dzięki temu możemy w każdej chwili cofnąć się do poprzedniego punktu migracyjnego. Migracje są tworzone w sposób przyrostowy, więc jeśli przypadkowo usuniemy jakiś pośredni plik migracyjny, migracja może zakończyć się błędem.



Każda migracja jest klasą w języku C# dziedziczącą po klasie *Migration*. Zawiera dwie metody *Up* oraz *Down*. *Up* wykonuje akcje podnoszące migrację. *Down* cofa migrację do poprzedniego stanu.

Aby stworzyć migrację należy postępować według poniższych kroków:

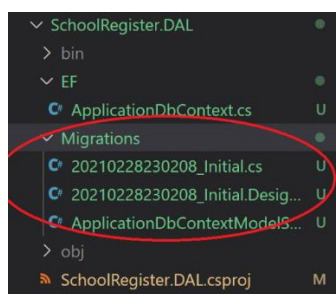
- 1) Proszę upewnić się że cała solucja się kompiluje, w kolejnym kroku proszę zaktualizować wersję narzędzi **dotnet-ef** do wersji 5.0.3 przy pomocy poniższej komendy:

```
dotnet tool update --global dotnet-ef --version 5.0.3
```

- 2) Następnie proszę stworzyć nową migrację o nazwie **Initial**, wykonując w konsoli poniższy kod:

```
dotnet ef migrations add Initial --project .\SchoolRegister.DAL\SchoolRegister.DAL.csproj --startup-project .\SchoolRegister.Web\SchoolRegister.Web.csproj
```

- 3) Jeżeli migracja wykonała się poprawnie zobaczymy że powstał nowy folder posiadający pliki migracyjne:



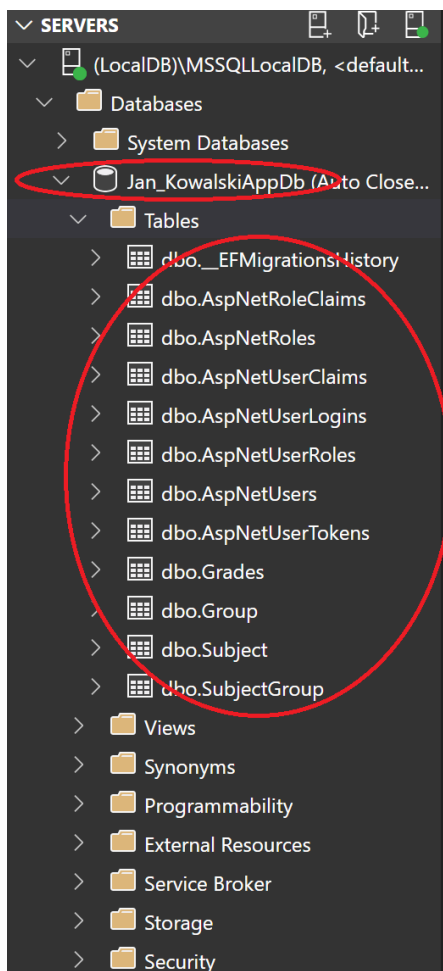
Przykładowy plik migracyjny zawiera wspomniane wcześniej metody *Up* oraz *Down*:

```
namespace SchoolRegister.DAL.Migrations
{
    0 references
    public partial class Initial : Migration
    {
        0 references
        protected override void Up(MigrationBuilder migrationBuilder) ...
        protected override void Down(MigrationBuilder migrationBuilder) ...
    }
}
```

- 4) **WAŻNE:** Stworzenie migracji nie oznacza, że zostały wprowadzone jakiekolwiek zmiany na bazie danych. Aby to zrobić należy wykonać poniższą komendę w celu zastosowania najnowszej (najwyższej) migracji.

```
dotnet ef database update --project .\SchoolRegister.DAL\SchoolRegister.DAL.csproj --startup-project .\SchoolRegister.Web\SchoolRegister.Web.csproj
```

- 5) Jeżeli zadanie zostało wykonane oraz wszystkie powyższe kroki zostały wykonane poprawnie proszę uruchomić **Azure Data Studio**, oraz połączyć się z serwerem **LocalDb**.



- 6) Jak można zauważyć stworzona została baza danych o nazwie którą podaliśmy w **appsettings.json**. Tabele z prefiksem *AspNet* są to tabele frameworka **ASP.NET Identity**. Niektóre z nich zostały zmodyfikowane przez nasz data model. Aby sprawdzić która migracja jest obecnie zastosowana należy wyświetlić tabelę: **__EFMigrationsHistory**. Proszę upewnić się baza danych posiada wszystkie zakładane wcześniej tabele, klucze główne, obce oraz ograniczenia.

Baza danych utworzona w ten sposób będzie podstawą pod dalsze elementy projektu. Proszę zwrócić szczególną uwagę na utworzoną migrację, poprawność relacji pomiędzy encjami, gdyż jest to jeden z głównych elementów, które mogą sprawić Państwu problemy. Proszę bardzo dokładnie porównać uzyskaną bazę danych z podanym wcześniej diagramem encji.