EUVIC: THE GOOD People

.NET
EntityFramework
Part 1



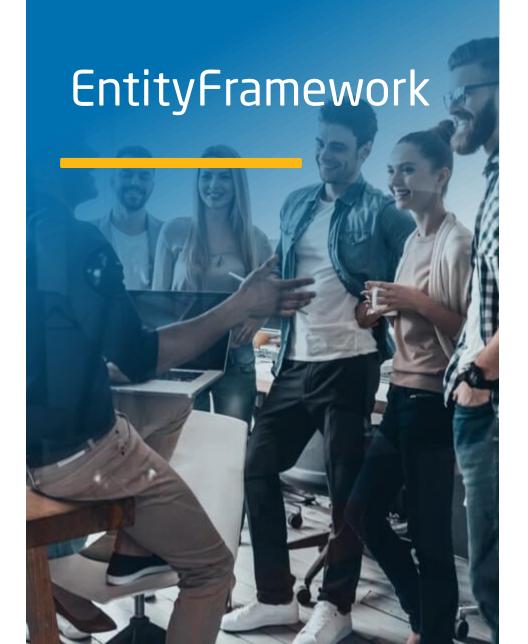
Czym jest ORM

ORM jest to technologia, która mapuje struktury bazy danych na klasy występujące w obiektowych językach programowania.

Przykłady technologi ORM dla dotneta

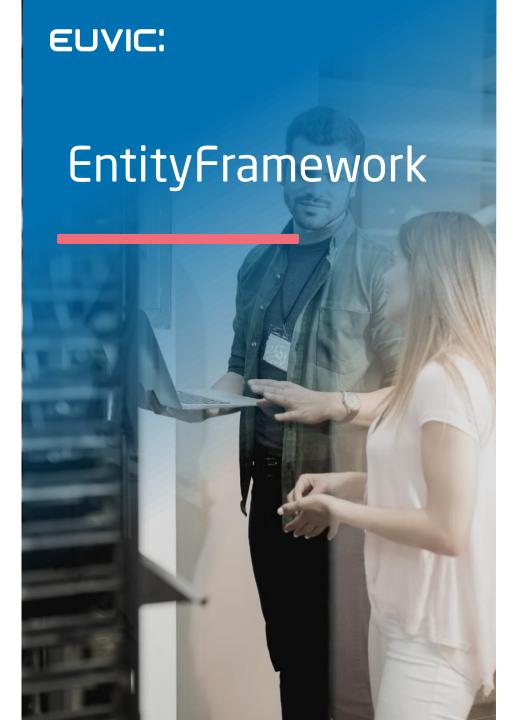
EntityFramework, Nhibernate, Dapper

EUVIC:



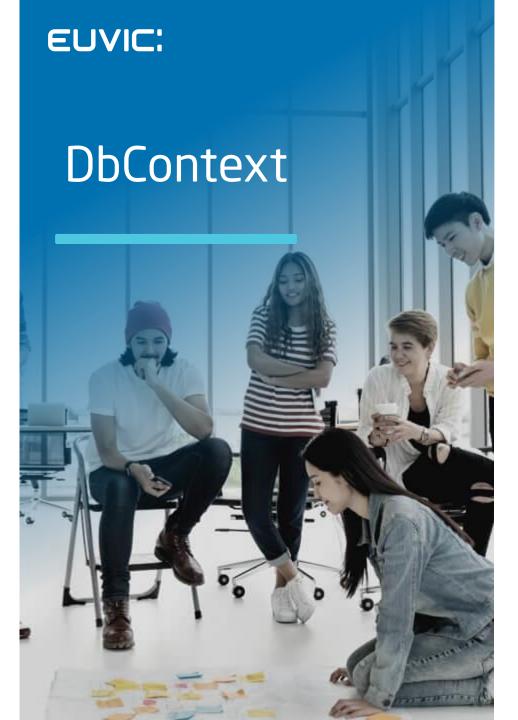
Wyróżnione możliwości EntityFramework

- Zmapowanie tabel bazodanowych na klasy
- Zmapowanie kolumn tabel na properties
- Mapowanie relacji bazodanowych
- Mapowanie widoków bazodanowych na klasy
- Generowanie migracji
- Wbudowana tranzakcyjność



Słownik pojęć

- Encja klasa reprezentująca tabelę
- **DbContext** klasa która jest źródłem dostępu do bazy danych. DbContext implementuje wzorzec "Unit of Work".
- Unit of work wzorzec który zbiera zmiany z repozytoriów i wykonuje je w jednej tranzakcji
- DbSet "Repozytorium" które jest przeznaczone dla pojedynczej encji
- Model Reprezenacja bazy danych jako zestawu encji znajdujących się w DbContexcie
- Migracja kod który reprezentuje zmianę w modelu której wynikiem będzie zmiana bazy danych
- Model Snapshot aktualny stan naszego modelu
- Migration Snapshot poprzedni stan modelu który był przed wygenerowaniem migracji



Przykładowy DbContext

- Encje nazywamy jako liczba pojedyncza np. "Attendee"
- DbSet nazywamy jako liczba mnoga np. "Attendees".
 Nazwa DbSet będzie automatycznie nazwą tabeli po wygenerowaniu migracji



Rejestracja DbContext

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<StaffTrainingContext>(
        options => options.UseSqlServer(Configuration.GetConnectionString("Sql")));
}

"ConnectionStrings": {
    "Sql": "Server=(localdb)\\mssqllocaldb;Database=StaffTraining"
}
```

- Aby móc zarejestrować DbContext potrzebujemy ConnectionString do bazy danych.
- Na potrzeby lokalne warto skorzystać z serwera "(localdb)\\mssqllocaldb"
- Connection String zawsze dostarczy nam usługodawca w przypadku wykupionego hostingu dotneta

EUVIC:

Konfiguracja encji

- Bardzo polecam aby skorzystać z interfacu lEntityTypeConfiguration, aby skonfigurować każdą encje
- Przeniesienie konfiguracji do osobnego pliku wprowdza porządek
- Konfiguracja może być w innym projekcie niż klasy domeny (encje), przez co projekt domeny, nie musi mieć zależności na EntityFramework

```
public class AttendeeConfiguration : IEntityTypeConfiguration<Attendee>
{
    Oreferences
    public void Configure(EntityTypeBuilder<Attendee> builder)
    {
        builder.HasKey(e => e.Id);
        builder.Property(x => x.Firstname)
            .HasMaxLength(50)
            .IsRequired();

        builder.Property(x => x.Lastname)
            .HasMaxLength(50)
            .IsRequired();
    }
}
```

Relacje

Konfiguracja "One To Many"

```
public class Training
{
    2 references
    public Guid Id { get; set; }
    1 reference
    public string Title { get; set; }
    1 reference
    public long LecturerId { get; set; }

    1 reference
    public Lecturer Lecturer { get; set; }
}
```

Relacje

Konfiguracja "Many To Many"

```
public class Training
{
    Oreferences
    public long Id { get; set; }
    Oreferences
    public string Title { get; set; }
    Oreferences
    public string Description { get; set; }
    1reference
    public long LecturerId { get; set; }

    1reference
    public Lecturer Lecturer { get; set; }
    2references
    public ICollection<Attendee> Attendees { get; set; }
}
```

EUVIC:

Relacje

Konfiguracja "Many To Many"

```
public class TrainingAttendeeConfiguration : IEntityTypeConfiguration<TrainingAttendee>
    0 references
    public void Configure(EntityTypeBuilder<TrainingAttendee> builder)
        builder.HasKey(x => new { x.TrainingId, x.AttendeeId });
        builder.HasOne(x => x.Training)
            .WithMany(x => x.Attendees)
            .HasForeignKey(x => x.TrainingId);
        builder.HasOne(x => x.Attendee)
         .WithMany(x => x.Trainings)
         .HasForeignKey(x => x.AttendeeId);
public class Training
    0 references
    public long Id { get; set; }
    public string Title { get; set; }
    1 reference
    public ICollection<TrainingAttendee> Attendees { get; set; }
```

```
public class TrainingAttendee
    public long TrainingId { get; set; }
    public long AttendeeId { get; set; }
    public TrainingAttendeeStatus Status { get; set; }
    public Training Training { get; set; }
    public Attendee Attendee { get; set; }
public class Attendee
    1 reference
    public long Id { get; set; }
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public ICollection<TrainingAttendee> Trainings { get; set; }
```

Relacje

Konfiguracja "One To One"

Relacje

Konfiguracja "Owns One"

```
internal class UserConfiguration : IEntityTypeConfiguration<User>
{
    Oreferences
    public void Configure(EntityTypeBuilder<User> builder)
    {
        builder.HasKey(x => x.Id);
        builder.OwnsOne(x => x.Profile);
    }
}

III identity.Users
II
```

Ignorowanie properties

- Domyślnie EntityFramework generuje migracje dla każdego publicznego property
- Czasem się zdarza, że chcemy stworzyć property, które oblicza cos z innego property
- W takim przypadku musimy zignorować takie property w konfiguracji

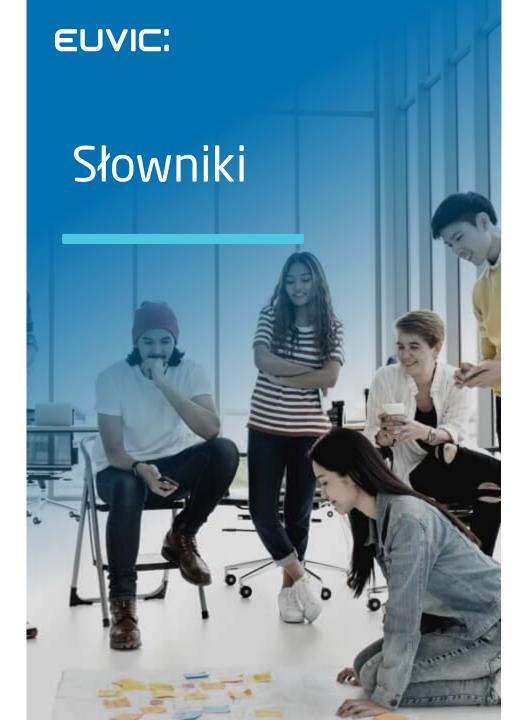
EUVIC:

Mapowanie widoków

```
public class AttendeesSummaryConfigurations : IEntityTypeConfiguration<AttendeeSummary>
    0 references
    public void Configure(EntityTypeBuilder<AttendeeSummary> builder)
        builder.HasNoKey();
                                                              public partial class AddViewAttendeesSummary : Migration
        builder.ToView("dbo.AttendeeSummary");
                                                                  0 references
                                                                  protected override void Up(MigrationBuilder migrationBuilder)
public class AttendeeSummary
                                                                      migrationBuilder.Sql(
    0 references
    public long AttendeeId { get; set; }
                                                                           CREATE VIEW [dbo].[AttendeeSummary]
    0 references
    public int TotalHours { get; set; }
                                                                                    dbo.Attendees.Id AS AttendeeId, SUM(dbo.Trainings.Duration) AS TotalHours
                                                                           SELECT
                                                                           FROM
                                                                                        dbo.Attendees
                                                                           INNER JOIN dbo.TrainingAttendee ON dbo.Attendees.Id = dbo.TrainingAttendee.AttendeeId
                                                                           INNER JOIN dbo.Trainings ON dbo.TrainingAttendee.TrainingId = dbo.Trainings.Id
                                                                           GROUP BY dbo.Attendees.Id
```

protected override void Down(MigrationBuilder migrationBuilder)

migrationBuilder.Sql("DROP VIEW [dbo].[AttendeeSummary]");



Słowniki (Lookups, Dictionaries)

- Zawsze w aplikacjach mamy dane słownikowe, które często nie są dodawane przez użytkowników tylko są ewentualnie dodawane przez programistów
- Typowym przykładem takiego słownika są statusy jakiegoś stanu w aplikacji

W takim przypadku mamy dwa podejścia:

- Możemy dodać stworzyć encje, którą możemy zaseedować danymi w migracjach
- Możemy stworzyć enum oraz zrobić converter

Relacje – podejście z Encją

```
public class TrainingAttendeeStatus
    5 references
    public int Id { get; set; }
    public string Name { get; set; }
public class TrainingAttendee
    2 references
    public Guid TrainingId { get; set; }
    2 references
    public long AttendeeId { get; set; }
    2 references
    public int StatusId { get; set; }
    public TrainingAttendeeStatus Status { get; set; }
    public Training Training { get; set; }
    public Attendee Attendee { get; set; }
public void Configure(EntityTypeBuilder<TrainingAttendeeStatus> builder)
   builder.HasKey(x => x.Id);
    builder.HasData(TrainingAttendeeStatusesSeed.GetSeed());
```

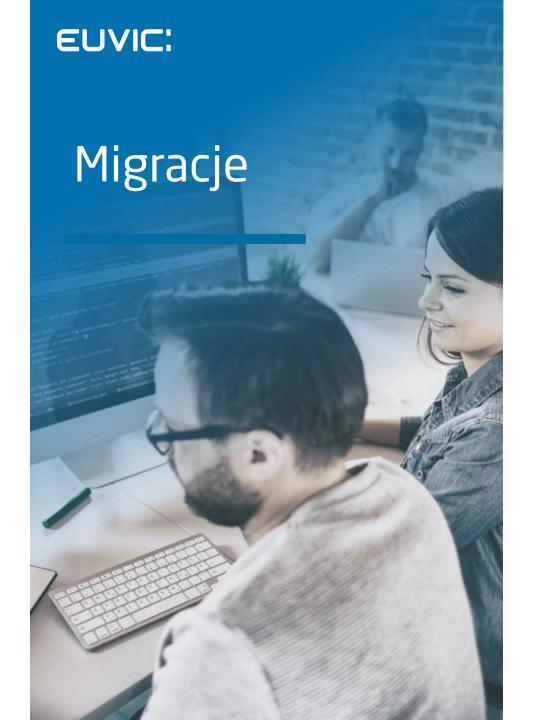
```
public static class TrainingAttendeeStatusesSeed
    1 reference
    public static IEnumerable<TrainingAttendeeStatus> GetSeed() =>
        new List<TrainingAttendeeStatus>()
            new TrainingAttendeeStatus()
                Id = (int)TrainingAttendeeStatuses.Interested,
                Name = TrainingAttendeeStatuses.Interested.ToString()
            new TrainingAttendeeStatus()
                Id = (int)TrainingAttendeeStatuses.Confirmed,
                Name = TrainingAttendeeStatuses.Confirmed.ToString()
            new TrainingAttendeeStatus()
                Id = (int)TrainingAttendeeStatuses.Declined,
                Name = TrainingAttendeeStatuses.Declined.ToString()
        };
```

Słowniki – podejście z enumem

```
public enum TechnologyScope
{
    Backend = 1,
    Frontend = 2,
    DevOps = 3,
}

public class Technology
{
    ¹reference
    public long Id { get; set; }
    ¹reference
    public string Name { get; set; }
    ¹reference
    public TechnologyScope Scope { get; set; }
}
```

| | ld | Name | Scope | CreateDate | UpdateDate |
|---|----|-----------------|---------|-----------------------------|-----------------------------|
| 1 | 2 | EntityFramework | Backend | 2022-09-05 00:00:00.0000000 | 2022-09-05 00:00:00.0000000 |



Co to są migracje

Migracje jest to mechanizm entity frameworka, który pozwala w łatwy sposób trzymać spójność modelu z bazą danych.

Każda zmiana w modelu, która będzie powodowała zmianę bazy danych musi być zakończona wygenerowaniem migracji

Migracje pozwalają utrzymać kontrolę w jakiej "wersji" jest aktualnie baza na środowisku lub lokalnie



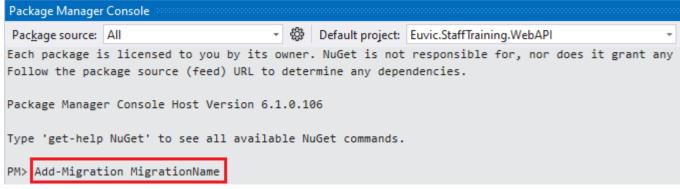
Kilka porad o migracjach

- Po wygenerowaniu pierwszej migracji, stworzy się nam folder "Migrations" w "root" projektu, folder ten możemy przenieść w dowolne miejsce w projekcie i nastepne migracje będą juz generowały się własnie tam.
- Każda z migracji zawiera nazwę migracji, polecam nazywać migracje aby odzwierciedlały to co rzeczywiście zawierają
- W folderze migrations znajduje się plik "ModelSnapshot.cs" który reprezentuje aktualny stan modelu
- Dodatkowo każda z migracji zwiera swój plik snapshot który jest wersją "ModelSnapshot" zanim został on zmieniony w związku z nową migracją
- Migracje możemy cofać do dowolnego punktu w przeszłości



Generowanie migracji THE GOOD People

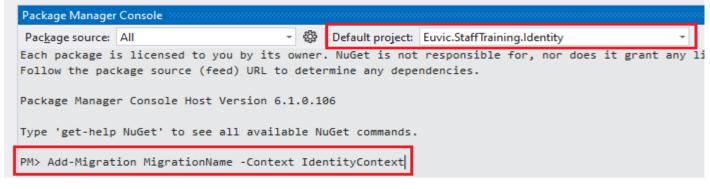
Aby wygenerować migracje należy wykonać komendę:



W przypadku gdy jest więcej niż 1 context musi miec dodatkowy parameter – Context

PM> Add-Migration MigrationName -Context StaffTrainingReadonlyContext

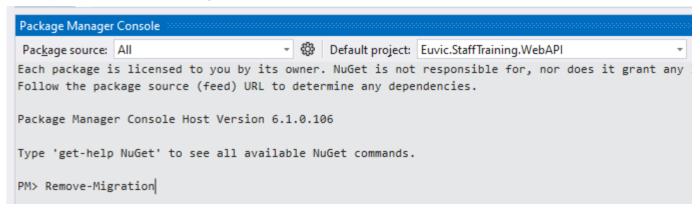
Gdy context znajduje się w innym projekcie niż projekt uruchomieniony (WebAPI) wtedy musimy wybrać "Default project"







Aby skasować migracje należy wykonać komendę:



W przypdku gdy jest wiele contextów trzeba dodać parameter -Context

PM> Remove-Migration -Context StaffTrainingContext

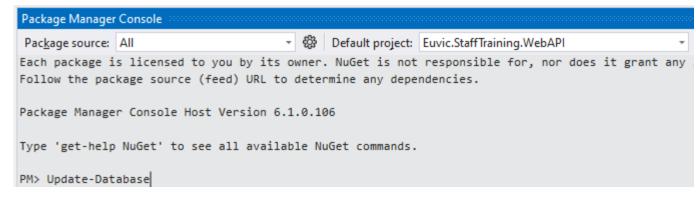
Nie możemy usunąć migracji, która została już zaaplikowana na bazę danych, w takim przypadku musimy przywrócić bazę do migracji poprzedzającej tą którą chcemy usunąć





Aplikowanie migracji

Aby zaaplikować migracje należy wykonać komendę:

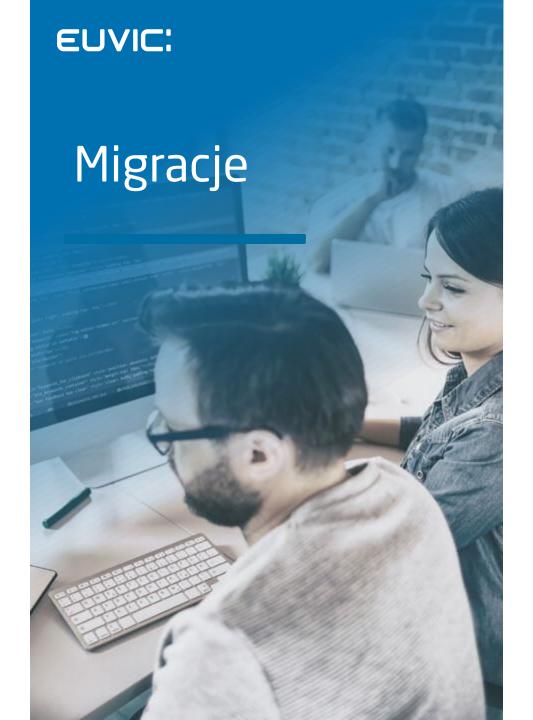


W przypdku gdy jest wiele contextów trzeba dodać parameter –Context

PM> Update-Database -Context StaffTrainingContext

Gdy potrzebujemy cofnąc bazę danych do jednej z poprzednich migracji należy wykonać

PM> Update-Database PreviousMigrationName -Context StaffTrainingContext





Automigracje

Poniżej przedstawiam extension method, które pozwala na automigracje na potrzeby lokalnego środowiska

```
public static async void Migrate<TDbContext>(this IApplicationBuilder applicationBuilder)
   where TDbContext : DbContext

{
   // AUTOMIGRATION ARE ALLOWED ONLY FOR LOCAL DEVELOPMENT

   var scopeFactory = applicationBuilder
        .ApplicationServices
        .GetRequiredService<IServiceScopeFactory>();

   using var scope = scopeFactory.CreateScope();
   using var dbContext = scope.ServiceProvider.GetRequiredService<TDbContext>();

   dbContext.Database.Migrate();
}
```

Uzycie wygląda następująco

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
   if (env.IsDevelopment())
   {
      app.Migrate<StaffTrainingContext>();
      app.Migrate<StaffTrainingReadonlyContext>();
      app.Migrate<IdentityContext>();
}
```



Migracje na środowisku

Automigracje nie nadają się do wykonania na środowisku z co najmniej 2 powodów:

- Gdy aplikacja będzie miala więcej niż 1 instacje migrację odpalą się w tym samym momencie i może się to kończyć błędem, ponieważ jedna z migracji będzie aplikowala ten sam skrypt SQL jako druga a poprzednia instancja zmieniła już bazę danych czego efektem będzie exception.
- Warto aby osoba aplikująca migracje świadomie przejrzała zmiany skryptu migracji, aby nie została zaaplikowana jakaś zmiana która ma nieoczekiwane konsekwencje na produkcji

Komenda wygenerowania skryptu migracji:

PM> Script-Migration --idempotent -Context StaffTrainingContext

Flaga –idempotent powoduje, że migracja która już istnieje na bazie danych zostanie pominięta





Wersjonowanie Migracji

- Po zaplikowaniu migracji na bazę danych stworzy się nam tabelka "__EFMigrationsHistory" która będzie zawierała historię zaaplikowanych migracji.
- Dzięki tej tabelce mamy kontrolę w jakiej wersji jest obecnie nasza baza danych i jakich migracji jeszcze brakuje

Jak to wygląda w bazie danych

| | MigrationId | Product Version |
|---|--|-----------------|
| 1 | 20220508192540_AddViewAttendeesSummary | 6.0.4 |
| 2 | 20220508192759_InitialMigration | 6.0.4 |



Osobny schemat dla MigrationHistory

- Bardzo polecam, aby stworzyć sobie osobne "MigrationHistory" dla każdego schematu bazy danych
- W naszym przypadku oddzieliłem "identity" od schematu "dbo"
- Aby stworzyć migration history dla innego schematu bazy danych należy dodać dodatkowy parameter do "UseSqlSever"





www.euvic.com