

PROJEKT MVC

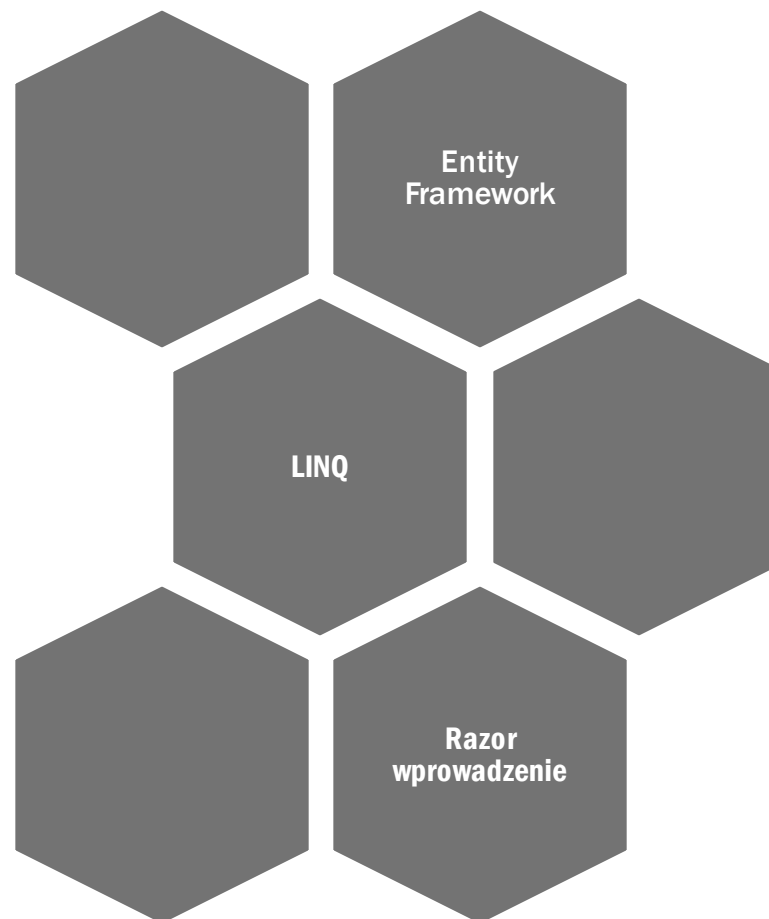
Programowanie wizualne

Wojciech Frohmberg, Instytut Informatyki, Politechnika Poznańska

2023



BAZA TECHNOLOGICZNA





ENTITY FRAMEWORK

ENTITY FRAMEWORK (EF)



Nowoczesny, łatwy w obsłudze
ORM



Transparentne śledzenie zmian w
modelach danych

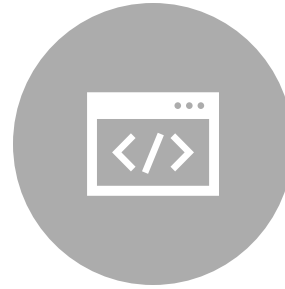


Automatyczne tworzenie oraz
aplikowanie migracji schematów
bazy danych



Integracja z LINQ oraz serwisami
MVC

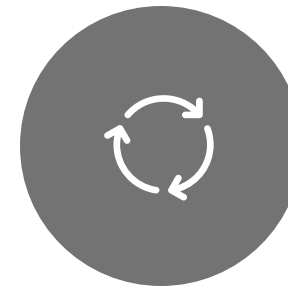
MOŻLIWE STRATEGIE ROZPOCZĘCIA PRACY Z EF



CODE FIRST

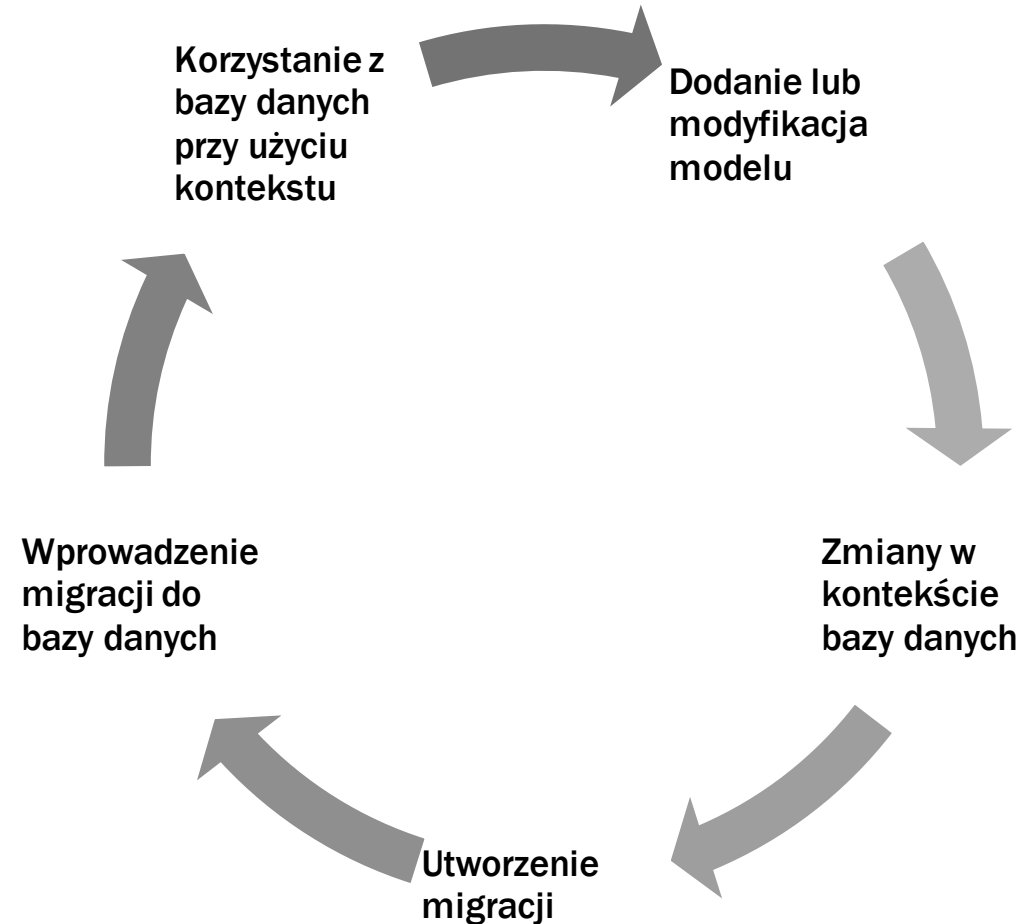


DATABASE FIRST



MODEL FIRST*

PROCES WDRAŻANIA ZMIAN W MODELACH W EF



ZAPŁON Z EF W KILKU KROKACH

1) TWORZENIE PROJEKTU MVC

W VISUAL STUDIO

1. Otwieramy Visual Studio
2. Wybieramy opcję Utwórz nowy projekt
3. Z dostępnych szablonów wybieramy "Aplikacja internetowa ASP.NET Core (Model-View-Controller)"
4. Wybieramy opcje korzystając z Wizarda.

WERSJA CLI (NA PRZYKŁADZIE VS CODE)

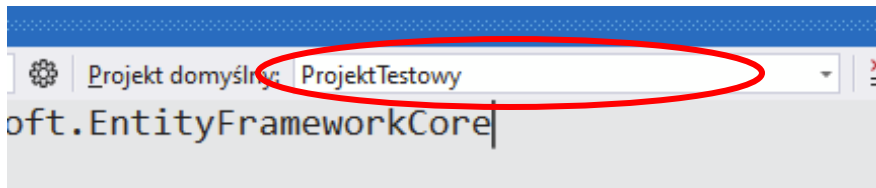
1. Otwieramy Visual Studio Code
2. Otwieramy konsolę (np. skrótem klawiszowym `<CTRL> ``)
3. Przechodzimy do folderu w którym chcemy umieścić folder naszego rozwiązania
4. Wpisujemy komendę:
`dotnet new sln --output MySolution`
5. Przechodzimy do folderu rozwiązania i wpisujemy:
`dotnet new mvc --name MyProject`

ZAPŁON Z EF W KILKU KROKACH

2) INSTALACJA PAKIETÓW

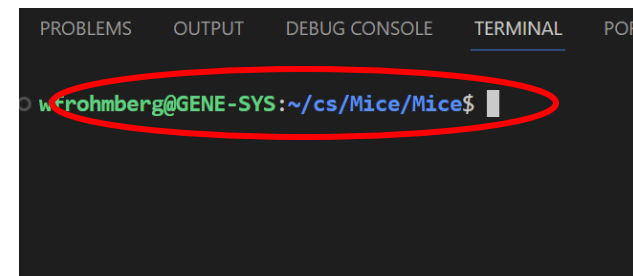
W VISUAL STUDIO (POWERSHELL)

1. Otwieramy konsolę menedżera pakietów (Narzędzia -> Menedżer pakietów NuGet -> Konsola menedżera pakietów lub skrót menu w wersji PL: <L ALT> M N K)
2. W konsoli wybieramy projekt, do którego chcemy dodać NuGet



WERSJA CLI

1. Otwieramy terminal (np. w VS Code skrótem klawiszowym <CTRL> `)
2. Upewniamy się że jesteśmy w folderze projektu do którego chcemy dodać NuGet



ZAPŁON Z EF W KILKU KROKACH

2) INSTALACJA PAKIETÓW CD.

W VISUAL STUDIO

3. Wpisujemy komendę Install-Package, po której wpisujemy pakiet, który chcielibyśmy zainstalować, tutaj:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Tools
- Microsoft.EntityFrameworkCore.Sqlite

WERSJA CLI

3. Wpisujemy komendy:

- dotnet add package Microsoft.EntityFrameworkCore
- dotnet add package Microsoft.EntityFrameworkCore.Design
- dotnet add package Microsoft.EntityFrameworkCore.Sqlite
- dotnet new tool-manifest
- dotnet tool install dotnet-ef

ZAPŁON Z EF W KILKU KROKACH

3) TWORZENIE KLASY MODELU

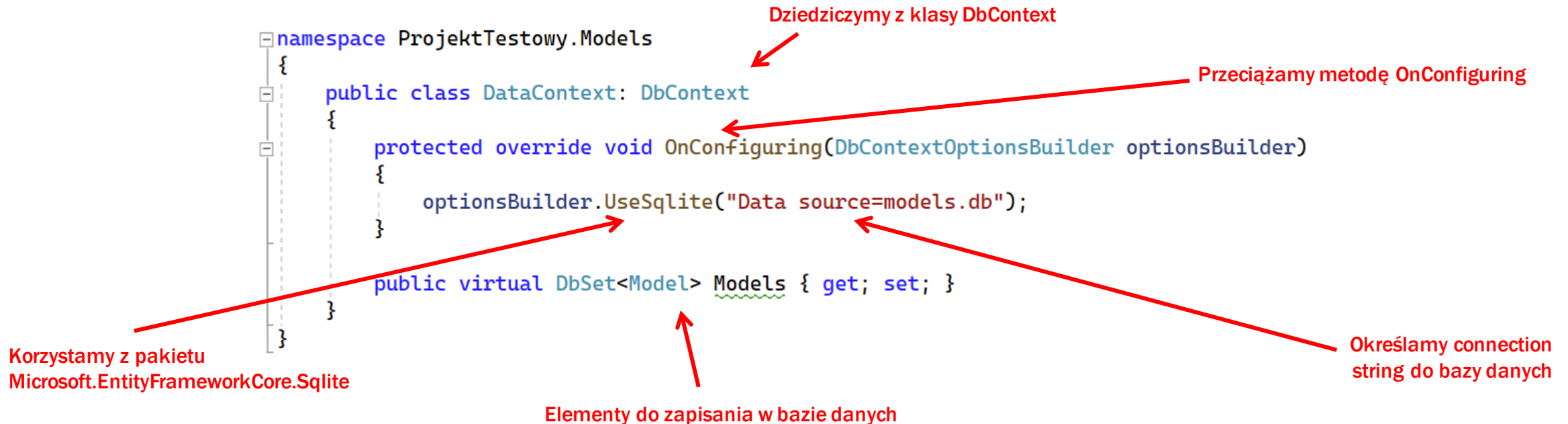
Tworzymy klasę(y) modelu. Może to być dowolna klasa z przynajmniej jedną właściwością, za pomocą której osiągniemy efekt unikalności wpisu do bazy danych:

```
namespace ProjektTestowy.Models
{
    public class Model
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
}
```

ZAPŁON Z EF W KILKU KROKACH

4) TWORZENIE KONTEKSTU BAZY

Dodajemy klasę kontekstu danych – klasa będzie odpowiadała za utrzymywanie kolekcji wszystkich modeli, które chcemy przechowywać w bazie danych.

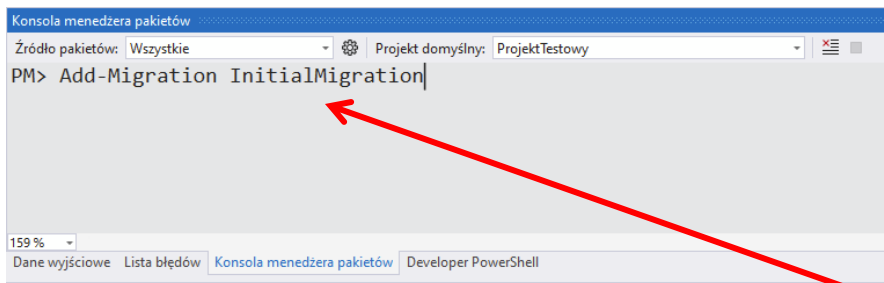


ZAPŁON Z EF W KILKU KROKACH

5) TWORZENIE MIGRACJI

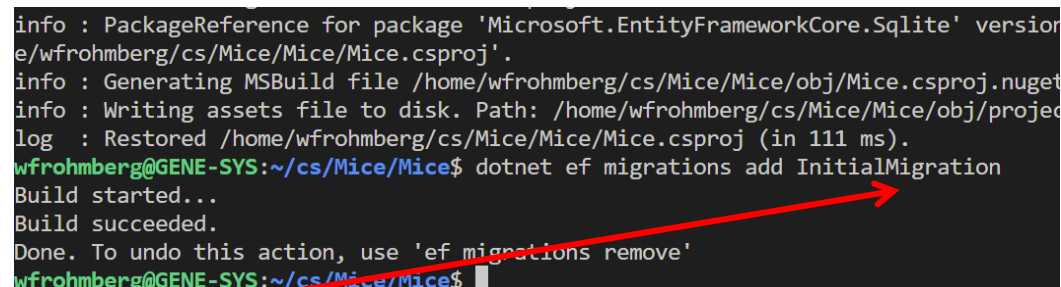
Korzystając z konsoli - tworzymy migrację kontekstu bazy danych. Migrację należy rozumieć jako sposób zmiany struktury bazy danych z poprzedniego stabilnego punktu na nowy tj. po dokonaniu jakichkolwiek strukturalnych w kodzie mające wpływ na przechowywanie danych. Przy czym zmiany, które chcielibyśmy odzwierciedlić to zarówno zmiany wynikające z dodania nowej klasy modelu, ale i te na poziomie istniejących modeli wynikające z modyfikacji klasy ich właściwości.

W Visual Studio (PowerShell)



```
Konsola menedżera pakietów
Źródło pakietów: Wszystkie Projekt domyślny: ProjektTestowy
PM> Add-Migration InitialMigration
```

Wersja cli (na przykładzie VS code)



```
info : PackageReference for package 'Microsoft.EntityFrameworkCore.Sqlite' version
e/wfrohmberg/cs/Mice/Mice/Mice.csproj'.
info : Generating MSBuild file /home/wfrohmberg/cs/Mice/Mice/obj/Mice.csproj.nuget
info : Writing assets file to disk. Path: /home/wfrohmberg/cs/Mice/Mice/obj/projec
log : Restored /home/wfrohmberg/cs/Mice/Mice/Mice.csproj (in 111 ms).
wfrohmb@GENE-SYS:~/cs/Mice/Mice$ dotnet ef migrations add InitialMigration
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
wfrohmb@GENE-SYS:~/cs/Mice/Mice$
```

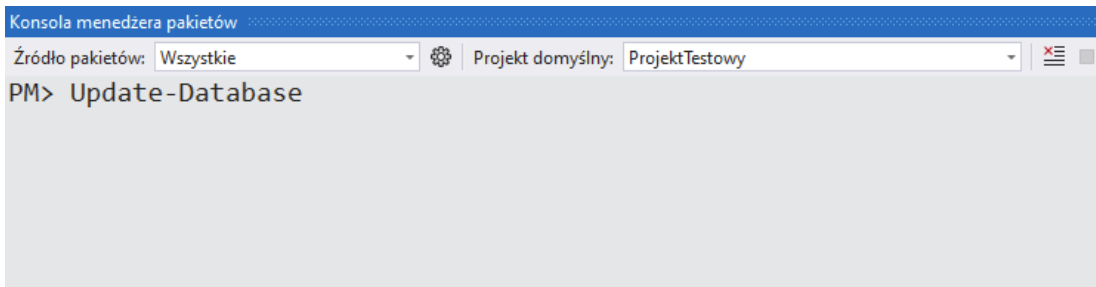
Wymagany parametr – nazwa migracji

ZAPŁON Z EF W KILKU KROKACH

6) UPDATE BAZY DANYCH

Ostatnim krokiem, który należy wykonać po dokonaniu migracji jest jej zaaplikowanie do bazy danych. Migracja bowiem stanowi tylko procedurę przejścia z jednej wersji struktury bazy do drugiej i póki nie zostanie zaaplikowana w bazie nie ma jej efektu.

W Visual Studio (PowerShell)



```
Konsola menedżera pakietów
Źródło pakietów: Wszystkie Projekt domyślny: ProjektTestowy
PM> Update-Database
```

Wersja cli (na przykładzie VS code)



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS 1
wfrohmb@GENE-SYS:~/cs/Mice/Mice$ dotnet ef database update
```


JAK KORZYSTAĆ Z MODELI EF

Po wprowadzeniu zmian w strukturze bazy chcielibyśmy móc się do niej odwołać. Do tego celu służy uprzednio utworzony kontekst danych. Żeby odwoływać się do elementów zgromadzonych w bazie tworzymy instancję kontekstu i operujemy na danych z wybranych zbiorów z danymi.

Odczyt danych

```
var context = new DataContext();  
foreach (var model in context.Models)  
{  
    Console.WriteLine($"{model.Id} {model.Name}");  
}
```

Dodawanie danych

```
var context = new DataContext();  
var model = new Model { Name = "Nowy model!" };  
context.Models.Add(model);  
context.SaveChanges();
```

POWIAZANIA MIĘDZY MODELAMI

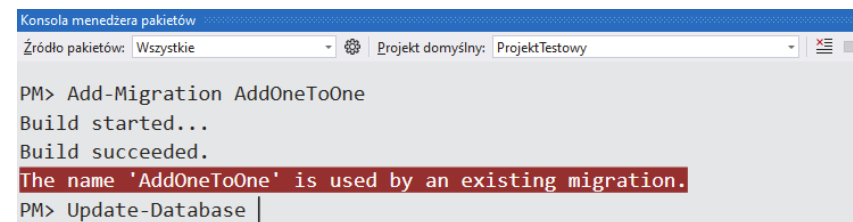
ASOCJACJE JEDEN DO JEDEN

Żeby zamodelować połączenia pomiędzy modelami wystarczy zapewnić dodanie modelu referowanego do zbioru modeli w kontekście danych oraz przechować referencję. Jeśli zależy nam na referowaniu w dwie strony musimy dodatkowo wyspecyfikować, w którą stronę asocjacja ma być przechowana poprzez jawne wyspecyfikowanie klucza.

```
public class SecondModel
{
    public int Id { get; set; }
    public string Name { get; set; }

    [ForeignKey("BaseModel")]
    public int BaseModelId { get; set; }
    public Model BaseModel { get; set; }
}
```

```
public class Model
{
    public int Id { get; set; }
    public string Name { get; set; }
    public SecondModel SecondModel { get; set; }
}
```



Konsola menedżera pakietów

Źródło pakietów: Wszystkie Projekt domyślny: ProjektTestowy

```
PM> Add-Migration AddOneToOne
Build started...
Build succeeded.
The name 'AddOneToOne' is used by an existing migration.
PM> Update-Database |
```

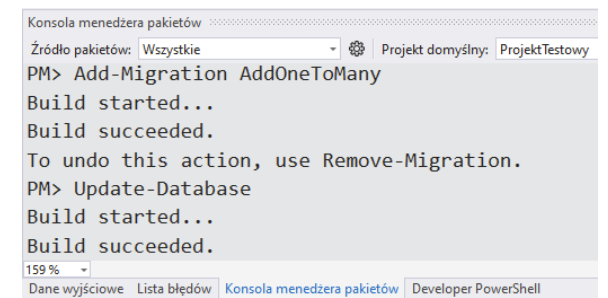
POWIAZANIA MIĘDZY MODELAMI

ASOCJACJE JEDEN DO WIELE

Asocjacje jeden do wiele w tym kontekście są mniej wymagające. Wystarczy w modelu, z którego ma zostać utworzona referencja utrzymać referencję pojedynczą na referowany model (lub tę referencję pominąć), w drugim modelu chcielibyśmy skorzystać z kolekcji referencji na ten pierwszy model (warto tą kolekcję przy okazji zainicjalizować).

```
public class Model
{
    public int Id { get; set; }
    public string Name { get; set; }
    public SecondModel SecondModel { get; set; }
}
```

```
public class SecondModel
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<Model> BaseModels { get; set; }
    = new List<Model>();
}
```



```
Konsola menedżera pakietów
Źródło pakietów: Wszystkie Projekt domyślny: ProjektTestowy
PM> Add-Migration AddOneToMany
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
159 %
Dane wyjściowe Lista błędów Konsola menedżera pakietów Developer PowerShell
```

POWIAZANIA MIĘDZY MODELAMI

ASOCJACJE WIELE DO WIELE

Połączenia wiele-do-wiele najpewniej już sami bylibyście w stanie zamodelować ale dla kompletności — wystarczy zatem przechować referującą do elementów alternatywnego modelu kolekcję w obu modelach.

```
public class Model
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<SecondModel> SecondModels { get; set; }
    = new List<SecondModel>();
}
```

```
public class SecondModel
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<Model> BaseModels { get; set; }
    = new List<Model>();
}
```

```
Konsola menedżera pakietów
Źródło pakietów: Wszystkie | Projekt domyślny: ProjektTestowy
PM> Add-Migration AddManyToMany
Build started...
Build succeeded.
An operation was scaffolded that may result in the loss of data.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
159 %
Dane wyjściowe Lista błędów Konsola menedżera pakietów Developer PowerShell
```

Więcej informacji o asocjacjach znajdziesz w pliku Asocjacje-EF-PW.pdf

DOŁADOWYWANIE REFERENCJI

METODA INCLUDE I MATERIALIZACJA DANYCH

Entity Framework jest z natury leniwy — korzysta z mechanizmu lazy loading i nie ładuje referencji jeśli nie ma pewności, że będą wykorzystywane. Żeby wymusić załadowanie danych, z których chcemy korzystać musimy zastosować metodę Include. Przy czym referowane kolekcje muszą być zainicjowane w ramach konstrukcji określonego obiektu modelu inaczej pomimo zastosowania metody Include kolekcja z referencjami nie będzie wypełniona.

```
var model = _context.Models
    .Include("SecondModels")
    .FirstOrDefault(m => m.Name == "Nowy model");
```


DEPENDENCY INJECTION (DI)

Entity Framework można wykorzystywać zupełnie niezależnie od typu projektu, którym się posługujemy. Możemy korzystać w ten wygodny sposób z zasobów danych zarówno w przypadku projektu biblioteki klas, aplikacji konsolowej, projektu Windows Forms, aplikacji WPF czy też projektu aplikacji webowej w tym zastosowanej przez nas MVC.

Często jednak w przypadku projektów wielowarstwowych chcielibyśmy oddzielić warstwę konfiguracji elementów składowych np. bazy danych od warstwy korzystania z tych elementów. Można w tym celu wykorzystać konstrukt składniowy zwany Dependency Injection, który ułatwia współpracę pomiędzy modułami i zapewnia transparentną architekturę komunikacyjną - poprzez zarejestrowane w aplikacji usługi zwane serwisami.

NA CZYM POLEGA DI?

Za pomocą serwisów na żądanie programisty tworzony lub reużyty jest obiekt określonego typu dokładnie w miejscu, w którym jest potrzebny. Dla przykładu z dependency injection możemy skorzystać w ramach kontekstu bazodanowego celem uzyskania konfiguracji naszej aplikacji:

```
public class DataContext: DbContext
{
    private readonly IConfiguration _configuration;

    public DataContext(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite(_configuration
            .GetConnectionString("DataContextConnectionString"));
    }

    public virtual DbSet<Model> Models { get; set; }
}
```

Do utworzenia kontekstu wymagamy podania konfiguracji w parametrze konstruktora

Korzystamy z konfiguracji w miejscu potrzeby pobrania ciągu ConnectionString

DODANIE SERWISU DO DI MVC

Serwisy zwyczajowo rejestrowane są w procedurze głównej aplikacji. Od tego z serwisem jakiego typu mamy do czynienia zależy wybór sposobu jego dodania np. serwis bazy danych dodawany jest przy użyciu metody `AddDbContext`, która w parametrze generycznym przyjmuje typ kontekstu bazy danych:

```
using ProjektTestowy.Models;


var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDbContext<DataContext>();
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline
```

Dodanie serwisu obsługującego tworzenie kontekstu bazy danych





LINQ

CZYM JEST LINQ?

LINQ to akronim od angielskiego Language-Integrated Query. Tradycyjne zapytania SQL nie najlepiej wpasowują się w składnię języków programowania, ponieważ zwyczajowo zapisuje się je w formie ciągów znaków. W takim podejściu nie mamy gwarancji poprawności składni zapytań dopóki nie uruchomimy i nie przetestujemy zapytania. Sprawę dodatkowo utrudnia fakt, iż część spośród zapytań jest dogenerowywana w trakcie działania programu w zależności od kontekstu i parametrów uruchomienia.

LINQ wychodzi naprzeciw tym problemom i oferuje język zapytań wbudowany w używany język programowania, co daje możliwość weryfikacji składni zapytania bezpośrednio przez kompilator danego języka!



SKŁADNIE LINQ

Query based syntax

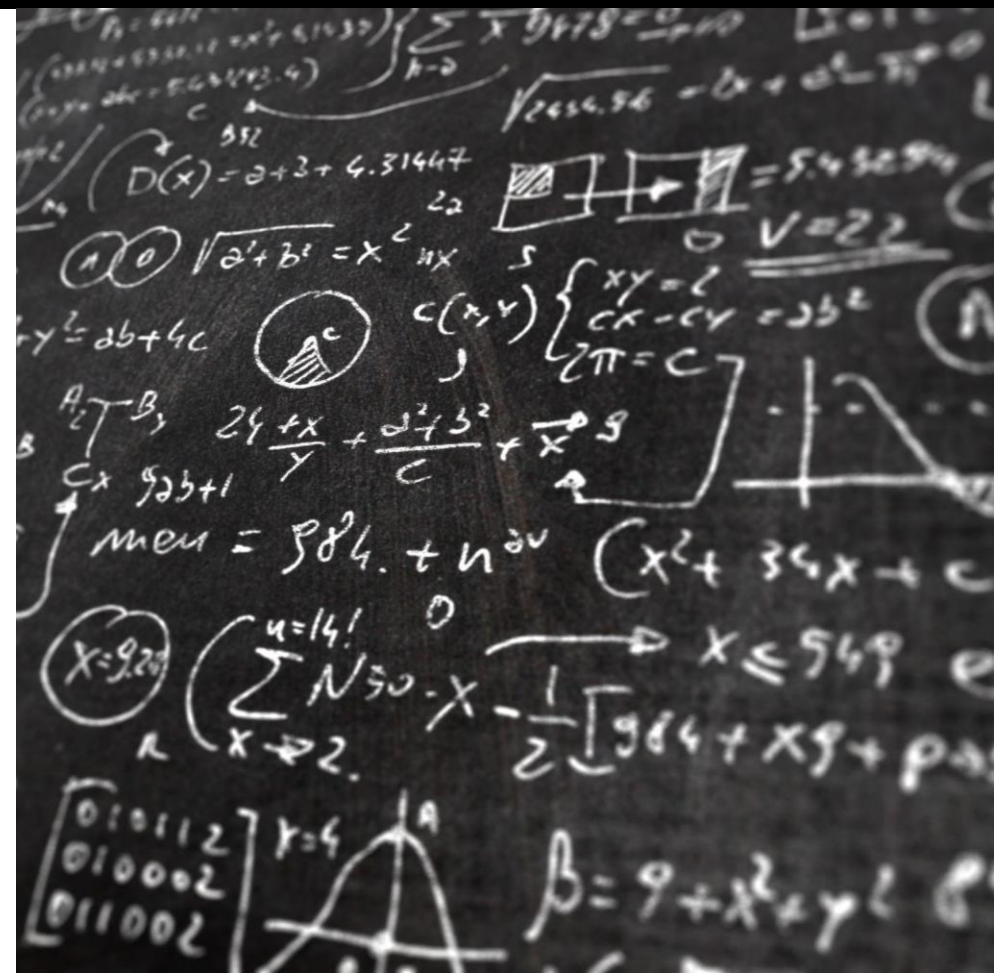
```
var query = from range_variable in enumerable_or_queryable
             where range_variable.field > 10
             select range_variable;
```

Method based syntax

```
var query = enumerable_or_queryable
             .Where(range_variable => range_variable.field > 10)
             .Select(range_variable => range_variable);
```

WARTO WIEDZIEĆ

Utworzenie zapytania LINQ nie jest równoznaczne z jego uruchomieniem i ewaluacją wyniku zapytania. Za pomocą kolejnych obiektów typu `Queryable` możemy "dobudowywać" właściwe zapytanie, które chcielibyśmy zaaplikować. Rzeczywiste zapytanie (np. SQL, gdy korzystamy z bazy danych) zostaje zmaterializowane dopiero w momencie zgłoszenia żądania uzyskania danych tj. podczas fizycznego przeiterowania po wyniku zapytaniu.



JAK DZIAŁA LINQ?

Niskopoziomowo LINQ opiera swoje działanie na mechanizmie Expression trees, za pomocą którego wnika w poszczególne wyrażenia przekazane do członów zapytania dokonując odpowiednich przekształceń np. tworząc kolejne fragmenty zapytania SQL.

```
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
    param.Name, left.Name, operation.NodeType, right.Value);
```

CA Konsola debugowania programu Microsoft Visual Studio

```
Decomposed expression: num => num LessThan 5
```

ELEMENTY SKŁADNI LINQ

KLAUZULA FROM

Query based syntax

Klauzula from – określa do jakiego iterowalnego obiektu referujemy i poprzez jaką zmienną zakresu, wielokrotnie użyta klauzula from daje możliwość połączenia różnowartościowego zbiorów.

```
var query = from range_variable in enumerable_or_queryable
            select range_variable;

var query = from range_variable in enumerable_or_queryable
            from other_variable in other_enumerable_or_queryable
            select new { R1 = range_variable, R2 = other_variable };
```

Method based syntax

Nie ma bezpośredniego odpowiednika dla klauzuli from w ramach tego podejścia. Do celu określenia źródłowego obiektu wywołujemy na tym obiekcie metodę. Z kolei do celu przeprowadzenia połączenia różnowartościowego używamy metody SelectMany.

```
var query = enumerable_or_queryable;

var query = enumerable_or_queryable
            .SelectMany(range_variable => other_enumerable_or_queryable,
                       (range_variable, other_variable) =>
                       new { R1 = range_variable, R2 = other_variable });
```

ELEMENTY SKŁADNI LINQ

KLAUZULA SELECT

Query based syntax

Klauzula select dokonuje projekcji wartości dla każdego z wpisu źródłowego elementu iterowalnego.

```
var query = from range_variable in enumerable_or_queryable
             select new
             {
                 Val1 = range_variable.field1,
                 Val2 = range_variable.field2
             };
```

Method based syntax

Odpowiednikiem klauzuli select jest metoda Select.

```
var query = enumerable_or_queryable.Select(range_variable =>
                                           new
                                           {
                                               Val1 = range_variable.field1,
                                               Val2 = range_variable.field2
                                           });
```


ELEMENTY SKŁADNI LINQ

KLAUZULA JOIN

Query based syntax

Klauzula join dokonuje połączenia równościowego z wyspecyfikowanym elementem iterowalnym przy użyciu określonej zmiennej zakresu po określonych i oddzielonych w ramach klauzuli on składowych (muszą one być oddzielone operatorem equals)

```
var query = from range_variable in enumerable_or_queryable
             join other_variable in other_enumerable_or_queryable
             on range_variable.field1 equals other_variable.field1
             select new
             {
                 Val1 = range_variable.field1,
                 Val2 = other_variable.field2,
             };
```

Method based syntax

Odpowiednikiem klauzuli join jest metoda Join.

```
var query = enumerable_or_queryable
             .Join(other_enumerable_or_queryable,
                  range_variable => range_variable.field1,
                  other_variable => other_variable.field2,
                  (range_variable, other_variable) =>
                      new
                      {
                          Val1 = range_variable.field1,
                          Val2 = range_variable.field2
                      });
```

ELEMENTY SKŁADNI LINQ

KLAUZULA JOIN Z GRUPOWANIEM

Query based syntax

Klauzula join może posłużyć do tego żeby przy okazji połączenia równościowego zagregować łączone elementy do elementu iterowalnego. W tym przypadku należy posłużyć się dodatkowo składnią into. Klauzula gwarantuje użycie elementu range_variable nawet jeśli nie ma on odpowiednika w elemencie połączeniowym (outer join)

```
var query = from range_variable in enumerable_or_queryable
            join other_variable in other_enumerable_or_queryable
            on range_variable.field1 equals other_variable.field1
            into g
            select new
            {
                Val1 = range_variable.field1,
                Group = g,
            };
```

Method based syntax

Odpowiednikiem klauzuli join z grupowaniem jest metoda GroupJoin.

```
var query = enumerable_or_queryable
    .GroupJoin(other_enumerable_or_queryable,
        range_variable => range_variable.field1,
        other_variable => other_variable.field2,
        (range_variable, g) =>
            new
            {
                Val1 = range_variable.field1,
                Group = g,
            });
```

ELEMENTY SKŁADNI LINQ

KLAUZULA GROUP BY

Query based syntax

Klauzula group by różni się od tej znanej z języka SQL tym, że przyjmuje dodatkowy parametr po słowie kluczowym group. W ramach parametru robimy projekcję elementów, które powinny znaleźć się w grupie. Dodatkowo przyna nam się jeszcze klauzula into, dzięki której nazywamy zmienną grupującą.

```
var query = from range_variable in enumerable_or_queryable
             group range_variable.field1 by range_variable.field2
             into g
             select new
             {
                 Val = g.Key,
                 Group = g
             };
```

Method based syntax

Odpowiednikiem klauzuli group by w składni opartej na metodach jest metoda GroupBy.

```
var query = enumerable_or_queryable
             .GroupBy(range_variable => range_variable.field2,
                     range_variable => range_variable.field1,
                     (key, g) => new { Val = key, Group = g });
```

ELEMENTY SKŁADNI LINQ

KLAUZULA ORDER BY

Query based syntax

Klauzula order by pozwala na zdeterminowanie kolejności wyników rosnącą (bądź malejącą) kolejnością elementów określonego wyrażenia

```
var query = from range_variable in enumerable_or_queryable
             orderby range_variable.field1,
                    range_variable.field2 descending
             select range_variable;
```

Method based syntax

Odpowiednikami klauzuli order by w składni opartej na metodach są metody OrderBy oraz kolejno wywoływane metody ThenBy. W przypadku wersji descending mamy tutaj odpowiednio dostępne również: OrderByDescending oraz ThenByDescending

```
var query = enumerable_or_queryable
             .OrderByDescending(range_variable => range_variable.field1)
             .ThenByDescending(range_variable => range_variable.field2);
```

ELEMENTY SKŁADNI LINQ

METODY AGREGUJĄCE

Przykładowe metody agregujące:

- Count
- Sum
- Average
- Min
- Max
- ...

The background of the entire image is a dense, repeating pattern. It features large, stylized white flowers with multiple layers of petals, interspersed with smaller, simpler white flowers and intricate swirling vine motifs. The pattern is set against a solid black background.

RAZOR WPROWADZENIE

CZYM JEST RAZOR?

Razor stanowi mieszaną składnię pozwalającą na osadzanie/wstrzykiwanie dynamicznych treści (bazując na języku C# lub VB) bezpośrednio w określone miejsca w HTML'u, którego kod staje się de facto szablonem na bazie którego ostateczny widok jest generowany. Ideą języka jest konstruowanie widoków materializowanych użytkownikowi przy minimalistycznym wkładzie programistycznym, sprowadzającym się co najwyżej do odniesienia do poszczególnych pól modelu, ewentualnie przeiterowaniu po określonej kolekcji.

Razor stanowi zatem uzupełnienie składni języków programowania C# oraz VB o możliwość template'owania tagów HTMLowych. Pozwala to na odseparowanie kwestii wyświetlania danych użytkownikowi przy jednoczesnym wykorzystaniu pojedynczej składni języka programowania a tym samym zachowaniu wzorca MVC.

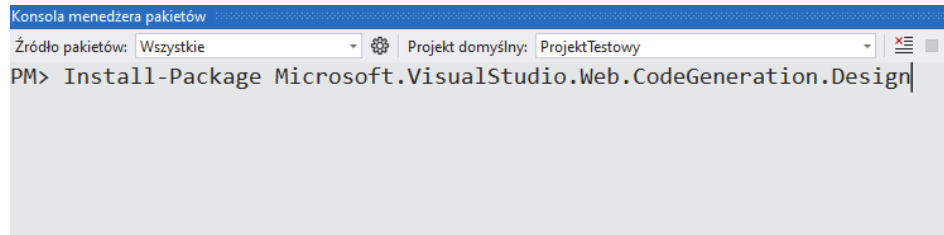
SCAFFOLDING WIDOKÓW

Siłą napędową Razora jest wielopoziomowe korzystanie ze wzorców. Oprócz tego, że sam kod napisany przy użyciu Razora stanowi wzorzec dokumentu HTML do zmaterializowania, istnieje silnik wzorcowania pozwalający na wygenerowanie gotowych szablonów takich wzorców bazujący bezpośrednio na modelach oraz kontekście bazodanowym. Wygenerowane widoki oczywiście są bardzo podstawowe ale wyglądają schludnie i są rozwijalne w wybranym przez użytkownika kierunku. Na dzisiejszych zajęciach skupimy się właśnie na możliwości łatwego automatycznego generowania widoków akcji CRUD (z ang. create, read, update, delete – utwórz, odczytaj, aktualizuj, usuń).

SCAFFOLDING JAK SKORZYSTAĆ?

INSTALOWANIE ZALEŻNOŚCI

W Visual Studio (PowerShell)



Wersja CLI

Musimy skorzystać z następujących komend:

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
```

```
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

```
dotnet tool install dotnet-aspnet-codegenerator
```

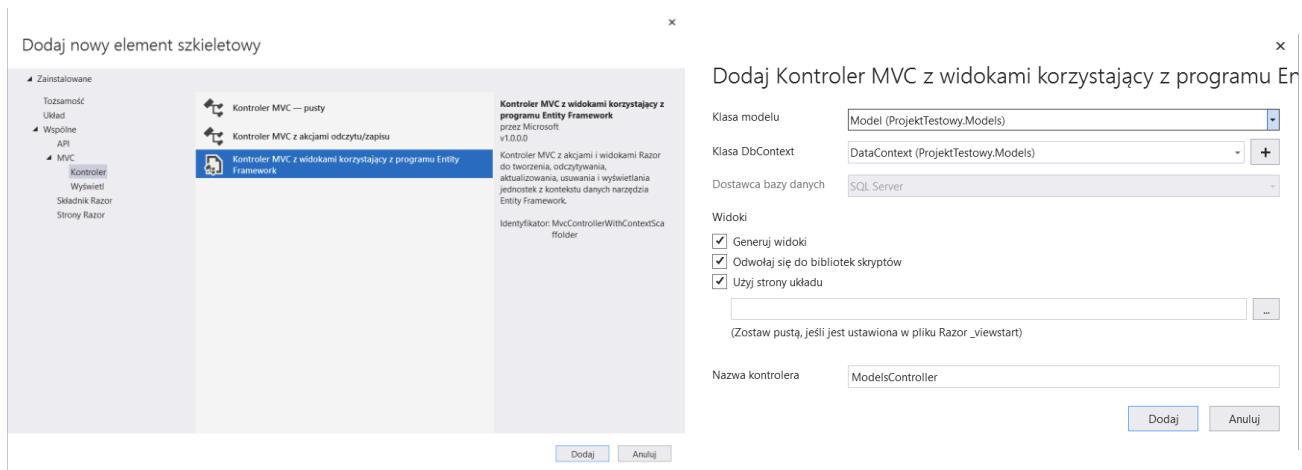
Warto przy tej komendzie upewnić się że mamy tool-manifest utrzymany w aktualnym folderze projektu (por. Slajd 9.)

SCAFFOLDING JAK SKORZYSTAĆ?

DODAWANIE NOWEGO KONTROLERA Z AKCJAMI CRUD

W Visual Studio

Najłatwiej nam będzie skorzystać z Wizarda i do folderu **Controllers** dodać nowy Kontroler przy użyciu wzorca "Kontroler MVC z widokami korzystający z programu Entity Framework"



Wersja CLI

Musimy skorzystać z następującej komendy:

```
dotnet aspnet-codegenerator controller  
--model Model  
-actions -async -useDefaultLayout  
--dbContext DataContext  
--databaseProvider 'sqlite'  
-name ModelsController  
-outDir Controllers
```