

PROJEKT MVC

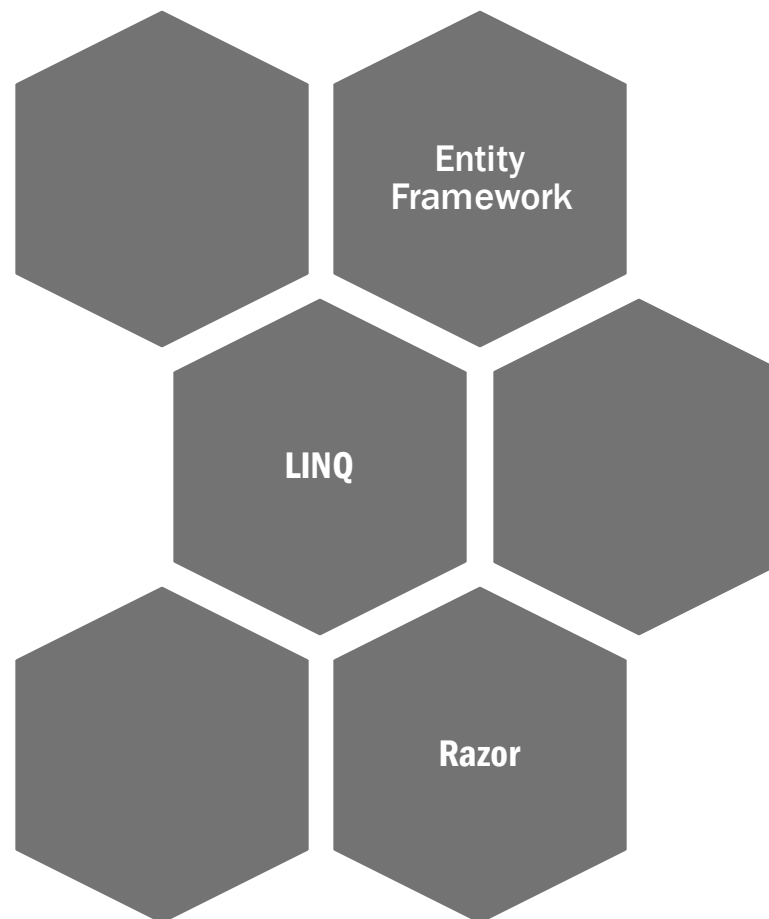
Programowanie wizualne

Wojciech Frohmberg, Instytut Informatyki, Politechnika Poznańska

2022



BAZA TECHNOLOGICZNA



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.

ENTITY FRAMEWORK

ENTITY FRAMEWORK (EF)



Nowoczesny, łatwy w obsłudze
ORM



Transparentne śledzenie zmian w
modelach danych

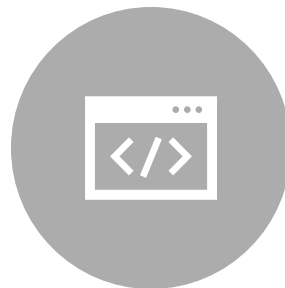


Automatyczne tworzenie oraz
aplikowanie migracji schematów
bazy danych



Integracja z LINQ oraz serwisami
MVC

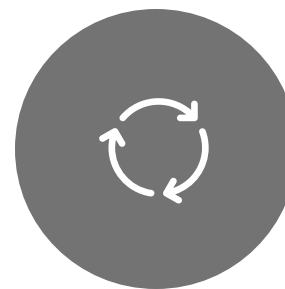
MOŻLIWE PODEJŚCIA KORZYSTANIA Z EF



CODE FIRST

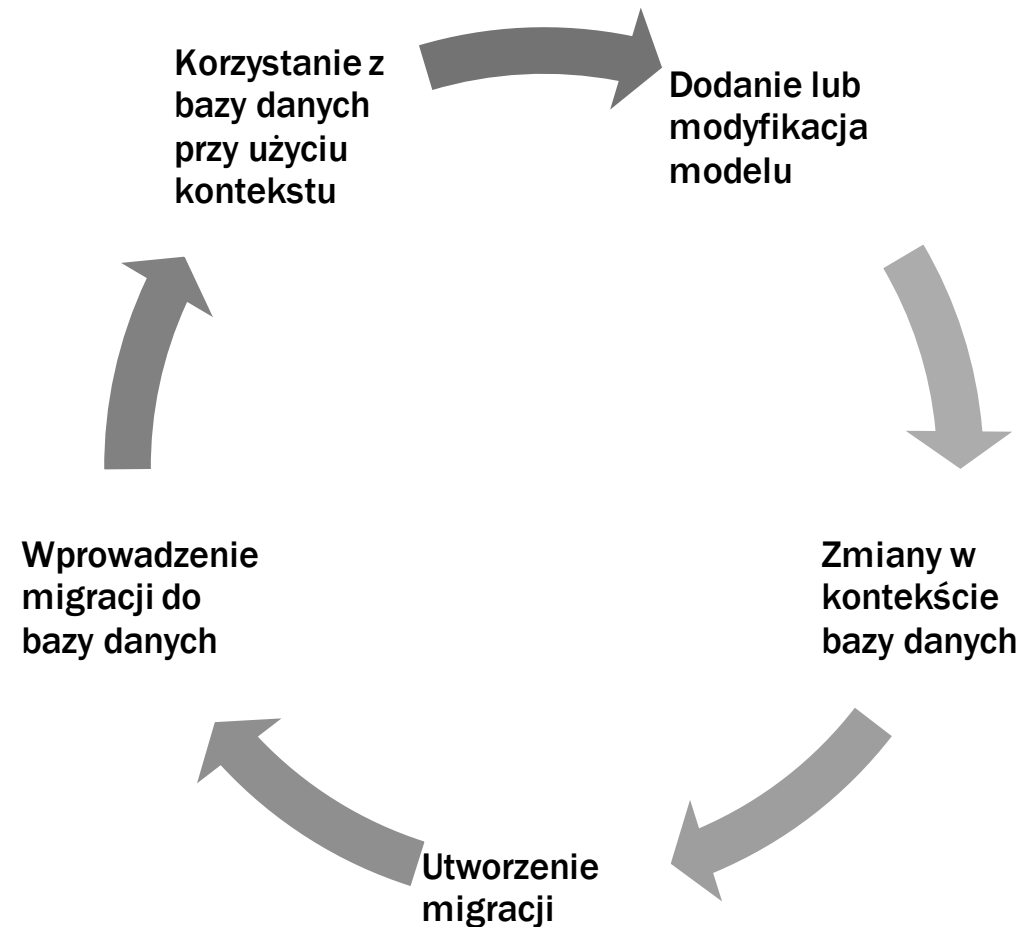


DATABASE FIRST



MODEL FIRST

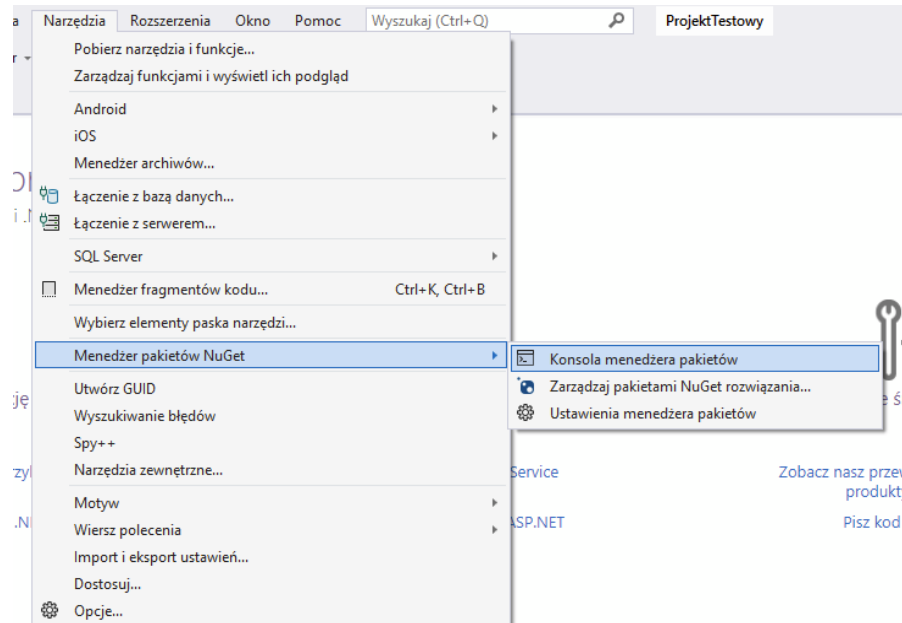
PROCES UŻYCIA EF (CODE FIRST)



JAK TO ZROBIĆ W VS?

OTWIERANIE KONSOLOWEGO MENEDŻERA PAKIETÓW

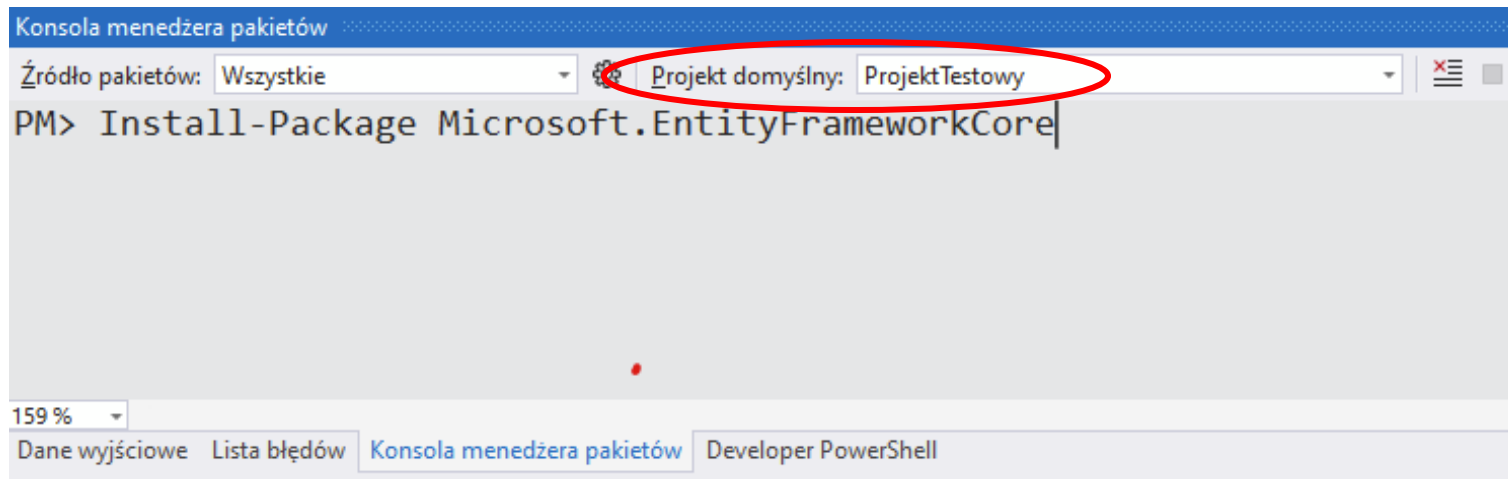
Najpierw musimy pobawić się z nugetami. Żeby to zrobić warto skorzystać z konsoli menadżera pakietów. Jeśli nie jest ona aktualnie uruchomiona można ją uruchomić z poziomu menu Narzędzia.



JAK TO ZROBIĆ W VS?

INSTALOWANIE PAKIETÓW

Po upewnieniu się, że konsola operuje na projekcie, w którym chcemy używać Entity Framework instalujemy niezbędne pakiety tj.: Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.Tools oraz Microsoft.EntityFrameworkCore.Sqlite korzystając przy tym z komendy Install-Package:



JAK TO ZROBIĆ W VS?

KLASA 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; }
    }
}
```

JAK TO ZROBIĆ W VS?

KONTEKST DANYCH

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

```
namespace ProjektTestowy.Models
{
    public class DataContext: DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Data source=models.db");
        }

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

Korzystamy z pakietu
Microsoft.EntityFrameworkCore.Sqlite

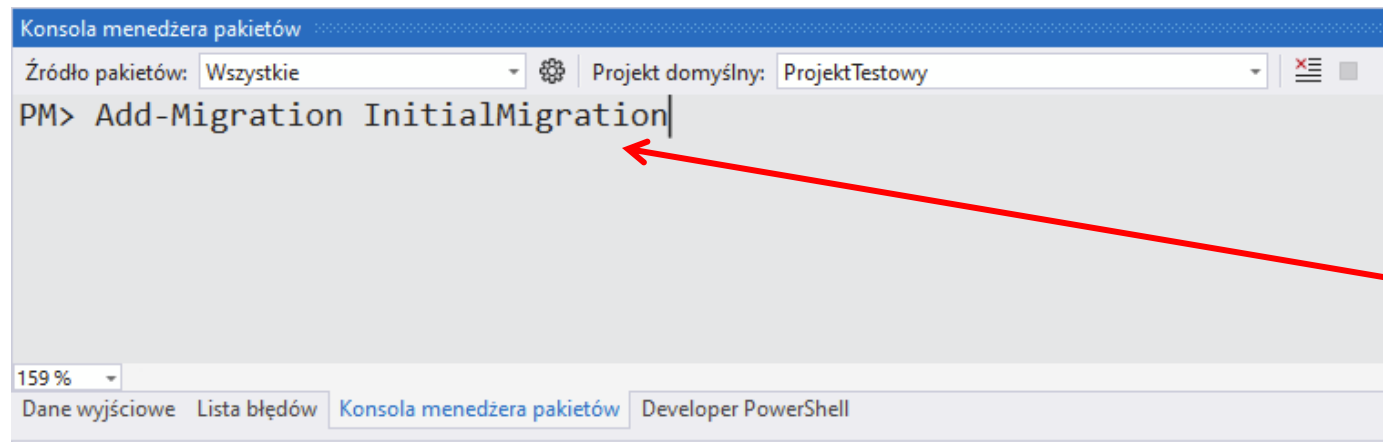
Elementy do zapisania w bazie danych

Określamy plik,
w którym powinna być przechowywana baza danych

JAK TO ZROBIĆ W VS?

TWORZENIE MIGRACJI

Korzystając z konsoli menedżera pakietów (oraz z tego, że uprzednio doinstalowaliśmy pakiet Tools EF) - 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.



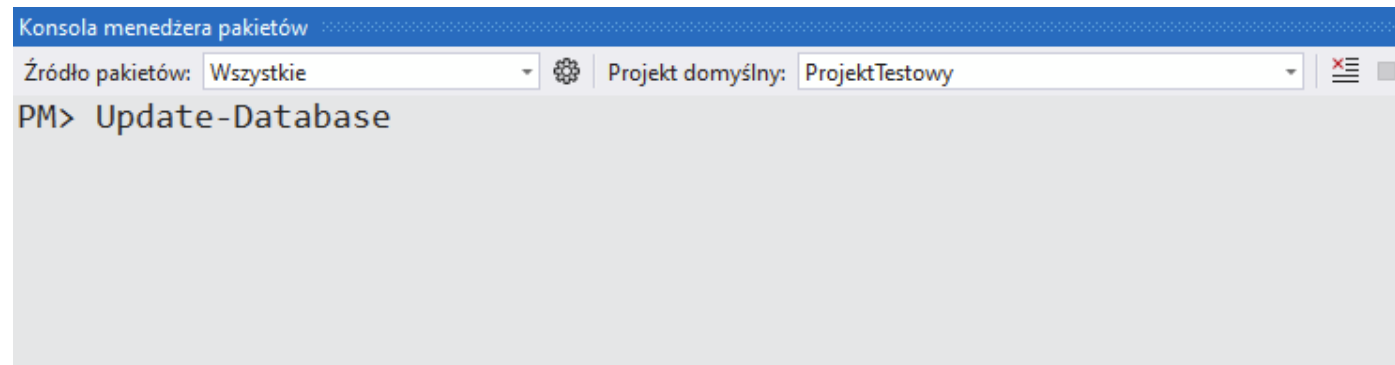
```
Konsola menedżera pakietów
Źródło pakietów: Wszystkie Projekt domyślny: ProjektTestowy
PM> Add-Migration InitialMigration
```

Wymagany parametr – nazwa migracji

JAK TO ZROBIĆ W VS?

APLIKOWANIE ZMIAN DO BAZY

Ostatnim krokiem, który należy wykonać po dokonaniu migracji jest jej zaaplikowanie do bazy danych. Migracja bowiem stanowi tylko procedurę przejścia z jednego punktu strukturalnego do drugiego i póki nie zostanie zaaplikowana w bazie nie ma jej efektu. Celem zaaplikowania zmian należy skorzystać z komendy Update-Database.



JAK TO ZROBIĆ W VS?

KORZYSTANIE Z BAZY DANYCH

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();
```

JAK TO ZROBIĆ W VS?

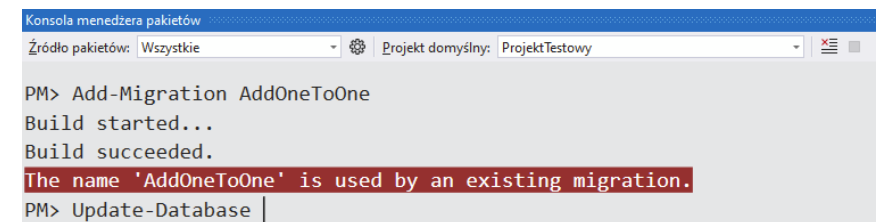
RELACJE JEDEN DO JEDEN

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 relacja 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 |
```

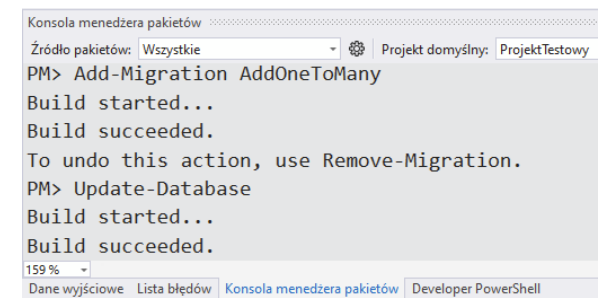
JAK TO ZROBIĆ W VS?

RELACJE JEDEN DO WIELE

Relacje 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>();
}
```



JAK TO ZROBIĆ W VS?

RELACJE 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
```


JAK TO ZROBIĆ W VS?

METODA INCLUDE I MATERIALIZACJA DANYCH

Entity Framework jest z natury leniuchem — 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. Warto zwrócić uwagę, że EF jest na tyle leniwy, że nie zainicjalizuje za nas kolekcji pomimo użycia metody Include, stąd ręczne inicjalizowanie kolekcji w modelach jest niezbędne.

```
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 np. 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 konstruować składniowy zwany dependency injection, który ułatwia współpracę pomiędzy modułami i zapewnia transparentną architekturę komunikacyjną.

NA CZYM POLEGA DI?

Dependency injection wykorzystuje globalną strukturę do utrzymywania tzw. serwisów. 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 użycia przekazania ciągu ConnectionString

DODANIE SERWISU DO DI MVC

Serwisy zwyczajowo inicjalizowane 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

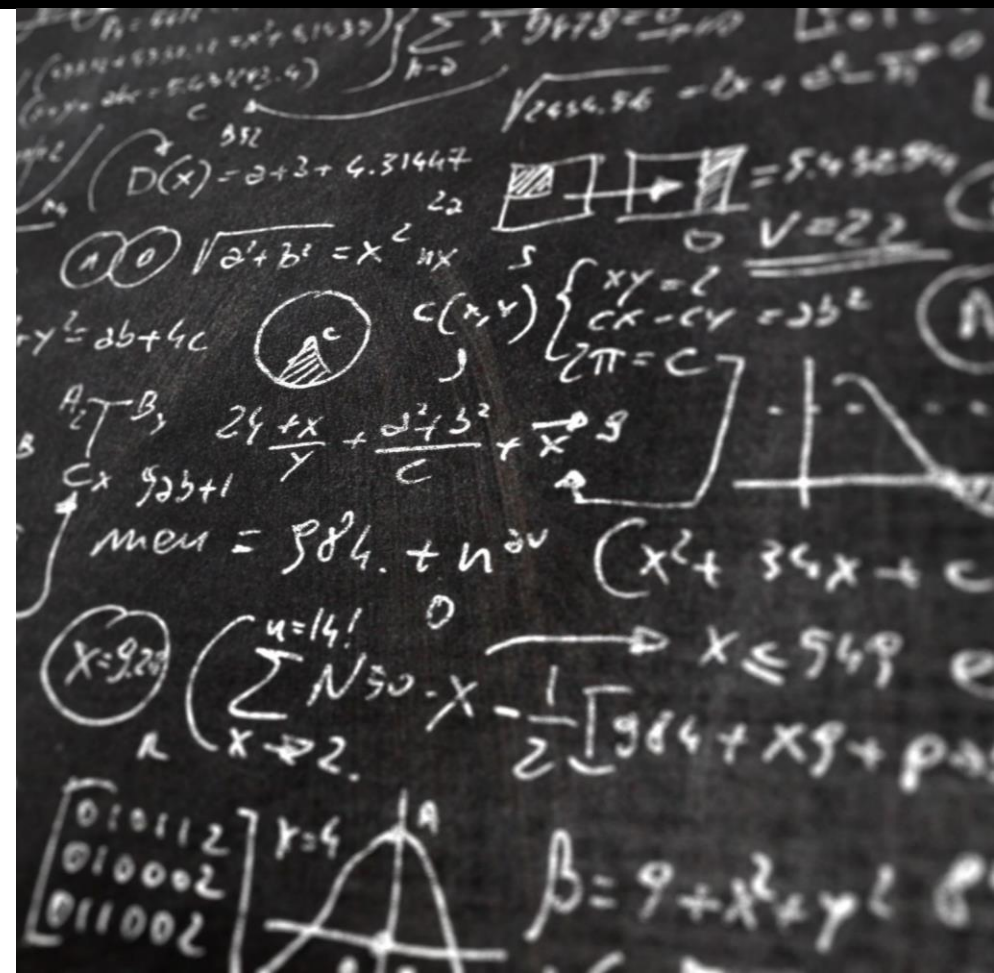
```
var query = from range_variable in enumerable_or_queryable
             where range_variable.field > 10
             select range_variable;
```

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 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 flowers with multiple layers of petals, interspersed with smaller, simpler circular floral motifs. Swirling lines and small leaf-like shapes fill the spaces between the larger flowers. The pattern is rendered in white and black, creating a high-contrast, graphic effect.

RAZOR

CZYM JEST RAZOR?

Razor stanowi mieszaną składnię pozwalającą na osadzanie/wstrzykiwanie dynamicznych treści (bazując na języku C# lub VB) bezpośrednio w template HTML'owy. 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 pozwalające na odcięcie kwestii widoków od kontrolerów obsługujących żądania użytkowników w kontekście wzorca projektowego Model-View-Controller (MVC).

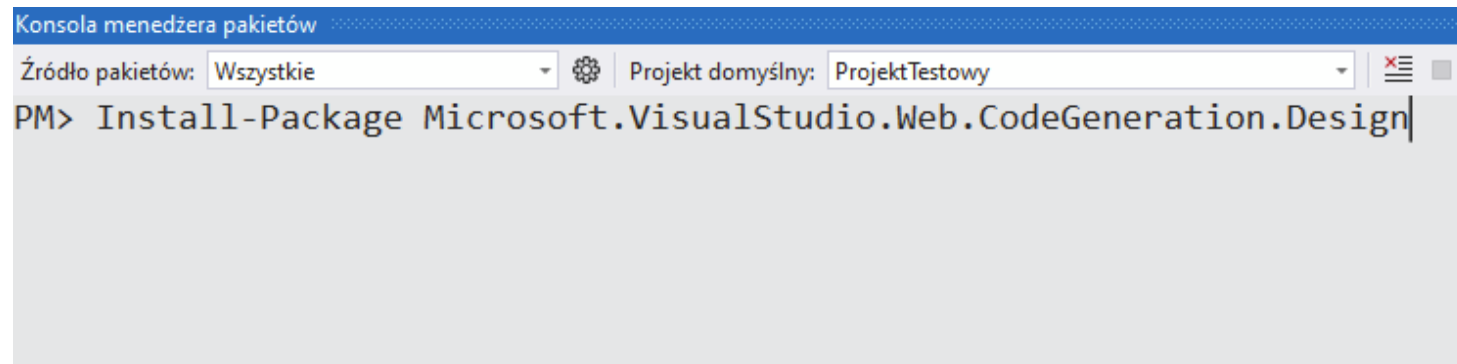
SCAFFOLDING WIDOKÓW

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 wygenerować gotowe podstawowe, ale schludne widoki do podstawowych akcji, które chcielibyśmy udostępnić w ramach nieomal każdego kontrolera modelu danych tj. tzw. akcji CRUD (z ang. create, read, update, delete – utwórz, odczytaj, aktualizuj, usuń)

SCAFFOLDING JAK SKORZYSTAĆ?

INSTALOWANIE ZALEŻNOŚCI

Żeby skorzystać się ze scaffoldingu nie obędzie się bez użycia menedżera pakietów. Potrzebny nam będzie nuget `Microsoft.VisualStudio.Web.CodeGeneration.Design`

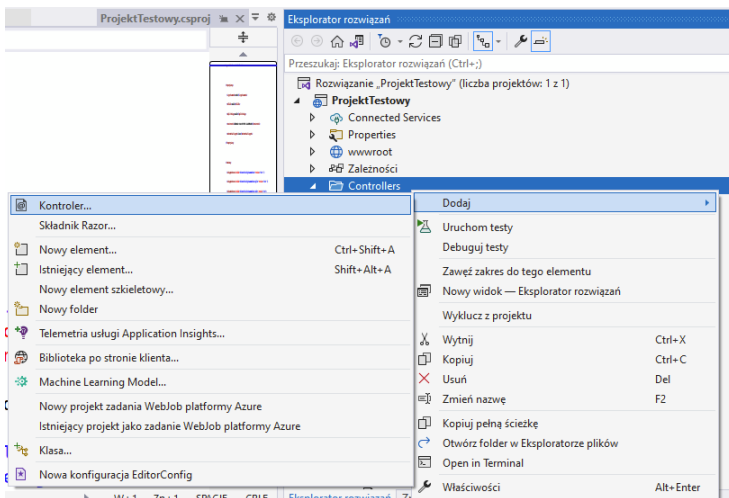


The screenshot shows the 'Konsola menedżera pakietów' (Package Manager Console) window in Visual Studio. At the top, there are two dropdown menus: 'Źródło pakietów:' (Package source) set to 'Wszystkie' (All) and 'Projekt domyślny:' (Default project) set to 'ProjektTestowy'. Below these, the command `PM> Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design` is entered into the console area.

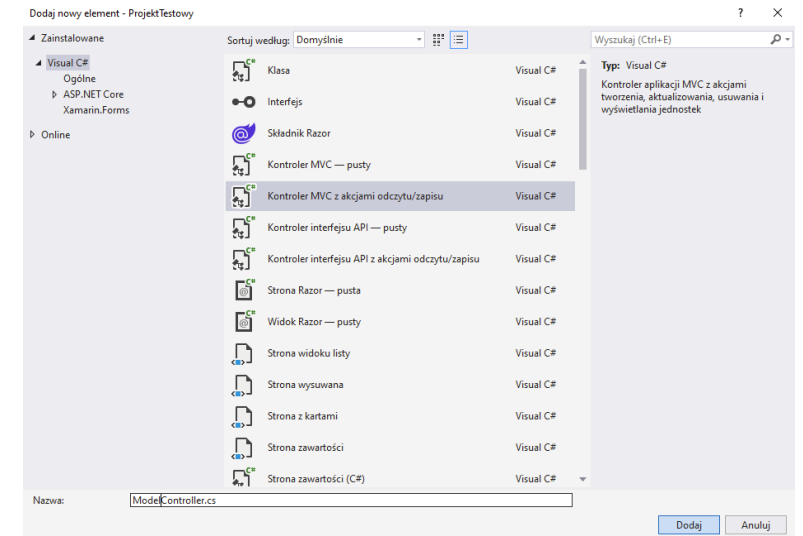
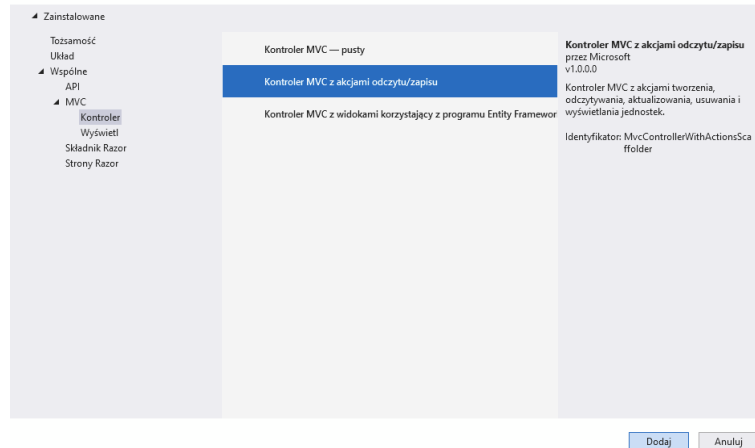
SCAFFOLDING JAK SKORZYSTAĆ?

DODAWANIE NOWEGO KONTROLERA Z AKCJAMI CRUD

Najpierw musimy zapewnić sobie kontroler, który udostępniłby akcje których wynikiem powinno być zmaterializowanie wzorca widoku:



Dodaj nowy element szkieletowy



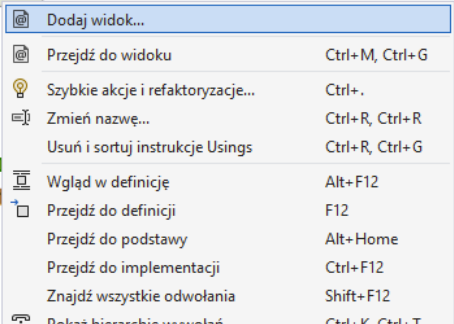
SCAFFOLDING JAK SKORZYSTAĆ?

DODAWANIE WIDOKU DO AKCJI (1)

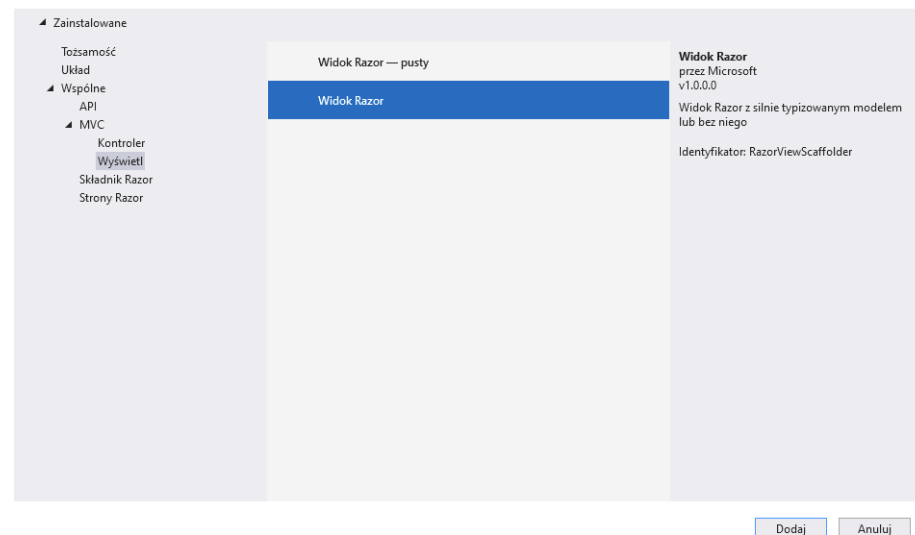
W ramach kontrolera możemy teraz wybrać akcję dla której widok chcielibyśmy wygenerować

```
namespace ProjektTestowy.Controllers
{
    public class ModelController : Controller
    {
        // GET: ModelController
        public ActionResult Index()
        {
            return View();
        }

        // GET: ModelController
        public ActionResult Detajl()
        {
            return View();
        }
    }
}
```



Dodaj nowy element szkieletowy



SCAFFOLDING JAK SKORZYSTAĆ?

DODAWANIE WIDOKU DO AKCJI (2)

Ostatnim krokiem jest dobranie odpowiednich parametrów scaffoldingu – szablonu z którego chcielibyśmy skorzystać (dostępne są: Create, Delete, Details, Edit, List), modelu, dla którego generujemy widok czy kontekstu danych przy użyciu, którego model jest składowany w bazie.

×

Dodaj Widok Razor

Nazwa widoku

Index

Szablon

List

Klasa modelu

Model (ProjektTestowy.Models)

Klasa kontekstu danych

DataContext (ProjektTestowy.Models)

Opcje

☐ Utwórz jako widok częściowy

☒ Odwołaj się do bibliotek skryptów

☒ Użyj strony układu

...

(Zostaw pustą, jeśli jest ustawiona w pliku Razor _viewstart)

Dodaj

Anuluj

A black and white photograph featuring a microphone on a stand in the foreground, angled towards the right. To the left, a portion of a keyboard is visible. The background is blurred, showing what appears to be a person's hands and other equipment. The text "LIVE DEMO" is overlaid in the center in a bold, white, sans-serif font.

LIVE DEMO