
# C# 6.0 i MVC 5

## Tworzenie nowoczesnych portali internetowych

Helion

Autorzy:
Krzysztof Żydzik, Microsoft Certified Solutions Developer, Microsoft Certified Professional, Microsoft Specialist
Tomasz Rak, Politechnika Rzeszowska, Wydział Elektrotechniki i Informatyki, Katedra Informatyki i Automatyki, Rzeszów, Polska



Redaktor prowadzący: Michał Mrowiec

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/c6mvc5*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe kolejnych kroków tworzenia przykładowej aplikacji dostępne są pod adresem:
*ftp://ftp.helion.pl/przyklady/c6mvc5.zip*

ISBN: 978-83-246-9496-9



Printed in Poland.

# Spis treści

Rozdział 4.

# Entity Framework 6

Entity Framework (EF) jest odpowiedzialny za kontakt aplikacji z bazą danych. Domyślne operacje wykonywane na bazie danych to operacje CRUD (ang. *Create*, *Read*, *Update*, *Delete*). Dzięki EF i Scaffolding (automatyczny generator kodu) można automatycznie wygenerować kod odpowiedzialny za operacje CRUD dla każdej tabeli z bazy danych.

## Podejście do pracy z modelem danych

EF udostępnia trzy podejścia do pracy z warstwą dostępu do danych DAL (ang. *Data Access Layer*):

- *Model First* — tworzy się diagram ERD (ang. *Entity-Relationship Diagram*) w Visual Studio za pomocą Model Designera i na podstawie diagramu generowana jest baza danych (kod SQL) oraz wszystkie klasy z modelem danych, po jednej dla każdej tabeli.
- *Code First* — tworzy się samodzielnie klasy (odpowiadające tabelom w bazie danych) i na podstawie klas generowana jest baza danych (kod SQL); dodatkowo, aby uaktualnić bazę danych po zmianach wprowadzonych w plikach z klasami, wykorzystuje się migracje (aktualizacje bazy danych).
- *Database First* — klasy i diagram ERD są generowane lub uaktualniane na podstawie istniejącej bazy danych.

### Porównanie różnych podejść

*Model First* pozwala na szybkie utworzenie bazy danych poprzez wstawianie elementów w edytorze graficznym, ale nie daje takiej kontroli nad kodem jak *Code First*. W przypadku korzystania z *Code First* trzeba ręcznie tworzyć pliki z klasami, co powoduje, że można się łatwo pomylić. *Database First* z kolei jest wykorzystywany w przypadku, gdy tworzy się aplikację na podstawie już istniejącej bazy danych. Dodatkowo po wybraniu bazy danych, na podstawie której chce się utworzyć projekt, wybiera się, czy w dalszej części zostanie użyte podejście *Code First*, czy *Model First*. W zależności od wyboru utworzone zostaną klasy (*Code First*) lub diagram (*Model First*).

Największą kontrolę nad kodem daje podejście *Code First*. Dla początkujących podejście *Model First* będzie bardziej intuicyjne, ponieważ baza jest prezentowana w formie diagramu ERD.

# Model dla podejścia Model First

Model danych dla podejścia *Model First* jest zawarty w pliku *.edmx*. Domyślnie model jest prezentowany jako diagram ERD w Entity Designerze (narzędzie w Visual Studio do tworzenia diagramu ERD).

Na pełny model danych w podejściu *Model First* składają się trzy części:

- model konceptualny (ang. *Conceptual Schema Definition Language*),
- model źródła danych (ang. *Storage Schema Definition Language*),
- część odwzorowująca (ang. *Mapping Specification Language*).

Wszystkie części znajdują się w jednym pliku *.edmx*. Jest to plik XML. Na podstawie pliku *.edmx* generowany jest schemat bazy danych — diagram ERD.

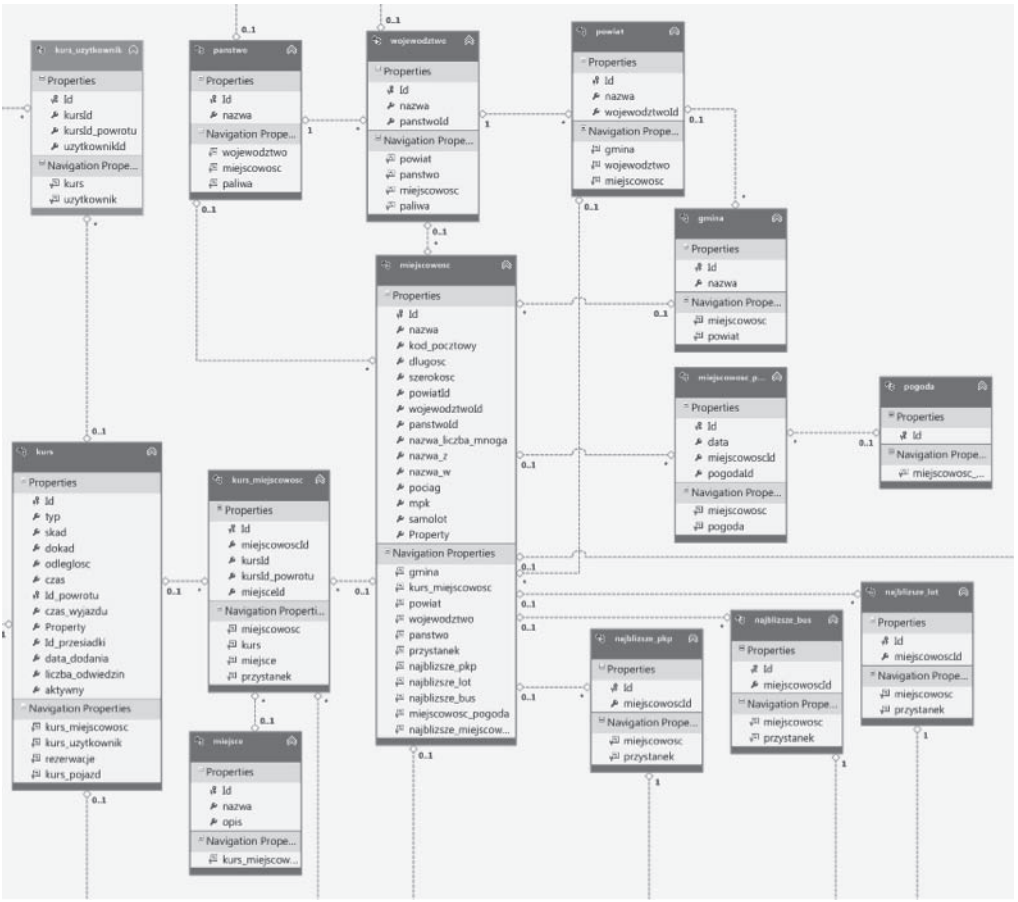
Przykładowy schemat bazy danych utworzony w Entity Designerze prezentuje rysunek 4.1.

# Model dla podejścia Code First

Przykładowa klasa z modelem dla podejścia *Code First* znajduje się na listingu 4.1. Jest to klasa POCO (ang. *plain-old CLR objects*), czyli klasa, która odwzorowuje strukturę tabeli z bazy danych na klasę w podejściu obiektowym.

**Listing 4.1.** *Przykładowa klasa POCO*

```
public partial class Kurs
{
    public Kurs()
    {
        this.KursUser = new HashSet<KursUser>();
    }
    public int Id { get; set; }
    public string Skad { get; set; }
    public string Dokad { get; set; }
    public System.DateTime DataWyjazdu { get; set; }
    public System.DateTime DataDodania { get; set; }

    public virtual ICollection<KursUser> KursUser { get; set; }
}
```




**Rysunek 4.1.** *Diagram ERD z Entity Designera*

# Nowości wprowadzane w kolejnych wersjach EF

## Nowości wprowadzone w EF 5

Najważniejsze nowości wprowadzone w EF 5 to:

- możliwość tworzenia wielu diagramów dla jednego modelu (*Model First*);
- wsparcie dla typu wyliczeniowego `enum` w *Code First* i *Model First*;
- wsparcie dla typów danych przestrzennych (ang. *spatial data types*) w *Code First* i *Model First*;

- TVF (ang. *Table-Valued Functions*) w *Model First* — podobne do procedur składowanych, z jedną małą różnicą — dane z TVF mogą być użyte w zapytaniach LINQ, natomiast dane z procedury składowanej nie.

## Nowości wprowadzone w EF 6

Najważniejsze nowości wprowadzone w EF 6 to:

- własne konwencje w *Code First*;
- poprawiona wydajność dla dużych modeli w podejściu *Code First*;
- migracje dla wielu kontekstów do jednej bazy danych;
- automatyczne ponawianie transakcji (ang. *Connection Resiliency*), w przypadku gdy zakończy się ona niepowodzeniem;
- tworzenie nowych kontekstów dla otwartego połączenia;
- poprawione wsparcie dla transakcji;
- wstrzykiwanie implementacji w czasie działania programu (ang. *Dependency Resolution*) oraz wsparcie dla wzorców *Service Locator* i *IoC*;
- łatwe przechwytywanie logów (ang. *Interception/SQL Logging*), zapytań SQL generowanych przez EF.

# Relacyjne bazy danych i EF

## Krótki opis baz relacyjnych

Bazy relacyjne powstały w czasach, gdy przestrzeń dyskowa była dość dużym ograniczeniem dla programów. Dyski miały pojemności liczone w megabajtach, a nie — jak obecnie — w terabajtach. Aby oszczędzić miejsce na dysku, dane musiały być przechowywane w oszczędny sposób bez duplikacji wpisów. Relacyjne bazy danych składają się z tabel, pomiędzy którymi zachodzą relacje. Każda tabela składa się z wierszy i kolumn. Na każdy wiersz składają się jednakowo ułożone kolumny wypełnione wartościami, które z kolei w każdym wierszu mogą być inne. Każdy wiersz ma swój unikalny numer zwany kluczem podstawowym PK (ang. *Primary Key*), po którym można rozpoznać konkretny wiersz danych (np. konkretnego pracownika z tabeli *Pracownik*). Aby powiązać wiersz z jednej tabeli z wierszem znajdującym się w innej tabeli, należy umieścić w niej klucz obcy FK (ang. *Foreign Key*). Jest to ta sama wartość, która w pierwszej tabeli jest kluczem podstawowym (PK). Druga tabela może np. zawierać miejsce zamieszkania pracownika, które może być wspólne dla kilku pracowników. Dzięki takiemu rozwiązaniu zostanie utworzona relacja pomiędzy tabelami. Istnieje kilka rodzajów relacji.

# Relacja „jeden do wielu”

Relacja „jeden do wielu” polega na tym, że do jednego wiersza w tabeli może być przypisanych kilka wierszy z innej tabeli. Przykładowo jeden klient może mieć kilka zamówień. Klucz obcy znajduje się w tabeli z zamówieniem i ma taką samą wartość jak klucz podstawowy z tabeli klienta. Po kluczu obcym można odczytać więcej danych na temat klienta, który złożył zamówienie, dzięki czemu nie trzeba do każdego zamówienia zapisywać tych samych danych pojedynczego klienta. Taka operacja odczytu dodatkowych danych z drugiej tabeli nazywana jest „joinem”. Na listingu 4.2 przedstawiono przykładową relację „jeden do wielu” dla podejścia *Code First*, a na rysunku 4.2 zaprezentowano tę samą relację w podejściu *Model First*.

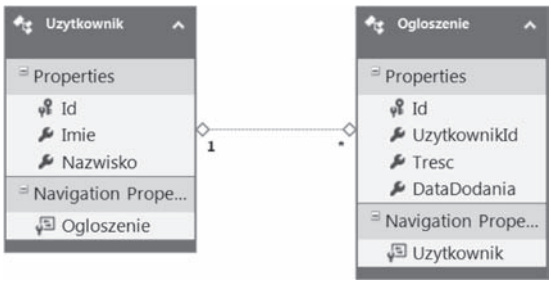
**Listing 4.2.** *Przykład relacji „jeden do wielu” w podejściu Code First*

```
public class Ogloszenie
{
    public Ogloszenie()
    {
    }
    public int Id { get; set; }
    public string Tresc { get; set; }
    public System.DateTime DataDodania { get; set; }

    public string UzytkownikId { get; set; }

    public Uzytkownik Uzytkownik { get; set; }
}

public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Ogloszenie = new HashSet<Ogloszenie>();
    }
    public int Id { get; set; }
    public string Imie { get; set; }
    public string Nazwisko { get; set; }

    public virtual ICollection<Ogloszenie> Ogloszenie { get; private
                ↪set; }
}
```

# Relacja „jeden do jednego”

Relacja „jeden do jednego” występuje, gdy jeden wiersz z pierwszej tabeli jest powiązany z tylko jednym rekordem z drugiej tabeli. Taka relacja może mieć sens np. wtedy, gdy chcesz przechowywać dodatkowe informacje na temat użytkownika, jednak nie chcesz, aby były one w tej samej tabeli, lub gdy użytkownik może pełnić tylko jedną funkcję w firmie. Na listingu 4.3 przedstawiono przykładową relację „jeden do jednego” dla podejścia *Code First*, a na rysunku 4.3 zaprezentowano tę samą relację w podejściu *Model First*.

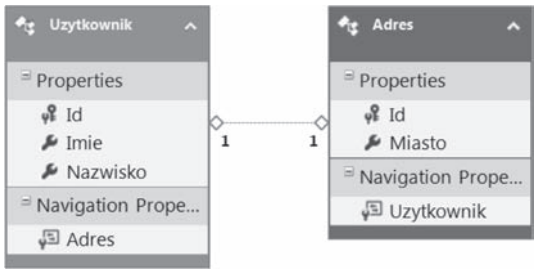
**Rysunek 4.2.** *Przykład relacji „jeden do wielu" w podejściu Model First*




**Listing 4.3.** *Przykład relacji „jeden do jednego" w podejściu Code First*

```
public class Uzytkownik
{
    public Uzytkownik()
    {
    }
    public int Id { get; set; }
    public string Imie { get; set; }
    public string Nazwisko { get; set; }

    [Required]
    public virtual Adres Adres { get; private set; }
}

public class Adres
{
    public Adres()
    {
    }
    [Key, ForeignKey("Uzytkownik")]
    public int Id { get; set; }
    public string Miasto { get; set; }

    public virtual Uzytkownik Uzytkownik { get; set; }
}
```

**Rysunek 4.3.** *Przykład relacji „jeden do jednego" w podejściu Model First*




# Relacja „wiele do wielu"

Relacja „wiele do wielu" występuje, gdy jeden wiersz z pierwszej tabeli może być powiązany z wieloma wierszami z drugiej tabeli, a jeden wiersz z drugiej tabeli może być powiązany z kilkoma wierszami z pierwszej tabeli. Przykładowo pracownik może

mieć przypisanych wiele adresów, a jeden adres może być przypisany do wielu pracowników. Oczywiście nie chodzi tu o to, że w tabeli są zapisane te same dane, tylko o to, że klucz obcy pokazuje na ten sam wiersz w tabeli. Relacje „wiele do wielu" rozkłada się na dwie relacje „jeden do wielu" i dodaje trzecią tabelę pomiędzy tabelami, które mają relację „wiele do wielu". Proces taki zabezpiecza przed redundancją danych (powtarzaniem tych samych danych). Nowa, środkowa tabela ma dwa klucze obce: jeden do tabeli pierwszej, a drugi do drugiej. W podejściu obiektowym w EF możliwe jest tworzenie relacji „wiele do wielu" bez dodatkowej klasy dla pośredniej tabeli. EF automatycznie przetłumaczy takie powiązanie i w bazie danych pojawi się trzecia tabela. Tabela pośrednia będzie niewidoczna z punktu widzenia kodu, jednak będzie się znajdowała w bazie danych. Na listingach przedstawiono przykładową relację „wiele do wielu" dla podejścia *Code First* bez dodatkowej tabeli (listing 4.4) i z dodatkową tabelą (listing 4.5), a na rysunkach zaprezentowano tę samą relację w podejściu *Model First* bez dodatkowej tabeli (rysunek 4.4) i z dodatkową tabelą (rysunek 4.5).

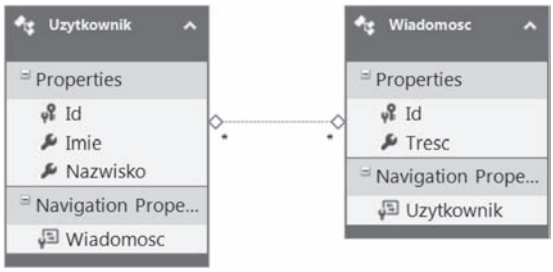
**Listing 4.4.** *Przykład relacji „wiele do wielu" w podejściu Code First bez dodatkowej tabeli*

```
public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Wiadomosc = new HashSet<Wiadomosc>();
    }
    public int Id { get; set; }
    public string Imie { get; set; }

    public string Nazwisko { get; set; }

    public virtual ICollection<Wiadomosc> Wiadomosc { get; private
                ↪set; }
}

public class Wiadomosc
{
    public Wiadomosc ()
    {
        this.Uzytkownik = new HashSet<Uzytkownik>();
    }
    public int Id { get; set; }
    public string Tresc { get; set; }

    public virtual ICollection<Uzytkownik> Uzytkownik { get; private
                ↪set; }
}
```

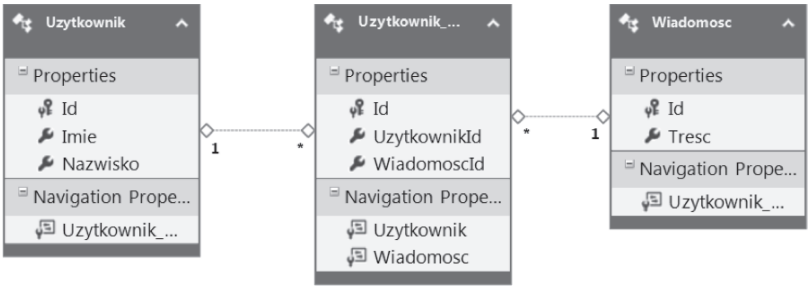
**Listing 4.5.** *Przykład relacji „wiele do wielu" w podejściu Code First z dodatkową tabelą*

```
public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Wiadomosc = new HashSet<Wiadomosc>();
    }
    public int Id { get; set; }
    public string Imie { get; set; }
```

```
    public string Nazwisko { get; set; }
    public virtual ICollection<Uzytkownik_Wiadomosc>
                ↪Uzytkownik_Wiadomosc { get; private set; }
}

public class Uzytkownik_Wiadomosc
{
    public int Id { get; set; }
    public string UzytkownikId { get; set; }
    public int WiadomoscId { get; set; }
    public virtual Uzytkownik Uzytkownik { get; set; }
    public virtual Wiadomosc Wiadomosc { get; set; }
}

public class Wiadomosc
{
    public Wiadomosc()
    {
        this.Uzytkownik_Wiadomosc = new
                ↪HashSet<Uzytkownik_Wiadomosc>();
    }
    public int Id { get; set; }
    public string Tresc { get; set; }

    public virtual ICollection<Uzytkownik_Wiadomosc>
                ↪Uzytkownik_Wiadomosc { get; private set; }
}
```




**Rysunek 4.4.** *Przykład relacji „wiele do wielu" w podejściu Model First bez dodatkowej tabeli*




**Rysunek 4.5.** *Przykład relacji „wiele do wielu" w podejściu Model First z dodatkową tabelą*

## Relacje opcjonalne

Relacje opcjonalne oznaczają, że dana relacja może, ale nie musi wystąpić. Duże znaczenie ma koniec, na którym znajduje się relacja opcjonalna. Przykładowo klient może, ale nie musi posiadać dodatkowych danych, dlatego relacja opcjonalna musi być po stronie tabeli klienta, ponieważ dane dodatkowe zawsze będą powiązane z klientem. Nie może być danych dodatkowych bez klienta, natomiast klient może być bez danych dodatkowych. Taka relacja to relacja `0..1 do 1`.

Podobnie wygląda sytuacja z relacją `0..1 do wielu`. W przypadku takiej relacji może istnieć wiele ogłoszeń, ale ogłoszenie niekoniecznie musi być powiązane z użytkownikiem.

# Obiekty DbContext i DbSet

## DbContext i DbSet

`DbContext` to obiekt odpowiedzialny za pracę z danymi. `DbContext` zajmuje się odczytem danych z bazy danych, śledzeniem stanu obiektów, zarządzaniem i powiązaniami pomiędzy danymi w pamięci, tłumaczeniem danych relacyjnych na obiektowe i zapisem danych do bazy. W klasie `DbContext` znajdują się właściwości `DbSet`, które reprezentują kolekcje określonych klas (tabel) znajdujących się w obiekcie kontekstu (listing 4.6).

**Listing 4.6.** *Przykładowa klasa DbContext*

```
public class ProduktContext : DbContext
{
    public DbSet<Kategoria> Kategorie { get; set; }
    public DbSet<Produkt> Produkty { get; set; }
}
```

Obiekt kontekstu jest przechowywany w pamięci do czasu, aż zostanie usunięty przez *Garbage Collector* lub zostanie wywołana metoda `Dispose()` na jego instancji. W aplikacji internetowej obiekt `DbContext` powinien być tworzony dla każdego żądania HTTP. Czas życia takiego obiektu powinien być jak najkrótszy, ponieważ nie zostanie on ponownie użyty.

## Metody Attach i Detach

Dane pobierane z bazy są automatycznie przypisywane do kontekstu. Wyjątkiem są tutaj zapytania `AsNoTracking()`, których wynik nie jest zapisywany do kontekstu, więc dane nie są śledzone (możliwe jest przyłączenie takich danych za pomocą metod). Ponieważ `DbContext` przechowuje dane w postaci obiektowej, możliwe jest odłączenie obiektu od kontekstu lub przyłączenie obiektu pochodzącego z innego źródła, innego kontekstu bądź

obiektu wcześniej odłączonego od kontekstu (również z innej instancji kontekstu). Obiekty odłączone od kontekstu nie mają śledzonego stanu przez `DbContext` i nie mogą zostać zapisane do bazy danych.

Istnieją metody pozwalające na przyłączenie obiektu do kontekstu:

- `AddObject()` dodaje i ustawia stan obiektu na `Added` (dodany). Dla obiektów ze stanem `Added` klucz podstawowy (PK) nie musi mieć unikalnej wartości, ponieważ podczas zapisu i tak zostanie wygenerowany przez bazę danych.
- `Attach()` dodaje do kontekstu i ustawia stan na `Unchanged` (niezmieniony). Wartość PK musi być unikalna, w przeciwnym wypadku EF wywoła wyjątek.
- `AttachTo()` działa identycznie jak `Attach()`, jednak zamiast wywoływać ją dla obiektu kontekstu, to nazwę kontekstu podaje się jako parametr.

Za pomocą metody `Detach()` można odłączyć obiekt od kontekstu, jednak niesie to za sobą pewne konsekwencje. Jeśli obiekt posiadał relacje do innych obiektów, to powiązane obiekty nie zostaną usunięte z kontekstu. Stan obiektu nie jest śledzony, odłączenie nie wpływa na dane w bazie danych, a jeśli istnieje, nie zostaje wywołane kaskadowe usuwanie w trakcie wykonywania operacji `Detach()`:

```
context.Detach(zamowienie.Szczegoly.First());
```

# Relacje poprzez klucz FK a relacje niezależne (obiektowe)

Relacje „jeden do jednego” zawsze są relacjami poprzez klucz (FK), natomiast relacje „wiele do wielu” zawsze są relacjami niezależnymi (ang. *Independent*). Dla relacji „jeden do wielu” można zastosować relację FK lub niezależną.

## Relacje poprzez klucz obcy — FK Association

Relacja FK (ang. *FK Association*) polega na dodaniu klucza obcego do klasy odwzorowującej tabele. Relacje pomiędzy klasami polegają na kluczu obcym, tak samo jak w bazie danych (listing 4.7). Takie rozwiązanie nie jest rozwiązaniem obiektowym, lecz relacyjnym. Natomiast klasy są częścią programowania obiektowego. Takie rozwiązanie ma swoje plusy i minusy.

Plusy relacji FK:

- znając wartość klucza, można dodać dane do sąsiedniej tabeli, nie trzeba posiadać wczytanego całego obiektu z bazy danych;
- podejście bardziej intuicyjne, identyczne jak w bazie danych i kodzie SQL.

Minusy relacji FK:

- podejście relacyjne, a nie obiektowe, w środowisku obiektowym;
- posiadanie dwóch relacji (zarówno FK, jak i relacji niezależnej) może powodować problem z synchronizacją pomiędzy tymi dwoma rodzajami relacji.

**Listing 4.7.** *Przykład relacji FK w EF*

```
public class Ogloszenie
{
    public Ogloszenie()
    {
    }
    public int Id { get; set; }
    public int UzytkownikId { get; set; }
    public Uzytkownik Uzytkownik { get; set; }
}

public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Ogloszenie = new HashSet<Ogloszenie>();
     }
    public int Id { get; set; }
    public string Imie { get; set; }
    public string Nazwisko { get; set; }

    public virtual ICollection<Ogloszenie> Ogloszenie { get; private
                ↪set; }
}
```

# Relacje niezależne — Independent Association

Relacja niezależna (ang. *Independent Association*) polega na tym, że nie dodaje się do klasy kluczy obcych, tylko obiekt klasy, z którą występuje relacja (listing 4.8). Konsekwencją takiego podejścia jest konieczność posiadania całego obiektu w pamięci — bez tego EF nie mógłby zamodelować relacji, ponieważ nie opiera się ona na FK.

Plusy relacji niezależnej:

- istnieje dostęp do wszystkich danych z sąsiedniej tabeli, a nie tylko do klucza, ponieważ w pamięci znajduje się cały obiekt;
- obiektowe podejście.

Minusy relacji niezależnej:

- skomplikowane tłumaczenie relacji pomiędzy obiektami na relacje w bazie danych;
- mniej intuicyjne niż relacje FK;
- jeśli tworzy się relacje, potrzebne są obiekty dla obu końców relacji (w FK wystarczyło podać wartość klucza obcego);

- nie ma stanu `modified` — za każdym razem relacja jest usuwana i dodawana jest nowa;
- mniej naturalne z punktu widzenia baz danych i języka SQL, trudniejsze do wychwycenia błędy.

**Listing 4.8.** *Przykład relacji niezależnej w EF*

```
public class Ogloszenie
{
    public Ogloszenie()
    {
    }
    public int Id { get; set; }

    public Uzytkownik Uzytkownik { get; set; }
}

public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Ogloszenie = new HashSet<Ogloszenie>();
    }
    public int Id { get; set; }
    public string Imie { get; set; }
    public string Nazwisko { get; set; }
    public virtual ICollection<Ogloszenie> Ogloszenie { get; private
                ↪set; }
}
```

# Odpytywanie bazy danych za pomocą EF i LINQ

## Wczytywanie zachłanne — Eager Loading

Wczytywanie zachłanne (ang. *Eager Loading*) to proces, w którym zapytanie o jeden typ tabeli pobiera również dane z tabeli, które są z nią powiązane (posiadają relacje). Użytkownik posiada wiele ogłoszeń. Jeśli pobiera się użytkownika, to pobierane są również jego ogłoszenia. Aby użyć *Eager Loading*, gdy włączone jest *Lazy Loading*, należy wywołać metodę `Include()` w zapytaniu LINQ (listing 4.9).

**Listing 4.9.** *Przykład zapytania Eager Loading*

```
using (var context = new AppContext())
{
    var uzytkownicy = context.Uzytkownik
                ↪.Include(b => b.Ogloszenia)
                ↪.ToList();
}
```

# Wczytywanie leniwe — Lazy Loading

Wczytywanie leniwe (ang. *Lazy Loading*) pozwala na opóźniony odczyt danych. Nie ma konieczności pobierania danych ogłoszeń, jeśli chce się pobrać tylko dane użytkownika, pomimo że w klasie `Uzytkownik` znajduje się kolekcja ogłoszeń. Gdy używa się *Lazy Loading*, podczas pobierania danych do klasy użytkownika nie są ładowane jego ogłoszenia (listingi 4.10 i 4.11). *Lazy Loading* wymaga oznaczenia właściwości relacji jako wirtualnej (`virtual`). Aby było możliwe leniwe ładowanie, tworzone są dynamiczne puste obiekty proxy, które nadpisują pola wirtualne właściwości. Jeśli użytkownik próbuje się odwołać do obiektu proxy po raz pierwszy (chce wyświetlić również informacje o ogłoszeniach użytkownika), to tworzona jest instancja prawdziwego obiektu (kolekcji ogłoszeń) w miejsce pustego obiektu proxy.

**Listing 4.10.** *Przykład klasy Lazy Loading*

```
public class Uzytkownik
{
    public Uzytkownik()
    {
        this.Ogloszenia = new HashSet<Ogloszenie>();
    }

    public string Imie { get; set; }
    public string Nazwisko { get; set; }

    public virtual ICollection<Ogloszenie> Ogloszenia { get; private
            ↪set; }
}
```

**Listing 4.11.** *Przykład zapytania Lazy Loading*

```
using (var context = new AppContext())
{
    var uzytkownicy = context.Uzytkownik.Take(10);
}
```

Podczas korzystania z Web API wymagana jest serializacja danych (zapis w formacie JSON lub XML). Aby serializacja była możliwa, konieczne jest wyłączenie *Lazy Loading*, ponieważ powoduje to ładowanie dużej ilości niepotrzebnych danych, a nawet zapętlenie. *Lazy Loading* można wyłączyć poprzez usunięcie modyfikatora `virtual` przed właściwością relacji lub globalnie w klasie z kontekstem, co zostało zaprezentowane w poniższym kodzie:

```
public AppContext()
{
    this.Configuration.LazyLoadingEnabled = false;
}
```

Zalety *Lazy Loading*:

- ♦ szybsze ładowanie aplikacji,
- ♦ mniejsze zużycie pamięci poprzez wczytywanie danych na życzenie,

- zmniejszenie liczby zapytań do bazy danych.

Wady *Lazy Loading*:

- konieczność sprawdzania, czy *Lazy Loading* jest włączone, co skutkuje nieznacznie mniejszą wydajnością, gdyż dane i tak muszą zostać wczytane.

# Jawne ładowanie — Explicit Loading

Jeśli chcemy, aby przy wyłączonym trybie *Lazy Loading* zapytanie zachowywało się tak jak w trybie *Lazy Loading*, trzeba skorzystać z zapytania *Explicit Loading*. Aby wykonać *Explicit Loading*, należy wywołać metodę `Load()` na powiązanej relacją tabeli (listing 4.12).

**Listing 4.12.** *Przykład Explicit Loading*

```
using (var context = new AppContext())
{
    var uzytkownik = context.Uzytkownik.Find(2);
    context.Entry(uzytkownik).Collection(p => p.Ogloszenia).Load();
}
```

# Problem N+1

*Problem N+1* występuje, gdy nie korzysta się z *Lazy Loading* i chce się pobrać w jednym zapytaniu do bazy listę obiektów, które posiadają relacje z innymi obiektami (listing 4.13). Następnie dla każdego pobranego obiektu jest wykonywane osobne zapytanie pobierające dodatkowe dane z tabel, które są powiązane (posiadają relacje). Problem występuje w zapytaniach bez wykorzystania operacji `join`. Przykładowo każdy klient (wiersz w tabeli *klient*) ma przypisany jeden adres (wiersz z tabeli *adres*). Przy pobieraniu listy klientów, gdy każda klasa klienta ma pole `adres`, wywołane zostaje zapytanie zwracające listę klientów (bez adresów), a potem automatycznie wywoływane są dodatkowo osobne zapytania o adres. Każde zapytanie jest wywoływane osobno, a więc będzie tyle zapytań do bazy danych, ile zostanie zwróconych klientów. W *Lazy Loading Problem N+1* wystąpi dopiero wtedy, gdy zajdzie potrzeba pobrania adresów. Jeśli tabela posiada `index`, dane zostaną bardzo szybko zwrócone — i takie podejście jest prawidłowe (szybsze niż operacja `join`), jednak gdy danych jest dużo i trzeba wykonać bardzo dużo pojedynczych zapytań do bazy, operacja `join` powinna być szybsza (listing 4.14). Najlepszym wyjściem jest napisanie dwóch zapytań i sprawdzenie, które jest szybsze w danym przypadku i w jakim stopniu obciąża bazę danych.

**Listing 4.13.** *Przykład kodu prezentującego Problem N+1*

```
using (var context = new AppContext())
{
    foreach (var uzytkownik in context.Uzytkownik)
    {
        foreach (var ogloszenie in uzytkownik.Ogloszenia)
        {
            Console.WriteLine("{0}: {1}", uzytkownik.Imie,
                ↪ogloszenie.Tytul);
```

```
        }
    }
}
```

**Listing 4.14.** *Przykład kodu korzystającego z operacji join (metoda Include) bez Problem N+1*

```
using (var context = new AppContext())
{
    foreach (var uzytkownik in
                ↪context.Uzytkownik.Include("Ogloszenia"))
    {
        foreach (var ogloszenie in uzytkownik.Ogloszenia)
        {
            Console.WriteLine("{0}: {1}", uzytkownik.Imie,
                ↪ogloszenie.Tytul);
        }
    }
}
```

# Metoda AsNoTracking()

Metoda rozszerzająca `AsNoTracking()` zwraca z zapytania dane, które nie będą śledzone przez EF ani nie będą przechowywane w pamięci obiektu `DbContext`. Zastosowanie `AsNoTracking()` skutkuje około dwukrotnie większą wydajnością przy odczycie. Jeśli dane mogą lub mają zostać zmodyfikowane, nie powinno się korzystać z tej metody. EF będzie wówczas śledził zmiany w obiekcie, co umożliwi edycję lub zapis danych do bazy (listing 4.15).

**Listing 4.15.** *Przykładowy kod wykorzystujący metodę AsNoTracking()*

```
using (var context = new WebAppContext())
{
    var dane = context.Tabela
                ↪.Where(d => d.Nazwa.Contains(".NET"))
                ↪.AsNoTracking()
                ↪.ToList();
}
```

# Odroczone i natychmiastowe wykonanie

Natychmiastowe (ang. *Immediate*) wykonanie zapytania pobiera całość danych od razu podczas wykonywania zapytania. Odroczone (ang. *Deferred*) wykonanie zapytania korzysta z iteratorów (tabela 4.1) i `yield return`, a więc musi zwracać typ danych, który dziedziczy po `IEnumerable`, i zwraca dane pojedynczo w momencie, kiedy są potrzebne. Domyślnie używane jest odroczone wykonanie. Aby wywołać natychmiastowe wykonanie, należy użyć jednej z następujących metod: `ToList()`, `ToDictionary()` lub `ToArray()`. Wszystkie zapytania zwracające pojedynczą wartość są wykonywane natychmiastowo, np. `Averange`, `Count`, `First` lub `Sum`, ponieważ nie ma możliwości użycia opóźnionego wykonania, jeśli jest tylko jeden element w kolekcji.

**Tabela 4.1.** *Lista operatorów, które posiadają zaimplementowane iteratory*

| Metoda | Iterator |
|---|---|
| Cast | CastIterator |
| Concat | ConcatIterator |
| DefaultIfEmpty | DefaultIfEmptyIterator |
| Distinct | DistinctIterator |
| Except | ExceptIterator |
| GroupJoin | GroupJoinIterator |
| Intersect | IntersectIterator |
| Join | JoinIterator |
| OfType | OfTypeIterator |
| Range | RangeIterator |
| Repeat | RepeatIterator |
| Reverse | ReverseIterator |
| Select | SelectIterator |
| SelectMany | SelectManyIterator |
| Skip | SkipIterator |
| SkipWhile | SkipWhileIterator |
| Take | TakeIterator |
| TakeWhile | TakeWhileIterator |
| Union | UnionIterator |
| Where | WhereIterator |
| Zip | ZipIterator |

# Entity SQL

*Entity SQL* to język zapytań bazujący na SQL (alternatywa dla *LINQ to Entities*). Pozwala na pisanie zapytań w języku SQL, jednak w treści zapytania zamiast nazw tabel podaje się nazwy klas z aplikacji. *Entity SQL* powinien być używany tylko w momencie, gdy zapytanie pisane za pomocą LINQ jest zbyt skomplikowane. Zapytania *Entity SQL* nie są kompilowane, lecz konstruowane są w czasie działania programu. Zapytania LINQ są kompilowane i typowane (zwiększa się bezpieczeństwo i szybkość), dlatego powinny być używane, jeśli tylko jest to możliwe. Aby wykonać zapytanie *Entity SQL*, należy uzyskać `ObjectContext` z obiektu `DbContext` i skorzystać z generycznej metody `Object ↪Query<T>` (listing 4.16)[1].

**Listing 4.16.** *Przykład zapytania Entity SQL*

```
using(ApplicationDbContext context = new ApplicationDbContext())
{
    var adapter = (IObjectContextAdapter)context;
    var objectContext = adapter.ObjectContext;
```

[1] *http://msdn.microsoft.com/en-us/library/bb399554.aspx*

```
    string esqlQuery = @"SELECT VALUE Ogloszenie FROM
                ↪Models.Ogloszenia as Ogloszenie
                ↪WHERE Ogloszenie.Id > @parametrId";

    ObjectQuery<Ogloszenie> query = new
                ↪ObjectQuery<Ogloszenie>(esqlQuery, objectContext,
                ↪MergeOption.NoTracking);

    query.Parameters.Add(new ObjectParameter("parametrId", 1));

    return query;
}
```

# Bezpośrednie zapytania SQL do bazy (Direct/RAW SQL) i procedury składowane w EF

Zapytania *Direct/RAW SQL* powinny być stosowane tylko wtedy, gdy nie ma możliwości skorzystania z zapytań *LINQ to Entities* lub *Entity SQL*. W zapytaniu SQL podaje się nazwy tabel w bazie danych, a nie nazwy klas, jak to było w *Entity SQL*. W *Entity SQL* operuje się na typach pochodzących z aplikacji, co wpływa na bezpieczeństwo, ponieważ typy danych są znane i walidowane. Dla zapytań *RAW SQL*, czyli składających się z czystego kodu SQL, występuje niebezpieczeństwo ataku SQL Injection. Aby pobrać dane za pomocą języka SQL, należy użyć metody `SqlQuery()`, a w celu wysłania zapytania do bazy niezwracającego żadnych wartości korzysta się z `SqlCommand()`.

Przykład zapytania zwracającego typ klasy POCO:

```
using (var context = new AppContext())
{
    var blogs = context.Ogloszenia.SqlQuery("SELECT * FROM dbo.Ogloszenia
").ToList();
}
```

Przykład wywołania procedury składowanej z parametrem:

```
using (var context = new AppContext())
{
    var ogloszenieId = 1;
    var blogs = context.Ogloszenia.SqlQuery("dbo.GetOgloszenieById
                ↪@p0", ogloszenieId).Single();
}
```

Przykład zapytania zwracającego typ, który nie jest klasą POCO:

```
using (var context = new AppContext())
{
    var tytuly = context.Database.SqlQuery<string>("SELECT Tytul FROM
                ↪ dbo.Ogloszenia").ToList();
}
```

Przykład zapytania SQL niezwracającego wartości:

```
using (var context = new AppContext())
{
    context.Database    .SqlCommand("UPDATE dbo.Ogloszenia SET Tytul =
                ↪'Jakis tytul' WHERE Id = 1");
}
```

# Skorowidz

## C



## D

## E



## F



## G



## H



## I

## J



## K

## L



## Ł



## M

## N



## O



## P

## Q



## R

## S

## Ś



## T



## U



## V



## W

## X



## Z

## Ż

# C# 6.0 i MVC 5

Dynamiczny rozwój sieci, która wkracza w coraz to nowe obszary życia, pozwala nam korzystać z większości usług z dowolnego miejsca na świecie i za pośrednictwem każdego urządzenia z dostępem do internetu. Aby użytkownik mógł swobodnie poruszać się po sieci, konieczne jest rozwijanie coraz bardziej zaawansowanych, wygodniejszych i szybszych aplikacji internetowych działających po stronie serwera. Do niedawna większość aplikacji tego rodzaju była pisana w takich językach jak PHP, Python lub Java, obecnie ogromną popularność zdobywają w tej dziedzinie język C# i platforma MVC.

Jeśli dysponujesz ogólną wiedzą na temat programowania i baz danych, opanowałeś lub właśnie opanowujesz podstawy języka C# i chcesz poznać platformę MVC oraz możliwości, jakie oferuje w zakresie tworzenia aplikacji webowych, sięgnij po tę książkę. Nie stanowi ona klasycznego podręcznika do programowania, lecz raczej zbiór praktycznych wskazówek i objaśnień pozwalających w krótkim czasie rozpocząć przygodę z tworzeniem aplikacji internetowych, a następnie logicznie ją kontynuować. Dzięki lekturze krok po kroku poznasz proces powstawania kodu aplikacji i jego wdrażania przy użyciu różnych narzędzi. Poznaj świat aplikacji internetowych!

- Podstawy języka C# i jego możliwości
- Wzorce projektowe i architektoniczne oraz ich stosowanie
- Możliwości webowych platform firmy Microsoft
- Tworzenie aplikacji i serwisów internetowych oraz ich wdrażanie
- Optymalizowanie aplikacji pod kątem wymagań wyszukiwarek i serwisów społecznościowych
- Budowa własnego portalu

## Buduj doskonałe serwisy internetowe w języku C# z platformą MVC!



Helion

22947 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-9496-9

9 788324 694969

Informatyka w najlepszym wydaniu

cena: 89,00 zł