



Programowanie aplikacji lokalnych 2025/26

Instrukcja laboratoryjna ORM Entity Framework



Prowadzący: Tomasz Goluch

Wersja: 1.0

I. ORM, DDD, Code-First.

Cel: Zapoznanie z mapowaniem obiektowo-relacyjnym (ORM) i jego kluczową rolą we wspieraniu podejścia Domain-Driven Design (DDD) poprzez realizację podejścia Code-First..

Zgodnie z zasadą Domain-Driven Design, projektując aplikację, powinniśmy najpierw skupić się na tworzeniu klas domenowych realizujących logikę aplikacji. Następnie, dzięki mapperowi obiektowo-relacyjnemu (ORM), możemy zająć się utrwalaniem stanu tych obiektów. ORM niweluje różnice między paradygmatem obiektowym a relacyjną bazą danych, pozwalając modelowi domeny na zachowanie tzw. persistence ignorance (klasy domeny nie muszą wiedzieć, jak są zapisywane).

Podejście Code-First jest tego naturalną konsekwencją: pozwala na automatyczne wygenerowanie schematu bazy danych na podstawie zdefiniowanych klas domeny. Dodatkowo ORM powinien zapewniać mechanizm łatwej migracji bazy danych w przypadku wprowadzenia zmian w modelu. Większość mapperów pozwala również na odwrotne podejście, czyli utworzenie modelu na podstawie istniejącej bazy danych (tzw. podejście Database-First).

II. Instalacja biblioteki i narzędzi EF.

Cel: Instalacja biblioteki Entity Framework Core, odpowiedniego dostawcy bazy danych (providera) oraz narzędzi..

Decydując się na wykorzystanie w projekcie Entity Framework Core, musimy zainstalować przynajmniej dwa pakiety NuGet:

1. **Główna biblioteka EF Core:** Microsoft.EntityFrameworkCore
2. **Dostawcy (Providera) bazy danych:** Pakietu, który uczy EF Core „rozmawiać” z konkretną bazą danych. W tym przykładzie jest to SQL Server).

Najlepiej wykorzystać do tego konsolę Menedżera Pakietów (Package Manager Console). Komenda: Install-Package <NazwaPakietu> [-Version <WersjaBiblioteki>] [Project <NazwaProjektu>]:

```
PM> Install-Package Microsoft.EntityFrameworkCore -Project
<nazwa_projektu>
PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer -Project
<nazwa_projektu>
```

Opis aktualnych stabilnych wersji EF Core wraz z informacją z jaką wersją .Net Core współpracują, można znaleźć tutaj: <https://learn.microsoft.com/en-us/ef/core/what-is-new/>. Najnowsze stabilne wersje to:

Release	Target framework	Supported until	Links
EF Core 9.0	.NET 8	November 10, 2026	What's new / Breaking changes
EF Core 8.0	.NET 8	November 10, 2026	What's new / Breaking changes
EF Core 7.0	.NET 6	Expired May 14, 2024	What's new / Breaking changes
EF Core 6.0	.NET 6	Expired November 12, 2024	What's new / Breaking changes

Następną planowaną wersją jest EF Core 10 (EF10), której wydanie zaplanowano na listopad 2025 r.

Przydatna będzie również instalacja narzędzi: Microsoft.EntityFrameworkCore.Tools:

```
PM> Install-Package Microsoft.EntityFrameworkCore.Tools -Project
<nazwa_projektu>
```

Pozwalają one między innymi na utworzenie, usunięcie i zmianę schematu bazy danych oraz na zarządzanie migracjami. Narzędzia po instalacji mogą być uruchamiane z konsoli menedżera pakietów. Można też zainstalować je globalnie wtedy będą widoczne z poziomu wiersza poleceń i Powershella.

Pakiet narzędzi EF pozwala dokonać zmian w strukturze bazy danych. Oto kilka komend które mogą się przykazać pomocne:

- **Add-Migration <NazwaMigracji>** – na podstawie dotychczasowych migracji dodaje (domyślnie do katalogu **Migrations**) nową migrację zawierającą różnice w modelu wprowadzone od czasu utworzenia ostatniej migracji.
- **Remove-Migration** – usuwa ostatnią migrację (cofa zmiany w kodzie, które zostały wprowadzone podczas migracji).
- **Drop-Database** – usuwa bazę danych.
- **Update-Database** – aktualizuje bazę danych do ostatniej (domyślnie) lub określonej migracji. Migracje mogą być określona według nazwy lub identyfikatora. Liczba 0 to szczególny przypadek, który oznacza moment przed pierwszą migracją i powoduje cofnięcie wszystkich migracji.

Szczegółowy opis komend udostępnianych poprzez pakiet narzędzi EF można znaleźć tutaj: <https://learn.microsoft.com/en-us/ef/core/cli/powershell>.

Zamiast instalować wymagane pakiety przez NuGet albo jego graficzny odpowiednik wystarczy dodać odpowiednie wpisy **PackageReference** do sekcji **ItemGroup** pliku projektu (miejsce **X.X.X** należy uzupełnić aktualnie wymaganą wersją bibliotek):

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="X.X.X"/>
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="X.X.X"/>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="X.X.X">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

III. Encje i obiekty wartości¹.

Cel: Zaznajomienie z podstawowymi informacjami na temat encji oraz obiektów reprezentujących wartości (value objects).

Encje – to najważniejsze byty z punktu widzenia modelowanego biznesu. Posiadają tożsamość określaną za pomocą jakiegoś identyfikatora – dwie encje o identycznym stanie nie są sobie równe, jeśli różnią się jedynie identyfikatorem. Zachowanie encji będzie śledzone w czasie.

Obiekty wartości – lub po prostu wartości – w przeciwnieństwie do encji nie posiadają tożsamości (dwa obiekty o identycznym stanie zawsze są sobie równe). Istnieją jako elementy członkowskie klas encji. Służą, np. do opisywania, wyliczania lub mierzenia wartości. Rodzi się zatem pytaniem czy nie są to po prostu typy proste? Obiekty wartości od typów prostych odróżnia to, że są one często podzbiorem wartości, które może reprezentować typ prosty (zakres dat, numer telefonu) albo zbiorem takich ograniczonych typów zamkniętych w klasę. Obiekty wartości reprezentujące typy proste powinny posiadać logikę, np., validację, co pozwala na jego łatwą lokalizację. Zgodnie z konceptem Whole Value Pattern zdefiniowanym przez Ward'a Cunningham'a w [The checks Pattern Language of Information Integrity](#) podczas tworzenia obiektu powinien podlegać validacji. Jeśli zostanie ona przeprowadzona, np. w encjach zawierających dany obiekt wartości to zostanie załamana pierwsza zasada SOLID, a kod będzie zduplikowany do wielu klas zawierających. Wartości w cyklu swojego życia po stworzeniu i validacji nie powinny już ulegać zmianie. Poza konstruktorem albo metodą wytwórczą nie powinny mieć metod zmieniających ich stan, np. setter. Jeśli chcemy w encji zmienić jakąś wartość to po prostu przypisujemy nową instancję obiektu wartości, która również ulegnie validacji. Do implementacji obiektów wartości można wykorzystać klasy albo rekordy. My posłużymy się **rekordami** (`record`), które od C# 9 stały się dedykowanym i preferowanym sposobem implementacji Obiektów Wartości. Są one specjalnym typem klasy (`record class`), który został zaprojektowany właśnie w celu hermetyzacji danych. Użycie ich jest zdecydowanie prostsze, ponieważ kompilator automatycznie implementuje za nas cały kluczowy kod (tzw. *boilerplate*), niezbędny do poprawnego działania Obiektów Wartości. Implementacja przykładowego rekordu `Money`, może być ograniczona do jednej linii:

```
public record Money(decimal Amount, string Currency);
```

Słowo kluczowe `record` to w istocie instrukcja dla kompilatora C#, aby automatycznie wygenerował kod pośredni (IL) implementujący pełną semantykę Obiektu wartości. Kompilator tworzy za nas cały powtarzalny kod (*boilerplate*), który musielibyśmy pisać ręcznie, na przykład poprzez dziedziczenie po niestandardowej klasie bazowej np. `ValueObject`. Dzięki `record` automatycznie otrzymujemy:

1. **Odpowiednik niemodyfikowalnej klasy (Immutability)** która dziedziczy po `IEquatable<Money>` wraz z podstawowymi składowymi: publicznymi właściwościami z setterami `init` oraz publicznym konstruktorem. Użycie setterów `init` gwarantuje, że stan obiektu wartości nie zmieni się po jego utworzeniu (można go ustawić tylko w konstruktorze lub bloku inicjalizacyjnym), co jest fundamentem dla obiektów wartości.

```
public class Money : IEquatable<Money>
{
    public decimal Amount { get; init; }
    public string Currency { get; init; }
```

¹ <https://enterprisecraftsmanship.com/posts/entity-vs-value-object-the-ultimate-list-of-differences/>

```

public Money(decimal Amount, string Currency)
{
    this.Amount = Amount;
    this.Currency = Currency;
}
// pozostałe funkcjonalności opisane niżej
}

```

2. **Porównywanie przez wartość:** Nadpisane metody Equals(object obj) oraz GetHashCode(). Dwa różne obiekty Money będą sobie równe, jeśli ich Amount i Currency są mają te same wartości a nie referencje. Metoda GetHashCode() łączy hashe wszystkich właściwości w bezpieczny sposób.

```

public override bool Equals(object? obj)
{
    return this.Equals(obj as Money);
}

public override int GetHashCode()
{
    return System.HashCode.Combine(this.Amount, this.Currency);
}

```

3. **Implementację IEquatable<T>:** Metodę Equals(Money other) dla wydajniejszego silnie typowanego porównywania (bez konieczności rzutowania). To w niej zawarta jest właściwa logika porównywania pól.

```

public virtual bool Equals(Money? other)
{
    return (other != null) &&
        EqualityComparer<decimal>.Default.Equals(this.Amount, other.Amount) &&
        EqualityComparer<string>.Default.Equals(this.Currency, other.Currency);
}

```

4. **Operatory równości:** Przeciążone operatory == i !=, które również działają w oparciu o porównywanie wartości.

```

public static bool operator==(Money? left, Money? right)
{
    if (left is null)
    {
        return (right is null);
    }
    return left.Equals(right);
}

public static bool operator!=(Money? left, Money? right)
{
    return !(left == right);
}

```

5. **Czytelną reprezentację ToString():** Koniec z NazwaTypu w logach! Metoda jest automatycznie nadpisywana, aby zwracać tekstową reprezentację obiektu i jego wartości (np. "Money { Amount = 100.50, Currency = PLN }"), co jest nieocenione podczas debugowania.

```

public override string ToString()
{
    var builder = new System.Text.StringBuilder();
}

```

```

builder.Append("Money");
builder.Append(" { ");
builder.Append("Amount = ");
builder.Append(this.Amount.ToString());
builder.Append(", ");
builder.Append("Currency = ");
// Poprawnie obsługuje potencjalny null dla typów referencyjnych
builder.Append((this.Currency == null) ? "null" : this.Currency.ToString());
builder.Append(" }");
return builder.ToString();
}

```

6. **Kopiowanie (Non-destructive mutation):** Możliwość tworzenia kopii obiektu ze zmodyfikowanymi wybranymi polami przy użyciu składni `with`. Gdy piszesz `var m2 = m1 with { Amount = 200 };`, kompilator tłumaczy to na coś w stylu: "stwórz nowy Money używając konstruktora kopiącego `m1`, a następnie (podczas inicjalizacji) ustaw jego `Amount` na 200".

```

protected Money(Money original)
{
    this.Amount = original.Amount;
    this.Currency = original.Currency;
}

```

7. **Dekonstruktor:** Metoda `Deconstruct` pozwalająca na łatwe rozpakowanie obiektu do zmiennych (np. `var (a, c) = myMoney;`).

```

public void Deconstruct(out decimal Amount, out string Currency)
{
    Amount = this.Amount;
    Currency = this.Currency;
}

```

Należy pamiętać, że kompilator robi jeszcze więcej i używa mechanizmów, które nie są wprost dostępne dla programisty piszącego zwykłą klasę.

Gdybyśmy skompilowali powyższy kod i porównali wygenerowany IL z tym z rekordu, można by zobaczyć różnice. Dobrym przykładem są ukryte metody generowane przez kompilator takie jak wirtualna metoda `PrintMembers` (używana przez `ToString`) oraz chroniony „konstruktor kopiący” (używany przez `with`), które są oznaczone atrybutem `[CompilerGenerated]`.

Podejście wykorzystujące rekordy jest preferowane, ponieważ pozwala programistie skupić się na tym, co najważniejsze z perspektywy DDD: **na logice biznesowej i regułach walidacji** zawartych w konstruktorze, zamiast na pisaniu i testowaniu powtarzalnego kodu do obsługi równości.

Aktualnie nadal nasz rekord `Money` bardziej przypomina głupi kontener na dane (DTO). Dodajmy zatem walidację w konstruktorze, aby pokazać prawdziwą moc obiektów wartości:

```

public record Money
{
    public decimal Amount { get; }
    // 3-literowy kod waluty zgodny z ISO 4217 (np. "PLN", "USD").
    public string Currency { get; }

    // Prywatny zbiór walidowanych walut (można to też przenieść do serwisu)
    private static readonly HashSet<string> ValidCurrencies =
        new() { "PLN", "USD", "EUR", "GBP", "CHF" };
}

```

```

public Money(decimal amount, string currency)
{
    // === 1. WALIDACJA ===
    // Reguły biznesowe są egzekwowane przy tworzeniu obiektu.
    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount),
            "Kwota pieniężna nie może być ujemna.");
    }

    if (string.IsNullOrWhiteSpace(currency))
    {
        throw new ArgumentException("Kod waluty nie może być pusty.",
nameof(currency));
    }

    // === 2. NORMALIZACJA ===
    // Standaryzujemy dane wejściowe przed ich zapisaniem.
    var normalizedCurrency = currency.Trim().ToUpperInvariant();

    // === 3. DALSZA WALIDACJA (na znormalizowanych danych) ===
    if (normalizedCurrency.Length != 3)
    {
        throw new ArgumentException(
            "Kod waluty musi być 3-znakowym kodem ISO 4217.", nameof(currency));
    }

    if (!ValidCurrencies.Contains(normalizedCurrency))
    {
        throw new ArgumentException(
            $"Waluta '{normalizedCurrency}' nie jest wspierana.",
nameof(currency));
    }

    // Przypisanie zwalidowanych i znormalizowanych wartości
    Amount = amount;
    Currency = normalizedCurrency;
}

// === 4. LOGIKA BIZNESOWA ===
// Obiekty Wartości mogą (i powinny) zawierać logikę
// operującą na wartościach, które przechowują.

// Dodaje dwie wartości pieniężne. Rzuca wyjątek, jeśli waluty są różne.
public Money Add(Money other)
{
    if (this.Currency != other.Currency)
    {
        // W bardziej złożonym systemie tu mogłaby być logika konwersji walut
        // throw new InvalidOperationException("Nie można dodawać pieniędzy w
różnych walutach.");
    }

    // Zwracamy NOWĄ instancję (niemodyfikowalność)
    return new Money(this.Amount + other.Amount, this.Currency);
}

// Mnoży wartość pieniężną przez stały współczynnik (np. podatek, ilość).
public Money MultiplyBy(decimal factor)
{
    if (factor < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(factor), "Współczynnik nie
może być ujemny.");
    }
}

```

```

// Zaokrąglenie jest ważne w operacjach finansowych
var newAmount = Math.Round(this.Amount * factor, 2);

return new Money(newAmount, this.Currency);
}

// Metoda fabrykująca dla wygody
public static Money Zero(string currency) => new Money(0, currency);
public static Money FromPln(decimal amount) => new Money(amount, "PLN");

// === 5. LEPSZA REPREZENTACJA TEKSTOWA ===
// Nadpisujemy domyślną implementację ToString() z rekordu,
// aby uzyskać ładniejsze formatowanie.
public override string ToString()
{
    // F2 formatuje do 2 miejsc po przecinku
    return $"{Amount:F2} {Currency}";
}

```

Najlepszym sposobem na utrwalanie obiektów wartości w Entity Framework Core jest mechanizm Encji posiadanych ([Owned Entity Types](#)) zostaną opisane w rozdziale poświęconym modelowi danych.

IV. Relacje, nawigacje oraz opóźnione ładowanie

Cel: Zaznajomienie z powiązaniem relacji w bazie danych z nawigacji i definiującymi ją właściwościami oraz mechanizmem opóźnionego ładowania danych.

Relacje w EF Core są definiowane przez klucze obce, dodatkowo aby zapewnić naturalny, zorientowany obiektywnie sposób do odczytu i manipulacji relacjami dodana została dodatkowa warstwa w postaci nawigacji². Dzięki nim aplikacja może działać na encjach nie martwiąc się o to, co dzieje się z wartościami klucza obcego. Nawigacje składają się z tzw. właściwości nawigacji⁵, które zapewniają możliwość nawigowania pomiędzy dwoma typami encji. Nawigacja od encji zależnej do głównej jest wymagana, jeśli wymagana jest relacja, co z kolei oznacza, że właściwość klucza obcego nie dopuszcza wartości `null`. I odwrotnie, nawigacja jest opcjonalna, jeśli klucz obcy dopuszcza wartość `null`, a zatem relacja jest opcjonalna. Nawigacje referencyjne od encji głównej do zależnej różnią się tym, że w większości przypadków encja główna może zawsze istnieć bez encji zależnych. Oznacza to, że relacja wymagana nie gwarantuje, że zawsze będzie istniała co najmniej jedna instancja encji zależnej. W modelu EF nie ma sposobu, a także nie ma standardowego sposobu w relacyjnej bazie danych, aby upewnić się, że element główny jest powiązany z określoną liczbą elementów podzielnych. Jedynym rozwiązaniem jest zaimplementowanie tego wymagania w logice biznesowej aplikacji. Wyjątkiem od tej reguły jest sytuacja kiedy element główny i zależny dzielą tę samą tabelę w relacyjnej bazie danych lub znajdują się w tym samym dokumencie w przypadku bazy dokumentowej.

² Relationship navigations – <https://learn.microsoft.com/en-us/ef/core/modeling/relationships/navigations> ⁵ Navigation properties - <https://learn.microsoft.com/en-us/ef/ef6/fundamentals/relationships#relationshipsin-ef>

Wyróżniamy dwie formy nawigacji:

1. odniesienie nawigacyjne – to proste odwołania do obiektów innej encji, reprezentują „jedną” stronę relacji jeden-do-jednego i jeden-do-wielu. Muszą posiadać zdefiniowany seter, niekoniecznie publiczny. I zawsze powinny być inicjowane do wartości `null` w przeciwnym przypadku było by to równoznaczne z twierdzeniem, że encja istnieje, gdy w istocie nie jest to prawda. Jeśli korzystamy z typów referencyjnych nullable w ramach włączonego nullable aware context³ dla relacji:
 - a. opcjonalnych – (domyślne w wyłączonym nullable aware context) musimy skorzystać z właściwości dopuszczających wartość null dodając znak zapytania po nazwie typu,
 - b. wymaganych – możemy wykorzystać zarówno typy dopuszczające jak i niedopuszczające wartość null.
2. kolekcja nawigacyjna – to instancja dowolnego typu implementującego `ICollection<T>` posiadająca działającą metodę `Add`. Często używa się:
 - a. Interfejsu `IList<T>` albo bezpośrednio typu `List<T>` – dla niewielkiej liczby powiązanych jednostek ale z zapewnieniem stabilnej kolejności,
 - b. Interfejsu `IEnumerable<T>`, `ICollection<T>`, `ISet<T>` albo bezpośrednio typu `HashSet<T>` – większa wydajność wyszukiwania, kosztem braku stabilnej kolejności.

Zawiera ona instancje encji zależnych, których może być dowolna liczba. Reprezentują stronę „wiele” relacji jeden-do-wielu i wiele-do-wielu. Seter nie jest wymagany i w takim przypadku kolekcja będzie tylko do odczytu. Powszechną praktyką jest inicjowanie kolekcji w wierszu, eliminując w ten sposób potrzebę sprawdzania, czy właściwość jest pusta. Nie należy stosować expression bodied property⁴ (o postaci: `member => expression;`) ponieważ przy każdym dostępie do właściwości nastąpi utworzenie nowej, pustej instancji kolekcji. Jeśli potrzebujemy typów, które implementują `INotifyPropertyChanging` i `INotifyPropertyChanged` to zamiast `List<T>` i `HashSet<T>` należy wykorzystać `ObservableCollection<T>` i `ObservableHashSet<T>`⁵.

Większość konfiguracji dotyczącej nawigacji odbywa się poprzez skonfigurowanie samej relacji. Konfigurację, która jest specyficzna dla samych właściwości nawigacji może być wykonywana za pomocą metody `Navigation` klasy `EntityTypeBuilder` w ciele metody `OnModelCreating`.

Każdemu typowi relacji w bazie danych powinien odpowiadać odpowiedni schemat właściwości nawigacji po stronie modelu obiektowego:

³ Working with Nullable Reference Types – <https://learn.microsoft.com/en-us/ef/core/miscellaneous/nullablereference-types>, C# 8: Nullable Reference Types – <https://www.meziantou.net/csharp-8-nullablenumbericatypes.htm>

⁴ Expression-bodied members – <https://learn.microsoft.com/en-us/dotnet/csharp/programmingguide/statements-expressions-operators/expression-bodied-members>

⁵ Change-tracking proxies – <https://learn.microsoft.com/en-us/ef/core/change-tracking/changedetection#change-tracking-proxies>

- jeden-do-jednego⁶ – jedna encja jest powiązana z co najwyżej jedną inną encją.
Przykład:

```
// Encja główna (rodzic) public
class Blog
{
    public int Id { get; set; }
    public BlogHeader? Header { get; set; } // Odniesienie nawigacyjne do encji zależnej
}

// Encja zależna (dziecko) public
class BlogHeader
{
    public int Id { get; set; }
    public int BlogId { get; set; } // Wymagana właściwość reprezentująca klucz obcy
    public Blog Blog { get; set; } = null!; // Wymagane odniesienie nawigacyjne do encji głównej
}
```

W encji głównej musi być zdefiniowana przynajmniej jedna właściwość klucza podstawowego lub alternatywnego. Domyślnie o nazwie `Id`. W encji zależnej musi być zdefiniowana przynajmniej jedna właściwość klucza obcego. Domyślnie o nazwie `{NazwaEncjiGłównej}Id`. Opcjonalnie, w encji głównej może być zdefiniowane odniesienie nawigacyjne do klasy zależnej i odwrotnie w klasie potomnej odniesienie nawigacyjne do klasy głównej. W przypadku kiedy zdefiniowane zostaną obydwie powyższe to taka relacja nazywana jest dwukierunkową. W relacji one-to-one nie zawsze jest to oczywiste która ze stron jest encją główną, kandydatką może być:

- jeśli istnieje naturalna relacja rodzic-dziecko to będzie to rodzic,
- encja bez której istnienia nie może istnieć powiązana z nią encja,
- jeśli tabele już istnieją to encja odpowiadająca tabeli z kluczami głównymi.

Korzystając z typów referencyjnych nullable, kiedy właściwość klucza obcego jest nullable to również odniesienie nawigacyjne z encji podrzędnej do głównej musi być nullable (referencja opcjonalna). W innym przypadku odniesienie nawigacyjne w encji zależnej może być not-nullable (referencja wymagana) albo nullable (referencja opcjonalna).

```
// Encja zależna (dziecko) public
class BlogHeader
{
    public int Id { get; set; }
    public Blog? Blog { get; set; } // Opcjonalne odniesienie nawigacyjne do encji głównej
}
```

- jeden-do-wielu⁷ – jedna encja jest powiązana z dowolną liczbą innych encji. Przykład:

⁶ Relacja one-to-one w EF – <https://learn.microsoft.com/en-us/ef/core/modeling/relationships/one-to-one>

⁷ One-to-many relationships – <https://learn.microsoft.com/en-us/ef/core/modeling/relationships/one-to-many>

```

// Encja główna (rodzic)
public class Blog
{
    public int Id { get; set; }
    public ICollection<Post> Posts { get; } = new List<Post>(); // Kolekcja
    nawigacyjna zawierająca encje zależne
}

// Encja zależna (dziecko)
public class Post
{
    public int Id { get; set; }
    public int BlogId { get; set; } // Wymagana właściwość reprezentująca klucz
    obcy
    public Blog Blog { get; set; } = null!; // Wymagane odniesienie nawigacyjne
    do encji głównej
}

```

- wiele-do-wielu⁸ – dowolna liczba encji jednego typu jest powiązana z dowolną liczbą encji tego samego lub innego typu. Przykład:

```

public class Post
{
    public int Id { get; set; }
    public List<Tag> Tags { get; } = new();
}
public class Tag
{
    public int Id { get; set; }
    public List<Post> Posts { get; } = new();
}

```

Relacje wiele-do-wielu różnią się od wcześniej opisanych relacji tym, że nie można ich przedstawić w prosty sposób przy użyciu samego klucza obcego. Zamiast tego potrzebny jest dodatkowy typ encji, aby „połączyć” dwie strony relacji. Jest to tzw. „join entity type”, który odwzorowuje tabelę skrzyżowań w relacyjnej bazie danych. Encje tego typu zawierają pary wartości kluczy obcych, gdzie każdy z nich wskazuje na encję po jednej z dwóch stron relacji. Każda encja „join entity type”, a zatem każdy wiersz w tabeli skrzyżowań, reprezentuje jedno powiązanie między typami encji w relacji. EF Core może ukryć typ „join entity type” i zarządzać nim w tle. Dzięki temu nawigacja relacji wiele-do-wielu może być używana w naturalny sposób, dodając lub usuwając jednostki z każdej strony w razie potrzeby. Więcej na ten temat można doczytać na stronach pomocy firmy Microsoft.

Entity Framework Core obsługuje wzorzec opóźnionego ładowania (Lazy Loading), który polega na automatycznym pobieraniu powiązanych danych z bazy danych w momencie, gdy po raz pierwszy odwołujemy się do danej właściwości nawigacyjnej.

Aby ten mechanizm zadziałał, konieczne jest spełnienie trzech warunków:

⁸Many-to-many relationships – <https://learn.microsoft.com/en-us/ef/core/modeling/relationships/many-to-many>

- Instalacja pakietu: Należy zainstalować dodatkowy pakiet NuGet:
Microsoft.EntityFrameworkCore.Proxies
- Konfiguracja kontekstu: Należy włączyć tworzenie proxy podczas konfiguracji DbContext (np. w OnConfiguring lub Program.cs), wywołując metodę:
.UseLazyLoadingProxies()
- Modyfikacja encji: Wszystkie właściwości nawigacyjne (zarówno kolekcje, jak i pojedyncze referencje), które mają być ładowane opóźnienie, muszą być oznaczone jako virtual.

Słowo kluczowe `virtual` jest kluczowe, ponieważ pozwala EF Core na stworzenie w locie dynamicznej klasy potomnej (tzw. *proxy*), która dziedziczy po naszej encji. To właśnie ta klasa proxy nadpisuje (`override`) wirtualną właściwość i przechwytuje moment odwołania się do niej, aby w tle wykonać zapytanie do bazy danych.

Mimo wygody, Lazy Loading jest jedną z najczęstszych przyczyn poważnych problemów z wydajnością, znaną jako problem N+1 zapytań. Problem ten występuje, gdy pobieramy listę encji (np. 'N' blogów), a następnie w pętli odwołujemy się do wirtualnej właściwości nawigacyjnej dla *każdego* elementu tej listy.

Przykład problemu N+1:

```
// 1. Pobieramy 100 blogów.  
//     To jest nasze PIERWSZE zapytanie (ten '1').  
var blogs = _context.Blogs.Take(100).ToList();  
  
// 2. Iterujemy po liście blogów  
foreach (var blog in blogs)  
{  
    // 3. PROBLEM: Wewnątrz pętli odwołujemy się do blog.Posts.  
    //     Dla każdego bloga EF Core wykona OSOBNE zapytanie,  
    //     aby dociągnąć jego posty.  
    Console.WriteLine($"Blog: {blog.Name} ma {blog.Posts.Count} postów.");  
}
```

Rezultat: Zamiast jednego dużego zapytania, aplikacja wykonała 101 zapytań do bazy danych (1 zapytanie po 100 blogów + 100 zapytań po posty dla każdego bloga). Przy dużej ilości danych może to całkowicie zablokować aplikację.

Zdecydowanie bezpieczniejszą i bardziej wydajną alternatywą, która pozwala uniknąć problemu N+1, jest Eager Loading (ładowanie chciwe). Polega ono na jawnym poinformowaniu EF Core, jakie dane powiązane ma pobrać *od razu*, w ramach jednego, większego zapytania SQL (typu `JOIN`). Robimy to za pomocą metody `.Include()`.

Poprawne rozwiążanie (bez Lazy Loading):

```
// 1. Pobieramy 100 blogów I OD RAZU ich posty.  
//     Metoda .Include() każe EF Core dołączyć tabelę 'Posts'.  
//     To jest JEDNO, złożone zapytanie do bazy.  
var blogsWithPosts = _context.Blogs  
    .Include(b => b.Posts) // <-- Kluczowa linia  
    .Take(100)  
    .ToList();  
  
// 2. Ta pętla jest teraz "darmowa".  
//     Wszystkie dane są już w pamięci.  
//     NIE są wykonywane żadne dodatkowe zapytania.  
foreach (var blog in blogsWithPosts)  
{
```

```
Console.WriteLine($"Blog: {blog.Name} ma {blog.Posts.Count} postów.");
}
```

W tym podejściu (bez używania `virtual` i `UseLazyLoadingProxies`) programista ma pełną kontrolę nad tym, kiedy i jakie dane są pobierane, co zapobiega niespodziewanym "burzom zapytań" do bazy danych.

Więcej informacji tutaj: <https://tutorials.eu/implementing-lazy-loading-with-entity-framework-core/>.

VII. Dziedziczenie

Cel: Zaznajomienie z trzema sposobami reprezentowania dziedziczenia w bazie danych.

Sposoby realizacji dziedziczenia:

1. model tabela na hierarchię (TPH):

- Jak działa: Cała hierarchia klas (rodzic i dzieci) jest mapowana na jedną tabelę w bazie danych.
- Identyfikacja: Dodatkowa kolumna, domyślnie nazwana `Discriminator`, jest automatycznie dodawana do tabeli. Przechowuje ona wartość (np. nazwę klasy), która identyfikuje, do jakiego typu należy dany wiersz.
- Zachowanie w EF Core: To jest domyślna strategia. Jeśli nie skonfigurujesz niczego, EF Core używa TPH.
- Konfiguracja (Opcjonalna): Możesz użyć `HasDiscriminator()`, aby zmienić nazwę kolumny dyskryminatora lub typy wartości.
- Zalety: Charakteryzuje się potencjalnie lepszą wydajnością zapytań ponieważ wszystkie dane znajdują się w jednej tabeli,
- Wady: Kolumny dla właściwości klas potomnych (jak `IpAddress`) muszą dopuszczać wartość `NULL`, nawet jeśli w klasie są oznaczone jako `required`, ponieważ wiersze należące do klasy bazowej (`Client`) nie będą miały wartości dla tej kolumny.

2. model tabela na typ (TPT):

- Jak działa: Każda klasa w hierarchii (bazowa i potomne) dostaje własną, oddzielną tabelę.
- Identyfikacja: Tabela klasy bazowej (np. `Clients`) przechowuje wspólne właściwości (`Id`, `Name`, `Address`). Tabela klasy potomnej (np. `ECommClients`) przechowuje tylko swoje dodatkowe właściwości (`IpAddress`) oraz klucz podstawowy, który jest jednocześnie kluczem obcym wskazującym na `Id` w tabeli `Clients`.
- Zachowanie w EF Core: Nie jest domyślne. Musi być jawnie włączona.
- Konfiguracja (Wymagana): Aby włączyć TPT, musisz jawnie wskazać mapowanie tabel dla każdej klasy:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Mówimy EF, że Client mapuje się na tabelę "Clients"
    modelBuilder.Entity<Client>().ToTable("Clients");

    // Mówimy EF, że ECommClient mapuje się na tabelę "ECommClients"
```

```
    modelBuilder.Entity<ECommClient>().ToTable("ECommClients");
}
```

- Zalety: Lepsza normalizacja. Kolumna `IpAddress` w tabeli `ECommClients` może być `NOT NULL`.
- Wady: Zapytania pobierające dane z różnych typów w hierarchii wymagają kosztownych operacji `JOIN` między tabelami.

3. model tabela na konkretny typ (TPC):

- Jak działa: Każda konkretna (nieabstrakcyjna) klasa potomna dostaje własną tabelę. Klasa bazowa (jeśli jest abstrakcyjna) nie ma swojej tabeli.
- Identyfikacja: Każda tabela (np. `ECommClients` i osobna tabela `StandardClients`, jeśli by istniała) przechowuje wszystkie właściwości danego typu, włącznie z tymi odziedzicznymi (`Id`, `Name`, `Address` oraz `IpAddress`). Właściwości bazowe są powielone w każdej tabeli.
- Zachowanie w EF Core: Nie jest domyślna. Dostępna od EF Core 7.
- Konfiguracja (Wymagana):

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>().UseTpcMappingStrategy();
    // Nie trzeba już wywoływać .ToTable() dla każdej encji
}
```

- Zalety: Najszybsza, gdy odpytujemy o konkretne klasy potomne, ponieważ w przeciwieństwie do TPT nie wymaga ona operacji `JOIN`. Dodatkowo, w odróżnieniu od TPH, eliminuje problem "rzadkich" tabel (pełnych `NULL`-i) i pozwala na stosowanie ograniczeń `NOT NULL` na właściwościach klas potomnych, co zapewnia lepszą integralność danych.
- Wady: Może być skomplikowana w zarządzaniu kluczami głównymi (wymaga sekwencji, aby uniknąć kolizji `Id`). Zapytania o wszystkie typy `Client` muszą używać `UNION ALL`.

W przypadku podejścia Code First, domyślną strategią dziedziczenia jest model tabela na hierarchię (TPH). Jest on często najlepszym wyborem ze względu na wysoką wydajność zapytań (brak konieczności używania operacji `JOIN`). Rozważmy poniższą hierarchię dziedziczenia:

```
public class Client
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public ICollection<Order> Orders { get; set; }
}

public class ECommClient : Client
{
    public string IpAddress { get; set; }
}
```

EF Core automatycznie utworzy jedną tabelę `Clients` i doda do niej kolumnę `Discriminator` typu `string`, która będzie przechowywać nazwy klas ("Client" i "ECommClient"), aby je rozróżnić. Jeśli chcemy dostosować domyślne zachowanie TPH – na

przykład, aby zmienić nazwę kolumny dyskryminatora lub użyć bardziej oszczędnego typu danych (np. `char` zamiast `string`) – możemy użyć metody `HasDiscriminator`:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
        // Zmieniamy nazwę kolumny na "clientType" i jej typ na 'char'
        .HasDiscriminator<char>("clientType")
        // Mapujemy konkretne wartości dla każdej klasy
        .HasValue<Client>('N') // 'N' = Normalny
        .HasValue<ECommClient>('E'); // 'E' = ECommerce
}
```

Dodatkowy materiał: <https://makolyte.com/ef-core-inheritance-mapping/>.

V. Model danych.

Cel: Zaznajomienie z podstawowymi informacjami na temat modelu danych.

W pierwszym kroku musimy utworzyć model danych domeny. Model to nic innego jak zbiór zwykłych klas reprezentujących encje, które mogą być powiązane ze sobą relacją dziedziczenia jak i kompozycji. Można stosować również kolekcje.

Właściwości klas mogą być też opatrzone przydatnymi adnotacjami (Data Annotations Attributes):

- **[Key]** – określa, że właściwość jest kluczem danej encji i ustawi odpowiadającą jej kolumnę jako PrimaryKey w bazie danych. Jeśli istnieje właściwość o nazwie Id to domyślnie stanie się ona kluczem głównym.
- **[Required]** – określa, że kolumna odpowiadająca tej właściwości jest NotNull w bazie danych.
- **[StringLength]** – Jest to kluczowy atrybut "hybrydowy", który służy jednocześnie do konfiguracji bazy danych i walidacji.
 - Wpływ na bazę danych: Jego główna wartość (np. `[StringLength(100)]`) określa maksymalną długość ciągu i jest tłumaczony na rozmiar kolumny w bazie danych (np. `nvarchar(100)`). Działa tu identycznie jak atrybut `[MaxLength]`.
 - Wpływ na walidację: Jest on również automatycznie wykorzystywany przez platformy (jak ASP.NET) do walidacji danych wejściowych. W przeciwieństwie do `[MaxLength]`, pozwala również na zdefiniowanie minimalnej długości ciągu za pomocą właściwości `MinimumLength` (np. `[StringLength(100, MinimumLength = 5)]`).
- **[NotMapped]** – określa, że właściwość lub klasa powinna zostać wykluczona z mapowania bazy danych.

Pewne atrybuty mają zastosowanie po stronie klienta (ASP.NET/Razor Pages) np. do walidacji, sposobu prezentacji danych:

- **[DataType]** – określa nazwę dodatkowego typu skojarzonego z polem danych, np.:
 - `[DataType(DataType.Date)]`
 - `[DataType(DataType.EmailAddress)]`

Będzie wykorzystane w widoku po stronie klienta.

- `[Display(Name = "Nazwa encji")]` – dostarcza atrybut ogólnego przeznaczenia, który może być wykorzystany, np. w widoku klienta do opisania encji.
- `[DisplayFormat(DataFormatString = "{0:dd.MM.yyyy}", ApplyFormatInEditMode = true)]` – określa sposób wyświetlania i formatowania pól danych przez dane dynamiczne ASP.NET. Właściwość `ApplyFormatInEditMode` ustawiona na `true` oznacza, że poprawność stringu będzie sprawdzana podczas jego edycji.
- `[RegularExpression(@"(?:[1-9][0-9]*|0)\/[1-9][0-9]*")]` – oznacza, że wartość w danych dynamicznych ASP.NET musi być zgodna z określonym wyrażeniem regularnym.

Jeśli nie chcemy zaciemniać naszego modelu adnotacjami możemy dokonać konfiguracji implementując przeciążoną metodę `OnModelCreating` klasy `DbContext`. Jest ona wywoływana przez platformę, podczas pierwszego utworzenia kontekstu wtedy też budowany jest model. Jako parametr otrzymuje ona obiekt typu `ModelBuilder`. W połączeniu z klasami innych builderów takimi jak `EntityTypeBuilder`, `PropertyBuilder`, `NavigationBuilder`, `IndexBuilder`, `KeyBuilder` wykorzystujemy Fluent API realizujące wzorzec płynnego interfejsu¹⁰, które jednocześnie jest bogatsze funkcjonalnie od opisanego wcześniej Data Annotations Attributes. Przykładowo, można w nim zdefiniować klucz złożony. Wartości takiego klucza wygenerowane są z co najmniej dwóch pól w bazie danych:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Order>()
        .HasKey(o => new { o.CustomerAbbreviation, o.OrderNumber }); }
```

Przydatne metody:

- `Entity< TEntity >` – zwraca obiekt, którego można użyć do skonfigurowania danego typu encji¹¹ w modelu. Jeśli dany typ encji nie jest jeszcze częścią modelu to zostanie dodany;
- `Property` – to samo co w przypadku typu encji ale dotyczy właściwości typu encji. Jako parametr przyjmuje wyrażenie lambda reprezentujące właściwość, która ma podlegać konfiguracji;
- `IsRequired` – (użyte z `true` lub bez parametru) konfiguruje kolumnę jako NOT NULL. `IsRequired(false)` konfiguruje kolumnę jako NULL (opcjonalną). Metody tej używamy, aby zmienić domyślne zachowanie wynikające z typu C#:

⁹Klasy te realizują wzorzec projektowy Builder, pozwalający budować (za pomocą składni łańcuchowania metod) obiekty klas charakteryzujących się złożonymi zasadami ich konstrukcji, które prowadzą do wielu reprezentacji wewnętrznych i wymagały by implementacji wielu konstruktorów w celu obsłużenia wszystkich przypadków.

¹⁰Fluent Interface to wzorzec składniowo podobny do Buildera (również oparty o składnię łańcuchowania metod) jednak zamiast skupiać się na budowie pojedynczego obiektu wykonuje bardziej złożone operacje na systemie złożonym z wielu obiektów jednocześnie nadal zapewniając czytelne API.

¹¹Entity Type – pojęcie o którym przechowywane są informacje w bazie. Typ encji zazwyczaj odpowiada jednej lub kilku powiązany tabelom w bazie danych.

- Przykład 1: Masz `string? Name { get; set; }` (typ nullowalny w C#). EF domyślnie ustawia kolumnę jako NULL. Użycie `.IsRequired(true)` zmusi ją do bycia NOT NULL.
- Przykład 2: Masz `string Name { get; set; }` (typ nienullowalny w C# z NRT). EF domyślnie ustawia kolumnę jako NOT NULL. Użycie `.IsRequired(false)` zmusi ją do bycia NULL.

```
modelBuilder.Entity<SomeEntity>().Property(se =>
    se.SomeProperty).IsRequired(true);
```

- `HasConversion` – konfiguruje właściwość tak, aby jej wartość była konwertowana do i z bazy danych przy użyciu danego `ValueConverter'a`:

```
modelBuilder.Entity<<nazwa_encji>>()
    .Property(e => e.<nazwa_właściwości>)
        .HasConversion(
            new ValueConverter<string, double>(
                v => double.Parse(v, NumberStyles.Any),
                v => v.ToString("e", CultureInfo.CurrentCulture))
        );
```

- `ApplyConfiguration` – pozwala na separację konfiguracji w mniejszych porcjach podawanych jako parametr implementujący `IEntityTypeConfiguration< TEntity >`. Przykład przeniesienia kodu konwertera do metody `Configure` powiązanej z konkretnym typem encji:

```
public void Configure(EntityTypeBuilder<<nazwa_encji>> builder)
{
    builder.Property(e => e.<nazwa_właściwości>)
        .HasConversion(
            new ValueConverter<string, double>(
                v => double.Parse(v, NumberStyles.Any),
                v => v.ToString("e", CultureInfo.CurrentCulture))
    );
}
```

- `HasDiscriminator< T >` – służy do konfigurowania aspektów kolumny dyskryminatora reprezentującej hierarchię dziedziczenia. Współpracuje z `HasValue< T >` konfigurującą wartości dyskryminatora. Zatem wartości w kolumnie dyskryminatora to identyfikatory podklas opisane typem `T` powiązanych z wykorzystaniem funkcji `HasValue< T >`. Więcej na ten temat w rozdziale poświęconym dziedziczeniu.
- `HasData` – pozwala zainicjować danymi pewien typ encji. Przykład wypełnienia typ encji zawartością typu:

```
public void Configure(EntityTypeBuilder<SomeEntity> builder)
{
    builder.HasData
    (
        Enum.GetValues(typeof(SomeEnum))
            .Cast<SomeEnum>()
            .Where(e => (int)e > 0)
            .Select(e => new SomeEntity()
            {
                Id = (int)e,
                Name = e.ToString(),
                Description = "Some description"
            })
    );
}
```

```

        Name = e.ToString()
    })
}
}

```

Jest to sposób na przechowywanie etykiet typu wyliczeniowego w bazie danych. Normalnie jeśli podamy jako właściwość typ wyliczeniowy zamiast wspomnianego typu encji w tabeli powiązanej pojawią się wartości numeryczne.

- **HasMany/WithOne** – służy do konfigurowania wielu relacji jeden-do-wielu w której dany typ encji T1 posiada kolekcję zawierającą inny typ encji T2, który z kolei musi zawierać właściwość typu T1. Metoda **HasMany** musi być używana w połączeniu z metodą **WithOne**, aby w pełni skonfigurować prawidłową relację, przestrzegając wzorca Has/With dla konfiguracji relacji. Przykład typów encji gdzie jeden zawiera kolekcję drugiego:

```

public class Company
{
    ...
    public ICollection<Employee> Employees { get; set; }
}
public class Employee
{
    ...
    public Company Company { get; set; }
}

```

Poprawnie skonfigurowana relacja jeden-do-wielu:

```

modelBuilder.Entity<Company>()
    .HasMany(c => c.Employees)
    .WithOne(e => e.Company);

```

Fluent API oferuje znacznie większe możliwości konfiguracji niż Data Annotations i pozwala na zdefiniowanie skomplikowanych mapowań, takich jak klucze złożone.

Do reprezentacji modeli wartości najlepiej użyć [Encje Owned](#), np. poprzez oznaczenie reprezentujących je klas atrybutem [\[Owned\]](#). Klasy w których będą się one zawierały (encje Owned) będą nazywane właścicielami. Encje Owned zawsze mają relację jeden do jednego z właścicielem i nie potrzebują klucza własnego, ponieważ wartości klucza obcego ich właściciela są dla nich unikalne. EF automatycznie na ich podstawie utworzy tzw. shadow property przejmujące ich rolę. Inna sytuacja wystąpi jeśli właściciel będzie posiadał kolekcję encji Owned, domyślnym scenariuszem dla EF będzie użycie klucza złożonego z klucza obcego właściciela oraz dodatkowej właściwości. Domyślnie w tablicy relacyjnej bazy danych zawartość encji Owned i ich właścicielami będą mapowane na tą same tabelę, nazywając kolumny właściwości encji Owned zgodnie ze wzorcem: *Navigation_OwnedEntityProperty*. Można pominąć tę konwencję za pomocą metody **ToTable** podając jako parametr nazwę tabeli w której będzie przechowywana zawartość encji Owned:

```

modelBuilder.Entity<EntityOwner>().OwnsOne(p => p.OwnedEntity, od => {
    od.ToTable("OwnedEntityName");
});

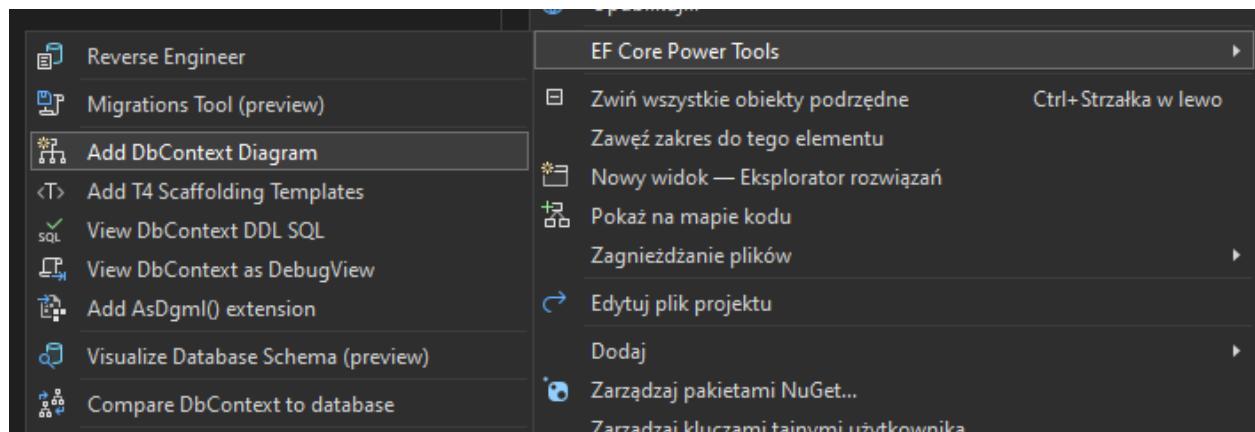
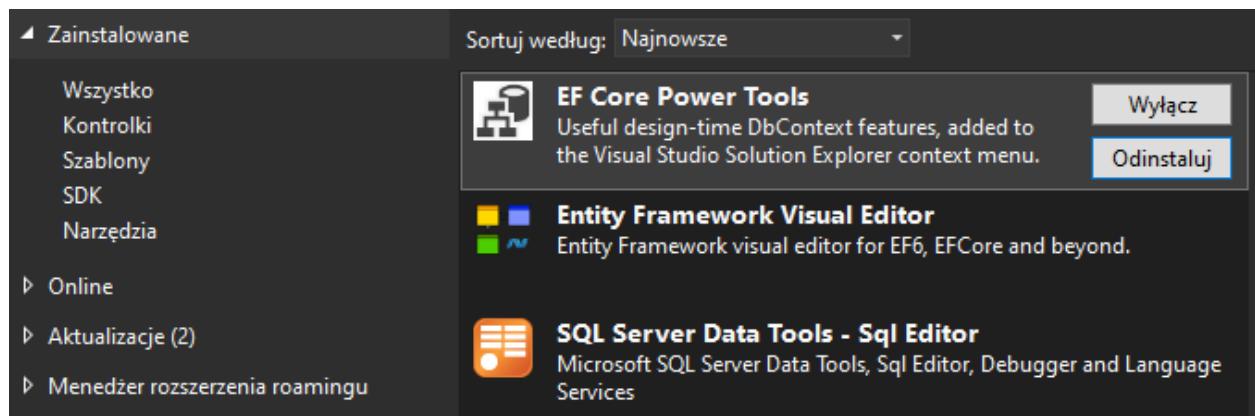
```

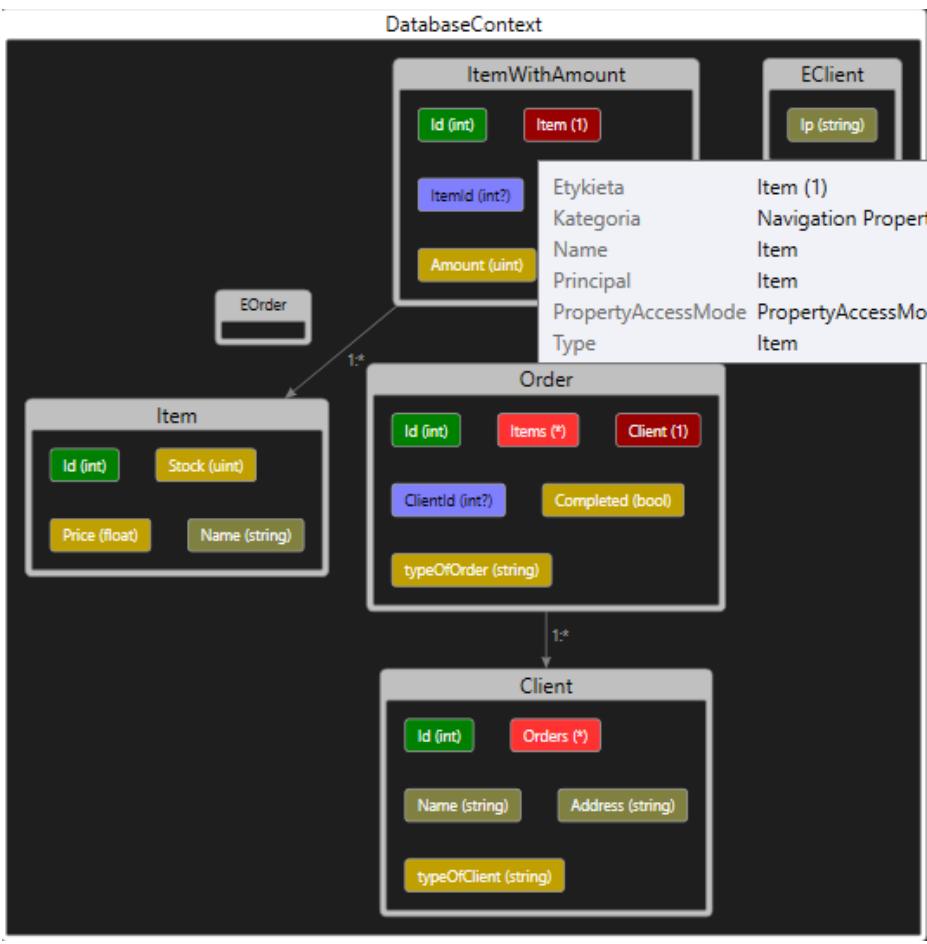
W celu zmiany tej nazwy można użyć metody `HasColumnName`. Podczas zapytania właściciela, posiadane typy Owned będą domyślnie uwzględniane. Nie jest konieczne użycie metody `Include`, nawet jeśli posiadane typy są przechowywane w oddzielnej tabeli.

Należy pamiętać, że nie można utworzyć `DbSet<T>` oraz wywołać w `ModelBuilder`, `Entity<T>()` gdzie `T` to encja Owned. Wystąpienia encji Owned nie mogą być współdzielone przez wielu właścicieli.

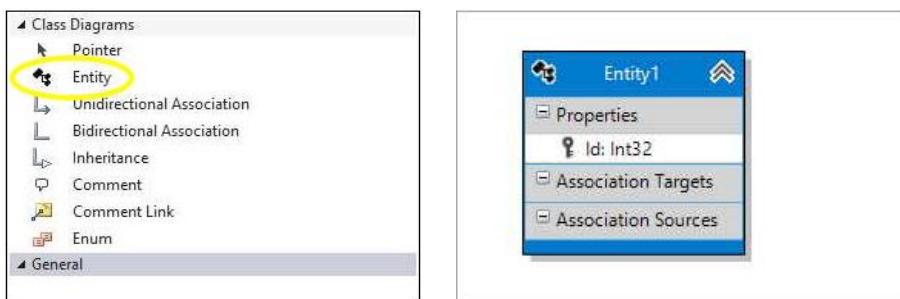
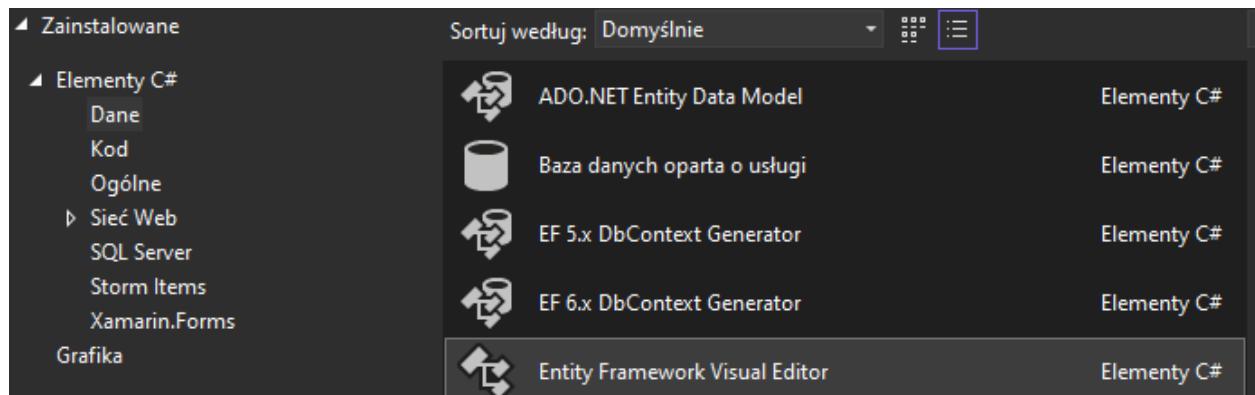
Podczas pracy z DBContextem mogą okazać się przydatne dwa rozszerzenia:

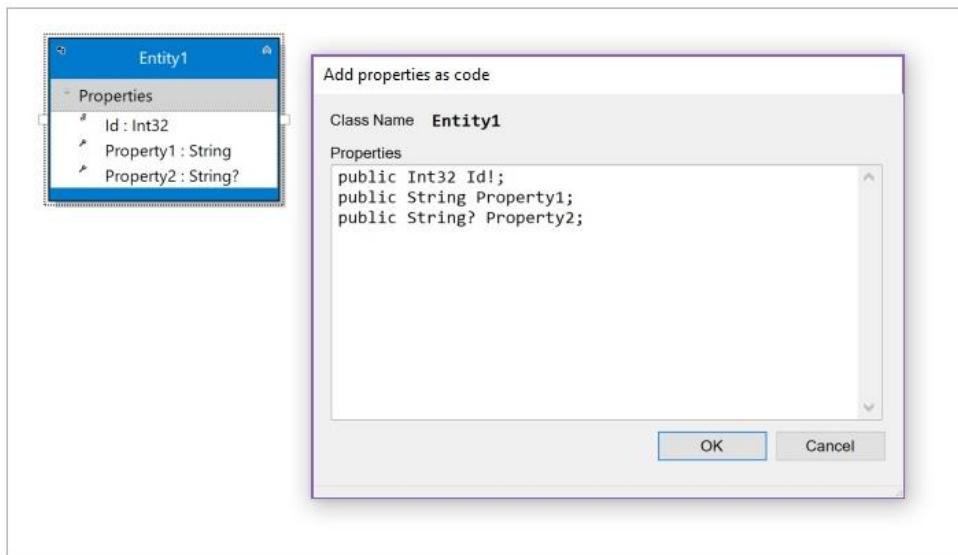
1. EF Core Power Tools – dodaje przydatne funkcje DbContext do menu kontekstowego Eksploratora rozwiązań.





2. Entity Framework Visual Editor – wizualny generator kodu, dodający nowy typ pliku (.efmodel), który pozwala na szybkie, łatwe projektowanie trwałych klas.





VIII. Połączenie z bazą danych i kontekst danych.

Cel: Konfiguracja połączenia z bazą danych i utworzenie kontekstu danych.

W celu utworzenia bazy danych musimy sprawdzić czy EF współpracuje z danym rodzajem bazy danych. Lista aktualnie wspieranych produktów jest spora i można ją zobaczyć tutaj: <https://learn.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli>.

Rozpoczynamy od utworzenia klasy reprezentującej kontekst danych naszej aplikacji.

Realizuje ona funkcjonalność wzorców projektowych:

- repozytorium – umożliwia realizację operacji CRUD (Create, Read, Update, Delete) dla danej klasy,
- jednostki pracy (unit of work) – założmy, że każdą zmianę w modelu obiektowym od razu odzwierciedlalibyśmy w bazie danych. Wymagało by to dużej liczby małych wywołań co spowodowało by spowolnienie systemu. Ponadto wymagało by to otwartej transakcji podczas całej tej operacji (aby móc odwołać wprowadzone zmiany w przypadku niepowodzenia), jest to niepraktyczne w przypadku złożonych transakcji biznesowych. Jednostka pracy śledzi wszystko, co może mieć wpływ na bazę danych. Zapis następuje poprzez wywołanie metody SaveChanges/SaveChangesAsync. Wszystkie wykryte operacje grupowane są w jedną transakcję, zapewniając albo jej powodzenie albo (w przypadku niepowodzenia) niezmieniony stan systemu.

W tym celu należy utworzyć klasę rozszerzającą klasę `DbContext` z EF, w której:

- dodajemy właściwości typu `DbSet< TEntity >` reprezentujące kolekcję wszystkich wystąpień klas naszego modelu które można odpisać z bazy danych. W miejscu parametru ogólnego `TEntity` podajemy główne klasy naszego modelu;
- Modyfikujemy klasę `DbContext`, aby przyjmowała konfigurację przez konstruktor, w postaci parametru obiekt `DbContextOptions`.
- W pliku `Program.cs` (dla .NET 6+), odczytujemy *connection string* (np. z `appsettings.json`) i przekazujemy go do metody `AddDbContext`. Używamy metod rozszerzeń `DbContextOptionsBuilder` pozwalających na konfigurację połączenia z bazą danych. Niektóre metody będą dostępne dopiero po zainstalowaniu dodatkowych

bibliotek a nowe metody powinny od razu widoczne ponieważ są one dodane jako metody rozszerzeń klasy `DbContextOptionsBuilder`:

- `UseSqlServer12` – konfiguruje kontekst do łączenia się z bazą danych lub plikiem bazy danych Microsoft SQL Server. Nas interesuje ta druga opcja. Jako parametr należy przekazać connection string w postaci: `"DataSource=(LocalDB)\\\<lokalna_instancja_SQL_Express>13;Initial Catalog=<nazwa_bazy_danych>;AttachDbFilename=./<bezwzględna_sciezka do_bazy_danych>.mdf;Integrated Security=True"`. Zaleca się stosowanie ścieżek bezwzględnych do pliku bazy danych.

- `UseSqlite14` – konfiguruje kontekst do łączenia się z bazą danych SQLite. Jako parametr należy przekazać connection string w postaci:

```
"DataSource=./<nazwa_pliku_bazy_danych>.db";
```

- `UseMySql15` – konfiguruje do łączenia się z bazą danych MySQL. Jako parametry należy przekazać:

- connection string w postaci: `"server=<adres_bazy_danych>;user id=<identyfikator_użytkownika>;password=<hasło użytkownika>;port=<port_na_którym_nasłuchuje_baza_danych_przewaźnie_3306>;database=<nazwa_bazy_danych>;"`;
- obiekt opisujący wersję serwera np. `new MariaDbServerVersion(new Version(10, 6, 5))`;

```
// Odczytanie connection stringa z appsettings.json
var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");

// Rejestracja DbContext
builder.Services.AddDbContext<MojaBazaDbContext>(options =>
{
    // Tutaj trafia logika wyboru dostawcy bazy danych:

    // o UseSqlServer[1] – konfiguruje kontekst do łączenia się z bazą
    // danych Microsoft SQL Server.
    // Zalecany connection string (np. dla LocalDB):
    //
    "Server=(localdb)\\mssqllocaldb;Database=NazwaBazyDanych;Trusted_Connection=True;" 
    options.UseSqlServer(connectionString);

    // o UseSqlite[3] – konfiguruje kontekst do łączenia się z
    // bazą danych SQLite.
    // Connection string: "DataSource=./nazwa_pliku.db"
    // options.UseSqlite(connectionString);

    // o UseMySql[15] – konfiguruje do łączenia się z bazą danych MySQL.
    // Connection string: "server=...;user id=...;password=...;database=..."
    // options.UseMySql(connectionString,
    //     new MariaDbServerVersion(new Version(10, 6, 5)));
});
```

¹² Wymaga zainstalowania biblioteki `Microsoft.EntityFrameworkCore.SqlServer`.

¹³ Do wyświetlenie listy aktualnie zainstalowanych instancji Microsoft SQL Server Express służy komenda: `SqlLocalDb info`.

¹⁴ Wymaga zainstalowania biblioteki `Microsoft.EntityFrameworkCore.Sqlite`.

¹⁵ Wymaga zainstalowania biblioteki `Pomelo.EntityFrameworkCore.MySql`.

```
// ... reszta pliku Program.cs ...
```

Connection String najlepiej przechowywać w pliku konfiguracyjnym `appsettings.json` (w nazwach ścieżek należy używać podwójnych znaków backslash `\\"`):

```
{  
    "ConnectionStrings": {  
        "msSqlCS": "Valid Connection String 1",  
        "mySqlCS": " Valid Connection String 2"  
    }  
}
```

Kontekst danych (a dokładniej klasa bazowa `DbContext`) udostępnia metody pozwalające manipulować stanem bazy danych:

- `SaveChanges` – Zapisuje w bazie danych wszystkie zmiany dokonane w tym kontekście.
- `Entry< TEntity >` – pobiera `EntityEntry< TEntity >` zapewniające dostęp do informacji o śledzeniu zmian i operacji dla danej encji. Posiada ono właściwość `State` typu `EntityState` określającą czy dany typ encji jest: { `Detached = 0`, `Unchanged = 1`, `Deleted = 2`, `Modified = 3`, `Added = 4` }.

Cykl życia instancji `DbContext` (od jej utworzenia do zwolnienia) jest zaprojektowany do użycia z pojedynczą jednostką pracy. Oznacza to, że okres jej istnienia jest zwykle bardzo krótki.

EF nie wspiera zarówno współbieżnych jak i asynchronicznych operacji wykonywanych na jednym kontekście danych. W związku z tym zawsze należy poczekać na zakończenie wywołania asynchronicznego lub używać oddzielnych instancji `DbContext` dla operacji wykonywanych równolegle.

Właściwości kontekstu danych reprezentujące wszystkie wystąpienia klas naszego modelu dziedziczą po klasie `DbSet< TEntity >`, która implementuje interfejs `IQueryable< TEntity >`. W takim przypadku naturalnym rozwiązaniem jest użycie LINQ do wykonywania zapytań, które zostaną przekonwertowane na zapytanie SQL.

```
public IEnumerable<Client> GetFilteredClients(string nameFilter = "") =>  
    _dbContext.Clients  
        // 1. Filtrowanie (WHERE) odbywa się w bazie danych  
        .Where(x => x.Name.StartsWith(nameFilter,  
            StringComparison.InvariantCultureIgnoreCase))  
        // 2. Sortowanie (ORDER BY) odbywa się w bazie  
        .OrderBy(x => x.Name)  
        // 3. Ograniczenie (TOP 10) odbywa się w bazie  
        .Take(10)  
        // 4. Dołączenie powiązanych danych (JOIN) tylko dla tych 10 rekordów  
        .Include(x => x.Orders)  
            .ThenInclude(x => x.Lines)  
            .ThenInclude(x => x.Item)  
        // 5. Pobranie 10 przefiltrowanych rekordów do pamięci  
        .ToList();
```

IX. Migracje.

Cel: Poznanie migracji w EF oraz podstawowych sposobów zarządzania nimi.

Po utworzeniu kontekstu danych i skonfigurowaniu połączenia z dostawcą bazy danych należy utworzyć pierwszą migrację. W tym celu przy pomocy zainstalowanych wcześniej narzędzi EF wykonujemy komendę: `add-migration FirstMigration` z wiersza poleceń menedżera pakietów. Polecenie to **nie modyfikuje bazy danych**, a jedynie tworzy w folderze `Migrations` dwa pliki:

- `<nazwa_kontekstu_danych>ModelSnapshot` – zawiera klasę dziedziczącą po klasie abstrakcyjnej `ModelSnapshot` z metodą `BuildModel`, pozwalającą na zbudowanie migawki modelu przy pierwszym żądaniu. Jest on "źródłem prawdy" dla EF. Kiedy tworzysz kolejną migrację, EF porównuje twój *aktualny* kod modelu z tym *snapshotem*, aby wygenerować różnice.)
- `<YYYYMMDDHHMMSS>_FirstMigration.cs` – zawiera klasę częściową `FirstMigration` dziedziczącą po `Migration` która zawiera kod do tworzenia tabeli na podstawie klas modelu. `<YYYYMMDDHHMMSS>` to moment (data i godzina) utworzenia migawki.

Aby fizycznie zastosować tę migrację do bazy danych, mamy dwie główne metody:

- Zastosowanie migracji w czasie deweloperskim (zalecane podczas pracy): W tej samej konsoli menedżera pakietów wykonujemy polecenie: `Update-Database` EF Core połączy się z bazą, sprawdzi tabelę `__EFMigrationsHistory` i uruchomi wszystkie oczekujące migracje.
- Zastosowanie migracji w czasie uruchomienia (Runtime): Możemy również skonfigurować aplikację tak, aby sama uruchamiała migracje przy każdym starcie. Służy do tego metoda `Database.Migrate()`.

Klasa `DbContext` z EF udostępnia właściwość która jest realizacją wzorca Fasada. Jego celem jest ujednolicenie dostępu, a w rezultacie ułatwienie wykorzystania złożonego systemu jakim poprzez wystawienie uproszczonego, uporządkowanego interfejsu programistycznego. W naszym przypadku jest to właściwość tylko do odczytu `Database` typu `DatabaseFacade` a systemem baza danych, zaś metody które mogą być przydatne to:

- `EnsureCreated` – Tworzy bazę danych jeśli nie istnieje, następnie jeśli w bazie nie istnieją żadne tabele to tworzony jest schemat bazy danych w oparciu o model EF, w przeciwnym przypadku nic się nie dzieje (istniejące tabele nie są sprawdzane pod kątem zgodności z modelem EF). Nie tworzy tabeli `__EFMigrationsHistory`, zatem nie jest wskazana jeśli chcemy korzystać z migracji, w jej miejsce lepiej wykorzystać metodę `Migrate`.
- `EnsureDeleted` – Usuwa bazę danych dla kontekstu jeśli taka istnieje. Często stosowana wraz z `EnsureCreated` do testowania albo prototypowania. Zapewnia że baza danych jest w stanie czystym przed każdym uruchomieniem testu/prototypu.
- `Migrate` – Jest to właśnie metoda do zastosowania migracji w czasie uruchomienia (Runtime). Utworzy bazę danych, jeśli jeszcze nie istnieje. Następnie aplikuje wszystkie oczekujące migracje kontekstu do bazy danych. Historia zaaplikowanych migracji przechowywana jest w dodatkowej tabeli `__EFMigrationsHistory` w `MigrationId` jako nazwy odpowiadających im plików (bez rozszerzeń .cs). Metoda ta jest dodawana w ramach klasy rozszerzającej `RelationalDatabaseFacadeExtensions` i wyklucza się wzajemnie z metodą `DatabaseFacade.EnsureCreated`, która nie używa migracji do tworzenia bazy

danych, a utworzona przez nią baza danych nie może być później aktualizowana za pomocą migracji.

X. Transakcje.

Cel: Zapoznanie ze wsparciem transakcji w EF.

Entity Framework Core zapewnia pełne wsparcie dla zarządzania transakcjami, ułatwiając możliwość samodzielne uruchamiania i wykonywania transakcji w ramach istniejącego kontekstu danych. Umożliwiająca łączenie kilku operacji w ramach tej samej transakcji, a zatem wszystkie one zostanie zatwierdzone albo wycofane. Pozwala także użytkownikowi łatwiej określić poziom izolacji transakcji. Fasada **Database** klasy **DbContext** udostępnia następujące metody:

- **BeginTransaction** – rozpoczyna nową transakcję.
- **UseTransaction** – pozwala wykorzystać przez **DbContext** transakcję, która została uruchomiona poza EF.

```
using (var dbContextTransaction = context.Database.BeginTransaction())
{
    // Używamy bezpiecznej metody ExecuteSqlInterpolated
    context.Database.ExecuteSqlInterpolated(
        $"UPDATE Blogs SET Rating = 5 WHERE Name LIKE '%Entity Framework%'");
}

var query = context.Posts.Where(p => p.Blog.Rating >= 5);      foreach (var post
in query)
{
    post.Title += "[Cool Blog]";
}
context.SaveChanges();
dbContextTransaction.Commit();
}
```

Pamiętajmy jednak, że `SaveChanges()` jest automatycznie transakcją. Domyślnie, każde wywołanie `context.SaveChanges()` jest automatycznie opakowywane w transakcję przez EF Core. Jeśli `SaveChanges()` ma zapisać 5 nowych Postów i zaktualizować 3 Blogi, EF Core sam uruchomi transakcję. Jeśli którakolwiek z tych 8 operacji się nie powiedzie, cała transakcja zostanie automatycznie wycofana (rollback), a baza danych pozostanie w nienaruszonym stanie. W 99% przypadków użytkownik nie musi ręcznie wywoływać `BeginTransaction`. Ręcznej transakcji używamy tylko w specjalnych przypadkach, gdy chcemy **połączyć wiele oddzielnych operacji w jedną**, atomową "jednostkę pracy".

XI. Błędy.

Cel: Opis najczęściej występujących wyjątków na laboratorium z EF Core i sposoby ich rozwiązywania.

System.InvalidOperationException:

- No database provider has been configured for this **DbContext**. A provider can be configured by overriding the '**DbContext.OnConfiguring**' method or by using '**AddDbContext**' on the application service provider. If '**AddDbContext**' is used, then also ensure that your **DbContext** type accepts a

`DbContextOptions<TContext>` object in its constructor and passes it to the base constructor for `DbContext`. – **Błąd ten oznacza, że `DbContext` nie wie, z jakiej bazy danych ma korzystać.** Należy sprawdzić:

- Czy w pliku `Program.cs` (lub `Startup.cs`) metoda `AddDbContext` jest poprawnie skonfigurowana i wywołuje metodę dostawcy (np. `.UseSqlServer(...)`).
- Jeśli nie używamy `AddDbContext`, należy sprawdzić, czy metoda `OnConfiguring` w klasie `DbContext` jest nadpisana i poprawnie konfiguruje dostawcę."

The entity type '`<nazwa_klasy_modelu>`' requires a primary key to be defined. If you intended to use a keyless entity type, call '`HasNoKey`' in '`OnModelCreating`'. For more information on keyless entity types, see <https://go.microsoft.com/fwlink/?linkid=2141943>. – **Błąd oznacza, że EF Core nie mógł znaleźć klucza głównego dla encji `<nazwa_klasy_modelu>`.** Należy sprawdzić:

- Czy klasa posiada właściwość pasującą do konwencji nazewniczej (np. `Id` lub `<nazwa_klasy_modelu>Id`).
- Jeśli klucz ma inną nazwę, czy został oznaczony atrybutem `[Key]` lub skonfigurowany za pomocą `.HasKey()` w `OnModelCreating`."
- The database name could not be determined. To use 'EnsureDeleted', the connection string must specify 'Initial Catalog'. – **EF Core obecnie nie obsługuje connection string'ów, które nie określają nazwy bazy danych podanych jako Initial Catalog.**
- Cannot attach file '`ściezka_do_pliku_bazy_danych`' as database '`<nazwa_bazy_danych>`' because this file is already in use for database '`<ściezka_do_pliku_bazy_danych>`'. – **Sprawdzić czy nie ma aktywnego połączenia z bazą danych. Przykładowo możemy posiadać aktywne połączenie w oknie Eksploratora serwera. W takim przypadku należy kliknąć prawym przyciskiem myszy na pliku bazy danych i z menu kontekstowego wybrać opcję „Zamknij połączenie”.**

`Microsoft.Data.SqlClient.SqlException`:

- A file activation error occurred. The physical file name '`<ściezka_do_pliku_bazy_danych>`' may be incorrect. Diagnose and correct additional errors, and retry the operation. "Cannot attach the file '`<ściezka_do_pliku_bazy_danych>.mdf`' as database '`<nazwa_bazy_danych>`'. – **sprawdzić czy ścieżka do pliku jest ścieżką względową, jeśli tak zmienić na bezwzględną, jeśli nie zobaczyć czy nie prowadzi ona do zasobu do którego mogą być wymagane specjalne uprawnienia.**

`Microsoft.EntityFrameworkCore.DbUpdateException`:

- Wewnętrzny wyjątek: `SqlException: Cannot insert the value NULL into column 'Id', table 'db dbo. <Nazwa_Encji>; column does not allow nulls. UPDATE fails.` **sprawdzić czy jak tworzona jest tablica w pliku migracji. Przykładowo dla `Sqlite` powinno być:**

```
migrationBuilder.CreateTable<
    name: "Entity",
    columns: table => new
    {
        Id = table.Column<int>(nullable: false)
```

```
.Annotation("Sqlite:Autoincrement", true),  
    ...  
, ...
```

Może się okazać, że migrację utworzono dla innego rodzaju bazy danych, np. MSSQL i wtedy powinno to wyglądać mniej więcej tak:

```
migrationBuilder.CreateTable(  
    name: "Items",  
    columns: table => new  
    {  
        Id = table.Column<int>(type: "int", nullable: false)  
            .Annotation("SqlServer:Identity", "1, 1"),  
        ...  
, ...
```

- Wewnętrzny wyjątek: SqlException: Invalid object name '<Nazwa_encji>'.
sprawdzić czy przeprowadzono migrację.