Bazy Danych

MINIPROJEKT

Autor: Tomasz Marek

Informatyka niestacjonarne

Semestr 7

1.Wstęp

1.1 Technologia

REST API przy użyciu **.Net 5.0**.

Komunikacja z bazą danych: **EF Core 5.0**.

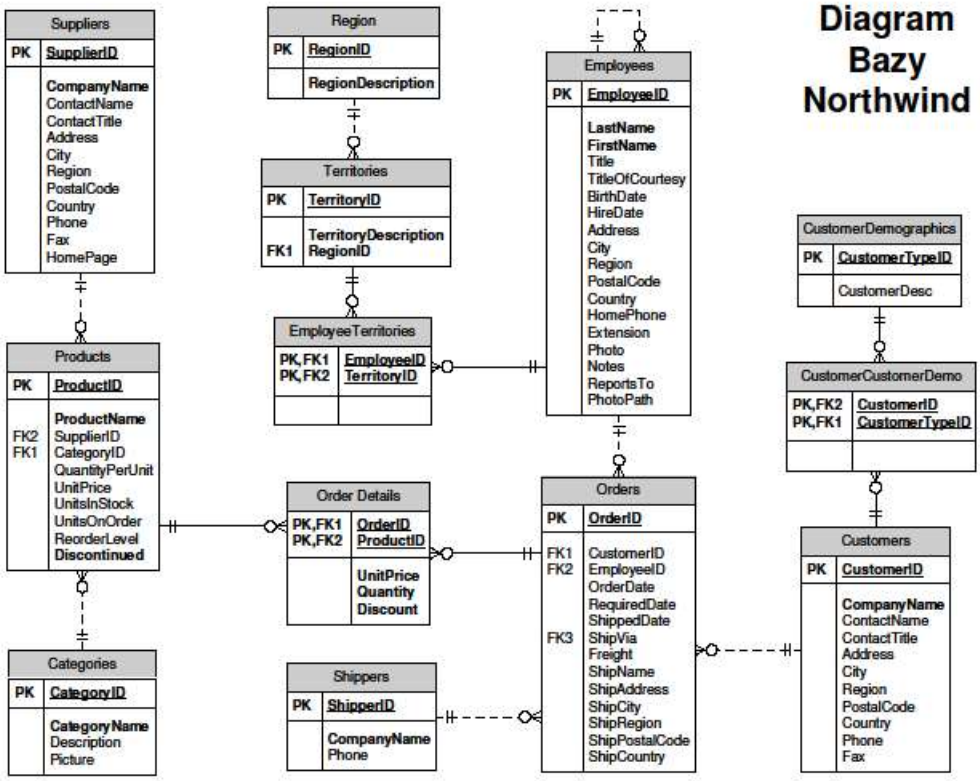
Baza danych: **MS SQL**.

Mapowanie między obiektami: **AutoMapper**.

Prezentacja endpointów oraz komunikacja z API: **Swagger**.

1.2 Baza danych

Założyłem że projekt ma na celu dostosowanie rozwiązania do narzuconego schematu bazy stąd baza została stworzona za pomocą zamieszczonego w opisie projektu skryptu, oraz testowana za pomocą zamieszczonych danych.



Do powyższego schematu nie wprowadzałem żadnych zmian.

W celu porównania i skorygowania różnić w mapowaniach pomiędzy modelami wykorzystałem mechanizmy migracji oraz scaffoldowania bazy zawarte w narzędziu **dotnet ef.**

2.Pominięte zagadnienia

Autoryzacja

W projekcie pominąłem zagadnienia autoryzacji osób komunikujących się z serwisem, jako że nie jest stricte związana z przedmiotem.

Obsługa wyjątków

Obsługa wyjątków wiązałaby się z implementacją middleware'a oraz klasy przechwytującej wyjątki, a następnie mapującej je na customowe komunikaty informujące o powodzie wyjątku w przyjazny dla użytkownika sposób(wraz z logowaniem oraz możliwością powiązania wpisu z komunikatem).

3.Struktura

3.1 Warstwy

Projekt składa się z 3 warstw

- warstwa dostępu do danych(DAL)
- warstwa serwisów(główna logika aplikacji)
- warstwa kontrolerów

3.1.1 Warstwa dostępu do bazy (DAL)

Warstwą dostępu do bazy jest klasa NorthwindDBContext.

Klasa ta odpowiada za:

- mapowania EF Core,
- uwidocznienie DBSet'ów
- generyczne operacje na nich wykonywane

W związku z faktem że operacje wykonywane na encjach są bardzo podobne(dużo boilerplate'u, wyodrębniłem zestaw generycznych operacji.

Są to operacje Get do zwracania:

- jednej encji dla klucza
- wielu encji dla wielu kluczy
- jednej encji dla zapytania zbudowanego dynamicznie przy pomocy obiektu GetSingleQueryOptions<T> zwracającego pierwszą encję po zastosowaniu

sortowania, filtrowania oraz dołączania powiązanych encji lub <T, TResult>, które dodatkowo umożliwia projekcję po stronie bazy.

- wielu encji dla obiektu GetQueryOption<T> lub <T, TResult>, analogicznie jak wyżej
- liczby encji zwracanych dla powyższych zapytań(wiele encji), na potrzeby kalkulacji liczby stron dla paginacji w interfejsie użytkownika.

Operacje Create, Update oraz Delete w wariantach pojedynczych oraz dla wielu encji

Funkcję SaveChangesAsync, pozwalającą na zachowanie transakcyjności przy wywoływaniu wielu operacji na poziomie serwisów.

Funkcję GetDBSet zwracającą DbSet odpowiadający podanemu typowi.

Opis

Aby zapewnić poprawne działanie funkcji generycznych poza klasą kontekstu(poza którą nie chcemy uwidaczniać DBSet'ów), zastosowałem refleksję, która w momencie tworzenia kontekstu tworzy statyczny słownik asocjujący typ encji z jej DBSet'em, dzięki temu możemy operować na Set'cie nie uwidaczniając go w serwisach, przekazując do funkcji generycznej tylko typ encji z którą chcemy pracować.

Aby udostępnić możliwość sortowania według wielu kolumn jednocześnie, musiałem zastosować refleksję do odnalezienia odpowiedniej metody rozszerzeń(limitacja języka oraz silnego typowania).

Zalety

Zaletą tego rozwiązania jest fakt, iż możemy wykonywać operacje na konkretnym set'cie w dowolnym serwisie, bez przymusu wstrzykiwania serwisów(pomaga uniknąć problemów związanych z cyklicznymi zależnościami, oraz pozwala na praktyczne wykorzystanie właściwości nawigacyjnych bez konieczności śmiecenia serwisów.

Warto wspomnieć, iż narzuca to pewną limitację(jeden DbSet na typ), co mimo wszystko jest bardzo łatwe do obejścia za pomocą dziedziczenia encji 1:1 np Order i ArchivizedOrder. Jest to nawet bardziej logiczne, gdyż te same pod względem struktury encje zazwyczaj mają inne znaczenie biznesowe, i nie powinny być rozpatrywane jako takie same.

3.1.2 Warstwa serwisów

Warstwa serwisów zawiera główną logikę aplikacji, tutaj także posłużyłem się generycznością i dziedziczeniem aby obsłużyć najbardziej standardowe operacje CRUD nie wymagające dodatkowej logiki.

To tutaj większość zapytań jest budowanych dynamicznie za pomocą wyżej wspomnianych obiektów. Są to przede wszystkim operacje niestandardowego mapowania encji, filtrowania, sortowania, załączania encji powiązanych oraz modyfikacji powiązanych encji.

3.1.3 Warstwa kontrolerów

Konwencje endpointów(ścieżek)

Endpointy(ścieżki) kierują się następującą konwencją.

- **{entities}** - nazwa encji w liczbie mnogiej
- **{entities}/{id}** - wskazanie konkretnej encji za pomocą modyfikatora
- **{entities1}/{id}/{entities2}** - wskazanie na encje(2) powiązane z encją(1)
- **{entities}/list** - wskazanie na widok listy(mapowanie encji na (w większości przypadków) lżejsze ich wersje na potrzeby wyświetlenia w widoku (np. listy)
- **{entities}/filter** - obsługa złożonych zapytań(filtrowanie, sortowanie, paginacja po stronie serwera)
- **{entities1}/{id1}/{entities2}/{id2}** - powiązanie encji(1) z encją(2) (dla relacji wiele do wielu)

Rezultaty dla endpointów

Powtarzelnymi rezultatami zapytań są

{Entity}ListView

Projekcja encji na model (w większości przypadków zminimalizowany) przeznaczony do wyświetlania w widoku listy.

{Entity}View

Projekcja encji stripująca właściwości nawigacyjne(relacji) modelu.

{Entity}

Encja odpowiadająca rekordowi w bazie

Endpointy

Endpointy /entities

Zwraca wszystkie encje, w większości przypadków służy tylko do debugu, możliwy do wykorzystania dla niewielkich encji (mała ilość, mały rozmiar)

Endpointy /entities/dictionary

Zwracają słownik wartości dla danej encji (tj. asocjacje id i wartości odpowiadającej nazwie danej encji). Do wykorzystania na potrzeby list wyboru w aplikacji. Pozwala na zminimalizowanie pobieranych danych i zmniejszenie zapytania w wypadku, gdy nie potrzebujemy wszystkich informacji związanych z daną encją, a chcemy jedynie zdefiniować relację do innej encji.

Endpointy /entities/{entityId}

Definiują szereg operacji zwracających pełen widok encji(wszystkie detale), oraz modyfikujących(PUT, POST, DELETE).

Endpointy /entities/list

Dla niektórych encji wyodrębniłem widok listy, jest to widok zawierający podstawowe dane dla encji, szerszy od słownika, przeznaczony do wyświetlenia w widoku grupowym np. listy. Na przykładzie zamówienia jest to widok zawierający ID, Status Zamówienia, wartość oraz nazwę klienta.

Endpointy entities/{entityId}/entities2

Podobnie jak powyższy zwraca widok listy dla encji powiązanych z daną encją

Endpointy entities1/{entity1Id}/entities2/{entity2Id}

Są to endpointy POST i DELETE pozwalające na tworzenie i usuwanie asocjacji pomiędzy encjami połączonymi relacją wiele do wielu, w przypadku aplikacji są to encje Terytoria-Pracownicy oraz Demografie-Klienci, nie obejmują za to relacji Zamównienia-Produkty, ponieważ ich encja asocjacyjna jest jawnie zaimplementowana w systemie, posiada własny kontroler.

Endpointy /entities/filter

Zwraca wynik skomplikowanych zapytań, pozwala na sortowanie, filtrowanie oraz paginację po stronie serwera za pomocą parametrów przesłanych z klienta(dynamicznie budowane zapytania do SQL).

Zrealizowanie jest za pomocą obiektu GetRequest, który zawiera grupy filtrujące pozwalające na budowanie złożonych filtrów oraz sortowania(znowu przy użyciu refleksji). Zaimplementowanie wymagało kilku obejść, które ponownie związane są z limitacjami języka statycznie typowanego.

W Api zapytania te zaimplementowane są za pomocą POST'a, co jest związane z limitacją Swaggera, która nie pozwala na przesłanie Body request'a w GET'cie. Docelowo jednak powinien być to GET. Do użytku w przypadku, w którym rozmiar i/lub ilość encji wpływają na znaczne spowolnienie działania aplikacji. W api zaimplementowane dla encji orders.

Raporty

Od powyższej konwencji (nie licząc pojedyńczych utility endpointów), nie przestrzegają raporty, które same z siebie ciężko rozwiązać generycznie. Każdy raport jest unikalny(mniej lub bardziej), a co za tym idzie dynamiczne budowanie zapytań wiązałoby się z parsowaniem i przesyłaniem z klienta lambd, co jest średnio bezpieczne. Można jednak w bardzo prosty sposób dodawać nowe raporty korzystając z generycznego interfejsu bazy(wymaga to jednak znajomości intrykacji EF Core, który ma problem z przetłumaczeniem zapytań.

Podsumowanie kontrolerów

Podsumowanie - udostępnienie takiej gamy endpointów pozwala na łatwą i szybką customizację działania systemu. W bardzo prosty sposób możemy przepiąć się z jednego enpointa na drugi, co zapewnia lepszą skalowalność i łatwiejsze utrzymanie.

Projekcje po stronie api oraz dynamiczne budowanie zapytań minimalizuje także problemy związane z utrzymaniem porządku w bazie(brak niepotrzebnych lub bardzo podobnych widoków, procedur itp.

Encje - założenia, relacje i wątpliwości

Przy implementacji UI każda encja posiadałaby osobny widok zawierający zakładkę z detalami oraz zakładki zawierające listy encji powiązanych.

Encje powiązane

Przy tworzeniu encji powiązanych relacją jeden do wielu obsłużyłem możliwość natychmiastowego tworzenia powiązanych encji. Po wstępnym utworzeniu encje powiązane edytowane byłyby pojedyńczo, z poziomu widoku encji nadrzędnej.

Regiony

Regiony pozwalają przy tworzeniu na natychmiastowe stworzenie powiązanych terytoriów. Następnie terytoria dodawane, usuwane i edytowane są już z poziomu widoku detali danego regionu.

Ordery

Ordery pozwalają na analogiczne tworzenie detali, powyższa logika jest jednak rozszerzona o modyfikację stanów magazynowych produktów. Dodatkowo detale są w pełni zależne od ordera, dlatego zastosowałem kaskadowe usuwanie.

Wątpliwości

Edycja zamówenia

Nie byłem pewien scope'u w jakim dozwolona powinna być edycja zamówienia, są to zagadnienia związane silnie z logiką biznesową aplikacji, tak więc postanowiłem zachować je w jak najprostszej formie. Rozważałem jednak dodanie endpointów zmieniających status

zamówienia(poinformowanie o wysłaniu i dostarczeniu zamówienia), np. na potrzeby integracji z serwisem doręczyciela(serwis informuje nasze api o dostarczeniu zamówienia).

Operacje na detalach zamówienia

Wątpliwości budziła także we mnie kwestia edycji detali zamówienia(w realnych warunkach taka operacja wiąże się zazwyczaj z anulacją zamówienia i stworzeniem nowego). Postanowiłem zostawić jednak taką możliwość ze względu na fakt, że system nie posiada informacji o statusie zamówienia(anulacji), a nie chcemy tracić takiej informacji + możliwość ta jest bardzo łatwa do zablokowania(ukrycie endpointa).

Pola dla zminimalizowanych widoków(/list)

Nie byłem także do końca pewien pól, które powinienem zamieścić w widokach zminimalizowany(tj. widokach listy) dla poszczególnych encji. Czasami ciężko jest określić które informacje są faktycznie potrzebne dla end usera, a które nie do końca. Problem ten jest jednak częściowo niwelowany przez powiązania nawigacyjne w aplikacji.

Ulepszenia, opcjonalne, refaktoryzacje

Terytoria są w pełni zależne od regionów

Zastosowałem kaskadowanie przy usunięciu, co skutkuje nullowaniem relacji z pracownikiem, opcjonalną operacją byłoby udostępnienie możliwości przepięcia terytorium na inny region, lub pracowników terytorium na inne terytorium co pozwoliłoby na zachowanie istniejących relacji.

Rozszerzenie orderów o status

(oprócz dat, z których teoretycznie możemy w taki czy inny sposób taki status wywnioskować).

Rozszerzenie kategorii o relację rekurencyjną

(zagnieżdżane kategorie, wiele kategorii dla jednego produktu).

Rozszerzenie grup demograficznych o nazwę.

Aktualna encja nie posiada skrótowej formy reprezentacji grupy na potrzeby asocjacji z klientami.

Encja pracowników powinna zawierać role, pozycję(wartość enumowa, nie varchar)

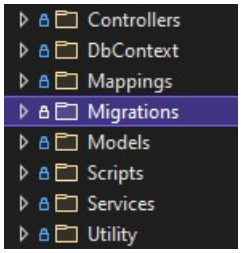
Na potrzeby filtrowania tylko pracowników dokonujących sprzedaży(raportowanie), w aktualnym schemacie nie jesteśmy w stanie rozgraniczyć pracowników najgorzej sprzedających od tych którzy nie sprzedają bo nie należy to do ich obowiązków.

4.Kod

Wstęp

W tej części zamieszczę konwencje oraz co ciekawsze kwestie od strony kodu.

Struktura kodu

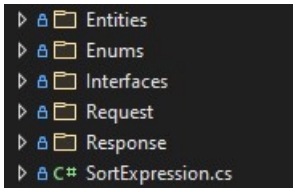


Folder Controllers zawiera kontrolery

DbContext zawiera Kontekst bazy oraz modele używane do querowania bazy.

Mappings zawiera mapowania AutoMappera

Models zawiera:



Encje, enumy, interfejsy, modele zapytań oraz modele odpowiedzi API

Scripts zawiera skrypty użyte w projekcie(w większości skrypty załączone do projektu)

Services zawiera serwisy

Utility zawiera klasy pomocnicze, między innymi rozszerzenia pozwalające na znajdowanie generycznych override'ów funkcji na typach, oraz PropertyAccessor, o którym mowa jest później

Generyczność

Kontekst bazy

Aby ograniczyć boilerplate i powtarzalność kodu stworzyłem zestaw generycznych operacji wykonywanych na encjach. Pozwala to na bardzo szybkie i łatwe zmiany w logice aplikacji.

Przykładowa funkcja

```
Odwołania: 23
public async Task<IEnumerable<TResult>> GetEntities<T, TResult>
    (GetQueryOptions<T, TResult> opt) where T : class
{
    var set = GetDbSet<T>();

    var query = set.AsQueryable();

    var projectedQuery = opt.BuildQuery(query);

    return await projectedQuery.ToListAsync();
}
```

Serwisy

Aby ograniczyć boilerplate dla standardowych operacji stworzyłem generyczny serwis **CrudServiceBase<T, KeyType>**, który rozszerza generyczność DbContext'tu na serwisy,

T odpowiada typowi encji

KeyType odpowiada typowi klucza encji (dla większości int, dla customera string).

Wymaganym jest, aby encja implementowała **IIdentificable<KeyType>** w celu zapewnienia właściwości **Id**, według której wykonywane są generyczne zapytania **Get**.

Refleksja

Kontekst bazy

Za pomocą refleksji na startupie rejestrowane są mapowania pomiędzy typami a DbSet'ami.

Dzięki temu możemy wywnioskować DbSet na którym powinniśmy pracować przekazując

tylko typ encji

```
private DbSet<T> GetDbSet<T>() where T : class
{
    Type t = typeof(T);

    var set = SetDictionary[t]?.GetValue(this) as DbSet<T> ?? null;

    return set;
}
```

Pozwala nam to na np. coś takiego

```
public async Task<IEnumerable<T>> Get(IEnumerable<KeyType> ids)
{
    return await _context.GetEntities<T, KeyType>(ids);
}
```

Co skutkuje takimi wywołaniami

```
var result = await _orders.Get(orderId);
```

Złożone wyrażanie zwracające Get

Refleksa użyta jest także aby dynamicznie składać zapytania zwracające wyniki na podstawie obiektów **GetQueryOptions<T>** oraz **GetSingleQueryOptions<T>**

```
public class GetQueryOptions<T> where T : class
{
    Odwołania: 19
    public Expression<Func<T, bool>> Filter { get; set; }
    Odwołania: 12
    public int Count { get; set; }
    Odwołania: 4
    public int Skip { get; set; }
    1 odwołanie
    public bool Reverse { get; set; }
    Odwołania: 18
    public Func<IQueryable<T>, IQueryable<T>> Includes { get; set; }

    Odwołania: 4
    public IEnumerable<SortExpression> Sort { get; set; }

    Odwołania: 3
    public IQueryable<T> BuildQuery(IQueryable<T> query)
```

Rozwiązanie to pozwala na budowanie zapytań według wcześniej ustalonego kontraktu

Przykładowe wywołanie może wyglądać tak

```
public async Task<IEnumerable<OrderListView>> GetFilteredOrdersListView(GetRequest req)
{
    var listBase = await _context.GetEntities(new GetQueryOptions<Order, OrderListViewBase>()
    {
        Includes = (IQueryable<Order> q) => q.Include(o => o.OrderDetails).ThenInclude(od => od.Product),
        Skip = req.ItemsPerPage * req.Page,
        Count = req.ItemsPerPage,
        Sort = req.Sorting?.GetSortExpressions<Order>(),
        Filter = req.Filters?.GetExpression<Order>(),
        Select = (IQueryable<Order> q) => q.Select(o => new OrderListViewBase()
        {
            Id = o.Id,
            OrderDate = o.OrderDate,
            RequiredDate = o.RequiredDate,
            ShippedDate = o.ShippedDate,
            Total = Convert.ToDouble(o.OrderDetails.Sum(od => od.UnitPrice * od.Quantity))
        }),
    });
```

Pojawia się tu także kolejna ciekawa kwestia, mianowicie dynamiczne budowanie zapytań według requesta(zapytania budowane są w ten sposób, aby wykonywane były po stronie bazy danych).

```
public class GetRequest
{
    Odwołania: 2
    public int Page { get; set; }
    Odwołania: 4
    public int ItemsPerPage { get; set; }

    Odwołania: 3
    public FilterGroup Filters { get; set; }

    Odwołania: 2
    public SortGroup Sorting { get; set; }
}
```

```
public class FilterGroup
{
    1 odwołanie
    public FilterGroupOperator Operator { get; set; }
    1 odwołanie
    public IEnumerable<Filter> Filters { get; set; }
    Odwołania: 7
    public IEnumerable<FilterGroup> Groups { get; set; }

    Odwołania: 6
    public Expression<Func<T, bool>> GetExpression<T>()
```

Przykładowe zapytanie do bazy wygląda następująco

```
{
  "page": 1,
  "itemsPerPage": 10,
  "filters": {
    "operator": 2,
    "filters": [],
    "groups": [
      {
        "operator": 1,
        "filters": [
          {
            "property": "name",
            "operator": 4,
            "value": "a"
          },
          {
            "property": "name",
            "operator": 4,
            "value": "b"
          }
        ]
      },
      {
        "operator": 1,
        "filters": [
          {
            "property": "name",
            "operator": 4,
            "value": "c"
          }
        ]
      }
    ]
  },
  "sorting": {
    "sort": [
      {
        "property": "string",
        "ascending": true
      }
    ]
  }
}
```

Zapytanie skutkuje to zwróceniem pierwszych 10 elementów które zawierają w nazwie a i b, lub c, posortowane rosnąco według nazwy.

Dostęp do właściwości po samej nazwie także wymagał refleksji

Udało się go zrealizować za pomocą klasy PropertyAccessor w połączeniu z zdefiniowanymi na modelu statycznymi mapami właściwości(co pozwala kontrolować, które właściwości można accessować).

```
public static class PropertyAccessor<T>
{
    private static Dictionary<Type, int> TupleTypeMap =
        new Dictionary<Type, int>() {
        { typeof(Category), 1 },
        { typeof(Order), 2 }
    };

    public static Tuple<
        Dictionary<string, LambdaExpression>,
        Dictionary<string, LambdaExpression>> PropAccessors
            = new Tuple<
                Dictionary<string, LambdaExpression>,
                Dictionary<string, LambdaExpression>>(
                    Category.PropDictionary,
                    Order.PropDictionary
                );

    public static Tuple<
        Dictionary<string, PropertyInfo>,
        Dictionary<string, PropertyInfo>> PropInfoAccessors =
        new Tuple<Dictionary<string, PropertyInfo>, Dictionary<string, PropertyInfo>>
        (
            Category.PropInfoDictionary,
            Order.PropInfoDictionary
        );
```

```
public class Order : IIdentificable<int>
{
    1 odwołanie
    public static Dictionary<string, LambdaExpression> PropDictionary { get; }
        = new Dictionary<string, LambdaExpression>()
        {
            { "orderDate", (Expression<Func<Order, DateTime?>>)((Order o) => o.OrderDate) },
            { "requiredDate", (Expression<Func<Order, DateTime?>>)((Order o) => o.RequiredDate) },
            { "shippedDate", (Expression<Func<Order, DateTime?>>)((Order o) => o.ShippedDate) },
            { "shipName", (Expression<Func<Order, string>>)((Order o) => o.ShipName) },
            { "shipAddress", (Expression<Func<Order, string>>)((Order o) => o.ShipAddress) },
            { "shipCity", (Expression<Func<Order, string>>)((Order o) => o.ShipRegion) },
            { "shipPostaCode", (Expression<Func<Order, string>>)((Order o) => o.ShipPostalCode) },
            { "shipCountry", (Expression<Func<Order, string>>)((Order o) => o.ShipCountry) }
        };

    1 odwołanie
    public static Dictionary<string, PropertyInfo> PropInfoDictionary { get; set; }
        = new Dictionary<string, PropertyInfo>()
        {
            { "orderDate", typeof(Order).GetProperty("OrderDate") },
            { "requiredDate", typeof(Order).GetProperty("RequiredDate") },
            { "shippedDate", typeof(Order).GetProperty("ShippedDate") },
            { "shipName", typeof(Order).GetProperty("ShipName") },
            { "shipAddress", typeof(Order).GetProperty("ShipAddress") },
            { "shipCity", typeof(Order).GetProperty("ShipCity") },
            { "shipPostaCode", typeof(Order).GetProperty("ShipPostalCode") },
            { "shipCountry", typeof(Order).GetProperty("ShipCountry") }
        };
```