

## 1. Wstęp

Naszym zadaniem jest stworzenie podstawowego asynchronicznego Web API dla serwisu zarządzającego najmem nieruchomości. Podstawowymi funkcjonalnościami serwisu mają być:

- Logowanie administratora
- Dodanie nowej nieruchomości do bazy
- Edycja nieruchomości
- Wyświetlanie nieruchomości
- Usuwanie nieruchomości
- Przypisanie właściciela oraz osoby która wynajmuje nieruchomość

## 2. Co obejmuje ten dokument?

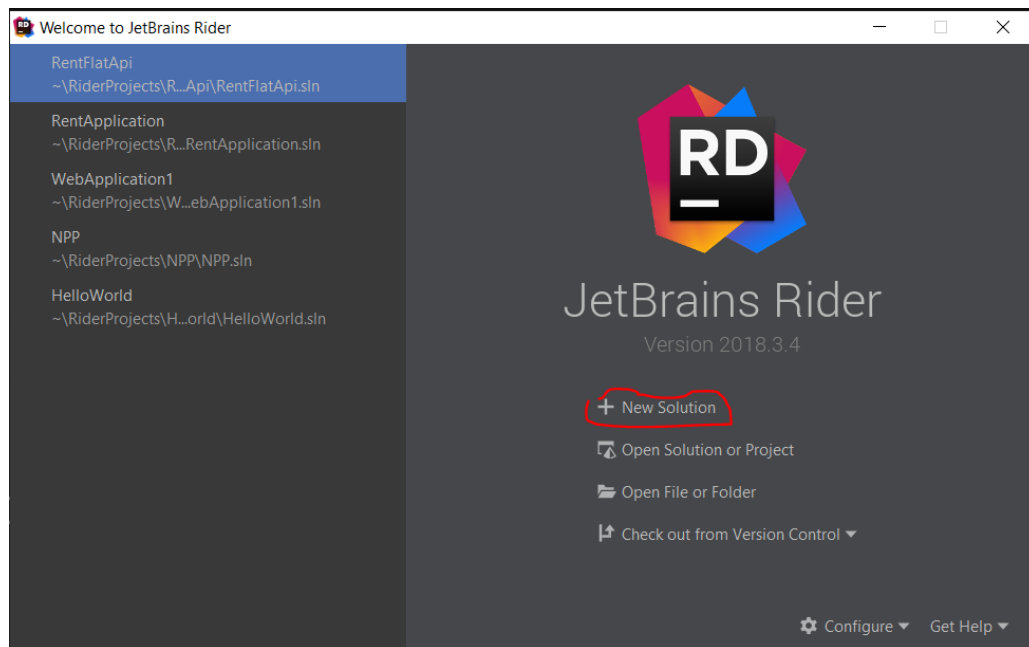
- Podstawy tworzenia Web API na podstawie ASP.NET Core 2.2
- Implementację mapowania obiektowo-relacyjnego przy pomocy EntityFramework Core 2.2
- Przykład Clean Code Architecture
- Całość dokumentu będzie oparta na pracy przy pomocy środowiska Rider od JetBrains. Osoby korzystające z Visual Studio przy problemach odsyłam do dokumentacji Microsoftu.

## 3. Co potrzebujemy?

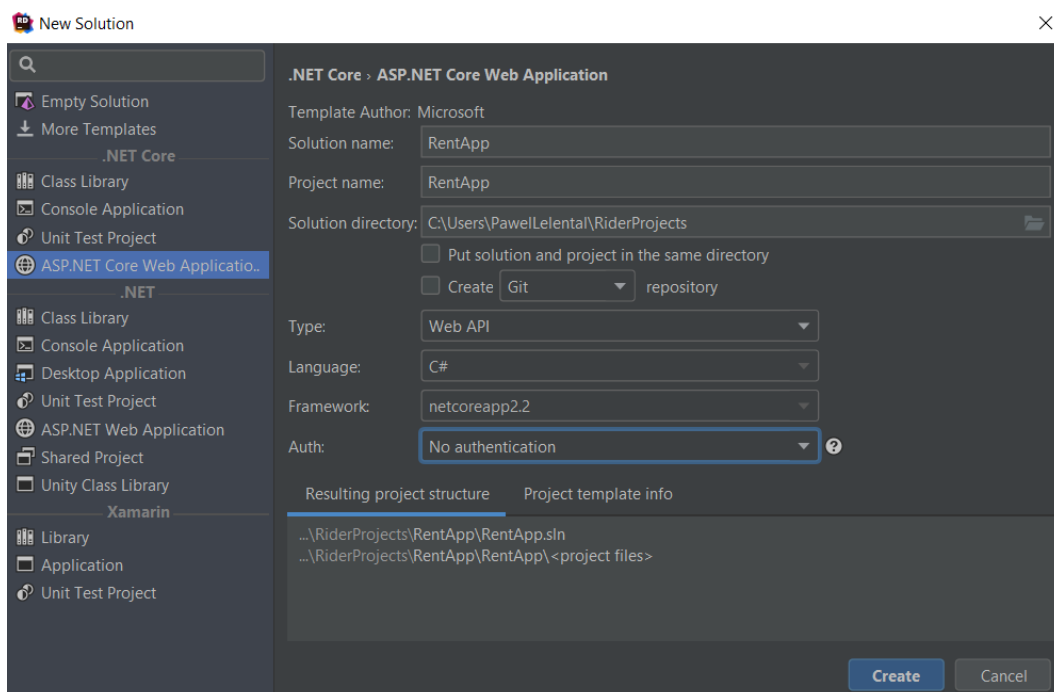
- a. .Net Core 2.2
- b. Sqlite 2.3
- c. DB Browser dla Sqlite
- d. JetBrains Rider/ Visual Studio Professional 2019

## 4. Tworzenie projektu

Zaczynamy od uruchomienia IDE. Pierwsze co pojawi się nam po załadowaniu środowiska to panel z listą wcześniejszych projektów. W okienku wybieramy *New Solution*.

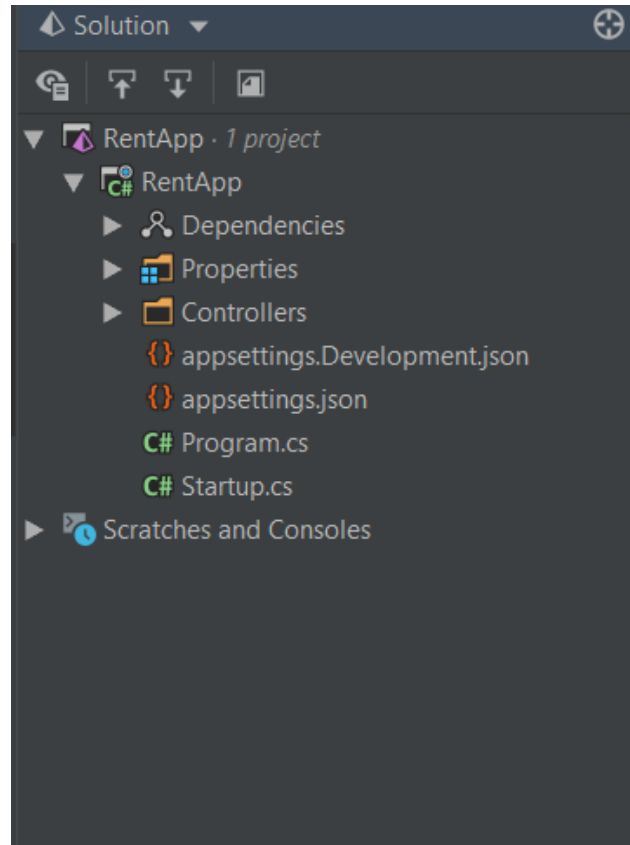


Następnie wybieramy rodzaj naszego nowo tworzonego projektu: *ASP.NET Core Web Application*. W oknie dialogowym wpisujemy główną nazwę solucji oraz nazwę projektu. W tym przypadku wpisujemy w oba pola *RentApp*. Niżej musimy jeszcze wybrać typ aplikacji, język, framework oraz czy chcemy by środowisko wygenerowało nam podstawową implementację uwierzytelniania. Kolejno zaznaczamy: Type – *Web API*, Language – *C#*, Framework – *netcoreapp2.2* i w ostatnim polu wybieramy *No authentication*. Po wszystkim klikamy przycisk *Create*.



## 5. Hierarchia projektu

Po stworzeniu projektu, widzimy że świeżo wygenerowany szkielet aplikacji prezentuje się następująco:



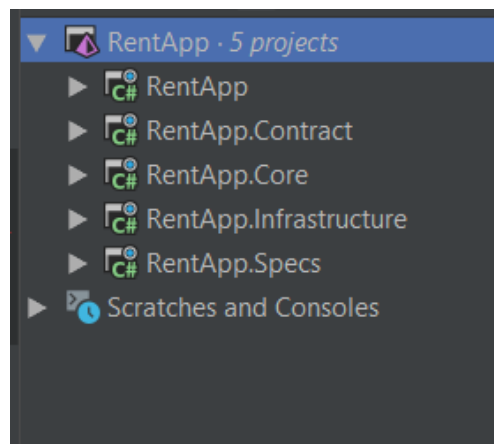
Gdzie odpowiednio każdy z folderów/plików odpowiada:

- **Dependencies** – zależności projektu
- **Properties** – pliki konfiguracyjne (rodzaj serwera, używane porty, środowisko)
- **Controllers** – katalog w którym umieszczone są kontrolery Restowe
- **Appsettings.json/appsettings.development.json** – pliki z ustawieniami aplikacji (w nich możemy trzymać url do połączenia z bazą danych)
- **Program.cs** – główna klasa uruchamiająca aplikację
- **Startup.cs** – klasa konfiguracyjna aplikacji

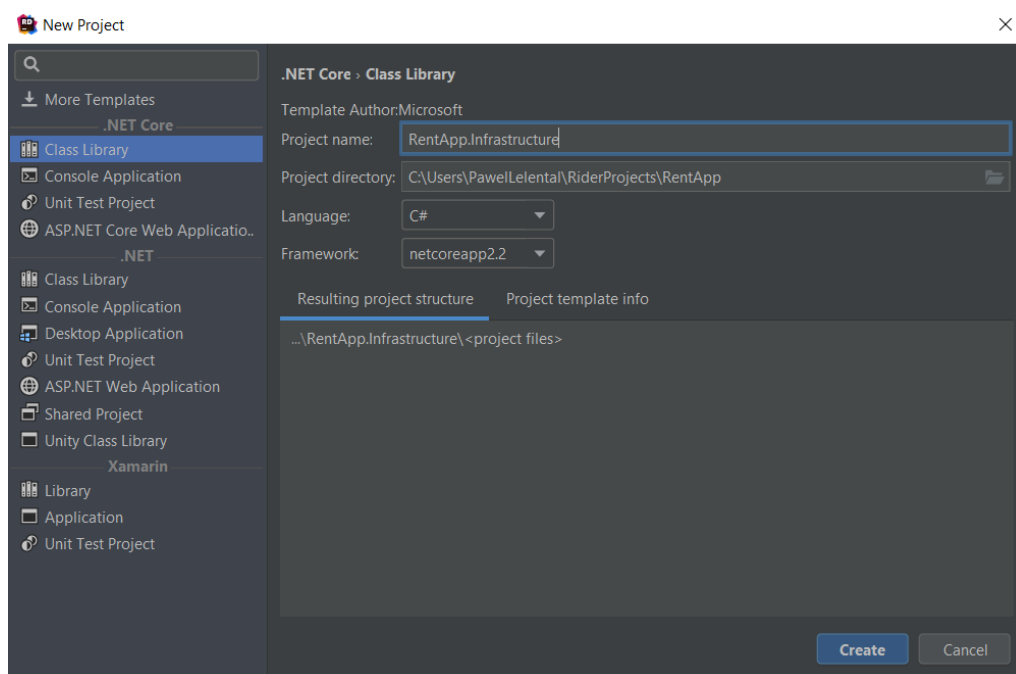
## 6. Projekt architektury aplikacji

Zgodnie Clean Architecture, stworzymy łącznie 5 podprojektów, odpowiednio rozdzielających logikę:

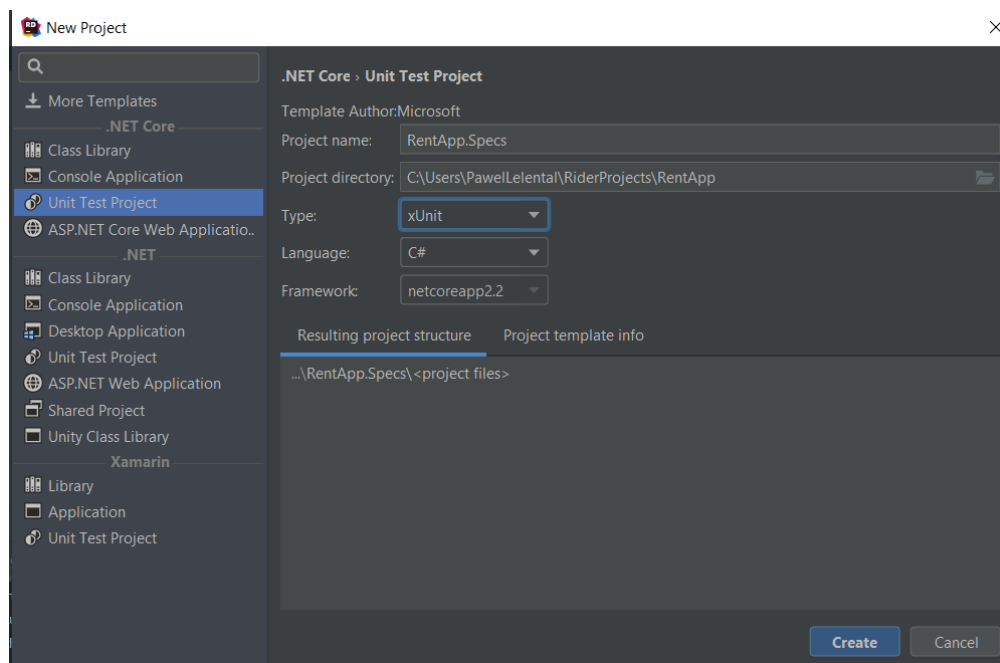
- **RentApp** – główny projekt, zawierający logikę do uruchomienia aplikacji i Controllery Restowe
- **RentApp.Infrastructure** – projekt który przechowuje całą logikę biznesową w tym komunikację z bazą danych
- **RentApp.Core** – projekt z logiką aplikacji, będącą pośrednikiem pomiędzy warstwą prezentacji, a warstwą biznesową
- **RentApp.Contract** – projekt przechowujący modele Dto (Data Transfer Object)
- **RentApp.Specs** – projekt z testami jednostkowymi



Aby stworzyć nowy projekt należy kliknąć prawym przyciskiem myszy w *Solution*, wybrać **Add->Class Library**.



Aby stworzyć nowy projekt należy kliknąć prawym przyciskiem myszy w Solution, wybrać **Add->Unit Test Project**. W okienku wybieramy jakiego frameworka do testów chcemy użyć. W tym projekcie wybór padł na **XUnit**.



## 7. Projekt encji wraz z klasami bazowymi

W tym momencie powinniśmy zastanowić się jak powinna wyglądać nasza aplikacja. Dobrą praktyką podczas analizowania naszych funkcjonalności jest faktyczne rozrysowanie modelu bazy danych. Źle przemyślane encje, potrafią się ciągnąć za programistą przez cały okres implementacyjny oraz utrzymywania aplikacji. W projekcie wykorzystamy ORM, co przekłada się na zaprojektowanie klas w C#, które to w dalszych krokach zostaną przełożone wraz ze swoimi proporcjami na odpowiednie typy bazodanowe, co zakończy się wygenerowaniem tabel i relacji w bazie. Plusem zastosowania tej metody jest elastyczność – nie jesteśmy ograniczeni do jednego dostawcy/producenta, przez co jeśli będzie istniała konieczność zmiany (np. z MS SQL na PostgreSQL), edycja ograniczy się do kilku linijek w kodzie, a nie do zmieniania całych skryptów SQL! (bazy danych różnią się od siebie np. nie wszystkie posiadają identyczne typy danych).

Zacniemy od zaprojektowania podstawowej klasy *Entity* w projekcie *RentApp.Infrastructure*, która to będzie dziedziczona przez pozostałe klasy. Jej podstawowe zmienne to: id, data utworzenia, data edycji. Id będzie naszym generowanym kluczem głównym. W tym celu powyżej proporcji dodajemy odpowiednie adnotacje.

```
4 usages 7 inheritors veok 0+16 exposing APIs
public class Entity
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    4 usages
    public long Id { get; set; }
    2 usages
    public DateTime DateOfCreation { get; set; }
    1 usage
    public DateTime DateOfUpdate { get; set; }
}
```

Jedną z podstawowych funkcjonalności jest zapewnienie logowania do systemu dla administratora. Jednakże musimy jeszcze zwrócić uwagę że aplikacja ma zawierać logikę do przypisywania do lokali osób które wynajmują oraz właściciela nieruchomości. W tym celu wydzielimy klasę *Person* która będzie dziedzyczyła klasę *Entity* oraz implementowała klasę *Address* zawierającą dane adresowe. Następnie stworzymy kolejno klasy: *User* – użytkownik systemu który będzie się logował, *Owner* – właściciel nieruchomości i *Tenant* – osoba wynajmująca. Wszystkie te trzy klasy będą rozszerzeniem klasy *Person*, a dzięki temu że już wcześniej do tej klasy przypisaliśmy *Entity*, będą one posiadały jej pola. Klasy *Owner* oraz *Tenant* z kolei posiadają informację o nieruchomościach. Dla wszystkich klas tworzymy folder *Model* i je w nim umieszczamy.

3 usages veok 4 exposing APIs

```
public class Address : Entity
{
    public string Street { get; set; }
    public string City { get; set; }
    public string ZipCode { get; set; }
    public string Country { get; set; }
}
```

3 usages 0 inheritors veok 0 exposing APIs

```
public class Person : Entity
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string PhoneNumber { get; set; }
    public Address Address { get; set; }
}
```

1 usage veok

```
public class User : Person
{
    public string Username { get; set; }
    public string PasswordHash { get; set; }
    public bool IsActive { get; set; }
}
```

2 usages veok 3 exposing APIs

```
public class Owner : Person
{
    1 usage
    public List<Flat> Flats { get; set; }
}
```

3 usages veok 3 exposing APIs

```
public class Tenant : Person
{
    1 usage
    public Flat Flat { get; set; }
}
```

Jak wyżej zostało wspomniane, *Owner* i *Tenant* posiadają informacje o nieruchomościach – przyjmijmy że właściciel może mieć kilka lokali, zaś wynajmujący może wynajmować tylko jeden. Implementacja klasy *Flat* natomiast wygląda następująco:

```
14 usages veok 6 exposing APIs
public class Flat : Entity
{
    public decimal Price { get; set; }
    public Address Address { get; set; }
    public string District { get; set; }
    public int NumberOfRooms { get; set; }
    public int SquareMeters { get; set; }
    2 usages
    public IEnumerable<Image> Images { get; set; }
    public int Floor { get; set; }
    public bool IsElevator { get; set; }
    1 usage
    public Owner Owner { get; set; }
    1 usage
    public Tenant Tenant { get; set; }
}
```

Jak widać ta klasa posiada odwołanie do *Image*. Będzie to model przechowujący *byte array* w naszej bazie danych. Jedna nieruchomość może mieć kilka/kilkanaście wgranych zdjęć.

```
2 usages veok
public class Image : Entity
{
    public byte[] Data { get; set; }
    1 usage
    public Flat Flat { get; set; }
}
```

Na ten moment mamy gotowy podstawowy model klas. W następnym punkcie zostanie omówiona konfiguracja EntityFramework Core 2.2, wraz z implementacją i wygenerowaniem tabel w bazie danych na podstawie stworzonego wyżej modelu.

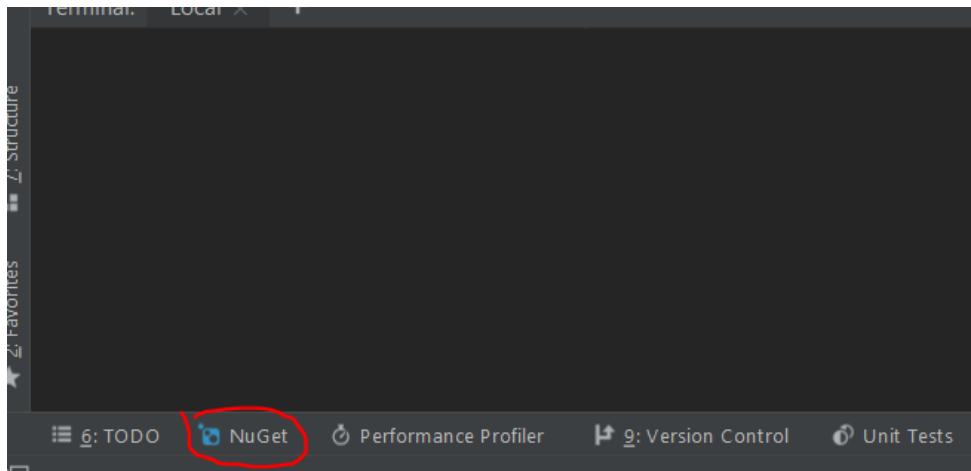


## 8. Połączenie z bazą danych, czyli konfiguracja EntityFramework Core 2.2

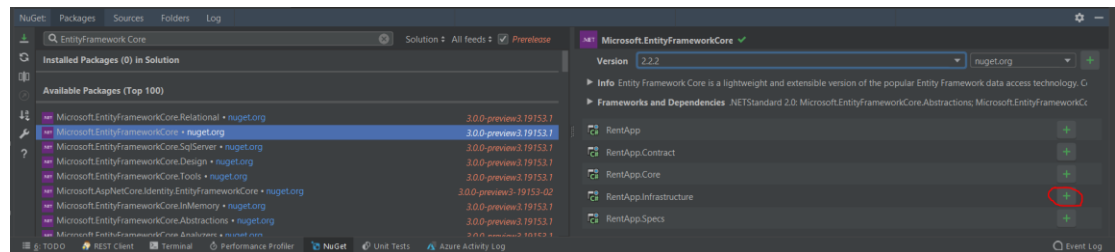
Entity Framework jest narzędziem zapewniającym mapowanie obiektowo-relacyjne. Dzięki niemu jesteśmy w stanie łatwy sposób zaimplementować klasy które będą przełożeniem na encje w bazie danych. Również zapewnia on podstawową logikę wykonywania zapytań. Plusem stosowania rozwiązania typu ORM jest elastyczne wybieranie bazy danych – w trakcie trwania developmentu jesteśmy w stanie zmienić dostawcę, nie martwiąc się o daną specyfikację danej bazy.

### a. Dodanie EntityFramework do projektu za pomocą Nuget Packages

Aby rozpocząć pracę z EntityFramework najpierw musimy dodać go do projektu. W tym celu użyjemy tzw. Paczek Nugetowych. Najpierw w Riderze w dolnym menu wybieramy Nuget.



W wyszukiwarce wpisujemy EntityFrameworkCore. Z listy należy wybrać Microsoft.EntityFrameworkCore i zaznaczyć interesującą nas wersję. W celu dodania narzędzia należy kliknąć w zielony plusik, który jest przy liście dostępnych projektów. W naszym przypadku klikamy RentApp.Infrastructure i RentApp.



Jednak to nie wszystko. Do projektu RentApp.Infrastructure potrzebujemy jeszcze dodać paczkę Microsoft.EntityFrameworkCore.Design oraz paczkę ze sterownikami do naszej bazy danych Microsoft.EntityFrameworkCore.Sqlite

## b. Stworzenie klasy DbContext<>

Po pociągnięciu paczek nuggetowych EntityFramework przechodzimy do stworzenia klasy implementującej *DbContext*, której logiką jest zapewnienie połączenia z bazą danych oraz tworzenie tabel.

Na sam początek tworzymy folder *Context* a w nim pustą klasę *RentContext*, która będzie dziedziczyła po *DbContext*.

```
5 usages  veok *  
public class RentContext : DbContext  
{  
  
}
```

Kolejnym krokiem jest wygenerowanie konstruktora który przyjmować będzie klasę *DbContextOptions* oraz metody przeciążającej *OnConfiguring*. W tej metodzie wskazujemy optionsBuilderowi by używał Sqlite wraz z connection stringiem.

```
5 usages  veok *  
public class RentContext : DbContext  
{  
    veok  
    public RentContext(DbContextOptions options) : base(options)  
    {  
    }  
  
    veok  
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  
    {  
        optionsBuilder.UseSqlite("DataSource=dbo.RentFlatApi.db");  
    }  
}
```

Ostatnim krokiem konfiguracyjnym *DbContextu* jest zarejestrowanie serwisu. W tym celu przechodzimy do klasy *Startup* w projekcie *RentApp* i tam w metodzie *ConfigureServices* dodajemy nasz *DbContext* wraz z przekazaniem takiego samego connectionstringa jak w *RentContextcie* i wskazaniem projektu w którym będą budowane migracje.

```
veok *  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);  
    services.AddDbContext<RentContext>(optionsAction: options =>  
        options.UseSqlite("DataSource=dbo.RentFlatApi.db",  
            sqliteOptionsAction: builder => builder.MigrationsAssembly("RentApp.Infrastructure")  
        ));  
}
```

### c. Tworzenie DbSet<>

Aby EntityFramework wygenerował encje bazodanowe na podstawie klas potrzeba jest wskazania tychże. Do tego wykorzystujemy *DbSet<>* który stworzy reprezentacje klasa-encja. Wracamy z powrotem do *RentContext* w której wskazujemy które tabele mają zostać wygenerowane poprzez stworzenie odpowiednich właściwości. Poza wskazaniem relacji, ta czynność jest wszystkim czego potrzebujemy.

```
public DbSet<Flat> Flat { get; set; }  
public DbSet<User> User { get; set; }  
public DbSet<Image> Image { get; set; }  
public DbSet<Owner> Owner { get; set; }  
public DbSet<Tenant> Tenant { get; set; }  
public DbSet<Address> Address { get; set; }
```

### d. Związki pomiędzy encjami

Przed ostatnim krokiem do stworzenia naszej bazy danych jest wskazanie relacji pomiędzy encjami. W klasie *RentContext* tworzymy metodą przeciążającą *OnModelCreating*. W jej ciele definiujemy relacje.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)  
{  
    modelBuilder.Entity<Flat>()  
    {  
        .HasMany( navigationExpression: x => x.Images )  
        .WithOne( navigationExpression: y => y.Flat )  
        .OnDelete(DeleteBehavior.Cascade);  
    }  
    modelBuilder.Entity<Flat>()  
    {  
        .HasOne( navigationExpression: x => x.Owner )  
        .WithMany( navigationExpression: y => y.Flats )  
        .OnDelete(DeleteBehavior.Cascade);  
    }  
    modelBuilder.Entity<Flat>()  
    {  
        .HasOne( navigationExpression: x => x.Tenant )  
        .WithOne( navigationExpression: y => y.Flat )  
        .HasForeignKey<Tenant>(z => z.Id);  
    }  
}
```

Po tej implementacji cała klasa *RentContext* prezentuje się następująco:

```
public class RentContext : DbContext
{
    5 usages
    public DbSet<Flat> Flat { get; set; }
    public DbSet<User> User { get; set; }
    public DbSet<Image> Image { get; set; }
    public DbSet<Owner> Owner { get; set; }
    public DbSet<Tenant> Tenant { get; set; }
    public DbSet<Address> Address { get; set; }

    veok
    public RentContext(DbContextOptions options) : base(options)
    {
    }

    veok
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("DataSource=dbo.RentFlatApi.db");
    }

    veok
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Flat>()
            .HasMany(navigationExpression: x => x.Images)
            .WithOne(navigationExpression: y => y.Flat)
            .OnDelete(DeleteBehavior.Cascade);

        modelBuilder.Entity<Flat>()
            .HasOne(navigationExpression: x => x.Owner)
            .WithMany(navigationExpression: y => y.Flats)
            .OnDelete(DeleteBehavior.Cascade);

        modelBuilder.Entity<Flat>()
            .HasOne(navigationExpression: x => x.Tenant)
            .WithOne(navigationExpression: y => y.Flat)
            .HasForeignKey<Tenant>(z => z.Id);
    }
}
```

#### e. Stworzenie pierwszej migracji i bazy danych

Ostatnim krokiem jest wygenerowanie migracji i stworzenie bazy danych. W tym celu otwieramy terminal/konsolę i przechodzimy do projektu w którym jest nasza klasa z DbContext. Następnie aby wygenerować migrację wpisujemy:

```
dotnet ef migrations add init -s ../RentFlat/ --context RentContext
```

Komenda wygeneruje folder z plikami migracyjnymi. *RentFlat* jest wskazaniem do głównego pliku z projektem. Jak możemy zauważyć jest to kod C#. Na poniższym screenie można zobaczyć kod generujący tabele Address oraz Owner.

```

public partial class newEntities : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Address",
            columns: table => new
            {
                Id = table.Column<long>(nullable: false)
                    .Annotation(name: "Sqlite:Autoincrement", value: true),
                DateOfCreation = table.Column<DateTime>(nullable: false),
                DateOfUpdate = table.Column<DateTime>(nullable: false),
                Street = table.Column<string>(nullable: true),
                City = table.Column<string>(nullable: true),
                ZipCode = table.Column<string>(nullable: true),
                Country = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey(name: "PK_Address", columns: x => x.Id);
            });

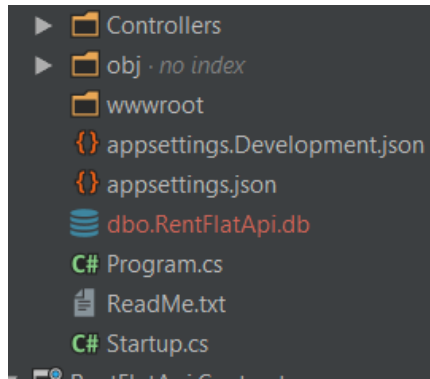
        migrationBuilder.CreateTable(
            name: "Owner",
            columns: table => new
            {
                Id = table.Column<long>(nullable: false)
                    .Annotation(name: "Sqlite:Autoincrement", value: true),
                DateOfCreation = table.Column<DateTime>(nullable: false),
                DateOfUpdate = table.Column<DateTime>(nullable: false),
                FirstName = table.Column<string>(nullable: true),
                LastName = table.Column<string>(nullable: true),
                Email = table.Column<string>(nullable: true),
                PhoneNumber = table.Column<string>(nullable: true),
                BankAccountNumber = table.Column<string>(nullable: true),
                Pesel = table.Column<string>(nullable: true),
                AddressId = table.Column<long>(nullable: true)
            },

```

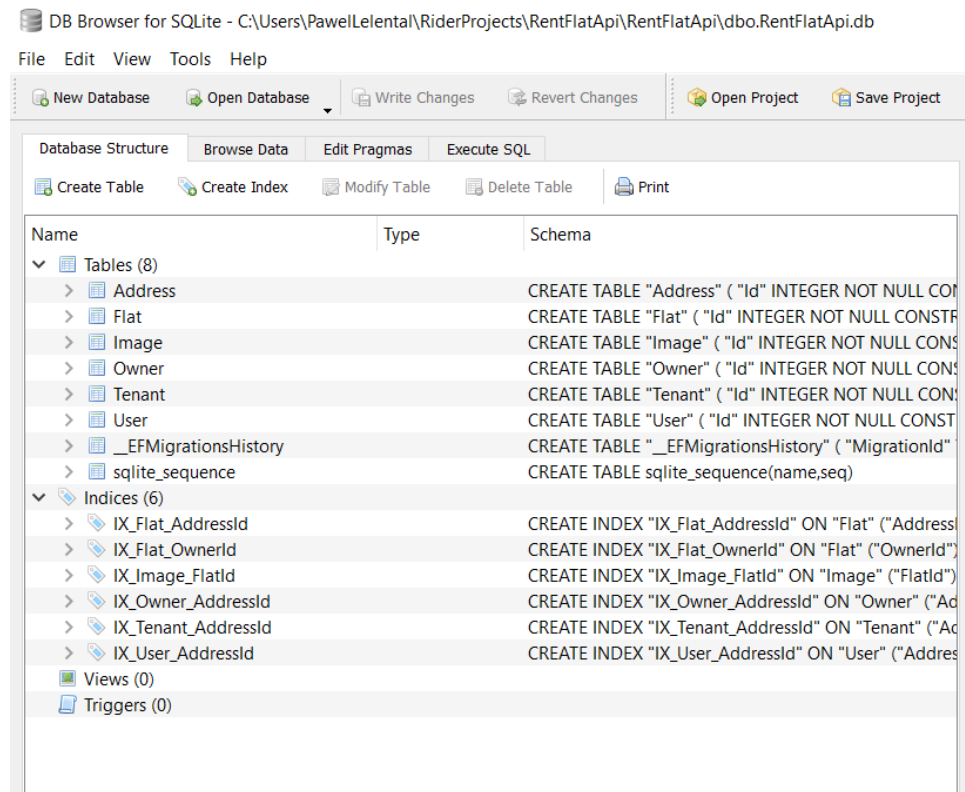
Następną komendą jest linijka która zaktualizuje naszą bazę (lub jak w tym przypadku, ją utworzy).

```
dotnet ef database update -s ../RentFlat/ --context RentContext
```

Po wpisaniu powyższej linijki w projekcie głównym RentApp powinien nam się pojawić plik z bazą danych.



Teraz za pomocą programu DB Browser możemy zaobserwować że baza została wygenerowana z zadeklarowanymi przez nas tabeli w DbSetcie.



## 9. Tworzymy pierwsze repozytorium

Po konfiguracji DataContext przejdziemy teraz do stworzenia logiki odpowiedzialnej za operacje nad bazą danych. Zaczniemy od stworzenia prostego generycznego interfejsu z podstawowymi funkcjonalnościami jakie powinny mieć nasze klasy repozytorium: pobierz wszystko, pobierz za pomocą identyfikatora, dodaj, usuń i edytuj encję. Ponieważ nasze Web API ma być asynchroniczne (wywoływania i przetwarzanie będzie realizowane na wielu wątkach, przez co nasza aplikacja będzie o wiele szybsza) każda z metod będzie używała metody *Task*.

```
1 usage 2 inheritors veok
public interface IRepository<TEntity>
{
    1 usage 1 implementation veok
    Task<IEnumerable<TEntity>> GetAll();
    1 implementation veok
    Task<TEntity> GetById(long id);
    1 usage 1 implementation veok
    Task Add(TEntity flat);
    1 implementation veok
    Task Update(TEntity entity);
    1 implementation veok
    Task Delete(long id);
}
```

Przejdźmy teraz do stworzenia klasy repozytorium *Flat*. Najpierw tworzymy interfejs *IFlatRepository* który jest rozszerzony przez *IRepository*. Stworzenie interfejsu dla *Flat* w tym przypadku ważne ponieważ jest wykorzystywane do wstrzykiwania zależności. Na razie przyjmijmy że *IFlatRepository* ma posiadać tylko logikę która została opisana w *IRepository*. Jeśli będziemy chcieli rozszerzyć funkcjonalność repozytorium, to należy w takim przypadku dodać metodę do interfejsu.

```
4 usages 1 inheritor veok
public interface IFlatRepository : IRepository<Flat>
{
}
```

Teraz stwórzmy klasę *FlatRepository* implementującą *IFlatRepository*. Ponieważ używamy tutaj opisanych wyżej interfejsów, musimy do klasy napisać implementację metod. Aby szybko wygenerować puste funkcje, w Riderze bądź w ReSharperze wystarczy kliknąć kombinację klawiszy ctrl + i. Zaznaczamy w okienku interesujące nas metody i gotowe! Puste metody zostały wygenerowane.



Generate



### Override members

Select members of base types to implement or override

- ☐ Equals(object obj):bool
- ☐ GetHashCode():int
- ☐ ToString():string
- ☒ GetAll():Task<IEnumerable<Flat>>
- ☒ GetById(long id):Task<Flat>
- ☒ Add(Flat flat):Task
- ☒ Update(Flat entity):Task
- ☒ Delete(long id):Task



No description available

☐ Make task-returning methods 'async'

Implement as: Public member

OK

Cancel



```

1 usage  2 veok *
public class FlatRepository : IFlatRepository
{
    0+1 usages  2 veok *
    public Task<IEnumerable<Flat>> GetAll()
    {
        throw new NotImplementedException();
    }

    2 veok *
    public Task<Flat> GetById(long id)
    {
        throw new NotImplementedException();
    }

    0+1 usages  2 veok *
    public Task Add(Flat flat)
    {
        throw new NotImplementedException();
    }

    2 veok *
    public Task Update(Flat entity)
    {
        throw new NotImplementedException();
    }

    2 new *
    public Task Delete(long id)
    {
        throw new NotImplementedException();
    }
}

```

Zanim przejdziemy do implementacji metod CRUD musimy napisać odwołanie do naszego wcześniej stworzonego DbContext, z którego to będziemy korzystać dla operacji bazodanowych. W tym celu definiujemy odpowiednią właściwość i przypisujemy ją do konstruktora. Dzięki wstrzykiwaniu zależności będziemy mieli natychmiastowy dostęp do funkcjonalności DbContext bez inicjalizacji.

```

private readonly DbContext _dbContext;

2 veok
public FlatRepository(DbContext dbContext)
{
    _dbContext = dbContext;
}

```

Pora na napisanie implementacji naszych funkcjonalności. Na pierwszy ogień pobieranie wszystkich encji – *GetAll()*. Aby w pełni korzystać z funkcjonalności asynchroniczności przed *Task* musimy dodać wymagane słowo kluczowe *async*. Cała implementacja jest bardzo prosta. Wystarczy że odwołamy się do naszego *RentContext* i wykorzystamy metodę *ToListAsync()*. Jednak jest w tym mały haczyk. Domyślnie EntityFramework korzysta z dociągania tzw. Lazy, przez to w tym momencie nasza metoda zwróci wszystkie Flats, ale bez zagęszczonych obiektów (np. Address będzie nullem). Dlatego musimy zmienić sposób pobierania danych. Z pomocą przychodzą nam dwa rozwiązania: pobieranie Explicit oraz Eager. Pierwsze z nich polega na tym że dane są dociągane „w locie”, zaś przy Eager wszystkie zagęszczone dane zostają wyciągnięte od razu, co przekłada się znacznie na wydajność zapytania. Dlatego dla naszego *GetAll()* użyjemy Explicit. Aby to zrobić należy dla każdego wyciągniętego obiektu wskazać referencję i ją wczytać. Również przy korzystaniu z metod asynchronicznych (w naszym wypadku *ToListAsync()* zapewniony przez Entity Framework) należy przed każdą metodą dodać słowo kluczowe *await*. Dlaczego je dodajemy? Kompilator wszystkie zdefiniowane zadania będzie wykonywał asynchronicznie dopóki nie napotka tej definicji. Jest to punkt synchronizujący. Jest on bardzo ważny ponieważ nie chcielibyśmy by różne metody wykonywały zapytania na bazie danych asynchronicznie. Najprostszym wytłumaczeniem jest tutaj przykład dodawania encji. Wyobraźmy sobie że dodajemy nową encję do bazy danych np. użytkownika, który musi mieć unikalny nick. Przed dodaniem potrzebujemy sprawdzić czy dany user jest już zarejestrowany. Przyjmując że mamy dwie metody, jedną która sprawdza i drugą która dodaje, po dopisaniu *await* wiemy że druga nie wywoła się przed pierwszą. Cała implementacja metody *GetAll()* prezentuje się następująco:

```
0+1 usages 2 veok
public async Task<IEnumerable<Flat>> GetAll()
{
    var flats = await _rentContext.Flat.ToListAsync();
    flats.ForEach( action: x => { _rentContext.Entry(x).Reference( propertyExpression: y => y.Address).LoadAsync(); });
    return flats;
}
```

Kolejną metodą CRUD będzie prostsze wyciąganie jednej encji za pomocą podania id. Implementacja jest bardzo podobna. Użycie *SingleOrDefault()* pozwala na wyciągnięcie danych, a jeśli w bazie danych nie będzie encji o podanym id, to zostanie zwrócony domyślny typ dla danego obiektu – w tym przypadku null. Jest to o tyle ważne że zastosowanie tej metody chroni nas przed ewentualnym pojawieniem się *NullPointerException*.

```
2 veok *
public async Task<Flat> GetById(long id)
{
    var flat = await _rentContext.Flat
        .Where(x => x.Id == id)
        .SingleOrDefaultAsync();
    await _rentContext.Entry(flat).Reference( propertyExpression: x => x.Address).LoadAsync();
    return flat;
}
```

Przejdźmy do Create. W metodzie ustawiamy flagę stworzenia danej encji – *DateTime.Now* - który jak można się domyślić zwróci nam obecną datę. Następnie przy pomocy *RentContext* wywołamy metody *Include()* które to wskazują aplikacji że dla każdego nowo tworzonego obiektu, powinny zastosować dodane do bazy takie rekordy Address, Owner, Tenant, Images. *AddAsync()* oraz *SaveChangesAsync()* dodają i zapisują nasze zmiany.

```
0+1 usages  veok
public async Task Add(Flat flat)
{
    flat.DateOfCreation = DateTime.Now;
    await _rentContext.Flat
        .Include( navigationPropertyPath: x => x.Address)
        .Include( navigationPropertyPath: x => x.Owner)
        .Include( navigationPropertyPath: x => x.Tenant)
        .Include( navigationPropertyPath: x => x.Images)
        .FirstAsync();
    await _rentContext.Flat.AddAsync(flat);
    await _rentContext.SaveChangesAsync();
}
```

Przedostatnią metodą do zaimplementowania jest Delete. Na samym początku powinniśmy sprawdzić czy żądany obiekt do usunięcia znajduje się w DB. Jeśli tak to za pomocą metody *Remove()* usuwamy go i dzięki *SaveChangesAsync()* zapisujemy zmiany.

```
veok
public async Task Delete(long id)
{
    var flatToDelete = await _rentContext.Flat.SingleOrDefaultAsync( predicate: flat => flat.Id == id);
    if (flatToDelete != null)
    {
        _rentContext.Flat.Remove(flatToDelete);
        await _rentContext.SaveChangesAsync();
    }
}
```

Ostatnią metodą jest aktualizacja danych. Jest ona bardzo podobna do pozostałych. Najpierw sprawdzamy czy dany obiekt istnieje w bazie. Zastosowanie *Include()* powoduje użycie dociągania zachłannego czyli Eager. Pozwoli nam to na edycje wszystkich danych. Jeśli dany obiekt istnieje, to robimy proste mapowanie wyciągniętej encji od tej którą przekazujemy w parametrze funkcji.

Dla dzieci encji, podczas edycji należy zaznaczyć by pola dziecka zostały zaktualizowane. W przeciwnym razie może zostać dodana nowa encja do bazy danych. Całość przykładowej metody Update wygląda tak:

```
public async Task Update(Flat entity)
{
    var flatToUpdate = await _rentContext.Flat
        .Include( navigationPropertyPath: x => x.Address)
        .Include( navigationPropertyPath: x => x.Owner)
        .Include( navigationPropertyPath: x => x.Tenant)
        .SingleOrDefaultAsync( predicate: x => x.Id == entity.Id);

    if (flatToUpdate != null)
    {
        flatToUpdate.Owner = entity.Owner;
        flatToUpdate.Images = entity.Images;
        flatToUpdate.Tenant = entity.Tenant;
        flatToUpdate.Floor = entity.Floor;
        flatToUpdate.Price = entity.Price;
        flatToUpdate.District = flatToUpdate.District;
        flatToUpdate.IsElevator = flatToUpdate.IsElevator;
        flatToUpdate.SquareMeters = flatToUpdate.SquareMeters;
        flatToUpdate.NumberOfRooms = flatToUpdate.SquareMeters;
        flatToUpdate.DateOfUpdate = DateTime.Now;

        if (entity.Address != null && flatToUpdate.Address != null)
        {
            entity.Address.Id = flatToUpdate.Address.Id;
            _rentContext.Entry(flatToUpdate.Address).CurrentValues.SetValues(entity.Address);
        }

        if (entity.Owner != null && flatToUpdate.Owner != null)
        {
            entity.Owner.Id = flatToUpdate.Owner.Id;
            _rentContext.Entry(flatToUpdate.Owner).CurrentValues.SetValues(entity.Owner);
        }

        if (entity.Tenant != null && flatToUpdate.Tenant != null)
        {
            entity.Tenant.Id = flatToUpdate.Tenant.Id;
            _rentContext.Entry(flatToUpdate.Tenant).CurrentValues.SetValues(entity.Tenant);
        }

        if (flatToUpdate.Images != null && entity.Images != null)
        {
            var imagesToUpdate = flatToUpdate.Images.ToList();
            foreach (var image in imagesToUpdate)
            {
                foreach (var entityImage in entity.Images)
                {
                    if (image.Id == entityImage.Id)
                    {
                        _rentContext.Entry(imagesToUpdate).CurrentValues.SetValues(entity.Images);
                    }
                }
            }
        }

        await _rentContext.SaveChangesAsync();
    }
}
```

**Zadania:**

- a. Dopisz brakujące dociąganie Explicit dla pozostałych obiektów i kolekcji w GetAll(), GetById().
- b. Napisz brakujące repozytoria dla pozostałych encji (User, Owner, Tenant, Address, Images)

## 10. Serwisy i DataContract, czyli oddzielenie logiki biznesowej od logiki aplikacji

Jednym z ważniejszych etapów tworzenia aplikacji sieciowej jest odseparowanie warstwy logiki aplikacji od warstwy biznesowej. Czynność ta ma za zadanie uchronić nas przed niechcianym udostępnieniem informacji. To co otrzymuje warstwa prezentacji, powinno być tylko niezbędnymi danymi. Również logika biznesowa powinna odpowiadać tylko za odpowiednie operacje, zaś wszystkie inne funkcjonalności (mapowanie, liczenie, hashowanie itd.) powinny być odseparowane. Dlatego też tworzymy serwisy, które wykonują logikę przed przekazaniem jej dalej, chociażby do Repository. W myśl Single Responsibility Principle (zasada pojedynczej odpowiedzialności) każda stworzona klasa, czy to Repository czy service, powinna wykonywać logikę do jakiej jest przeznaczona i nie odpowiadać za inne obszary w naszym projekcie. Single Responsibility Principle jest jedną z głównych zasad programowania obiektowego wchodzą w skład mnemonika **SOLID**:

**S** – Single Responsibility Principle

**O** – Open Closed Principle – kod powinien być zamknięty na modyfikację ale otwarty na rozszerzenia

**L** – Liskov Substitution Principle – klasa dziedzicząca powinna tylko rozszerzać logikę klasy bazowej bez dokładnej znajomości tych obiektów.

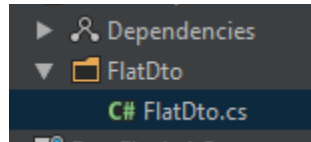
**I** – Interface Segregation Principle – posiadanie wielu interfejsów jest lepsze niż posiadanie jednego ogólnego

**D** – Dependency Inversion Principle – wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych

By lepiej zrozumieć powyższe zasady polecam przeczytać artykuły z linków poniżej:

- <https://blog.helion.pl/mnemonik-solid-s-single-responsibility-principle/>
- <https://blog.helion.pl/mnemonik-solid-o-openclosed-principle/>
- <https://blog.helion.pl/mnemonik-solid-l-liskov-substitution-principle/>
- <https://blog.helion.pl/mnemonik-solid-interface-segregation-principle/>
- <https://blog.helion.pl/mnemonik-solid-d-dependency-inversion-principle/>
- <https://www.samouczekprogramisty.pl/solid-czyli-dobre-praktyki-w-programowaniu-obiektowym/>

Wróćmy jednak to dalszej implementacji naszej aplikacji. Dlatego że nie chcemy by warstwa prezentacji aplikacji wiedziała o encjach biznesowych, tworzymy modele **DTO** (Data Transfer Object). Są to obiekty których zadaniem jest dzielenie się danymi przez różne warstwy aplikacji. Wszystkie obiekty **DTO** tworzymy w projekcie *RentApp.Contract*. Znając specyfikacje wymagań aplikacji, w tych modelach powinny znajdować się te dane które chcemy by zostały przekazane dalej. Przy naszym serwisie nieruchomości chcielibyśmy dostać prosty obiekt *FlatDto*, posiadający wszystkie informacje o danym lokum. Klasę *FlatDto* stwórzmy w Directory o takiej samej nazwie.



```
9 usages  veok
public class FlatDto
{
    public decimal Price { get; set; }
    2 usages
    public string City { get; set; }
    public string District { get; set; }
    public string Street { get; set; }
    public string ZipCode { get; set; }
    public int NumberOfRooms { get; set; }
    public int SquareMeters { get; set; }
    public int Floor { get; set; }
    public bool IsElevator { get; set; }
}
```

Teraz po stworzeniu obiektu **DTO** przejdźmy dalej. Zaczniemy od stworzenia Directory „Services” w projekcie *RentApp.Core*, który będzie miejscem naszej logiki aplikacji. Będzie ona pośredniczyć między warstwą prezentacji a warstwą biznesową. W *Services* stwórzmy generyczny interfejs. Na sam początek będzie on miał podobne funkcje jak interfejs z projektu z logiką biznesową.

```
namespace RentFlatApi.Core.Services
{
    [1 usage] [2 inheritors] [veok]
    public interface IService<TEntity>
    {
        [1 usage] [1 implementation] [veok]
        Task<IEnumerable<TEntity>> GetAll();
        [1 implementation] [veok]
        Task<TEntity> GetById(long id);
        [1 usage] [1 implementation] [veok]
        Task Add(TEntity flat);
        [1 implementation] [veok]
        Task Update(TEntity entity);
        [1 implementation] [veok]
        Task Delete(long id);
    }
}
```

Następnie tworzymy klasę *FlatService*, która powinna implementować interfejs *IFlatService*. Interfejs z kolei dziedziczy po naszym świeżo stworzonym generycznym *IService* i używa on właśnie *FlatDto* jako otrzymywany obiekt z warstwy prezentacji. Całość jest podobna do postępowania z *RentApp.Infrastructure* i może nam się wydać że duplikujemy kod. Jednak stworzenie serwisu pośredniczącego jest bezpieczniejsze niż wywoływanie klas *Repository* z poziomu *Controllera* (który będziemy tworzyli później). *FlatService* do komunikacji z *FlatRepository* będzie używał *IFlatRepository* której to instancje będzie tworzył w konstruktorze dzięki Dependency Injection (wstrzykiwaniu zależności).



```

public interface IFlatService : IService<FlatDto>
{
}

1 usage 2 week *
public class FlatService : IFlatService
{
    private readonly IFlatRepository _iFlatRepository;

    2 week *
    public FlatService(IFlatRepository iFlatRepository)
    {
        _iFlatRepository = iFlatRepository;
    }

    0+1 usages 2 week

```

Jednak by wstrzykiwanie zależności prawidłowo zadziałało, musimy zarejestrować odpowiednią zależność łączenia między interfejsem *IFlatRepository* a *FlatRepository* w ustawieniach aplikacji. W tym celu dodajemy specjalny Scope w *Startup.cs*, znajdujący się w projekcie *RentApp*. To samo też robimy dla *IFlatService* i *FlatService*.

```

public IConfiguration Configuration { get; }

// This method gets called by the runtime. Use this method to add services to the container.
2 week
public void ConfigureServices(IServiceCollection services)
{
    services.AddAutoMapper();
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    services.AddScoped<IFlatRepository, FlatRepository>();
    services.AddScoped<IFlatService, FlatService>();
    services.AddDbContext<RentContext>(optionsAction: options =>
    {
        options.UseSqlite(connectionStrings: "DataSource=dbo.RentFlatApi.db",
            sqliteOptionsAction: builder => builder.MigrationsAssembly("RentFlatApi.Infrastructure"));
    });
}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
2 week
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{

```

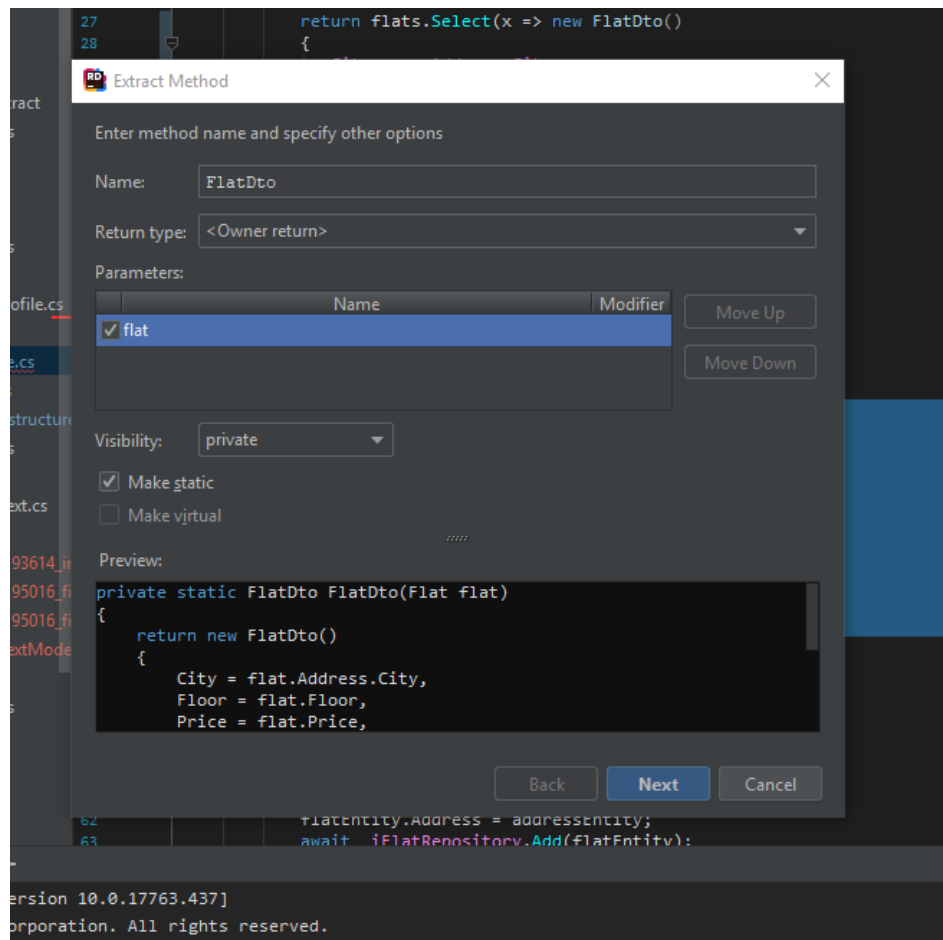
Kolejnym etapem jest teraz napisanie implementacji logiki zwracającej dane funkcje z metod interfejsu. Zaczniemy od metody *GetAll()*. Pierwsze co musimy zrobić to wyciągnąć dane wszystkich naszych obiektów. W tym celu wykorzystamy metodę *GetAll()*; z *Repository*. Jednak dane które otrzymamy są encjami *Flat* a nie *FlatDto*. Dlatego też musimy w tym miejscu zrobić przemapowanie. Na rynku i w Internecie jest wiele frameworków które pomagają ułatwić pracę z mapowaniem obiektów. Jednym z nich jest **AutoMapper**. Nie będziemy go wykorzystywać w czasie ćwiczeń, ale wspominam nim jako ciekawostkę. Przemapowanie obiektów możemy łatwo rozwiązać również dzięki LINQ. Dla wyciągniętej listy obiektów *Flat*, używamy metody *Select* w której dzięki użyciu delegacji możemy łatwo przypisać dany obiekt do nowego *FlatDto* i po wszystkim zwrócić wszystko jako listę dzięki metodzie *ToList()*.

```
0+1 usages  veok *
public async Task<IEnumerable<FlatDto>> GetAll()
{
    var flats = await _iFlatRepository.GetAll();
    return flats.Select(x => new FlatDto()
    {
        City = x.Address.City,
        Floor = x.Floor,
        Price = x.Price,
        District = x.District,
        IsElevator = x.IsElevator,
        SquareMeters = x.SquareMeters,
        NumberOfRooms = x.NumberOfRooms,
        Street = x.Address.Street,
        ZipCode = x.Address.ZipCode
    }).ToList();
}
```

Przejdźmy do *GetById()*. Implementacja metody jest bardzo podobna, używamy *Repository* do pobrania obiektu *Flat*, a w return umieszczamy mapowanie do *FlatDto*.

```
veok *
public async Task<FlatDto> GetById(long id)
{
    var flat = await _iFlatRepository.GetById(id);
    return new FlatDto()
    {
        City = flat.Address.City,
        Floor = flat.Floor,
        Price = flat.Price,
        District = flat.District,
        IsElevator = flat.IsElevator,
        SquareMeters = flat.SquareMeters,
        NumberOfRooms = flat.NumberOfRooms,
        Street = flat.Address.Street,
        ZipCode = flat.Address.ZipCode
    };
}
```

Na tym etapie rzuca się w oczy niepotrzebna redundancja mapowań. Zgodnie z zasadą programowania obiektowego **DRY** (don't repeat yourself) możemy się zastanowić czy nie lepiej w tym miejscu byłoby wydzielenie mapowania pełnego obiektu do osobnej metody, bądź klasy. Aby zrobić to szybko i przyjemnie zaznaczymy interesujący nas kawałek kodu (np. `return FlatDto()`) i wciśniemy kombinację klawiszy `ctrl + alt + m`. Kombinacja ta sprawia że przy kodzie pojawi nam się małe okienko dialogowe w którym możemy zobaczyć funkcje do ekstrakcji zaznaczonego kodu do nowej metody. Wybieramy *extract method*, po czym naszym oczom powinno okazać się nowe okno z ustawieniami dla nowej metody.



Nazwijmy naszą metodę *FlatDtoMapper*, odznaczmy pole static, klikijmy next i gotowe! Return w *GetById()* został zamieniony na odwołanie do nowej metody a ona sama została wygenerowana poniżej.

```

    public async Task<FlatDto> GetById(long id)
    {
        var flat = await _iFlatRepository.GetById(id);
        return FlatDtoMapper(flat);
    }

    1 usage 2 new *
    private FlatDto FlatDtoMapper(Flat flat)
    {
        return new FlatDto()
        {
            City = flat.Address.City,
            Floor = flat.Floor,
            Price = flat.Price,
            District = flat.District,
            IsElevator = flat.IsElevator,
            SquareMeters = flat.SquareMeters,
            NumberOfRooms = flat.NumberOfRooms,
            Street = flat.Address.Street,
            ZipCode = flat.Address.ZipCode
        };
    }

```

Teraz w *GetAll()* usuńmy całą logikę przypisywania *FlatDto* i wsadźmy tam odwołanie do *FlatDtoMapper*. Dzięki delegacji nie musimy nic podawać w parametrach funkcji. Całość teraz wygląda prosto i przejrzysto.

```

    0+1 usages 2 veok *
    public async Task<IEnumerable<FlatDto>> GetAll()
    {
        var flats = await _iFlatRepository.GetAll();
        return flats
            .Select(FlatDtoMapper)
            .ToList();
    }

```

Następne metody *Add()* oraz *Update()* też są bardzo podobne. Musimy stworzyć teraz metodę mapującą w drugą stronę -> Dto do Entity.

```
2 usages 2 veok *
private static Flat FlatMapper(FlatDto flat)
{
    return new Flat()
    {
        Floor = flat.Floor,
        Price = flat.Price,
        District = flat.District,
        IsElevator = flat.IsElevator,
        SquareMeters = flat.SquareMeters,
        NumberOfRooms = flat.NumberOfRooms,
        Id = flat.Id,
        Address = new Address()
        {
            City = flat.City,
            Street = flat.Street,
            ZipCode = flat.ZipCode,
        }
    };
}
```

Następnie implementujemy ją w *Add()* oraz *Update()*, zaś te metody wywołują już nasze gotowe funkcje z *Repository*.

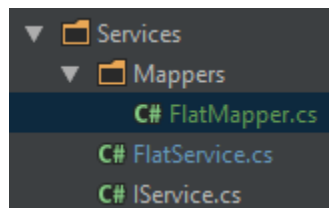
```
0+1 usages 2 veok *
public async Task Add(FlatDto flat)
{
    await _iFlatRepository.Add(FlatMapper(flat));
}

new *
public async Task Update(FlatDto entity)
{
    await _iFlatRepository.Update(FlatMapper(entity));
}
```

Ostatnią metodą CRUD jest *Delete()*. Która poprzez przekazanie *Id* wykorzysta metodę *Delete()* z *Repository*.

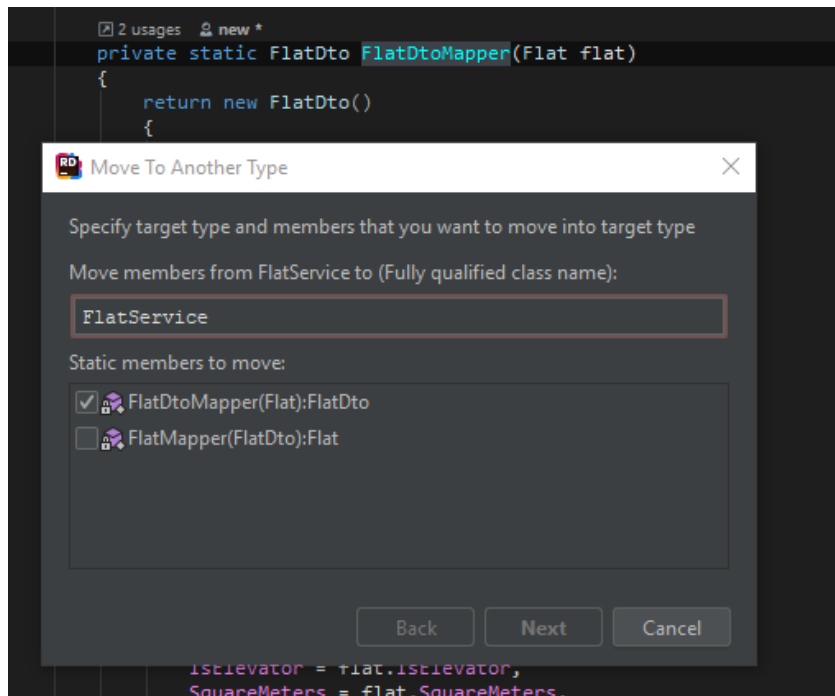
```
veok *  
public async Task Delete(long id)  
{  
    await _iFlatRepository.Delete(id);  
}
```

Posiadając teraz dwie metody mapujące możemy je wydzielić do osobnej klasy. Stworzymy zatem *Directory Mappers* w *Services*, a tam pustą statyczną i wewnętrzną klasę *FlatMapper*.

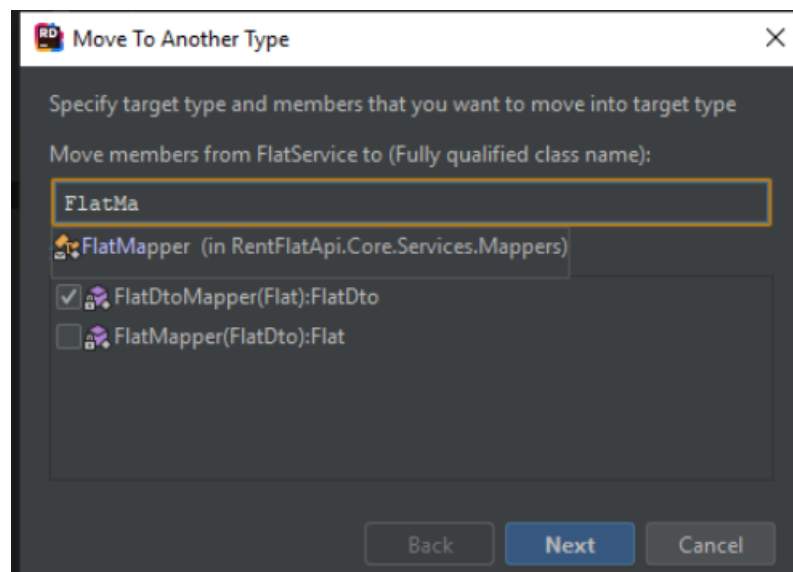


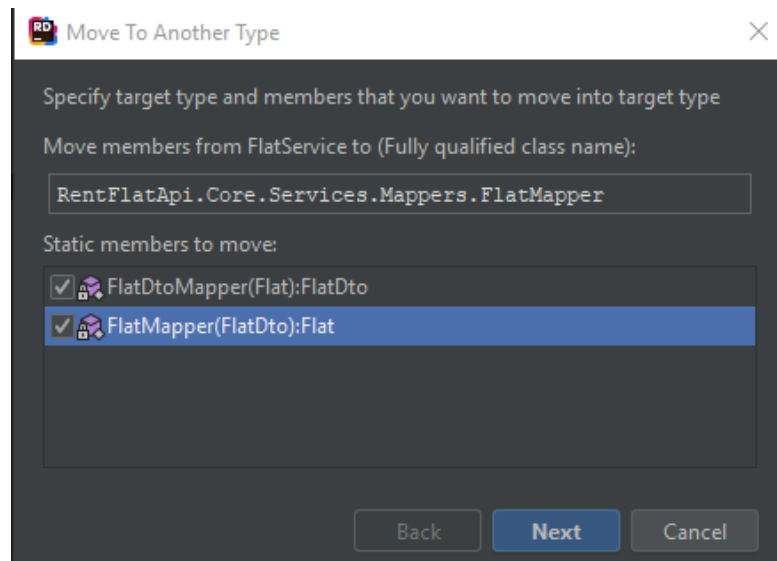
```
internal static class FlatMapper  
{  
  
}
```

Dzięki Riderowi bądź Resharperowi możemy łatwo przenieść obie metody do nowo utworzonej klasy, oszczędzając przy tym czas na zbędne kopiowanie funkcji i edycję bieżących odwołań. Wystarczy że ustawimy wskaźnik na nazwie metody i wciśniemy przycisk *F6*. Po tym powinno pojawić się nam okienko *Move To Another Type*.

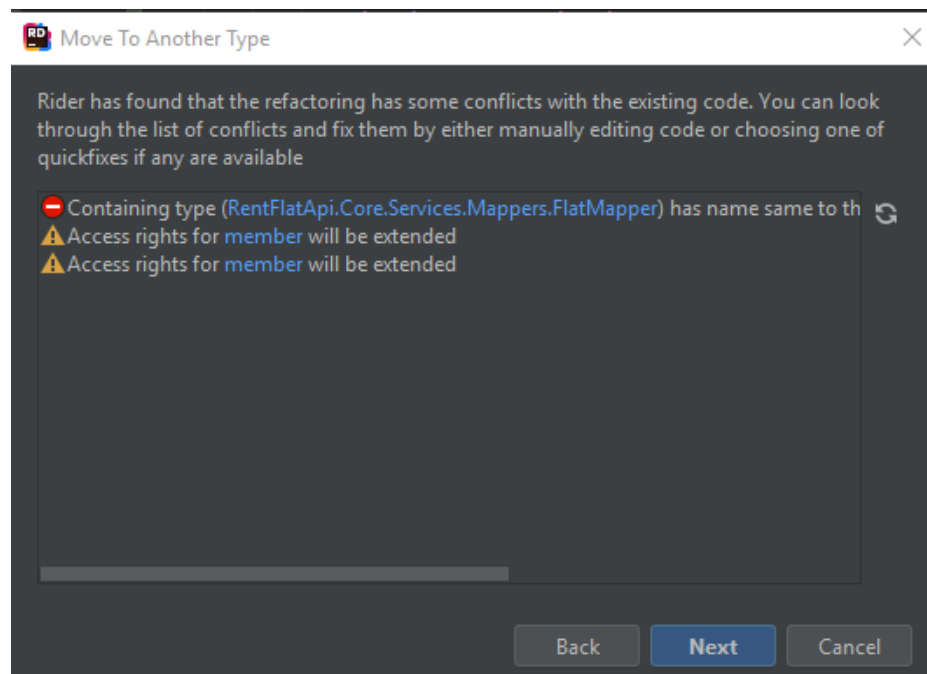


Zaznaczamy *FlatDtoMapper* oraz *FlatMapper*, a w miejscu *FlatService* wpisujemy *FlatMapper*. Podczas wpisywania nazwy klasy, Rider powinien sam nam podpowiedzieć która (uwaga, przykład jest przeze mnie pokazywany na innym projekcie. Dlatego pokazuje *RentFlatApi*, zamiast *RentApp*).





Klikamy next, po czym powinno nam się pojawić okienko informujące o trzech ostrzeżeniach. Pierwsze z nich informuje nas o tym iż posiadamy nazwę metody która nazywa się tak samo jak konstruktor klasy do której ma zostać przeniesiona. Dwa pozostałe ostrzeżenia informują o tym że prawa dostępu do metod zostaną rozszerzone.



Klikamy next i cały proces powinien zostać zakończony.



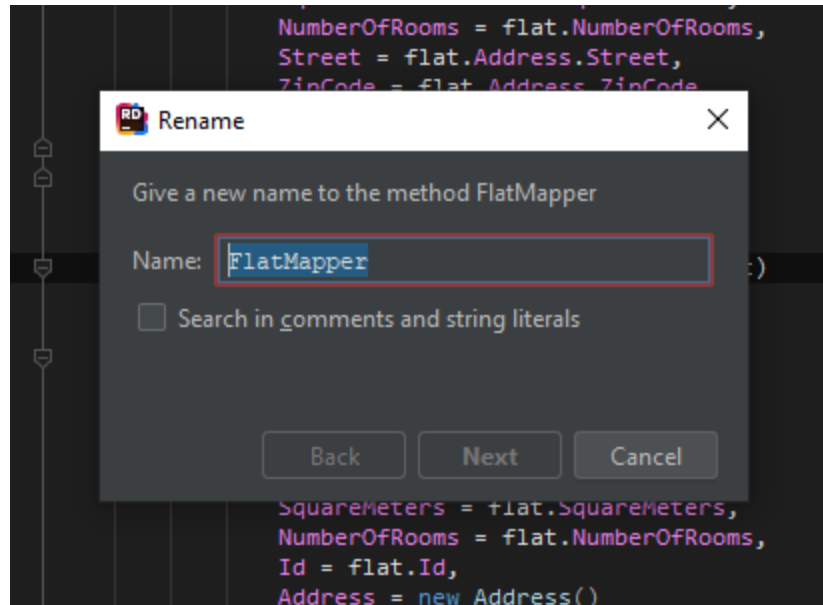
```

internal static class FlatMapper
{
    2 usages
    public static FlatDto FlatDtoMapper(Flat flat)
    {
        return new FlatDto()
        {
            City = flat.Address.City,
            Floor = flat.Floor,
            Price = flat.Price,
            District = flat.District,
            IsElevator = flat.IsElevator,
            SquareMeters = flat.SquareMeters,
            NumberOfRooms = flat.NumberOfRooms,
            Street = flat.Address.Street,
            ZipCode = flat.Address.ZipCode,
            Id = flat.Id
        };
    }

    2 usages
    public static Flat FlatMapper(FlatDto flat)
    {
        return new Flat()
        {
            Floor = flat.Floor,
            Price = flat.Price,
            District = flat.District,
            IsElevator = flat.IsElevator,
            SquareMeters = flat.SquareMeters,
            NumberOfRooms = flat.NumberOfRooms,
            Id = flat.Id,
            Address = new Address()
            {
                City = flat.City,
                Street = flat.Street,
                ZipCode = flat.ZipCode,
            }
        };
    }
}

```

Zostaje nam jeszcze zmiana nazwy *FlatMapper()*. Zmieńmy jej nazwę na *MapFlatToDto()*, a dla porządku zmieńmy też nazwę *FlatDtoMapper* na *MapDtoToFlat()*. By szybko zmienić nazwę metody, najeżdżamy na jej nazwę i wciskamy kombinację klawiszy *shift+F6*, po której pojawi się okno Rename.



Wpisujemy *MapDtoToFlat* i naciskamy next. Tą samą czynność powtarzamy dla *FlatDtoMapper()*. Po całej refaktoryzacji możemy zobaczyć że nazwy metod zostały zmienione, a w klasie *FlatService* mamy odwołania do nowo stworzonego Mapper'a.

```

public class FlatService : IFlatService
{
    private readonly IFlatRepository _iFlatRepository;

    veok *
    public FlatService(IFlatRepository iFlatRepository)
    {
        _iFlatRepository = iFlatRepository;
    }

    0+1 usages veok *
    public async Task<IEnumerable<FlatDto>> GetAll()
    {
        var flats = await _iFlatRepository.GetAll();
        return flats
            .Select(FlatMapper.MapFlatToDto)
            .ToList();
    }

    veok *
    public async Task<FlatDto> GetById(long id)
    {
        var flat = await _iFlatRepository.GetById(id);
        return FlatMapper.MapFlatToDto(flat);
    }

    0+1 usages veok *
    public async Task Add(FlatDto flat)
    {
        await _iFlatRepository.Add(FlatMapper.MapDtoToFlat(flat));
    }

    veok *
    public async Task Update(FlatDto entity)
    {
        await _iFlatRepository.Update(FlatMapper.MapDtoToFlat(entity));
    }

    veok *
    public async Task Delete(long id)
    {
        await _iFlatRepository.Delete(id);
    }
}

```

#### Zadania:

- a. Stwórz brakujące serwisy dla pozostałych obiektów oraz stwórz klasy DTO.

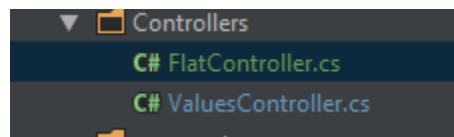
## 11. Tworzymy pierwszy Controller

Architektura Restowa wykorzystuje ściśle metody protokołu HTTP. Podczas tworzenia API wykorzystywane są cztery podstawowe metody:

- GET – pobieranie zasób danych
- POST – wysyłanie i tworzenie zasobu danych
- PUT – wysyłanie i aktualizacja zasobu danych
- DELETE – usunięcie zasobu danych

Protokół HTTP jest bezstanowy, przez co za pomocą niego są wysyłane tylko i wyłącznie informacje. Najpopularniejszym formatem to wysyłania i przekształcania danych jest JSON.

Przejdźmy do stworzenia pierwszego Controllera, który będzie posiadał tylko niezbędne metody tzw. Endpointy za pomocą których będzie można komunikować się z naszą aplikacją z zewnątrz. Na sam początek tworzymy klasę FlatController w folderze Controllers w głównym projekcie:



FlatController powinien dziedziczyć po podstawowej klasie ControllerBase, która dostarcza podstawowe funkcjonalności do komunikowania się po protokole HTTP oraz powinien zostać on oznaczony dwoma atrybutami: ApiController którego dołączenie włącza obsługę API oraz Route który określa ścieżkę do której będziemy wykorzystywać by odwołać się do stworzonych EndPointów.

```
[ApiController]
[Route("api/[controller]")]
public class FlatController : ControllerBase
{
}
}
```

Zacznijmy od zaimplementowania podstawowego zapytania GET, który po podaniu Id, zwróci nam dany rekord opakowany w FlatDto. Tworzymy metodę GetFlatById, która przyjmuje long id. Metoda ta będzie określona atrybutemHttpGet wraz odpowiednią ścieżką, dzięki której się do niej odwołamy. Ponieważ nasza aplikacja ma działać asynchronicznie, dodajemy jak w poprzednich serwisach potrzebne słowa kluczowe async, Task oraz await. Do uzyskania rekordu użyjemy wcześniej stworzonego przez nas FlatService. Ponieważ wcześniej nie zdefiniowaliśmy żadnej walidacji dodajemy ją teraz opatrując metodę FlatService w blok try catch. Funkcja GetFlatById zwracać będzie

metody odpowiadające kodom protokołu http. Przy pomyślnym pobraniu danych – 200, Ok wraz danymi oraz w przypadku niepowodzenia – 404, NotFound wraz z naszym kodem błędu. Całość implementacji wygląda następująco:

```
[ApiController]
[Route("api/[controller]")]
public class FlatController : ControllerBase
{
    private readonly IFlatService _flatService;

    public FlatController(IFlatService flatService)
    {
        _flatService = flatService;
    }

    [HttpGet("GetFlat/{Id}")]
    public async Task<IActionResult> GetFlatById(long id)
    {
        try
        {
            var flat = await _flatService.GetById(id);
            return Ok(flat);
        }
        catch (NullReferenceException e)
        {
            return NotFound(value: $"Can't found flat with id = {id}");
        }
    }
}
```

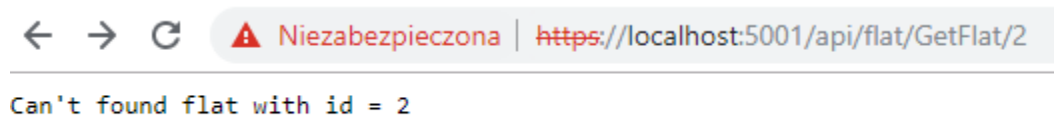
Ponieważ w FlatRepository podczas loadingu z Address, nie mamy również handlingu przypadku w którym obiekt Flat nie posiada relacji do żadnego obiektu z Adresem, to dodajmy odpowiedni block try catch. Uchroni to nas przed niespodziewanym Exception:

```
0+1 usages 2 weeks *
public async Task<Flat> GetById(long id)
{
    var flat = await _rentContext.Flat
        .Where(x => x.Id == id)
        .SingleOrDefaultAsync();

    try
    {
        await _rentContext.Entry(flat).Reference(x => x.Address).LoadAsync();
    }
    catch (ArgumentException e)
    {
        return null;
    }

    return flat;
}
```











Aby przetestować metodę, dla zapytań GET wystarczy odwołać się po URL w przeglądarce internetowej (bądź w programach typu Postman, bądź przy pomocy innych wbudowanych narzędzi) tak jak na zrzucie poniżej:



← → ↻ Niezabezpieczona | <https://localhost:5001/api/flat/GetFlat/2>

Can't found flat with id = 2

Dodajmy jeszcze jedną metodę GET, a mianowicie GetAll. Dla ułatwienia przykładu część pól FlatDto zamierzmy na nullable i zmodyfikujmy odpowiednio metodę mapującą. Cała implementacja prezentuje się następująco:

```
public class FlatDto
{
     2 usages
    public decimal? Price { get; set; }
     4 usages
    public string City { get; set; }
     2 usages
    public string District { get; set; }
     2 usages
    public string Street { get; set; }
     2 usages
    public string ZipCode { get; set; }
     2 usages
    public int? NumberOfRooms { get; set; }
     2 usages
    public int? SquareMeters { get; set; }
     2 usages
    public int? Floor { get; set; }
     2 usages
    public bool IsElevator { get; set; }
     2 usages
    public long? Id { get; set; }
}
```

```

internal static class FlatMapper
{
    [2 usages]
    public static FlatDto MapFlatToDto(Flat flat)
    {
        return new FlatDto()
        {
            City = flat.Address?.City,
            Floor = flat.Floor,
            Price = flat.Price,
            District = flat.District,
            IsElevator = flat.IsElevator,
            SquareMeters = flat.SquareMeters,
            NumberOfRooms = flat.NumberOfRooms,
            Street = flat.Address?.Street,
            ZipCode = flat.Address?.ZipCode,
            Id = flat.Id
        };
    }

    [2 usages]
    public static Flat MapDtoToFlat(FlatDto flat)
    {
        return new Flat()
        {
            Floor = flat.Floor.GetValueOrDefault(),
            Price = flat.Price.GetValueOrDefault(),
            District = flat.District,
            IsElevator = flat.IsElevator,
            SquareMeters = flat.SquareMeters.GetValueOrDefault(),
            NumberOfRooms = flat.NumberOfRooms.GetValueOrDefault(),
            Id = flat.Id.GetValueOrDefault(),
            Address = new Address()
            {
                City = flat.City,
                Street = flat.Street,
                ZipCode = flat.ZipCode,
            }
        };
    }
}

```

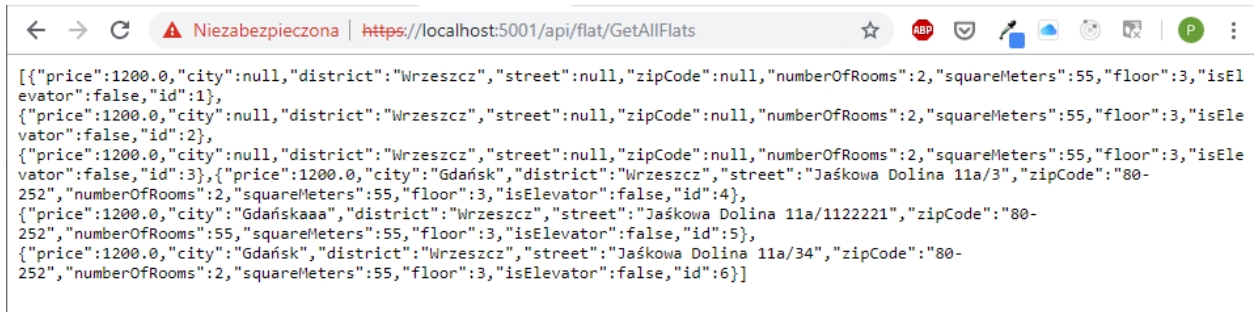
```

[HttpGet("GetAllFlats")]
public async Task<IActionResult> GetAllFlats()
{
    var flats = await _flatService.GetAll();
    return Ok(flats);
}

```



Teraz gdy odwołamy się po URL GetAllFlats i jeśli mamy rekordy w bazie danych dostaniemy zwrot wszystkich nieruchomości:



```
[{"price":1200.0,"city":null,"district":"Wrzeszcz","street":null,"zipCode":null,"numberOfRooms":2,"squareMeters":55,"floor":3,"isElevator":false,"id":1}, {"price":1200.0,"city":null,"district":"Wrzeszcz","street":null,"zipCode":null,"numberOfRooms":2,"squareMeters":55,"floor":3,"isElevator":false,"id":2}, {"price":1200.0,"city":null,"district":"Wrzeszcz","street":null,"zipCode":null,"numberOfRooms":2,"squareMeters":55,"floor":3,"isElevator":false,"id":3}, {"price":1200.0,"city":"Gdańsk","district":"Wrzeszcz","street":"Jaśkowa Dolina 11a/3","zipCode":"80-252","numberOfRooms":2,"squareMeters":55,"floor":3,"isElevator":false,"id":4}, {"price":1200.0,"city":"Gdańsk","district":"Wrzeszcz","street":"Jaśkowa Dolina 11a/1122221","zipCode":"80-252","numberOfRooms":55,"squareMeters":55,"floor":3,"isElevator":false,"id":5}, {"price":1200.0,"city":"Gdańsk","district":"Wrzeszcz","street":"Jaśkowa Dolina 11a/34","zipCode":"80-252","numberOfRooms":2,"squareMeters":55,"floor":3,"isElevator":false,"id":6}]
```

Implementacja metod POST i PUT będzie w tym do siebie bliźniaczo podobna. Metodę POST wykorzystuje się do tworzenia nowych obiektów z kolei PUT do aktualizacji. Obie funkcje przyjmują FlatDto, które określamy atrybutem [Body] – to właśnie przez body requestu http prześlemy wymagane pola do stworzenia lub edycji nieruchomości. CreateFlat zwracać będzie przy pomyślnym stworzeniu kod 201 – Created, zaś aktualizacja, kod 200- OK. Każdej z metod również przypisujemy odpowiedni atrybut. Ciekawostką jest to że przy wszystkich atrybutach, pod warunkiem że ścieżka jest unikalna, nie musimy tworzyć żadnej dodatkowej – wywołanie nastąpi poprzez odpowiednie wskazanie zapytania GET/POST/PUT/DELETE po podaniu tylko głównej ścieżki endpointa. Logika powinna wyglądać jak na zdjęciu poniżej:

```
[HttpPost]
public async Task<IActionResult> CreateFlat([FromBody] FlatDto flat)
{
    if (flat == null)
    {
        return BadRequest();
    }

    await _flatService.Add(flat);
    return Created(uri: "Created new flat", flat);
}

[HttpPut("UpdateFlat")]
public async Task<IActionResult> UpdateFlat([FromBody] FlatDto flat)
{
    if (flat == null)
    {
        return BadRequest();
    }

    await _flatService.Update(flat);
    return Ok(value: $"Updated flat with id = {flat.Id}");
}
```

Teraz by przetestować dodawanie i edytowanie napiszmy odpowiedniego JSONa. Musimy w nim podać adres, co on akceptuje, odpowiedni content-type (w tym przypadku oczywiście json) oraz body. Do przetestowania tych zapytań użyję wbudowanego narzędzia REST Client z Ridera:

```
POST https://localhost:5001/api/flat
Accept: */*
Cache-Control: no-cache
Content-Type: application/json

{
  "Price": 1200,
  "City": "Warszawa",
  "District": "Bemowo",
  "Street": "Jerozolimska 1/2",
  "ZipCode": "00-007",
  "NumberOfRooms": 2,
  "SquareMeters": 15,
  "Floor": 8
}
```

Zaś po uruchomieniu powinniśmy otrzymać komunikat o poprawnym utworzeniu:

```
Run: rest-api#1 x
POST https://localhost:5001/api/flat

HTTP/1.1 201 Created
Date: Fri, 31 May 2019 18:23:10 GMT
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked
Location: Created new flat

{
  "price": 1200.0,
  "city": "Warszawa",
  "district": "Bemowo",
  "street": "Jerozolimska 1/2",
  "zipCode": "00-007",
  "numberOfRooms": 2,
  "squareMeters": 15,
  "floor": 8,
  "isElevator": false,
  "id": null
}

Response code: 201 (Created); Time: 35ms; Content length: 176 bytes
```

Sprawdźmy jeszcze czy faktycznie nasza nieruchomość się dodała. Wykorzystajmy GetAllFlats do sprawdzenia.

```
▶ GET https://localhost:5001/api/flat/GetAllFlats
Accept: */*
Cache-Control: no-cache
Content-Type: application/json
```

Oraz po wywołaniu widzimy że faktycznie nieruchomość z Bemowa widnieje w naszej liście:

```
  },
  {
    "price": 1200.0,
    "city": "Warszawa",
    "district": "Bemowo",
    "street": "Jerozolimska 1/2",
    "zipCode": "00-007",
    "numberOfRooms": 2,
    "squareMeters": 15,
    "floor": 8,
    "isElevator": false,
    "id": 9
  }
]
Response code: 200 (OK); Time: 51ms; Content length: 1522 bytes
```

Przetestujemy jeszcze edytowanie. Uznaliśmy jednak że cena 1200.0 zł jest za mała w tym celu zmienimy ją na 3569.21 zł.

```
PUT https://localhost:5001/api/flat/UpdateFlat
Accept: */*
Cache-Control: no-cache
Content-Type: application/json

{"Price": 3569.21,
 "City": "Warszawa",
 "District": "Bemowo",
 "Street": "Jerozolimska 1/2",
 "ZipCode": "00-007",
 "NumberOfRooms": 2,
 "SquareMeters": 15,
 "Floor": 8,
 "Id": 9}
```

Oraz wynik:

```
PUT https://localhost:5001/api/flat/UpdateFlat

HTTP/1.1 200 OK
Date: Fri, 31 May 2019 18:29:36 GMT
Content-Type: text/plain; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

Updated flat with id = 9

Response code: 200 (OK); Time: 29ms; Content length: 24 bytes
```

Użyjmy teraz metody GetById by sprawdzić czy faktycznie dla tego jednego rekordu zmieniła nam się wartość Price:

```
GET https://localhost:5001/api/flat/GetFlat/9
Accept: */*
Cache-Control: no-cache
Content-Type: application/json
```

Oraz poprawny wynik:

```
Run: rest-api_1#1 x
GET https://localhost:5001/api/flat/GetFlat/9

HTTP/1.1 200 OK
Date: Fri, 31 May 2019 18:31:57 GMT
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

{
  "price": 3569.21,
  "city": "Warszawa",
  "district": "Bemowo",
  "street": "Jerozolimska 1/2",
  "zipCode": "00-007",
  "numberOfRooms": 15,
  "squareMeters": 15,
  "floor": 8,
  "isElevator": false,
  "id": 9
}

Response code: 200 (OK); Time: 48ms; Content length: 175 bytes
```

Ostatnią podstawową implementacją jest Delete. Metoda również jest bardzo krótka, przez to iż cała potrzebna logika została zawarta i opakowana w niższych warstwach aplikacji:

```
[HttpDelete("DeleteFlat/{id}")]
public async Task<IActionResult> DeleteFlat(long id)
{
    await _flatService.Delete(id);
    return Ok(value: $"Flat with id = {id} deleted");
}
```

Dla testów usuńmy rekord o numerze id = 9.

```
DELETE https://localhost:5001/api/flat/DeleteFlat/9
Accept: */*
Cache-Control: no-cache
Content-Type: application/json
```

Zgodnie z komunikatem, rekord został pomyślnie usunięty:

```
Run: rest-api_1#1 x
DELETE https://localhost:5001/api/flat/DeleteFlat/9
HTTP/1.1 200 OK
Date: Fri, 31 May 2019 18:42:39 GMT
Content-Type: text/plain; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

Flat with id = 9 deleted

Response code: 200 (OK); Time: 65ms; Content length: 24 bytes
```

Sprawdźmy więc to odwołując się do Endpointa GetFlatById:

```
1 GET https://localhost:5001/api/flat/GetFlat/9
2 Accept: */*
3 Cache-Control: no-cache
4 Content-Type: application/json
5
```

Run: rest-api\_1#1 x

```
GET https://localhost:5001/api/flat/GetFlat/9

HTTP/1.1 404 Not Found
Date: Fri, 31 May 2019 18:51:25 GMT
Content-Type: text/plain; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

Can't found flat with id = 9

Response code: 404 (Not Found); Time: 459ms; Content length: 28 bytes
```

Oraz dla pewności sprawdzimy również poprzez GetAllFlats:

```
{
  "price": 1200.0,
  "city": "Gdańsk",
  "district": "Wrzeszcz",
  "street": "Jaśkowa Dolina 11a/34",
  "zipCode": "80-252",
  "numberOfRooms": 2,
  "squareMeters": 55,
  "floor": 3,
  "isElevator": false,
  "id": 6
},
{
  "price": 1200.0,
  "city": "Warszawa",
  "district": "Mokotów",
  "street": "Koszykowa 1/2",
  "zipCode": "00-002",
  "numberOfRooms": 5,
  "squareMeters": 155,
  "floor": 6,
  "isElevator": false,
  "id": 7
},
{
  "price": 1200.0,
  "city": "Warszawa",
  "district": "Mokotów",
  "street": "Koszykowa 1/2",
  "zipCode": "00-002",
  "numberOfRooms": 5,
  "squareMeters": 155,
  "floor": 6,
  "isElevator": false,
  "id": 8
}
]
Response code: 200 (OK); Time: 168ms; Content length: 1348 bytes
```

Jak widać ostatni rekord kończy się na Id = 8, więc operacja przebiegła pomyślnie.

**Zadania:**

- a. Stwórz brakujące Controlery
- b. Dodaj logowanie do krytycznych miejsc aplikacji np. przy blokach try catch



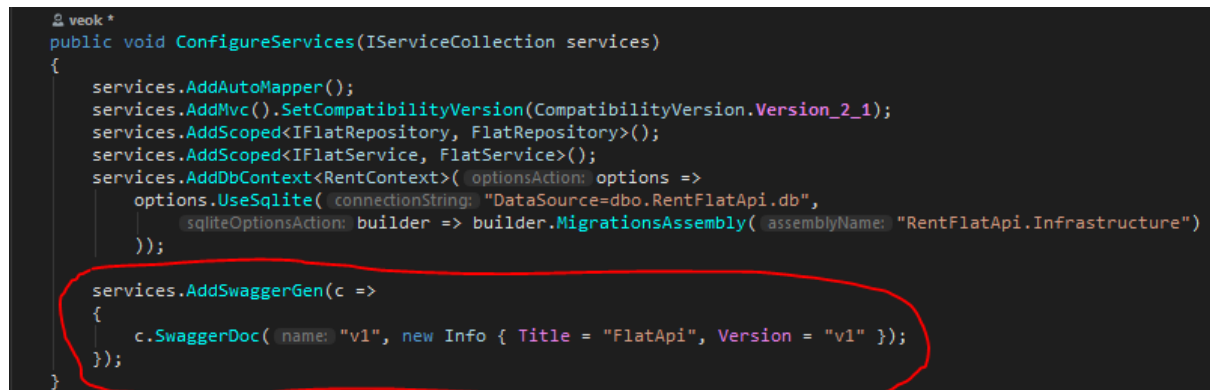
## 12. Swagger

Wraz z rozwijaniem się aplikacji i postawianiem nowych endpointów, coraz trudniej jest ogarnąć wszystkie możliwe adresy. Możemy prowadzić własną dokumentację projektu, ale również możemy skorzystać z gotowych rozwiązań. Najpopularniejszym do dnia dzisiejszego jest biblioteka Swagger (dostępna również w javie jak i innych technologiach). Narzędzie to w przejrzysty sposób dokumentuje nasze endpointy w formie strony internetowej oraz pozwala przetestować logikę wysłać odpowiednie metody.

Aby dodać bibliotekę do naszego projektu, użyjmy linii komend i w folderze z solucją wywołajmy poniższy skrypt:

```
dotnet add <nazwa głównego projektu>.csproj package Swashbuckle.AspNetCore
```

Teraz wystarczy dodać odpowiednie linijki kodu do naszej klasy Startup.cs, które odpowiednio skonfigurują użycie swaggera:



```
using Microsoft.Extensions.DependencyInjection;

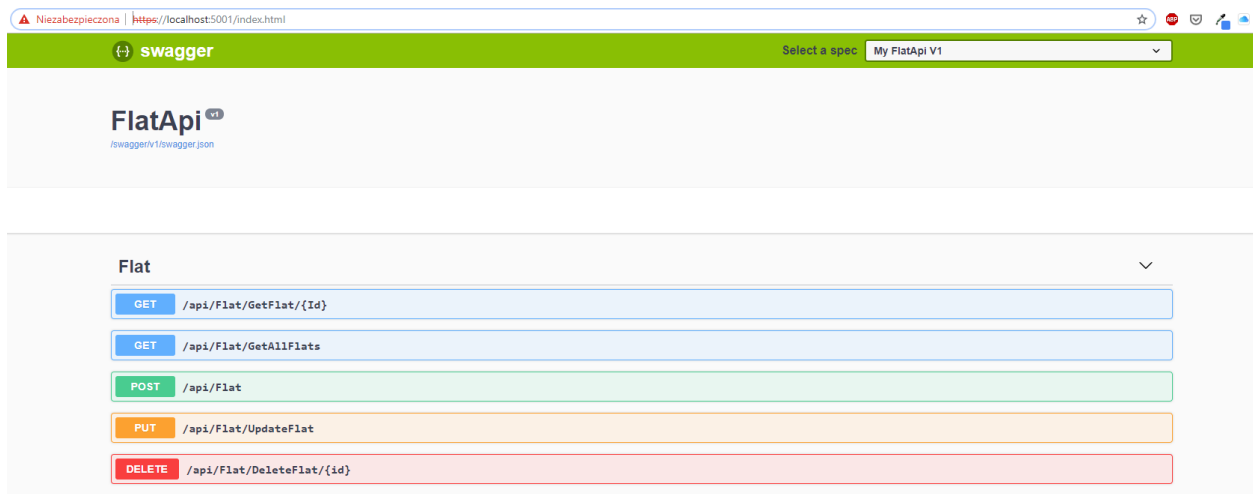
public void ConfigureServices(IServiceCollection services)
{
    services.AddAutoMapper();
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    services.AddScoped<IFlatRepository, FlatRepository>();
    services.AddScoped<IFlatService, FlatService>();
    services.AddDbContext<RentContext> ( optionsAction: options =>
        options.UseSqlite( connectionString: "DataSource=dbo.RentFlatApi.db",
            sqliteOptionsAction: builder => builder.MigrationsAssembly( assemblyName: "RentFlatApi.Infrastructure")
        ));

    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc( name: "v1", new Info { Title = "FlatApi", Version = "v1" });
    });
}
```

```
app.UseSwagger();

app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint( url: "/swagger/v1/swagger.json", name: "My FlatApi V1");
    c.RoutePrefix = string.Empty;
});
}
```

I tyle! Odpalamy teraz naszą aplikację i pod podstawowym adresem np. <https://localhost:5001/index.html> ukaże się nam dokumentacja naszego REST Api. Wszystko łatwo, prosto i przyjemnie.



Po rozwinięciu odpowiedniego przycisku odpowiadającego danemu endpointowi możemy przeprowadzić daną operację, np. GetAllFlats:

GET /api/Flat/GetAllFlats

Parameters Cancel

No parameters

Execute

Responses Response content type: application/json

Code	Description
200	Success

I po kliknięciu execute dostaniemy odpowiedź wraz ze wszystkimi informacjami:

GET /api/Flat/GetAllFlats

Parameters Cancel

No parameters

Execute Clear

Responses Response content type: application/json

Curl

```
curl -X GET "https://localhost:5001/api/Flat/GetAllFlats" -H "accept: application/json"
```

Request URL

```
https://localhost:5001/api/Flat/GetAllFlats
```

Server response

Code	Details
------	---------

Server response

Code

Details

200

Response body

```
[
  {
    "price": 1200,
    "city": null,
    "district": "Wrzeszcz",
    "street": null,
    "zipCode": null,
    "numberOfRooms": 2,
    "squareMeters": 55,
    "floor": 2,
    "isElevator": false,
    "id": 1
  },
  {
    "price": 1200,
    "city": null,
    "district": "Wrzeszcz",
    "street": null,
    "zipCode": null,
    "numberOfRooms": 2,
    "squareMeters": 55,
    "floor": 3,
    "isElevator": false,
    "id": 2
  },
  {
    "price": 1200,
    "city": null,
```

Download

Response headers

```
content-type: application/json; charset=utf-8
date: Fri, 31 May 2019 19:34:05 GMT
server: Kestrel
transfer-encoding: chunked
```

Responses

Code

Description

200

Success