

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?

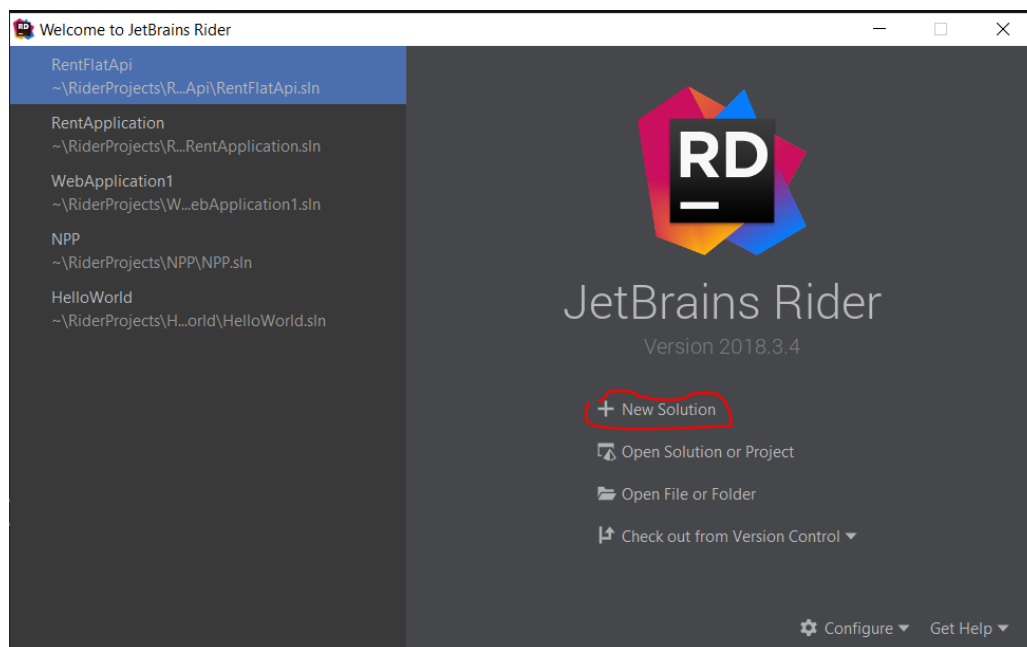
- Podstawy tworzenia Web API na podstawie ASP.NET Core 2.2
- Implementację mapowania obiektowo-relacyjnego przy pomocy EntityFramework Core 2.2
- Przykład Clean Code Architecture
- Całość dokumentu będzie oparta na pracy przy pomocy środowiska Rider od JetBrains. Osoby korzystające z Visual Studio przy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

3. Co potrzebujemy?

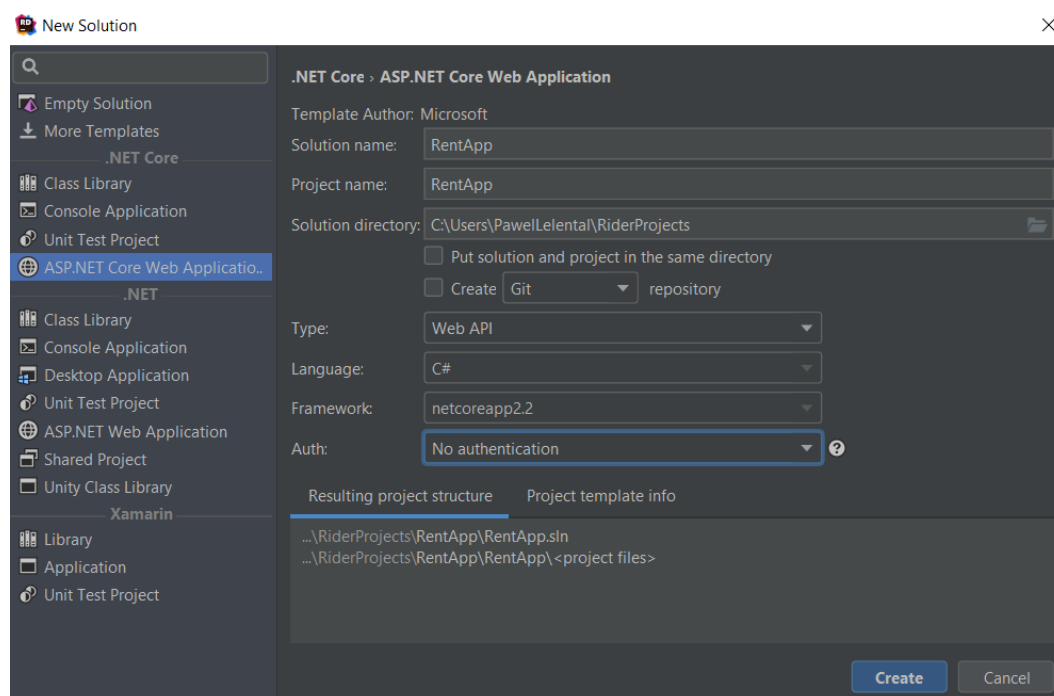
- a. .Net Core 2.2
- b. Sqlite 2.3
- c. DB Browser dla Sqlite
- d. JetBrains Rider/ Visual Studio Professional 2019

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*.

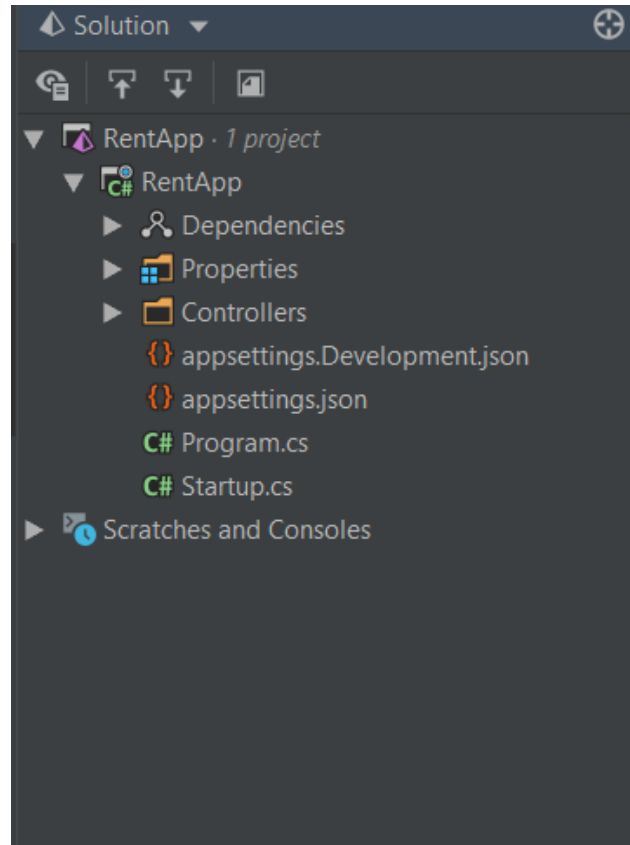


Następnie wybieramy rodzaj naszego nowo tworzonego projektu: *ASP.NET Core Web Application*. W oknie dialogowym wpisujemy główną nazwę solucji oraz nazwę projektu. W tym przypadku wpisujemy w oba pola *RentApp*. Niżej musimy jeszcze wybrać typ aplikacji, język, framework oraz czy chcemy by środowisko 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:



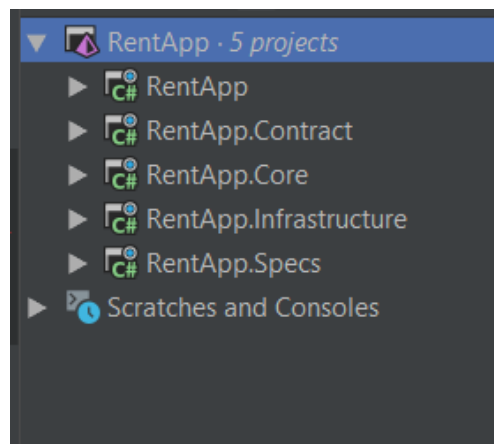
Gdzie odpowiednio każdy z folderów/plików odpowiada:

- **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

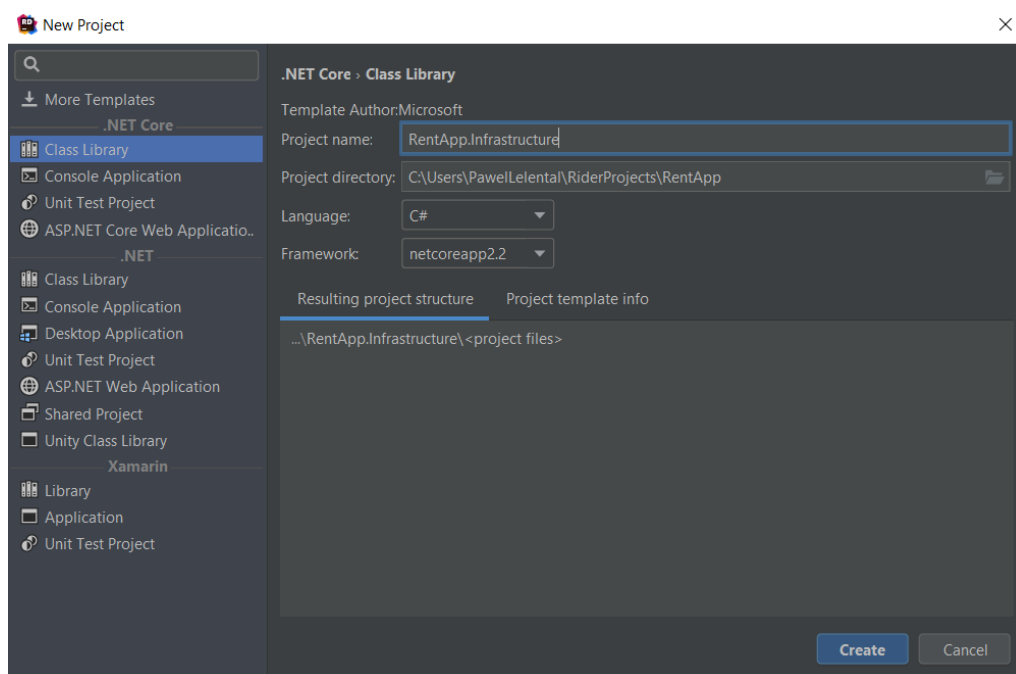
6. Projekt architektury aplikacji

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

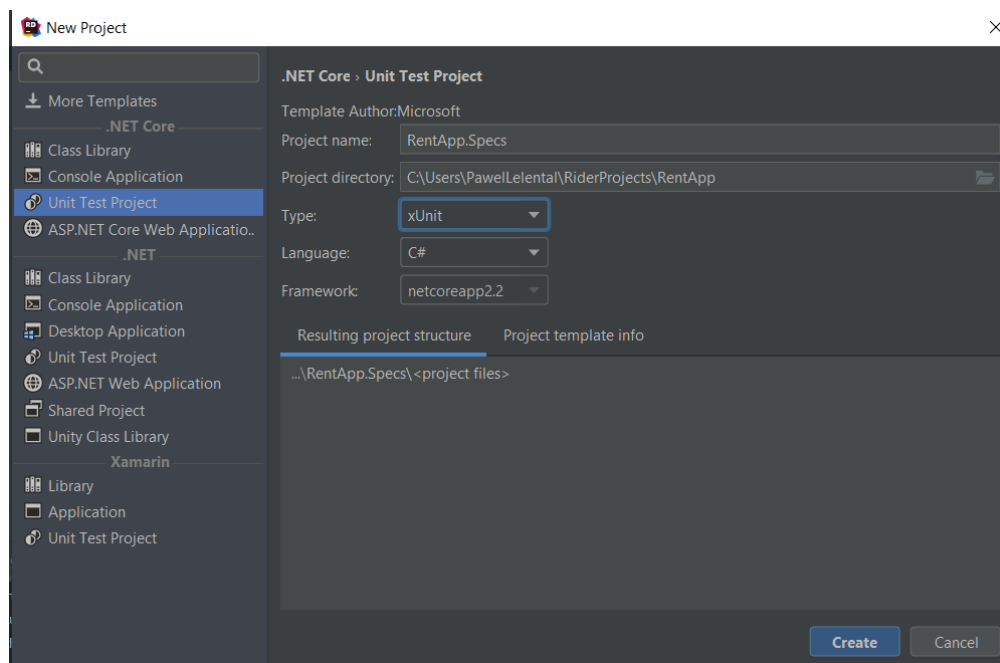
- **RentApp** – główny projekt, zawierający logikę do uruchomienia aplikacji i Controllery Restowe
- **RentApp.Infrastructure** – projekt który przechowuje całą logikę biznesową w tym komunikację z bazą danych
- **RentApp.Core** – projekt z logiką aplikacji, będącą pośrednikiem pomiędzy warstwą prezentacji, 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**.



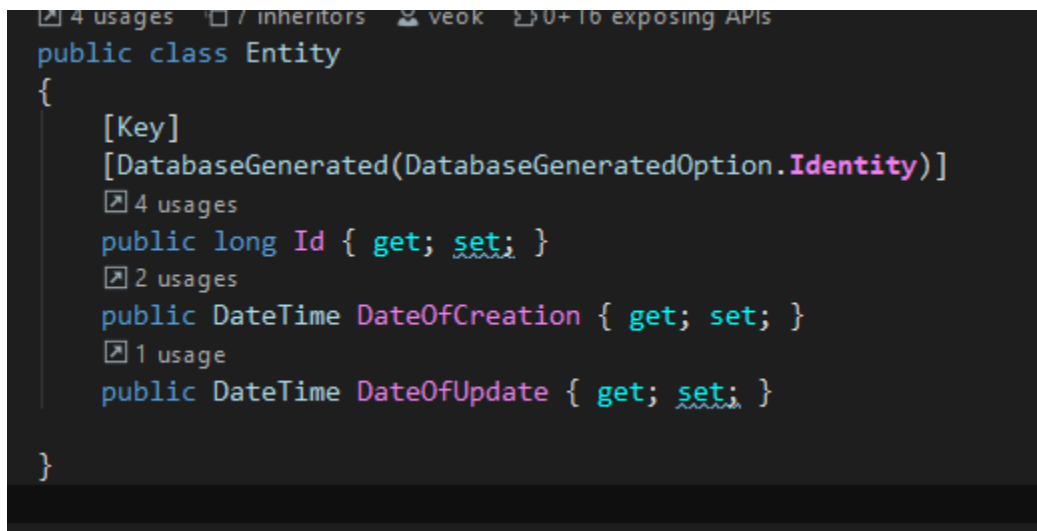
Aby stworzyć nowy projekt należy kliknąć prawym przyciskiem myszy w Solution, wybrać **Add->Unit Test Project**. W okienku wybieramy jakiego frameworka do testów chcemy użyć. W tym projekcie wybór padł na **XUnit**.



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.



```
public class Entity
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public long Id { get; set; }
    public DateTime DateOfCreation { get; set; }
    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; }
}
```

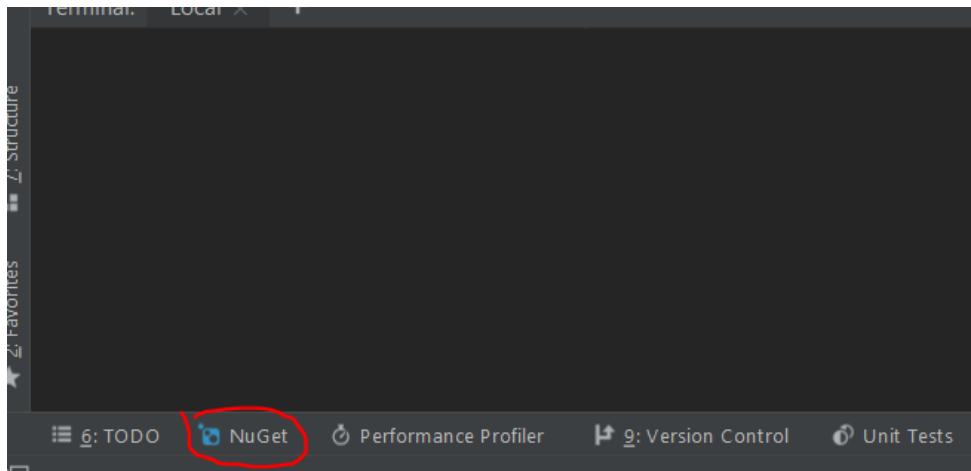
Na ten moment mamy gotowy podstawowy model klas. W następnym punkcie zostanie omówiona konfiguracja EntityFramework Core 2.2, wraz z implementacją i wygenerowaniem tabel w bazie danych na podstawie stworzonego wyżej modelu.

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

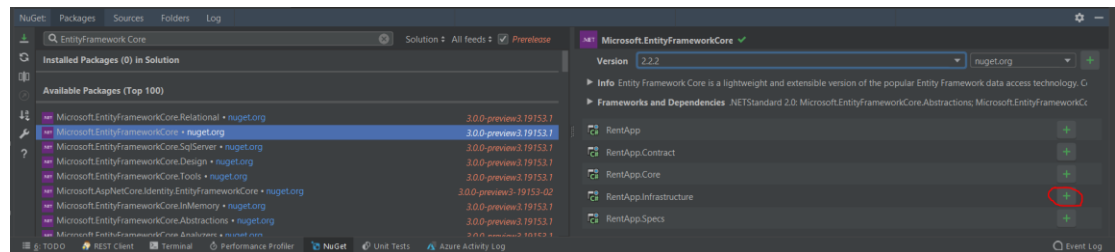
Entity Framework jest narzędziem zapewniającym mapowanie obiektowo-relacyjne. Dzięki niemu jesteśmy w stanie łatwy sposób zaimplementować klasy które będą przełożeniem na encje w bazie danych. Również zapewnia on podstawową logikę wykonywania zapytań. Plusem stosowania rozwiązań typu ORM jest elastyczne 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 dołączeniu 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 propercji. 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);
            });

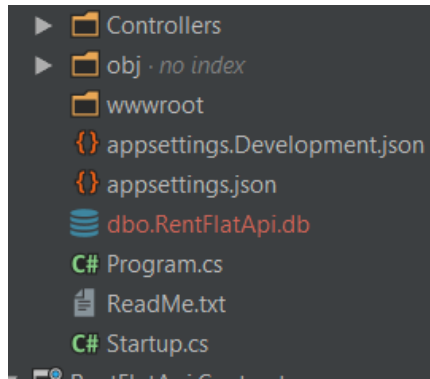
        migrationBuilder.CreateTable(
            name: "Owner",
            columns: table => new
            {
                Id = table.Column<long>(nullable: false)
                    .Annotation(name: "Sqlite:Autoincrement", value: true),
                DateOfCreation = table.Column<DateTime>(nullable: 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)
            },

```

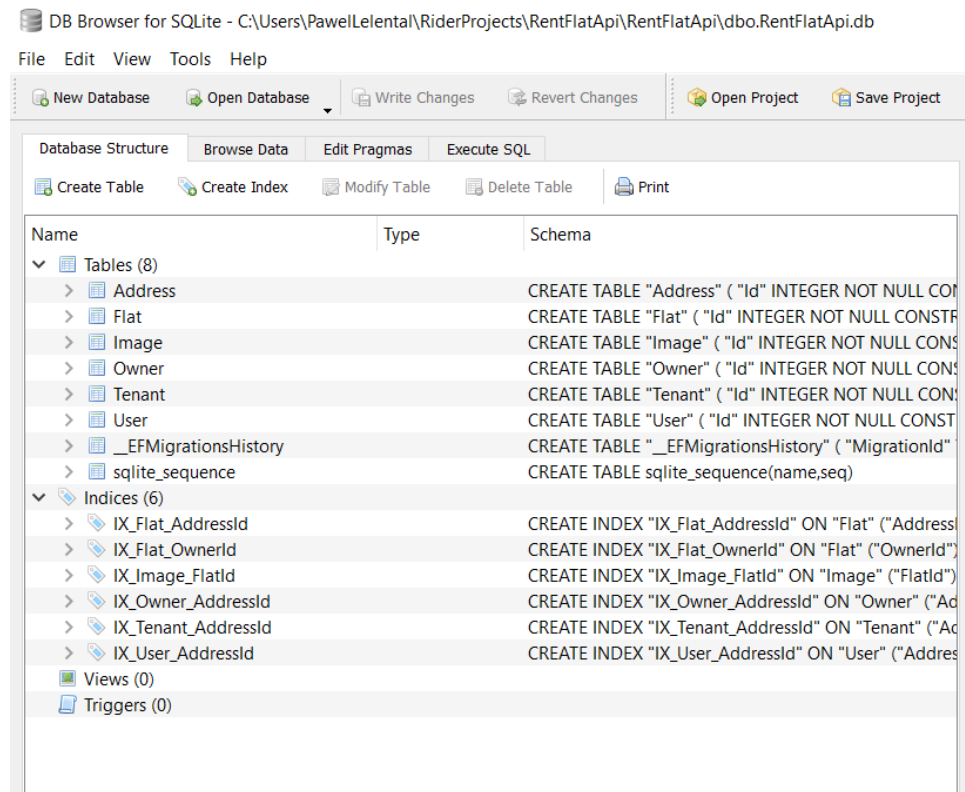
Następną komendą jest linijka która zaktualizuje naszą bazę (lub jak w tym przypadku, ją utworzy).

```
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 RentContext _rentContext;

2 veok
public FlatRepository(RentContext rentContext)
{
    _rentContext = rentContext;
}

```

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.

```
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)
        .Include(navigationPropertyPath: x => x.Images)
        .SingleOrDefaultAsync(predicate: x => x.Id == entity.Id);

    if (flatToUpdate != null)
    {
        flatToUpdate.Owner = entity.Owner;
        flatToUpdate.Images = entity.Images;
        flatToUpdate.Tenant = entity.Tenant;
        flatToUpdate.Address = entity.Address;
        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;
        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)