

Mariusz Wieczorek

mariusz.wieczorek@kabat.pl

Streszczenie

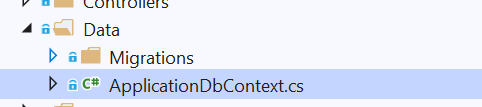
Separacja klas, rozdzielenie modeli na foldery Core i Persistence  
Tworzenie klas domenowych.  
Ustworzenie klasy ApplicationUser dziedziczącej po klasie IdentityUser  
Utworzenie bazy danych i tabel

asp.net Core

Tworzenie Modeli

**Tworzymy Modele**

Najpierw stworzymy model, a na jego podstawie bazę danych.  
Wykorzystamy ponownie podejście code first. Tym razem będziemy wykorzystywać  
wersję Entity Framework dedykowane do wersji Core.   
W katalogu Data jest plik ApplicationDbContext czyli nasz główny kontekst za pomocą którego będziemy odwoływać się do bazy za pomocą Entity Framework.



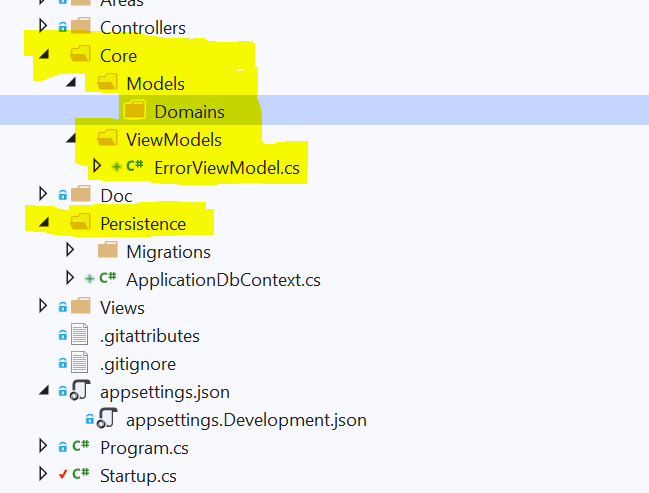
Zastosujemy dobre praktyki, podzielimy modele na dwa różne foldery  
folder **Core** w którym będą klasy domenowe, interfejsy i klasy bezpośrednio nie powiązane z żadnym frameworkiem, nas  
Oraz **Persistence** (trwałość), czyli klasy, które są już powiązane z konkretnymi frameworkami. Migracje są ściśle powiązane z frameworkami, więc przenosimy je do katalogu Persistence podobnie robimy z klasą ApplicationDbContext. Zmieniamy namespace i poprawiamy usingi w kodzie.

|  |
| --- |
| using Microsoft.AspNetCore.Identity.EntityFrameworkCore;  using Microsoft.EntityFrameworkCore;  using System;  using System.Collections.Generic;  using System.Text;  namespace MyTasks.Persistence  {  public class ApplicationDbContext : IdentityDbContext  {  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)  : base(options)  {  }  }  } |

W Core tworzymy nowe foldery takie jak Models, możemy przenieść ten, który pozostał.  
Data – usuwamy. Tworzymy katalog **ViewModels**, w którym będą klasy, które zazwyczaj są przekazywane z kontrolera do widoku.

W folderze Models potrzebujemy nowego folderu **Domains**, a w nim będą nasze obiekty domenowe, bazodanowe, na podstawie których będziemy tworzyć tabele w bazie danych.

ErrorViewModel przenosimy z katalogu Models do ViewModels.



W kolejnych krokach będziemy dodawać kolejne klasy do odpowiednich folderów i zobaczymy w jaki sposób pomoże nam taka separacja tych klas.

**Tworzymy klasy domenowe**

Więc w katalogu Domains tworzymy klasy Category i Task

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Collections.ObjectModel;  using System.ComponentModel.DataAnnotations;  using System.Linq;  using System.Threading.Tasks;  namespace MyTasks.Core.Models.Domains  {  public class Category  {  public Category()  {  Tasks = new Collection<Task>();  }  public int Id { get; set; }  [Required]  public string Name { get; set; }  public ICollection<Task> Tasks;  }  } |

String jest typem domyślnie nulowalnym więc musimy go oznaczyć jako wymagane.

|  |
| --- |
| W bardziej rozbudowanych aplikacjach preferowane **fluent Api**, gdzie bardziej można odseparować konfigurację od modelu. Ale w małej aplikacji możemy sobie pozwolić na **data annotation**. Jeżeli użylibyśmy fluent API to nasza konfiguracja powinna trafić do folderu Persistence. |

Zadanie może posiadać jedną kategorię, ale jedna kategoria może być przypisana do wielu zadań.  
Dlatego w klasie z kategoriami dodamy listę zadań i ją inicjujemy w konstruktorze.

Klasa Task.

|  |
| --- |
| Uwaga nasza klasa będzie się nazywała **Task**, a taka klasa już istnieje tylko w innym namespace. Co gorsza jest zawsze domyślnie dodawany ten namespace przy tworzeniu nowej klasy.  using System.Threading.Tasks;  Dlatego zawsze musimy uważać przy odwołaniu do klasy Task czy brany jest pod uwagę odpowiedni namespace. Tutaj usuwamy go z usingów na górze klasy |

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.ComponentModel.DataAnnotations;  using System.Linq;  using System.Threading.Tasks;  namespace MyTasks.Core.Models.Domains  {  public class Task  {  public int Id { get; set; }  [MaxLength(50)]  [Required(ErrorMessage ="Pole tytułjest wymagane.")]  [Display(Name ="Tytuł")]  public string Title { get; set; }  [MaxLength(250)]  [Required(ErrorMessage = "Pole opis jest wymagane.")]  [Display(Name = "Opis")]  public string Description { get; set; }  [Required(ErrorMessage = "Pole kategoria jest wymagane.")]  [Display(Name = "Kategoria")]  public int CategoryId { get; set; }  [Display(Name = "Termin")]  public DateTime? Term { get; set; }  [Display(Name = "Zrealizowane")]  public bool IsExecuted { get; set; }  public int UserId { get; set; }  public Category Category { get; set; }  public ApplicationUser User { get; set; }  }  } |

|  |
| --- |
| Do klasy z użytkownikiem za bardzo nie mamy dostępu, dlatego aby dodać nowe pola do użytkownika musimy utworzyć nową klasę **ApplicationUser**, która będzie dziedziczyła po klasie IdentityUser. |

|  |
| --- |
| using Microsoft.AspNetCore.Identity;  using System;  using System.Collections.Generic;  using System.Collections.ObjectModel;  using System.Linq;  using System.Threading.Tasks;  namespace MyTasks.Core.Models.Domains  {  public class ApplicationUser : IdentityUser  {  public ApplicationUser()  {  Tasks = new Collection<Task>();  }  public ICollection<Task> Tasks;  }  } |

Zadanie może zostać przypisane do konkretnego użytkownika, konkretny użytkownik może mieć przypisane wiele zadań.

Jeżeli chodzi o naszą klasę ApplicationUser, to musimy w aplikacji poprawić odwołania tam gdzie mamy już odwołania do IdentityUser.

Musimy te odwołania zmienić w trzech plikach:  
ApplicationDbContext  
Klasa dziedziczy po IdentityDbContext, musimy jeszcze mu przekazać ApplicationUser

|  |
| --- |
| namespace MyTasks.Persistence  {  public class ApplicationDbContext : IdentityDbContext  {  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)  : base(options)  {  }  }  } |
| namespace MyTasks.Persistence  {  public class ApplicationDbContext : IdentityDbContext<ApplicationUser>  {  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)  : base(options)  {  }  }  } |

W klasie Startup.cs w miejscu gdzie konfigurujemy nasze serwisy musimy zmienić IdentityUser  
na ApplicationUser.

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddDbContext<ApplicationDbContext>(options =>  options.UseSqlServer(  Configuration.GetConnectionString("DefaultConnection")));  services.AddDatabaseDeveloperPageExceptionFilter();  services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)  .AddEntityFrameworkStores<ApplicationDbContext>();  services.AddControllersWithViews();  } |
| public void ConfigureServices(IServiceCollection services)  {  services.AddDbContext<ApplicationDbContext>(options =>  options.UseSqlServer(  Configuration.GetConnectionString("DefaultConnection")));  services.AddDatabaseDeveloperPageExceptionFilter();  services.AddDefaultIdentity<ApplicationUser>(options => options.SignIn.RequireConfirmedAccount = true)  .AddEntityFrameworkStores<ApplicationDbContext>();  services.AddControllersWithViews();  } |

I w widoku częściowym \_LoginPartial.cshtml

|  |
| --- |
| @using Microsoft.AspNetCore.Identity  @inject SignInManager<IdentityUser> SignInManager  @inject UserManager<IdentityUser> UserManager |
| @using Microsoft.AspNetCore.Identity  @using MyTasks.Core.Models.Domains  @inject SignInManager<ApplicationUser> SignInManager  @inject UserManager<ApplicationUser> UserManager |

W ApplicationDbContext

|  |
| --- |
| namespace MyTasks.Persistence  {  public class ApplicationDbContext : IdentityDbContext<ApplicationUser>  {  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)  : base(options)  {  }  public DbSet<Task> Tasks { get; set; }  public DbSet<Category> Categories { get; set; }  }  } |

Na koniec jeszcze konfigurujemy connectionstring.  
W przypadku aplikacji ASP.NET Core robimy to w appsetting.json

|  |
| --- |
| {  "ConnectionStrings": {  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-MyTasks-7258E40F-6E2C-434A-9455-1D4848C8A564;Trusted\_Connection=True;MultipleActiveResultSets=true"  },  "Logging": {  "LogLevel": {  "Default": "Information",  "Microsoft": "Warning",  "Microsoft.Hosting.Lifetime": "Information"  }  },  "AllowedHosts": "\*"  } |
| {  "ConnectionStrings": {  "DefaultConnection":"Server=127.0.0.1;Database=MyTasks;Uid=user1;Pwd=alamakota;"  },  "Logging": {  "LogLevel": {  "Default": "Information",  "Microsoft": "Warning",  "Microsoft.Hosting.Lifetime": "Information"  }  },  "AllowedHosts": "\*"  } |

Przechodzimy do Package Manager Console i dodajemy nową migrację

Dodajemy nową migrację

|  |
| --- |
| Package Manager Console Host Version 5.8.1.7021  Type 'get-help NuGet' to see all available NuGet commands.  PM> add-migration init  Build started...  Build succeeded.  To undo this action, use Remove-Migration.  PM> update-database  Build started...  Build succeeded.  Done.  PM> |

Tworzymy konto, potwierdzamy I logujemy się do aplikacji

