Mariusz Wieczorek

mariusz.wieczorek@kabat.pl

Streszczenie

Co to jest Dependency Injection  
Utworzenie abstrakcji w postaci interfejsów

asp.net Core

Dependency Injection

**Dependency Injection**

Jest to wzorzec polegający na usuwaniu bezpośrednich zależności w kodzie.  
Dzięki niemu możemy uzyskać tzw. luźnie powiązania [ loose coupling ] do czego powinniśmy dążyć pisząc aplikację.

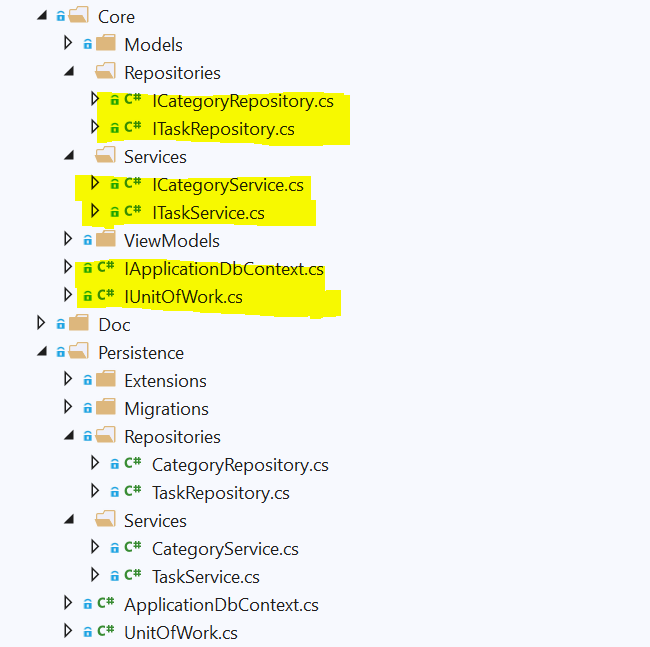
Chcemy aby nasze aplikacje były bardziej elastyczne, a ewentualne zmiany w przyszłości były jak najłatwiejsze do wprowadzania, oraz co bardzo ważne dążymy do tego aby nasza aplikacja była testowalna, czyli aby łatwo było napisać do niej testy jednostkowe.

Aby to zrobić musimy opierać nasze rozwiązania nie na konkretnych implementacjach ale na interfejsach, co sprawia, że mamy w kodzie małe zależności.

Interfejsy są stabilne, co oznacza, że jeżeli zrobimy zmianę w interfejsie, to ta zmiana jest również powiązana z implementacją tego interfejsu. Jeżeli natomiast zrobimy zmianę w konkretnej implementacji, to zazwyczaj nie musimy zmieniać tego interfejsu.

Pierwszym krokiem do wprowadzenia DI jest stworzenia abstrakcji do klas np. w postaci interfejsów, a następnie operowanie w aplikacji nie na konkretnych implementacjach ale na interfejsach.  
Konkretne implementacje zostaną właśnie dzięki DI wstrzyknięte do naszej aplikacji.  
Będziemy musieli później w klasie Startup.cs zdefiniować takie powiązania.

Interfejsy będziemy tworzyć w folderze Core.  
Tworzymy w nim folder Repositories i Services



Tworzymy Interfejsy dla repozytoriów, serwisów, UnitOfWork i ApplicationDbContext.

|  |
| --- |
| namespace MyTasks.Core.Repositories  {  interface **ICategoryRepository**  {  public IEnumerable<Category> GetCategories();  }  } |

|  |
| --- |
| namespace MyTasks.Core.Repositories  {  public interface **ITaskRepository**  {  IEnumerable<Task> Get(string userId,  bool isExecuted = false,  int categoryId = 0,  string title = null);  Task Get(int id, string userId);  void Add(Task task);  void Update(Task task);  void Finish(int id, string userId);  void Delete(int id, string userId);  }  } |

|  |
| --- |
| namespace MyTasks.Core  {  public interface **IApplicationDbContext**  {  DbSet<Task> Tasks { get; set; }  DbSet<Category> Categories { get; set; }  int SaveChanges();  }  } |

|  |
| --- |
| namespace MyTasks.Core  {  public interface **IUnitOfWork**  {  ITaskRepository Task { get; set; }  ICategoryRepository Category { get; set; }  void Complete();  }  } |

|  |
| --- |
| namespace MyTasks.Core.Services  {  public interface **ITaskService**  {  IEnumerable<Task> Get(string userId,  bool isExecuted = false,  int categoryId = 0,  string title = null);  Task Get(int id, string userId);  void Add(Task task);  void Update(Task task);  void Finish(int id, string userId);  void Delete(int id, string userId);  }  } |

|  |
| --- |
| namespace MyTasks.Core.Services  {  public interface **ICategoryService**  {  IEnumerable<Category> GetCategories();  }  } |

W poszczególnych klasach ustawiamy, że implementują one utworzony interfejs.

|  |
| --- |
| namespace MyTasks.Persistence.Repositories  {  public class **CategoryRepository** : **ICategoryRepository**  {  private readonly ApplicationDbContext \_context; |

|  |
| --- |
| namespace MyTasks.Persistence.Repositories  {  public class **TaskRepository** : **ITaskRepository**  {  private readonly **IApplicationDbContext** \_context;  public **TaskRepository**(**IApplicationDbContext** context)  {  \_context = context;  } |

|  |
| --- |
| namespace MyTasks.Persistence.Services  {  public class **CategoryService** : **ICategoryService**  {  private readonly IUnitOfWork \_unitOfWork; |

|  |
| --- |
| namespace MyTasks.Persistence.Services  {  public class **TaskService** : **ITaskService**  {  private readonly IUnitOfWork \_unitOfWork; |

|  |
| --- |
| namespace MyTasks.Persistence  {  public class **ApplicationDbContext** : IdentityDbContext<ApplicationUser> , **IApplicationDbContext**  { |

|  |
| --- |
| namespace MyTasks.Persistence  {  public class **UnitOfWork** : **IUnitOfWork**  {  // readonly przy polu oznacza, że jego wartość  // możemy zmienić tylko w konstruktorze  private readonly IApplicationDbContext \_context; |

Następnie w tych klasach wszystkie wystąpienia typów będącymi konkretnymi klasami zamieniamy na odpowiadające im nazwy interfejsów. Do konstruktorów tych klas teraz możemy przekazać dowolne klasy, które implementują wskazany interfejs.

Wprowadzamy również zmiany w kontrolerze

|  |
| --- |
| namespace MyTasks.Controllers  {  [Authorize]  public class **TaskController** : Controller  {  private readonly **TaskService** \_taskService;  private readonly **CategoryService** \_categoryService;    public TaskController(ApplicationDbContext context)  {  \_taskService = new TaskService(new UnitOfWork(context));  \_categoryService = new CategoryService(new UnitOfWork(context));  } |

Dzięki DI inicjalizacja w konstruktorze będzie wyglądała w inny sposób.

Dodamy nowy parametr **ITaskService**, mówimy, że za pomocą DI zostanie wstrzyknięty ten serwis do tego konstruktora. Nie musimy wstrzykiwać tylko do konstruktora, możemy również do właściwości, czy metody ale my wykorzystamy podejście związane z wstrzyknięciem do konstruktora.  
To co zostało przekazane zostanie przypisane do \_taskService. Nie potrzebujemy już ApplicationDbContext.

|  |
| --- |
| namespace MyTasks.Controllers  {  [Authorize]  public class **TaskController** : Controller  {  private readonly **ITaskService** \_taskService;  private readonly **ICategoryService** \_categoryService;  public TaskController(**ITaskService** **task**, **ICategoryService** **category**)  {  \_taskService = **task**;  \_categoryService = **category**;  } |

Teraz musimy dodać jeszcze konfigurację dla Dependency Injection, będziemy ją wpisywać w Startup.cs w metodzie ConfigureServices.  
Tutaj mówimy tak: W każdym miejscu naszej aplikacji, gdzie użyłem ITaskService wstaw implementację TaskService.

|  |
| --- |
| // dla każdego request'a jedna instancja tej klasy  **services.AddScoped<ITaskService, TaskService>();**  // ciągle będziemy pracować na jednej instancji  **services.AddSingleton<ITaskService, TaskService>();**  // każde użycie powoduje powstanie nowej instancji  **services.AddTransient<ITaskService, TaskService>();** |

