Wstrzykiwanie serwisów - wzorzec Dependency Injection. Zastosowanie Data Transfer Objects.

Programowanie aplikacji WWW w technologii .NET, 2021/2022

Przygotowała: I. Kartowicz-Stolarska

CEL

1. Obsługa zapytań w aplikacji webowej z wykorzystaniem wzorca Dependency Injection.
2. Elementy czystej architektury w Razor Page i ASP.NET Core.

SŁOWNIK

Serwis/ Services
Jest odpowiedzialny za logikę biznesową aplikacji, czyli wymagania danej aplikacji. Łączy różne elementów aplikacji i zawiera kod, który jest odpowiedzialny za reagowanie na żądanie użytkownika. Przygotowuje i zwraca dane z repozytoriów (w EntityFrameworkCore repozytoriami mogą być DbSet'y) w odpowiednim formacie np. przygotowuje dane do zapis do bazy danych np. ustawia datetime, ale nie zapisuje tych danych do bazy.

Repozytorium
Operuje na bazie danych np. zapisuje dane przygotowane w serwisie do bazy danych. Nie powinny zawierać żadnej logiki związanej z przygotowaniem danych, jak również logiki biznesowej aplikacji, czy instrukcji warunkowych. Repozytoria mogą tylko sprawdzać, czy dane przekazane przez serwis nie są null'em i zwracać informację o wyjątku do serwisu.

DependencyInjection
DependencyInjection to mechanizm, w którym odpowiednie zależności są ładowane w taki sposób, aby serwis, aż do momentu inicjalizacji, nie nie wiedział jaka będzie implementacja danego serwisu. Operowanie na inferface'ach umożliwia niezależność od implementacji i bibliotek np. EntityFramework do obsługi bazy danych. W ten sposób, to aplikacja umożliwia sterowanie kodem, a szczegóły dotyczące implementacji nie są ukryte w kodzie.

Data Transfer Objects (DTO)
DTO to są takie klasy, które odpowiadają za odwzorowanie modeli Entity, ale nie posiadają żadnych metod i konstruktorów. Posiadają tylko wybrane właściwości i nie przechowują żadnych zachowań. DTO dla aplikacji webowych trzymamy w katalogu ViewModels. Dla każdego serwisu najlepiej utworzyć oddzielny model.

TUTORIAL

Przygotowanie środowiska pracy

1. Utwórz **nowy** projekt z klasami Person, Address, Group i PersonGroup z zajęć "Dostęp do bazy danych".
2. Zainstaluj wymagane pakiety Nuget.
3. Utwórz context PeopleContext bazując na poprzednich zajęciach.
4. Utwórz nową bazę danych i skonfiguruj połączenie.
5. Wygeneruj migracje i zaktualizuj bazę danych.
6. Nie pomijaj pierwszych pięciu punktów xD. Zrób je, to bardzo ważne.
7. Do klasy Person.cs dodaj nową właściwość isActive:
```
public bool IsActive { get; set; }
```
8. Ponownie wygeneruj migrację ze zmianą i zaktualizuj bazę danych.

1. Proste wstrzykiwanie serwisów

W tym tutorialu zrealizujesz prosty serwis zwracający listę osób, które mają ustawioną flagę *IsActive* na wartość *true* (są "aktywne"). W tym rozwiązaniu serwis będzie ściśle powiązany i zależny od frameworka ORM - *EntityFrameworkCore*. W tutorialu "*Wstrzykiwanie serwisów z wykorzystaniem repozytoriów i DTO*" poznasz bardziej elastyczne rozwiązanie pozwalające tworzyć serwisy niezależne od użytej biblioteki ORM.

Na początek przygotuj środowisko pracy według pierwszych 8 punktów z sekcji Tutorial.

Proces wstrzykiwanie serwisu wymaga:
- zaimplementowania interface'u serwisu oraz klasy serwisu implementującego ten interface
- modyfikacji modelu strony, który będzie wywoływał serwis
- wywołania wstrzykiwania serwisu w pliku Program.cs (w starszych wersjach Starup.cs)

Przygotowanie interface'ów

W projekcie dodaj nowy katalog *Interfaces* i utwórz nowy interface IPersonService.

Interface będzie implementował jedną metodą *GetActivePeople* zwracającą listę aktywnych osób z tabeli *Person.*

```
public interface IPersonService {
     public IQueryable<Person> GetActivePeople();
}
```

Wywołanie interface'ów w modelach strony

1. Przejdź do pliku *Pages/IndexModel.cshtml.cs*
2. Zadeklaruj nowe właściwości oraz zmodyfikuj konstruktor. W metodzie OnGet wywołaj metodę serwisu *GetActivePeople* zwracającą wszystkie aktywne rekordy z modelu Person.

```
public class IndexModel : PageModel {
     private readonly ILogger<IndexModel> _logger;
     private readonly IPersonService _personService;
     public IQueryable<Person> Records { get; set; }

     public IndexModel(ILogger<IndexModel> logger,
IPersonService personService)
     {
          _logger = logger;
          _personService = personService;
     }

     public void OnGet()
     {
         Records = _personService.GetActivePeople();
     }
}
```

3. Uzupełni szablon *Pages/IndexModel.cshtml* o wyświetlanie danych z bazy:

```
<div class="text-center">
     <p>W bazie jest @Model.Records.Count() obiektów. Lista
aktywnych osób:</p>

   @foreach (var item in Model.Records) {
     <p>@item.FirstName @item.LastName</p>
     }
</div>
```

Implementacja klasy serwisu

1. W projekcie dodaj nowy katalog *Services* i utwórz nową klasę *PersonService* implementującą serwis *IPersonService*.

```
 public class PersonService: IPersonService {
```

```
        public IQueryable<Person> GetActivePeople()
        {
              throw new NotImplementedException();
        }
}
```

2. W serwisie zaimplementujemy właściwość prywatną umożliwiającą podłączenie do bazy danych. W tym rozwiązaniu serwis będzie ściśle związany z frameworkiem EntityFrameworkCore i jego kontekstem.
```
private readonly PeopleContext _context;
```

3. W metodzie *GetActivePeople()* wywołuję pobranie wszystkich aktywnych osób z kontekstu.
```
private readonly PeopleContext _context;

public PersonService(PeopleContext context) {
      _context = context;
}
public IQueryable<Person> GetActivePeople() {
     return _context.Person.Where(p => p.IsActive);
}
```

Wstrzykiwanie zależności

Zwróć uwagę, że w modelach strony posługujemy się interface'ami serwisów. Aby aplikacja wiedziała, z której klasy implementującej serwis ma korzystać, należy w *Program.cs* wstrzyknąć właściwą zależność. W naszym wypadku będzie to *PersonService* implementujący interface *IPersonInterface*.

```
builder.Services.AddRazorPages();
…..
builder.Services.AddTransient<IPersonService,
PersonService>();
```

Serwisy można dołączyć na 3 sposoby przez:

- AddSingleton
- AddTransient
- AddScoped

Jaka jest różnica?

Uruchom aplikację i sprawdź, czy działa xD.

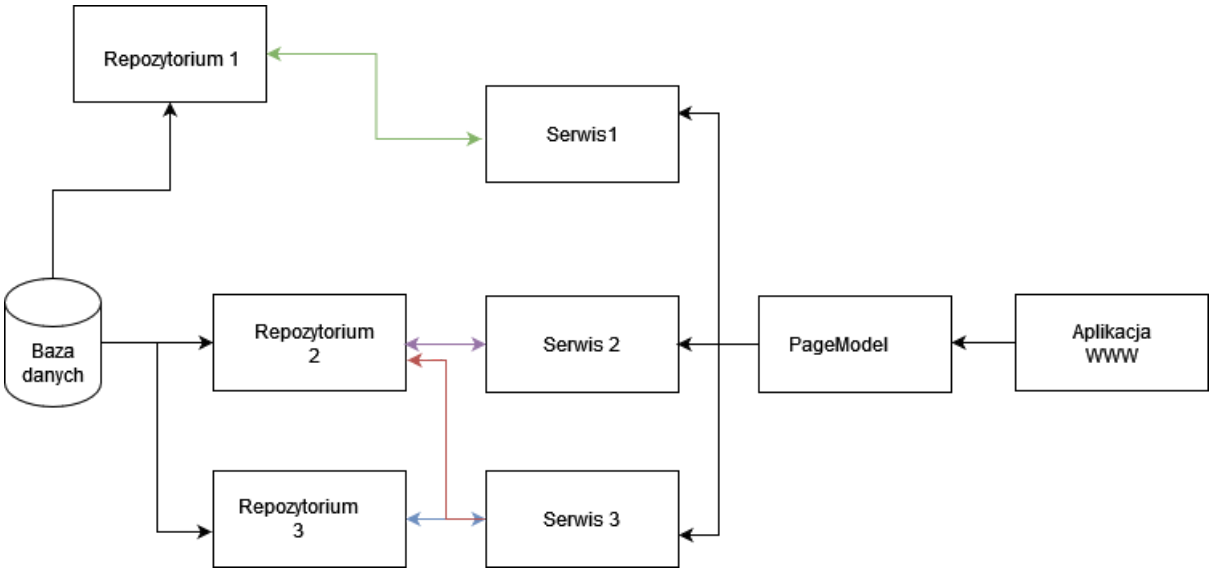
2. Wstrzykiwanie serwisów z wykorzystaniem repozytoriów i DTO

W tym tutorialu zrealizujesz serwis zwracający listę osób, które mają ustawioną flagę *IsActive* na wartość *true* (są "aktywne"). W tym rozwiązaniu serwis będzie niezależny od frameworka ORM - *EntityFrameworkCore*. Dodatkowo utworzysz klasy pomocnicze do pobierania danych z bazy danych - DTO oraz repozytoria, które umożliwią pobieranie danych z bazy danych niezależnie od użytej biblioteki ORM.

Założenia:

- model strony (w MVC kontroler) powinien być możliwie prosty. Nie powinien zawierać żadnej logiki biznesowej poza walidacją danych
- repozytoria odpowiadają za komunikację z bazą danych
- seriwsy odpowiadają za przygotowanie danych do dodania do bazy/lub za ich wyświetlanie i za całą logikę biznesową aplikacji, która łączy różne jej elementy

Przepływ danych:



Na początek przygotuj środowisko pracy według pierwszych 8 punktów z sekcji Tutorial.

Realizacja tego tutoriala wymaga:

- przygotowania modeli DTO
- zaimplementowanie interface'ów repozytoriów
- zaimplementowania interface'u serwisu oraz klasy serwisu implementującego ten interface

- modyfikacji modelu strony, który będzie wywoływał serwis
- wywołania wstrzykiwania serwisu w pliku Program.cs (w starszych wersjach Starup.cs)

Tworzenie Data Transfer Objects (DTO)

1. Utwórz katalog *ViewModels*, a w nim katalog Person.
2. W katalogu *ViewModels/Person* utwórz nową klasę *PersonForListVM*. Naszym celem będzie stworzenie takiej klasy, która będzie zwracała listę obiektów Person do wyświetlenia na stronie głównej aplikacji. Sufix VM pozwoli odróżnić modele Entity od modeli View Model.
3. W klasie *PersonForListVM* będziemy przechowywać tylko te elementy, które będziemy wyświetlać na stronie głównej aplikacji np. imię i nazwisko.
```
public class PersonForListVm {
      public int Id { get; set; }
      public string FullName { get; set; }
}
```
4. Dodatkowo utwórz klasę do wyświetlania listy osób, które będą zawierać listę obiektów PersonForListVM oraz ilość obiektów Person zapisanych w bazie. Taki sposób przygotowania danych zminimalizuje ilość zapytań związanych z zapytaniem bazy danych o ilość elementów zapisanych w bazie.
```
public class ListPersonForListVM {
      public List<PersonForListVm> People { get; set; }
      public int Count { get; set; }
}
```

Przygotowanie interface'ów

Utwórz nowy katalog *Interfaces*, który będzie zawierał klasy inferface'ów związanych z naszym projektem.

Interface serwisu

1. W katalogu Interfaces utwórz nową klasę o nazwie *IPersonService*. Będzie to interface dla serwisu związanego z modelem Person. Interface będzie deklarował metody zwracające zwracające konkretne obiekty DTO dla wybranej akcji.
2. W interface *IPersonService* zdefiniuj jedną metodę *GetPeopleForList* o zwracanym typie *ListPersonForListVM*.
```
      public interface IPersonService {
            ListPersonForListVM GetPeopleForList();
      }
```

Interface repozytorium

1. W katalogu Interfaces utwórz interface *IPersonRepository*. Interface IPersonRepository będzie odpowiedzialny za połączenie do bazy danych za pomocą wybranej biblioteki ORM i użycie właściwego kontekstu.
2. W pliku *IPersonRepository* zadeklaruj metody *GetAllActivePeople*.

```
public interface IPersonRepository {
      IQueryable<Person> GetAllActivePeople();
}
```

Tworzenie serwisów

1. W projekcie dodaj nowy katalog Services i utwórz nową klasę PersonService implementujący interface *IPersonService*.

```
public class PersonService : IPersonService {
      public ListPersonForListVM GetPeopleForList()
      {
            throw new NotImplementedException();
      }
}
```

2. W serwisie dodaj właściwość prywatną umożliwiającą podłączenie do bazy danych. Nie chcemy, aby serwis był związany z frameworkiem EntityFramework, a tylko odwoływał się do interface'u odpowiedzialnego za podłączenie do bazy danych.

```
private readonly IPersonRepository _personRepo;
```

3. Utwórz konstruktor ustawiający repozytorium:

```
public PersonService(IPersonRepository personRepo) {
      _personRepo = personRepo;
}
```

4. W metodzie *GetPeopleForList()* wywołaj pobranie wszystkich aktywnych osób z repozytorium odpowiedzialnego za pobranie danych z bazy danych.

```
public class PersonService : IPersonService {
      private readonly IPersonRepository _personRepo;
      public PersonService(IPersonRepository personRepo) {
                  _personRepo = personRepo;
      }

       public ListPersonForListVM GetPeopleForList() {
                  return _personRepo.GetAllActivePeople();
       }
}
```

5. W pliku *Services/PersonService.cs* uzupełnij metodę *GetPeopleForList*. Ta metoda zwróci nam tylko te dane, które są wymagane do wyświetlenia w widoku głównym strony. W metodzie zostanie użyte mapowanie obiektów Person na obiekty DTO ListPersonForListVM

```
public class PersonService : IPersonService {
	private readonly IPersonRepository _personRepo;
	public PersonService(IPersonRepository personRepo) {
		_personRepo = personRepo;
	}

	public ListPersonForListVM GetPeopleForList() {
		var people =  _personRepo.GetAllActivePeople();
		ListPersonForListVM result = new
	ListPersonForListVM();
		result.People = new List<PersonForListVm>();
		foreach (var person in people) {
		// mapowanie obiektow
		var pVM = new PersonForListVm()
		{
			Id = person.Id,
			FullName = person.FirstName + " " +
		person.LastName
		};
		result.People.Add(pVM);
	}
	result.Count = result.People.Count;
	return result;
	}
}
```

* Mapowanie obiektów można zoptymalizować za pomocą biblioteki AutoMapper;

Tworzenie repozytoriów

1. Utwórz nowy katalog *Repositories* i w nim klasę *PersonRepository* z pustą implementacją metod z interface'u *IPersonRepository*.

```
public class PersonRepository : IPersonRepository
{
	public IQueryable<Person> GetAllActivePeople()
	{
		throw new NotImplementedException();
	}
}
```

2. Następnie rozszerz implementację *Respositories/PersonRepository.cs* o context *PeopleContext* i metodę, która zwraca listę aktywnych osób (rekordy

z flagą *IsActive* ustawioną na *true*). Klasa *PersonRepository* jest powiązana z *EntityFrameworkCore* i jej kontekstem.

```
public class PersonRepository : IPersonRepository {
   private readonly PeopleContext _context;

   public PersonRepository(PeopleContext context) {
        _context = context;
   }
   public IQueryable<Person> GetAllActivePeople()
   {
     return _context.Person.Where(p => p.IsActive);
   }
}
```

Wstrzykiwanie zależności

Na ten moment są już utworzone serwisy, interface'y i repozytoria, ale aplikacja nie ma informacji w jaki sposób te komponenty będą z sobą współpracować. Współpracę wymienionych wyżej komponentów umożliwi wzorzec DependencyInjection. W przykładzie posłużymy się wstrzykiwaniem zależności za pomocą konstruktora.

1. W konfiguracji aplikacji Program.cs wskaż z jakiej implementacji interface'u *IPersonRepository* korzystasz. W naszym wypadku będzie to *PersonRepository*. Wstrzyknij też zależność od serwisu.

```
builder.Services.AddRazorPages();
…..
builder.Services.AddTransient<IPersonService,
PersonService>();
builder.Services.AddTransient<IPersonRepository,
PersonRepository>();
```

2. Aby uniknąć zbyt wielu wpisów *builder.Services.AddTransient<>* możesz utworzyć dodatkową klasę statyczną o nazwie np. DependencyInjection.cs. Klasę umieść bezpośrednio w projekcie:

```
public static class DependencyInjection
     {
     public static IServiceCollection AddProjectService(this
IServiceCollection services)
     {
          services.AddTransient<IPersonService,
PersonService>();
          services.AddTransient<IPersonRepository,
PersonRepository>();
          return services;
     }
```

```
}
```

A następnie w Program.cs zamiast

```
builder.Services.AddTransient<IPersonService,
PersonService>();
builder.Services.AddTransient<IPersonRepository,
PersonRepository>();
```

wywołaj

```
builder.Services.AddProjectService();
```

Wywołanie interface'ów w modelach strony

1. Uzupełnij Pages/IndexModel.cshtml.cs o pobranie obiektów z bazy danych za pomocą serwisów i ViewModel.

```
public class IndexModel : PageModel
      {
      private readonly ILogger<IndexModel> _logger;
      private readonly IPersonService _personService;
      public ListPersonForListVM Records { get; set; }

      public IndexModel(ILogger<IndexModel> logger,
IPersonService personService)
      {
            _logger = logger;
            _personService = personService;
      }

      public void OnGet()
      {
            Records = _personService.GetPeopleForList();
      }
}
```

2. Dodaj losowe dane do bazy danych.
3. Uzupełni szablon o wyświetlanie danych z bazy:

```
<div class="text-center">
      <p>W bazie jest @Model.Records.Count obiektów. Lista
      aktywnych osób:</p>

   @foreach (var item in Model.Records.People) {
      <p>@item.FullName</p>
      }
</div>
```

Uruchom aplikację i porównaj wydajność aplikacji implementującej serwisy z i bez wykorzystania dodatkowych repozytoriów i obiektów DTO.

PRZYDATNE LINKI

- https://youtu.be/KZxTOCuTuc0 - wstrzykiwanie serwisów video
-
- https://docs.microsoft.com/pl-pl/dotnet/core/extensions/dependency-injection-guidelines
- https://www.learnrazorpages.com/advanced/dependency-injection
- https://www.tutorialsteacher.com/core/dependency-injection-in-aspnet-core
- https://auth0.com/blog/dependency-injection-in-dotnet-core/
- https://stackoverflow.com/questions/38138100/addtransient-addscoped-and-addsingleton-services-differences

DLA ZAAWANSOWANYCH

- https://automapper.org/