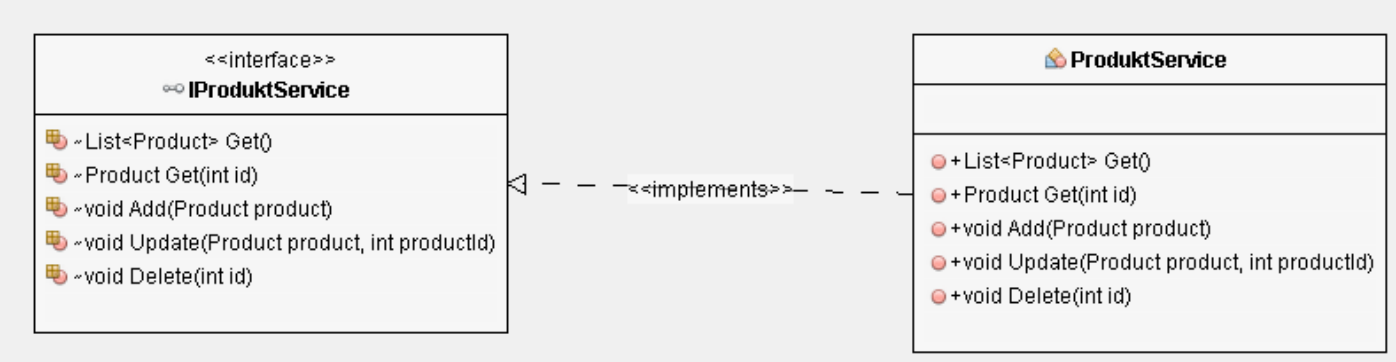
Aplikacje WWW
(studia niestacjonarne)
Laboratorium 3

SOA, Creating Services, AutoMapper

1. Wprowadzenie do SOA (Service-Oriented Architecture)

Architektura zorientowana na usługi (SOA) jest metodyką wytwarzania oprogramowania, w której główny nacisk jest kładziony na tworzenie autonomicznych usług, które to są później wykorzystywane w innych komponentach systemu. Usługa jest logicznie odseparowanym elementem kodu, dzięki czemu nie powinna wykazywać zależności pomiędzy innymi usługami. Posiada interfejs, który pozwala na ukrycie konkretnej implementacji dla klientów danej usługi. Tworząc usługę najpierw budujemy dla niej abstrakcyjny interfejs, następnie implementujemy wcześniej utworzony interfejs w postaci klasy (klas).

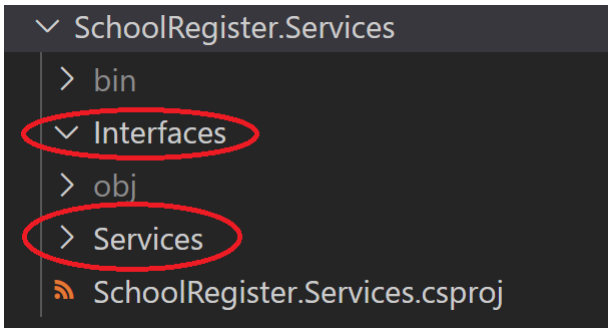


Dzięki temu podejściu możliwe jest szybkie testowanie, oraz podmiana poszczególnych komponentów jak i zastosowanie wzorców Strategy oraz Dependency Injection. Często również stosuje się klasę bazową (BaseService) która przechowuje referencję do obiektu DbContext. Dzięki temu nie musimy w konkretnych usługach powielać kodu inicjalizującego obiekt kontekstu. Aby zbudować podstawową architekturę typu SOA należy wydzielić wszystkie klasy oraz interfejsy usług do osobnej biblioteki (w naszym przypadku jest to SchoolRegister.Services). Dzięki temu możemy zaimportować bibliotekę usług w dowolnym momencie do projektu aplikacji (Web). W przypadku tzw. Micro-Services oraz dużej logiki biznesowej zawartej w usługach, często spotyka się podejście: jeden projekt – jedna usługa. Lecz w naszym przypadku wystarczy jeden projekt dla wszystkich usług.

2. Tworzenie usług (Creating Services)

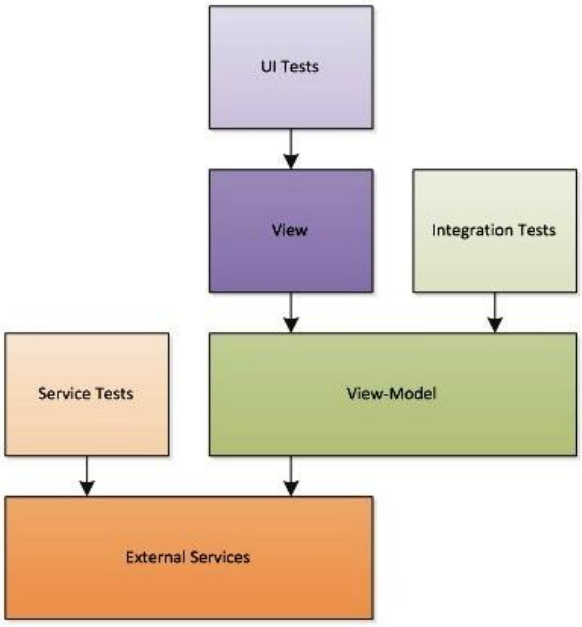
Aby stworzyć usługi należy wykonać poniższe kroki:

1) W projekcie ***SchoolRegister.Services*** powinny się znajdować dwa foldery ***Interfaces*** oraz ***Services***. Jeżeli nie istnieją, należy je utworzyć.



2) W folderze *Interfaces* tworzymy nowy nowy plik, który będzie interface'em o nazwie ***ISubjectService.cs***. Należy również zachować odpowiednią konwencję nazewniczą. Wszystkie interfejsy powinny się rozpoczynać od wielkiej litery „I". Natomiast interfejsy będące abstrakcją usług powinny się kończyć się sufiksem "Service" np. ***ISubjectService, IGroupService, ITeacherService, etc*** .Utworzony interfejs powinien być publiczny oraz zawierać deklaracje metod które będą później zaimplementowane przez klasę implementującą interfejs.

3) Aby poprawnie zadeklarować metody potrzebujemy wprowadzić pojęcia ***view modelu***. View modele są to klasy odpowiadające za agregację oraz dostarczanie danych między modelem a widokiem. Najczęściej klasy te „zbierają" dane z wielu modeli do jednej klasy lub redukują dane otrzymane z modelu. Takie podejście jest często stosowane we modelu programowania MVVM. Równie często view modele stosowane są do ukrywania niechcianych właściwości modelu w widoku. Takie podejście daje wiele korzyści, szczególnie na etapie testowania (testy jednostkowe). View Modele najczęściej stosowane są przy pobieraniu i wyświetlaniu danych.



4) W projekcie ***SchoolRegister.ViewModels*** mamy obecnie jeden folder VM posiadający jeden view model, który został przeniesiony z projektu Web. View modele będziemy umieszczać w folderze VMs. Każda klasa znajdująca się w tym folderze powinna zawierać suffix **Vm**, np. **SubjectVm, AddOrUpdateSubjectVm**.

```
using System.ComponentModel.DataAnnotations;

namespace SchoolRegister.ViewModels.VM
{
    0 references
    public class AddOrUpdateSubjectVm
    {
        0 references
        public int? Id { get; set; }
        [Required]
        0 references
        public string Name { get; set; }

        [Required]
        0 references
        public string Description { get; set; }

        [Required]
        0 references
        public int TeacherId { get; set; }
    }
}
```

W klasie ***AddOrUpdateSubjectVm*** annotacja ***[Required]*** pozwoli na walidację view modelu przy pomocy ***ModelState*** na poziomie kontrolera.

```
using System.Collections.Generic;

namespace SchoolRegister.ViewModels.VM
{
    0 references
    public class SubjectVm
    {
        0 references
        public int Id { get; set; }

        0 references
        public string Name { get; set; }

        0 references
        public string Description { get; set; }

        0 references
        public IList<GroupVm> Groups { get; set; }

        0 references
        public string TeacherName { get; set; }
        0 references
        public int? TeacherId { get; set; }
    }
}
```

5) Oprócz dwóch wspomnianych view modeli należy również stworzyć ***GroupVm***

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace SchoolRegister.ViewModels.VM
{
    1 reference
    public class GroupVm
    {
        0 references
        public int Id { get; set; }
        [Required]
        0 references
        public string Name { get; set; }
    }
}
```

Teraz możemy wykorzystać view modele do zadeklarowania metod w interfejsie.

```
using System;
using System.Collections.Generic;
using System.Linq.Expressions;
using SchoolRegister.Model.DataModels;
using SchoolRegister.ViewModels.VM;

namespace SchoolRegister.Services.Interfaces
{
    public interface ISubjectService
    {
        SubjectVm AddOrUpdateSubject(AddOrUpdateSubjectVm addOrUpdateVm);
        SubjectVm GetSubject(Expression<Func<Subject, bool>> filterExpression);
        IEnumerable<SubjectVm> GetSubjects(Expression<Func<Subject, bool>> filterExpression = null);
    }
}
```

6) Aby móc korzystać z ***ApplicationDbContext*** można stworzyć klasę bazową serwisu **BaseService**, po której będą dziedziczyć wszystkie serwisy. Klasę tę należy stworzyć w folderze ***Services*** w projekcie

SchoolRegister.Services. Dzięki temu wszystkie klasy dziedziczące po klasie bazowej będą mogły korzystać ze składowych klasy bazowej, np. ***ApplicationDbContext.***

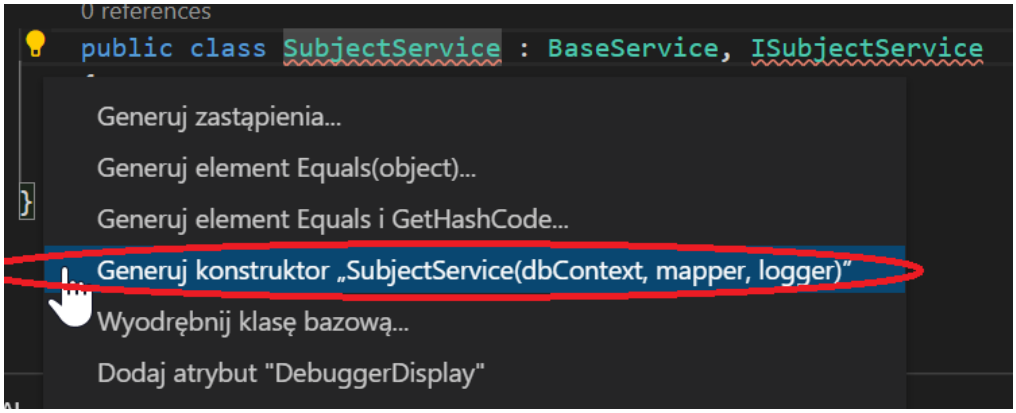
7) Aby klasa ***BaseService*** działała poprawnie należy zainstalować dodatkowo z NuGet-a poniższe pakiety dla projektu ***SchoolRegister.Services***

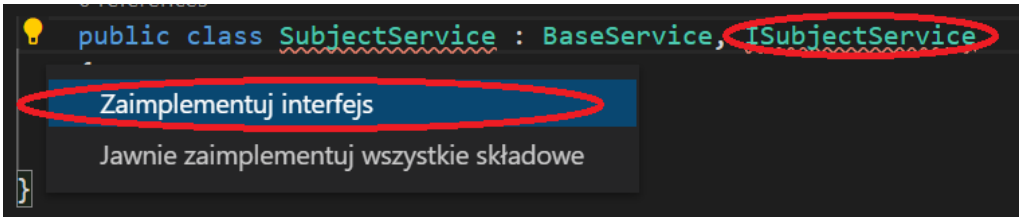
- `dotnet add .\SchoolRegister.Services\SchoolRegister.Services.csproj package Microsoft.Extensions.Logging.Abstractions --version 5.0.0`
- `dotnet add .\SchoolRegister.Services\SchoolRegister.Services.csproj package AutoMapper --version 10.1.1`
- `dotnet add .\SchoolRegister.Services\SchoolRegister.Services.csproj package AutoMapper.Extensions.Microsoft.DependencyInjection --version 8.1.1`

```
using System;
using AutoMapper;
using Microsoft.Extensions.Logging;
using SchoolRegister.DAL.EF;

namespace SchoolRegister.Services.Services {
    public abstract class BaseService {
        protected readonly ApplicationDbContext DbContext;
        protected readonly ILogger Logger;
        protected readonly IMapper Mapper;
        public BaseService (ApplicationDbContext dbContext, IMapper mapper, ILogger logger) {
            DbContext = dbContext;
            Logger = logger;
            Mapper = mapper;
        }
    }
}
```

8) Następnie można zaimplementować interfejs ***SubjectService*** w postaci klasy. Po jej stworzeniu **(w folderze Services)** możemy stworzyć konstruktor oraz zaimplementować metody. Aby pojawiła się żółta ikona żarówki proszę kliknąć na podkreślony na czerwono element.





Stworzony konstruktor klasy umożliwiaja wstrzyknięcie obiektu typu ***ApplicationDbContext, IMapper, oraz ILogger***. Następnie wywoływany jest konstruktor klasy bazowej (***BaseService***) przekazując argumenty parametrów.

Uzyskany zostanie poniższy efekt:

```
using AutoMapper;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using SchoolRegister.DAL.EF;
using SchoolRegister.Model.DataModels;
using SchoolRegister.Services.Interfaces;
using SchoolRegister.ViewModels.VM;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;

namespace SchoolRegister.Services.Services
{
    public class SubjectService : BaseService, ISubjectService
    {
        public SubjectService(ApplicationDbContext dbContext, IMapper mapper, ILogger logger) : base(dbContext, mapper, logger)
        {
        }

        public SubjectVm AddOrUpdateSubject(AddOrUpdateSubjectVm addOrUpdateVm)
        {
            throw new NotImplementedException();
        }

        public SubjectVm GetSubject(Expression<Func<Subject, bool>> filterExpression)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<SubjectVm> GetSubjects(Expression<Func<Subject, bool>> filterExpression = null)
        {
            throw new NotImplementedException();
        }
    }
}
```

9) Dzięki zastosowania interfejsu możliwe jest zaimplementowanie klasy serwisu. Jak widać dziedziczymy po bazowym serwisie oraz implementujemy interfejs.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using AutoMapper;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using SchoolRegister.DAL.EF;
using SchoolRegister.Model.DataModels;
using SchoolRegister.Services.Interfaces;
using SchoolRegister.ViewModels.VM;

namespace SchoolRegister.Services.Services {
    public class SubjectService : BaseService, ISubjectService {
        public SubjectService (ApplicationDbContext dbContext, IMapper mapper, ILogger logger) : base (dbContext, mapper, logger) { }

        public SubjectVm AddOrUpdateSubject (AddOrUpdateSubjectVm addOrUpdateVm) {
            try {
                if (addOrUpdateVm == null)
                    throw new ArgumentNullException ($"View model parameter is null");
                var subjectEntity = Mapper.Map<Subject> (addOrUpdateVm);
                if (!addOrUpdateVm.Id.HasValue || addOrUpdateVm.Id == 0)
                    DbContext.Subjects.Add (subjectEntity);
                else
                    DbContext.Subjects.Update (subjectEntity);
                DbContext.SaveChanges ();
                var subjectVm = Mapper.Map<SubjectVm> (subjectEntity);
                return subjectVm;
            } catch (Exception ex) {
                Logger.LogError (ex, ex.Message);
                throw;
            }
        }

        public SubjectVm GetSubject (Expression<Func<Subject, bool>> filterExpression) {
            try {
                if (filterExpression == null)
                    throw new ArgumentNullException ($" FilterExpression is null");
                var subjectEntity = DbContext.Subjects.FirstOrDefault (filterExpression);
                var subjectVm = Mapper.Map<SubjectVm> (subjectEntity);
                return subjectVm;
            } catch (Exception ex) {
                Logger.LogError (ex, ex.Message);
                throw;
            }
        }

        public IEnumerable<SubjectVm> GetSubjects (Expression<Func<Subject, bool>> filterExpression = null) {
            try {
                var subjectEntities = DbContext.Subjects.AsQueryable ();
                if (filterExpression != null)
                    subjectEntities = subjectEntities.Where (filterExpression);
                var subjectVms = Mapper.Map<IEnumerable<SubjectVm>> (subjectEntities);
                return subjectVms;
            } catch (Exception ex) {
                Logger.LogError (ex, ex.Message);
                throw;
            }
        }
    }
}
```

Jak można zauważyć w metodzie ***AddOrUpdateService*** najpierw sprawdzamy: Jeżeli argument parametru ***addOrUpdateVm*** jest równy null, zgłaszany jest wyjątek. Następnie mapujemy przy pomocy ***AutoMappera*** obiekt view modelu na encję typu ***Subject***. Kolejno w zależności czy właściwość ***Id*** jest pusta lub równa 0, encja jest dodawana (Add), w przeciwnym wypadku modyfikowana (Update). Jednakże na tym etapie dane zmieniane są tylko na poziomie aplikacji (oczywiście w pamięci serwera aplikacji). Dopiero metoda ***SaveChanges*** pozwala na zapisanie tych zmian na bazie danych przy pomocy EF.

W metodzie ***GetSubject*** przekazujemy wyrażenie lambda jako parametr. Dzięki temu będzie możliwe filtrowanie po dowolnej właściwości obiektu typu ***Subject***. Jeśli argument tego parametru jest równy null, zgłaszany jest wyjątek. Następnie pobieramy encję zgodną z predykatem podanym jako parametr. Następnie uzyskana encja ***subject*** jest mapowana na view model – ***SubjectVm*** a następnie zwracana.

Metoda ***GetSubjects*** jest łudząco podobna ***GetSubject***, lecz jej zdaniem jest zwrócenie wielu obiektów. Stąd filtrowanie odbywa się przy pomocy ***Where***. Jeśli predykat podany jako argument parametru jest pusty, zwracane są wszystkie encje (spełniające predykat) znajdujące się w bazie danych. Jeśli argument istnieje to jest przekazywany do metody ***Where,*** która zwraca wszystkie encje zgodne w predykatem podanym jaki wyrażenie lambda.

3. AutoMapper

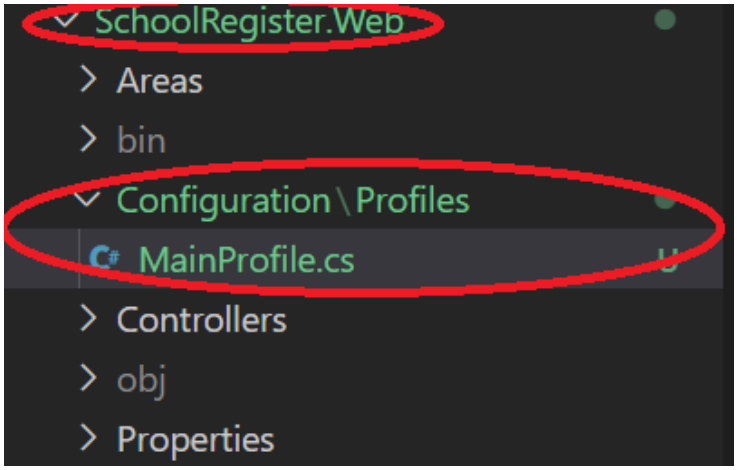
Bardzo często podczas tworzenia nowych serwisów pojawia się potrzeba skopiowania właściwości obiektu jednego typu do obiektów drugiego typu. Np z typu będącego encją do obiektu view modelu (Subject do SubjectVm lub odwrotnie). Kopiowanie właściwości po kolei jest bardzo nieefektowne oraz mało eleganckie ze względu na to że w przypadku gdy dany obiekt posiada wiele właściwości, kod tego typu rozrasta się bardzo szybko. Aby przeciwdziałać tej sytuacji należy stosować tzw. O-OM (Object-Object Mapper). Aby móc dokonywać mapowania, np. pomiędzy modelem a view modelem korzystamy z jednego z tzw. Object to Object Mapperów o nazwie ***AutoMapper***. Pozwala on na automatyczne przepisanie właściwości **o tej samej nazwie oraz o tym samym typie** pomiędzy obiektami rożnych typów. Aby zainstalować bibliotekę dla projektu ***SchoolRegister.Web*** korzystamy z NuGeta tak samo jak w przypadku projektu ***SchoolRegister.DAL***. Proszę zainstalować poniższe biblioteki przy pomocy terminala:

- `dotnet add .\SchoolRegister.Web\SchoolRegister.Web.csproj package AutoMapper --version 10.1.1`

- `dotnet add .\SchoolRegister.Web\SchoolRegister.Web.csproj package AutoMapper.Extensions.Microsoft.DependencyInjection --version 8.1.1`

Korzystając z dobrodziejstw AutoMappera należy pamiętać o zdefiniowaniu tzw. map oraz odpowiednim skonfigurowaniu biblioteki ***AutoMapper***. W tym celu proszę postępować wg. poniższych kroków:

1) W projekcie ***SchoolRegister.Web*** tworzymy nowy folder o nazwie ***Configuration*** a następnie w tym folderze tworzymy kolejny folder ***Profiles.*** W folderze ***Profiles*** tworzymy nowy plik o nazwie ***MainProfile.cs***



2) Klasa tego typu powinna wyglądać następująco:

```
using AutoMapper;
using SchoolRegister.Model.DataModels;
using SchoolRegister.ViewModels.VM;
using System.Linq;

namespace SchoolRegister.Web.Configuration.Profiles{
    public class MainProfile : Profile
    {
        public MainProfile()
        {
            //AutoMapper maps
            CreateMap<Subject, SubjectVm>() // map from Subject(src) to SubjectVm(dst)
                     // custom mapping: FirstName and LastName concat string to TeacherName
                    .ForMember(dest => dest.TeacherName, x => x.MapFrom(src => $"{src.Teacher.FirstName} {src.Teacher.LastName}"))
                    // custom mapping: IList<Group> to IList<GroupVm>
                    .ForMember(dest => dest.Groups, x => x.MapFrom(src => src.SubjectGroups.Select(y => y.Group)));

            CreateMap<AddOrUpdateSubjectVm, Subject>();
            CreateMap<Group, GroupVm>();
            CreateMap<SubjectVm, AddOrUpdateSubjectVm>();
            //other maps...
        }
    }
}
```

Klasa ta powinna wyglądać podobnie do powyższego przykładu. Nie musi wyglądać tak samo ze względu na sposób stworzenia serwisów oraz logiki biznesowej w nich umieszczonej.

Weźmy pod uwagę linijkę `CreateMap<AddOrUpdateSubjectVm, Subject>()`, mamy tutaj przykład prostej rejestracji mapy ***AddOrUpdateSubjectVm*** (Source) do **Subject** (destination). Mówimy wtedy o mapowaniu view modelu na encję modelu. Jeżeli typy oraz nazwy właściwości tych klas, będą takie same w obu klasach, nic poza rejestracją mapy nie musi być wykonane. W przeciwnym przypadku musimy zdefiniować mapy dla wybranych właściwości. np. w przypadku `CreateMap<Subject, SubjectVm>()` do właściwości ***TeacherName*** (string) musimy zdefiniować mapę, gdyż nie ma takiej właściwości w klasie ***SubjectVm***. Jest natomiast właściwość ***Teacher*** klasy ***Teacher***, więc możemy z niej zbudować stringa zawierającego imię oraz nazwisko nauczyciela, co będzie stanowiło ***TeacherName***. Taka złączenia stringów nazywamy template strings.

Powyższa klasa nie jest kompletna. Należy do niej dodać pozostałe mapy które będą nam potrzebne przy mapowaniu użytym w serwisach.

3) Proszę w projekcie ***SchoolRegister.Web*** w pliku ***Startup.cs*** w metodzie ***ConfigureServices*** zarejestrować nowy serwis

```
services.AddAutoMapper(typeof(Startup));
```

```
public class Startup
{
    2 references
    public IConfiguration Configuration { get; }
    0 references
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    // This method gets called by the runtime. Use this method to add services to the c
    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddAutoMapper(typeof(Startup));
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")
        );
```

4. Zadanie

Po zapoznaniu się z powyższą instrukcją, proszę zaprojektować oraz zaimplementować następujące usługi (services):

- ***TeacherService***
- ***GradeService***
- ***GroupService***
- ***StudentService***

Każda usługa powinna składać się z dwóch elementów: interfejs oraz klasa go implementująca. Proszę pamiętać aby dziedziczyć również po klasie *BaseService*. Proszę również stworzyć odpowiednie klasy view modelu, analogicznie jak w przypadku wcześniej utworzonej usługi *SubjectService*. Proszę przemyśleć funkcjonalność poszczególnych usług, być może możliwe jest wykorzystanie stworzonych wcześniej usług.

Metody w poszczególnych usługach należy zbudować w oparciu o funkcjonalności systemu podane poniżej:

- Wystawianie ocen studentom przez użytkowników z rolą ***Teacher***.
  *Podpowiedź: Proszę w celu sprawdzenia czy dany użytkownik należy do danej roli (np. Teacher), proszę wykorzystać klasę **UserManager<User>**, konkretnie metodę tej klasy **IsInRoleAsync(user, roleName)**. Proszę klasę UserManager wstrzyknąć do konstruktora jako parametr.*

  *Podpowiedź2: Proszę w celu pobrania encji konkretnego typu wykorzystać metodę **OfType<Type>**. Np.*

```
var teacher = DbContext.Users.OfType<Teacher>().FirstOrDefault(t => t.Id == addGradeToStudentVm.TeacherId);
```

- Możliwość wyświetlania swoich ocen (lub ocen swoich dzieci) przez użytkowników z rolą ***Parent*** lub ***Student***. Podpowiedź:
- Użytkownicy z rolą ***Teacher*** posiadają możliwość wysyłania emaila do użytkowników z rolą ***Parent***.
  *Podpowiedź: W celu wysłania emaila proszę wykorzystać klasę **SmtpClient***

- Zarządzanie grupami (klasami) (tworzenie oraz usuwanie grup)
- Dodawanie oraz usuwanie uczniów do i z grup.