Laborator 3
Dependency injection & Repository pattern

To be clear about some terms:
serviciu = bucata de cod / clasa care proceseaza date si rezolva un anumit scop
instanta = vezi OOP: class vs instance; basically, toti suntem oameni, dar eu sunt o instanta de om, tu esti alta instanta de om
Pana acum nu cred ca am explicat asta, asa ca o sa vin acum:
IQueryable vs IEnumerable vs List; la baza toate sunt colectii, dar functioneaza asa:

- IQueryable = asta tine minte query-ul spre baza de date; nu tine minte datele, ci cum sa ia datele; de fiecare date cand accesam datele dintr-un IQueryable, se face query-ul spre baza again.
- IEnumerable = the same thing, dar cu date in-memory; nu tine minte datele, si cum sa ia datele, si mereu ia datele din nou la accesarea lor.
- List = asta tine minte datele, nu cum sa le ia; ia datele doar la inceput si le copie pe propria lui memorie.

Hai sa incepem cu **Dependency Injection**. Dar pentru asta, sa fim sigur ce-i o dependinta si unde o vedem, ca sa intelegem de ce facem ceea ce facem.

In exemplele date pana cum, noi ma folosit clasa noastra dbContext direct in Controller (which is bad practice, dar asta o sa intelegem in partea a doua), asa ca hai sa luam exemplul asta:

```
public class BookController : ControllerBase
{
    public IActionResult FindByName(string title)
    {
        using var context = new LibraryDbContext();
        var books = context.Books.Where(x => x.Title.Contains(title));
        return Ok(books.ToList());
    }
}
```

Aici, BookController, pentru a-si face treaba si a functiona metoda ei (FindByName), are nevoie neaprata ca o instanta de DbContext sa fie up. Daca vin si rescriu putin de mai sus treaba pentru a arata asa:

```
public class BookController : ControllerBase
{
    public IActionResult FindByName(string title)
    {
        var books = GetBooksFilteredByTitle(title);
        return Ok(books.ToList());
    }
    private IQueryable<Book> GetBooksFilteredByTitle(string title)
    {
        using var context = new LibraryDbContext();
        var books = context.Books.Where(x => x.Title.Contains(title));
        return books;
    }
}
```

E, acum o sa avem o problema. Cand va ajunge la books.ToList(), el va incerca sa acceseze obiectul *context*; surpriza! Obiectul *context* a fost distrus la finalul metodei `GetBooksFilteredByTitle`, HOP eroare de dependinta.

Din acest motiv, spunem ca obeictul `BookController` este dependent de obiectul `LibraryDbContext`.

Evident ca intr-o aplicatie mai mare, in Controller-ul o sa depinde de o clasa „Service" care la randul ei o sa depinde de o alta clasa „Service" sau de „DbContext" direct, deci o ierarhie intreaga de dependinte, greu de controlat manual.

Deci, avand dependinte, greu de lucrat.. Rezolvarea? Hmm, hai sa inversam dependintele. Pai, in loc sa fie instanta de `BookController` dependenta de `LibraryDbContext`, de ce sa nu fie invers? Instanta de `LibraryDbContext` sa depinda de instanta de `BookController`. Boy, am zis „instanta" foarte des..

Bun, pai cum facem context-ul de depinda de Controller? Simplu, adaugam conterxt-ul ca proprietate in controller-ul nostru. Daca Controller-ul il detine, el va avea controlul, cat exista controller-ul exista si context-ul.

Asta inseamna inversion of control, schimbam controlul intre cele 2 clase.

Totusi, si ca proprietate, pentru ca ea sa aiba valoare, sa nu fie null, cineva sau ceva tot trebuie sa fie responsabil de a-i da o valoare.. aaa n-am chef, las' ca face .NET-ul; nu glumesc, chiar face .NET-ul. Noi trebuie doar sa ii zicem sa faca asta,

si rezolva cu o instanta configurata cat sa mearga bine, si sa nu depinda la randul ei de alta clase.

Cum noi vrem ca Controller-ul sa se contruiasca cu tot cu Context de la inceput, cerem din constructor instanta si o salvam in proprietate. It should look something like this:

```
public class BookController : ControllerBase
{
    private readonly LibraryDbContext _context; // am facut proprietatea privata
    public BookController(LibraryDbContext context) // am cerut prin constructor
    {
        this._context = context; // am salvat in proprietate
    }
    public IActionResult FindByName(string title)
    {
        var books = _context.Books.Where(x => x.Title.Contains(title));
        return Ok(books.ToList());
    }
}
```

And this ^ is Dependency Injection :)

Framework-ul cauta tipul cerut de noi in colectia sa de servicii (imediat zic de asta), creaza o instanta daca nu este deja creata, si o paseaza in constructor.

Colectia de servicii: asta este colectia care este confiurata in Startup (remember, laboratorul 1 :D); este prima functie de care ziceam pe atunci *ConfigureServices*.

Cand creem orice clasa (serviciu, repository, dbContaxt, etc) pe care il vrem folosit prin dependency injection, va trebui sa venim in Startup sa inregistrem respectiva clasa in colectia de servicii; e simplu, pur si simplu mergem si ii dam AddTransient / AddScoped / AddSingleton; Transient / Scoped / Singleton zice cum sunt controlate instantele respectivului serviciu:

- Transient = la fiecare injectare se creaza o instanta noua
- Scoped = in cadrul unul request, exista o singura instanta pentru toate injectiile; instante diferite pentru requesturi diferite
- Singleton = o singura instanta orice ar fi

One more thing to say here, would be: it's a good practice to use interfaces (cu exceptia DbContext); „teoria" din spate e greu de inteles din prima: interfetele astea creaza un lebel de abstractizare si imparte defintiile metodelor de implementarea lor, ajuta putin pentru a fi cat mai decuplate componentele intre

ele + ajuta pentru crearea de teste (see TDD online); e smecher ca se inejcteaza de obiecei interfata, in startup mentionezi ca de la interfata *IBookService* exista implementarea *BookService*, si stie el ce instanta sa-ti dea.

Basically, in Startup ar arata cam asa:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    // inregistram Db Contextul; asta are metoda lui speciala care il face Scoped
    services.AddDbContext<LibraryDbContext>(options => options.UseSqlServer("De-
    faultConnection"));
    services.AddTransient<IBookService, BookService>(); // inregistram si serviciul
    nostru ca Transient
}
```

Acum putem zice 2 cuvinte si despre **Repository Pattern**.
Povestioara: sa faci sa mearga ceva, e simplu. Cod care doar sa mearga poate scrie toata lumea; sa scrii cod organizat, care sa functioneze si pe o echipa intreaga de programare, that's the real deal.. asta face un developer sa fie bun. De obicei, se lucreaza in layere, fiecare layer stiind sa rezolve un anumit lucru; usually, o sa avem

- Data Access Layer (DAL) care se ocupa de cum sunt organizate datele, cum sunt datele retrieved si configurarile lor.. asta este cel care comunica cu baza de date.
- Business Logic Layer (BLL), care stie cum trebuiesc procesate datele in functie de logica custom a aplicatiei; asta este cea care comunica cu DAL-ul de mai sus.
- Layer-u de Controllere, care tot ce stie sa faca este sa preia cerintele userilor si sa expuna datele inapoi; el foloseste ce ii da BLL-ul de mai sus.

Aceste layere, comunica unul cu altul prin Dependency Injetion-ul de mai devreme, iar ceea ce numim Repository Pattern, este un design pattern ce, in general, organizeaza putin codul scris in DAL

Ce ne zice el? Sa creem un Repository pentru fiecare entitate; basically, vazut din afara, este depozitoul de date. Acest repository este singurul care stie cum si de

unde sa ia datele; Cand am nevoie sa iau date despre din tabela de Books, injectezi BookRepository si ma folosesc de ce metode imi expune acesta.

Mai mult de atat, pentru ca refolosirea codului e un lucru super util, se obisnuieste crearea unui GenericRepository, care sa aiba metode comune, care au aceeasi implementare indiferent de entitate. Dupa, repository-ul specific (BookRepository) vine, il mosteneste pe cel generic, si automat pot fi folosite si cele din generic; pe langa ele, fiecare repo specific isi scrie metodele specifice de accesare a datelor.

Exemplu de Generic Repository (ca interfata):

```
public interface IGenericRepository<TEntity> where TEntity : class
{
    Task<List<TEntity>> GetAll();
    void Create(TEntity entity);
    void Update(TEntity entity);
    void Delete(TEntity entity);

    void CreateRange(IEnumerable<TEntity> entities);
    void UpdateRange(IEnumerable<TEntity> entities);
    void DeleteRange(IEnumerable<TEntity> entities);
    Task CreateAsync(TEntity entity);
    Task CreateRangeAsync(IEnumerable<TEntity> entities);

    TEntity FindById(object id);

    bool Save();
}
```

Sincer, mai mult de atat nu prea e de zis despre acest Repository Pattern; lucrurile vin si se imbina mai bine cu cat il folosesti; pentru detalii, insa, o sa las link-ul asta aici:

https://www.programmingwithwolfgang.com/repository-pattern-net-core/