

# Loosely coupled .NET MVC Solution Structure with Service – Repository as Model

Georgi Kolev

Примерът съдържа:

- Структура на MVC Solution базирана на слаба връзка (Loose coupling) между компонентите и
- Inversion of Control pattern с Dependency Injection за инжектиране и управление на компонентите
- Service – Repository pattern като Model от MVC
- Подходи, които улесняват разширяването и поддръжката на кода (Scalability)

---

С разрастването на проектите, поддръжката и възможността им за разширяване се затруднява. Често това е следствие от не достатъчно добра структура.

Целта на този пример е да ви демонстрира структура на MVC сайт, която е: гъвкава, разширяема, с използвани компоненти, тествема и налага начин на писане на код, който е лесен за поддържане.

Бележка: Използват се схеми за изобразяване на идеята, кодът може да се види в приложеният проект.

[Първо пускане на проекта](#)

---

Накратко за всяка от практиките, които ще бъдат използвани:

**Repository pattern** – Разделяне на логиката, която изтегля информация от базата и я трансформира в обект, от бизнес логиката, която работи върху нея. По този начин бизнес логиката не се интересува от начина, по който се изтегля/записва информацията в базата от данни.

<http://msdn.microsoft.com/en-us/library/ff649690.aspx>

**Service layer pattern** – Дефиниране и имплементация на бизнес логика, която не е обвързана с потребителския интерфейс и не издава детайли как работи.

<http://programmers.stackexchange.com/questions/162399/how-essential-is-it-to-make-a-service-layer>

**Dependency Injection (DI)** - Начин за вмъкване на зависимостите в обект, вместо те да се инстанцират вътре в него. Контролира и техния lifecycle. В проекта се използва библиотека за DI с името Ninject.

<http://stackoverflow.com/questions/130794/what-is-dependency-injection>

**Loose coupling** – Взаимоотношенията между компонентите стават само чрез техните интерфейси. Това дава възможност за един интерфейс да има няколко различни имплементации, които да се използват в различни ситуации. Също така функционалност може да бъде променена без това да се отрази на използващите я.

[http://en.wikipedia.org/wiki/Loose\\_coupling](http://en.wikipedia.org/wiki/Loose_coupling)

В последвалото описание се предполага, че сте запознати с идеята на споменатите практики/подходи.

# Структура

Стандартно проектите биват изградени от 3 основни компонента:

Потребителски интерфейс – чрез който потребителят взаимодейства с приложението;

Бизнес слой – в която е имплементирана логическата част от приложението;

Операции за достъп до данни – които се грижат да изтеглят/записват данните на приложението, например от/в база данни, файлове, отдалечени сървъри посредством уеб сървиси и др.

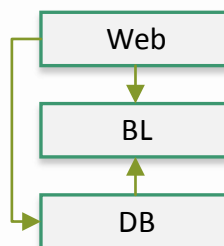
Всеки от тях ще е в отделен проект:

Web – потребителски интерфейс;

BL – бизнес логика;


DB – връзка към базата от данни.

Референции между проектите



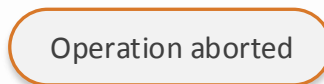
## Легенда

AccountService	Клас
BaseController	Абстрактен / Базов клас
IAccountRepository	Интерфейс
ValidateCurrPassword()	Част от операция (метод)
IQueryable<Account>GetAll()	Метод
Ctor(IAccountRepository repository)	Конструктор
IAccountRepository repository	Поле

 Преработка



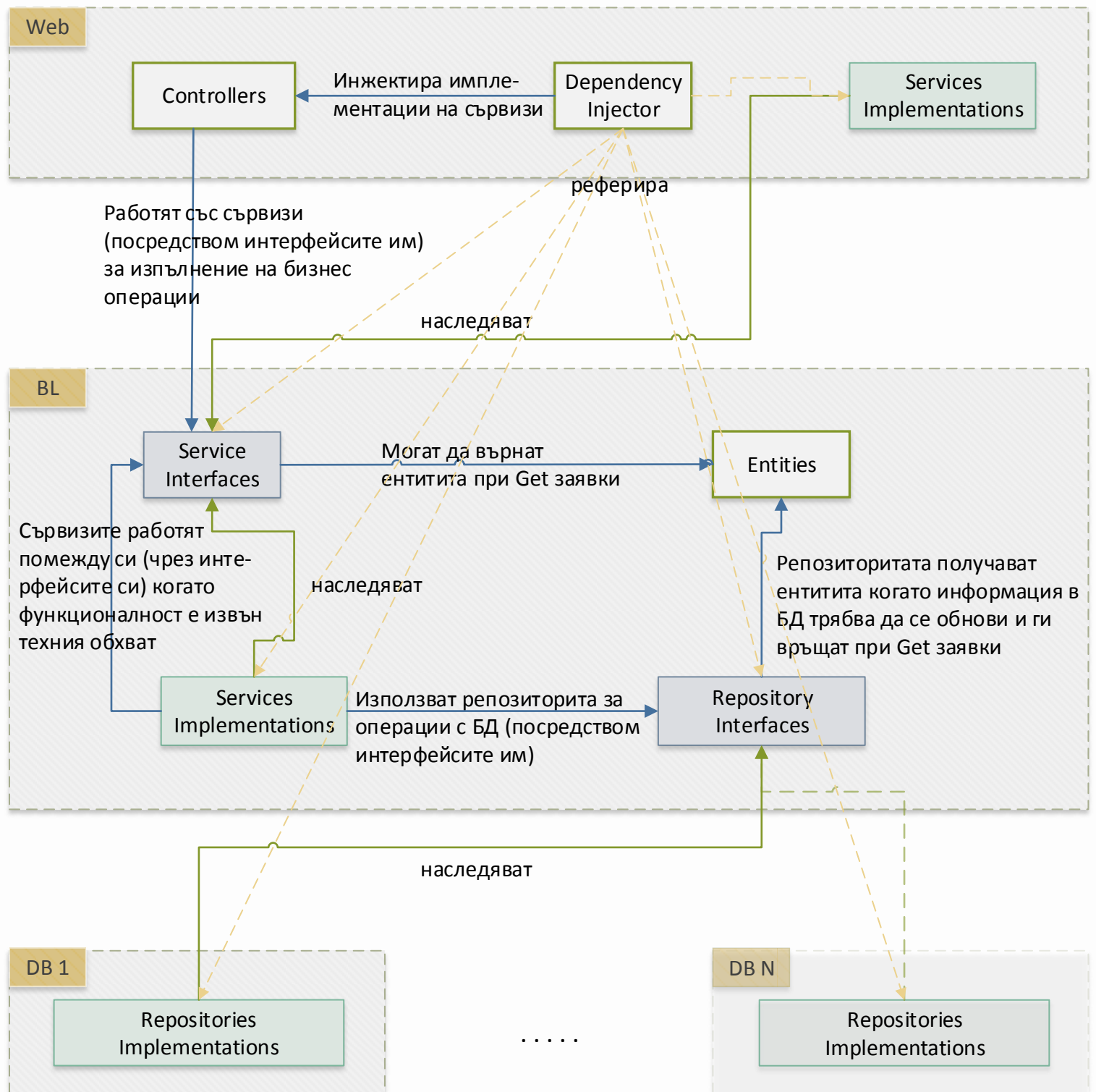
Логически  
IF блок



Край на блок  
схема



Схема на основните компоненти и взаимодействията между тях. Повече за техните функции на следващите страници.



## 1. Web : Потребителски интерфейс

Потребителският интерфейс работи с бизнес логиката посредством нейните интерфейси. Имплементациите им биват инстанцирани и вмъквани в контролерите и филтрите на Web чрез **Dependency Injection (DI)**.

За тази цел на DI обекта трябва да се укаже кои класове отговарят на необходимите интерфейси. Това включва и периода им на живот (по време на рекуест, винаги едни и същи, всеки път нови,...).

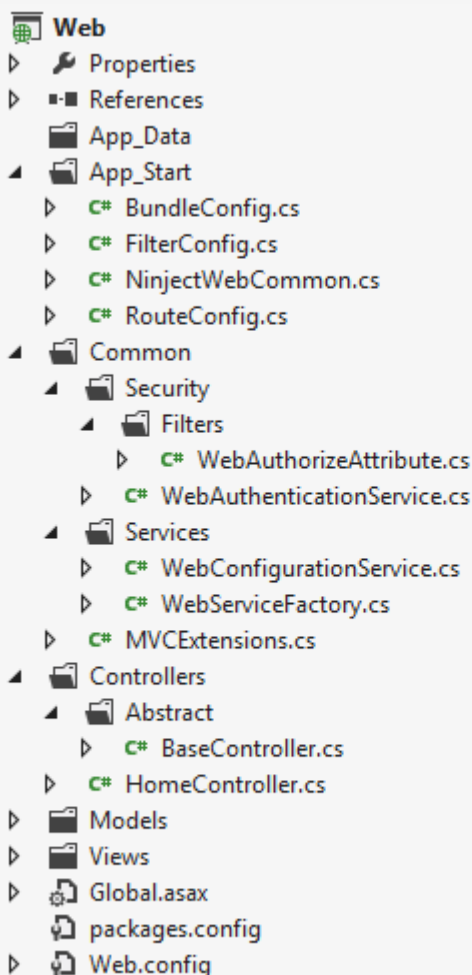
С използването на DI получаваме следните предимства:

- Премахване необходимостта от писането на factory-та, които да създават инстанции на интерфейси;
- Централизирано място, от което се управляват използваните компоненти в приложението;
- Възможност за лесна подмяна на вмъкваните обекти.

Важно:

Избягвайте реферирането на DI контейнера в кода (извън мястото в което се настройва).

В приложения проект за DI се използва Ninject. Неговата конфигурация се намира в NinjectWebCommon.



### Как изглежда настройката на Ninject

Вмъкване на имплементация за всеки сървиз интерфейс. За всяка http заявка се създава 1 инстанция на имплементацията, която се вмъква където се срещне интерфейса.

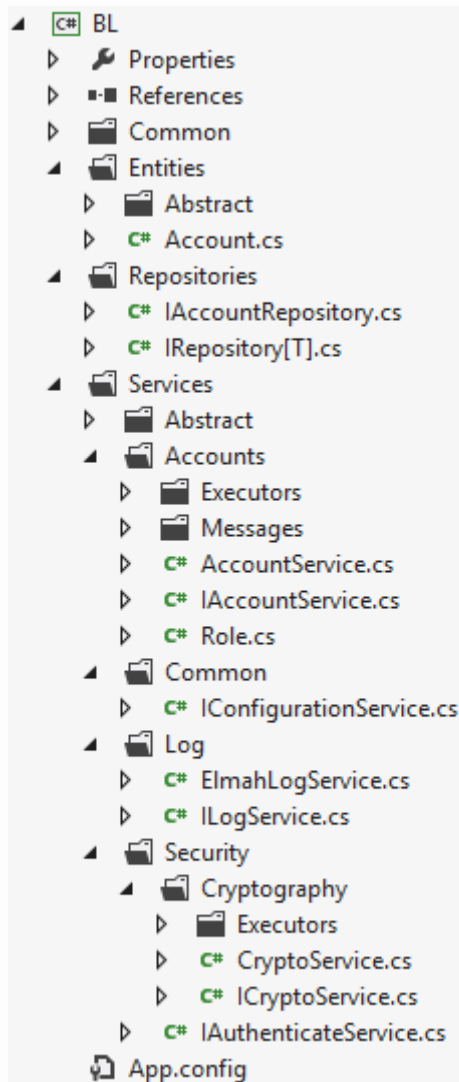
```
Bind<IAuthenticationService>().To<WebAuthenticationService>().InRequestScope()  
Bind<IAccountService>().To<AccountService>().InRequestScope()
```

Вмъкване на имплементация за всеки репозитори интерфейс.

```
Bind<IAccountRepository>().To<AccountRepository>().InRequestScope()
```

Вмъкване на една инстанция на DbContext-а във всяко репозитори. Инстанцията е жива, докато не изтече актуалната http заявка. По този начин се имплементира Unit of Work pattern.

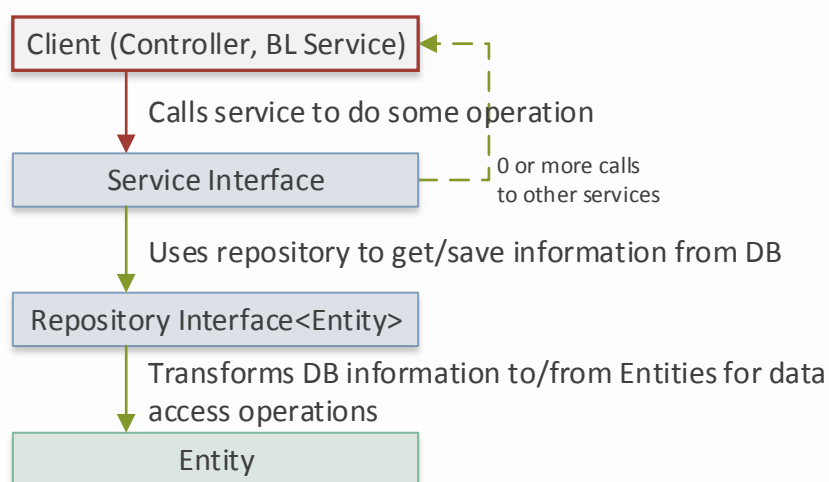
```
Bind<DbContext>().ToSelf().InRequestScope()
```



Бизнес логиката е съставена от 3 основни части:

- Entities – обекти представляващи информация от базата;
- Интерфейси на репозитория – дефиниращи операции за работа с базата от данни посредством ентити обекти;
- Services – съдържащи основната бизнес логика на приложението.

Взаимодействия при изпълнение на операция



**Entities** – Обекти отразяващи редове от таблици в базата от данни. Връщат се от репозиторията при изтегляне на информация и се подават когато трябва да се обновят редове в базата.

**Repositories** - Интерфейси на репозитория, изпълняват работата на посредник между базата от данни и бизнес логиката. В тях са дефинирани Create/Read/Update/Delete (CRUD) методи работещи с Entity обекти.

Интерфейсите са необходими, за да се осигури Loose coupling между базата от данни и бизнес логиката.

Освен това дават възможност за използване на паметта за съхранение на информацията в Unit тестовите, чрез което се заобикаля обвързването им с база от данни.

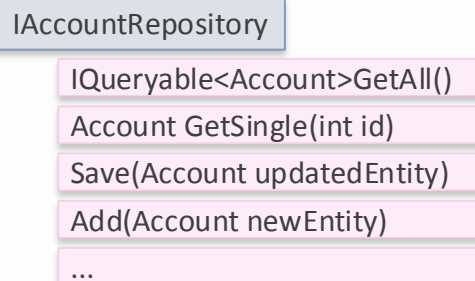
Имплементират се в DB Проекта.

DI Контейнера в Web се настройва да вмъква имплементациите (намиращи се в ДБ проекта/ите) за интерфейсите на репозиторията.

Важно:

- Всяко репозитори отговаря само за една таблица
- > separation of concerns;
- Репозиторията се използват само от сервизите;
- Препоръчително е репозиторията да работи (приема/връща) само неговия ентити обект;
- Трябва да връщат IQueryable за Get заявки когато е възможно;
- С тях се работи посредством техните интерфейси
- > loose coupling.

Примерен интерфейс на репозитори



**Services** – Основната (бизнес) логика на приложението представяна от интерфейси и техните имплементации. Интерфейсите са необходими, за да се постигне loose coupling между бизнес логиката и използващият го клиент.

DI контейнера в Web се настройва да вмъква имплементация когато е необходима инстанция на сървиз .

Пример: В базата има таблица за потребители, в нея могат да се добавят нови или да се променят наличните (от потребителския интерфейс). Операциите (добавяне, промяна, изтриване..) биват дефинирани в сървиз интерфейс и имплементирани в клас, който го наследява. Интерфейса се вкарва като конструктор параметър в контролерите, които предоставят тези операции на потребителя, а DI контейнера се настройва да вмъкне имплементиращия клас.

В последствие се добавя нова таблица за адреси на потребители. Това налага операциите за добавяне/промяна на потребители да се обновят така, че да попълват и тази информация. Операциите за работа с новата таблица се дефинират в нов сървиз интерфейс, който се имплементира в нов клас. Сървиза за потребители ще трябва да бъде обновен да получава допълнителна информация (в параметри на методите му) и референция към интерфейса на новия сървиз, който да използва за извикване на операциите в него. Тоест сървиза за потребители ще работи със сървиза за адреси когато е необходима промяна на адреси.

В някои случаи имплементациите трябва да бъдат създадени в клиентския проект, защото бизнес логиката не знае за тяхната специфика.

Пример: **IConfigurationService** дефинира метод за изтегляне на стойност на конфигурируема настройка. Тъй като местата, в които могат се държат настройките са специфични за главният проект – е оставена реализацията на него. В случая той е реализиран в **Web** като **WebConfigurationService**, вземащ настройките от Web.config.

Сървизите работят с репозиторитата посредством техните интерфейси, за да се постигне Loose coupling между бизнес логиката и базата от данни. Репозитори се вмъква посредством конструктор инжекшън.

Важно:

Един сервиз отговаря за функционалността около една специфична сфера (Потребители, Сигурност, Писма..), без да е зависим от клиента (Web / Console / Windows Forms app / another Service)

-> separation of concerns;

**\*\*Всеки сервиз може да използва (реферира) само до 1 репозитори като 1 репозитори не може да бъде използвано в повече от един сервиз -> separation of concerns;**

Трябва да връщат IQueryable за Get заявки когато е възможно -> за да може интерфейса да прилага допълнителна филтрация при нужда;

С тях се работи само посредством интерфейсите им -> loose coupling.

**\*\*Нормално е операция да използва няколко таблици, което налага достъпа до няколко репозиторита. Често необходимите репозиторита биват вмъквани в сервиза изпълняващ операцията .**

Въпреки, че тази реализация е по-лесна за използване, тя има следните недостатъци:

1 Репозитори се използва на повече от 1 място → от няколко места се променя таблица;

Не могат да се изолират (mock) частите на операцията, за да се тества само основата;

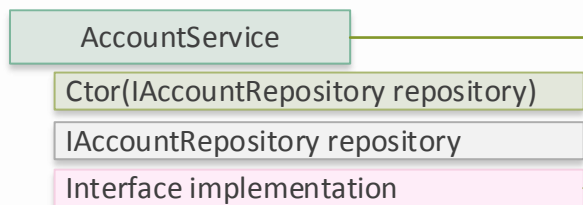
Сервизите се натрупват с репозиторита;

Предразполага към дублиране на функционалност;

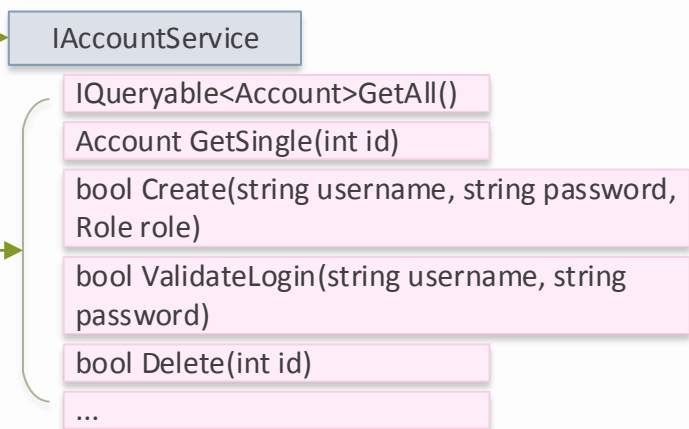
Увеличават се задълженията на сървиза, нарушава се separation of concerns.

По-добрият вариант е сервизите да работят един с друг, за да има разделение на функционалността, и за да могат да се тестват поотделно компонентите на операциите. **Повече за връзка между сървизи в т. 4.**

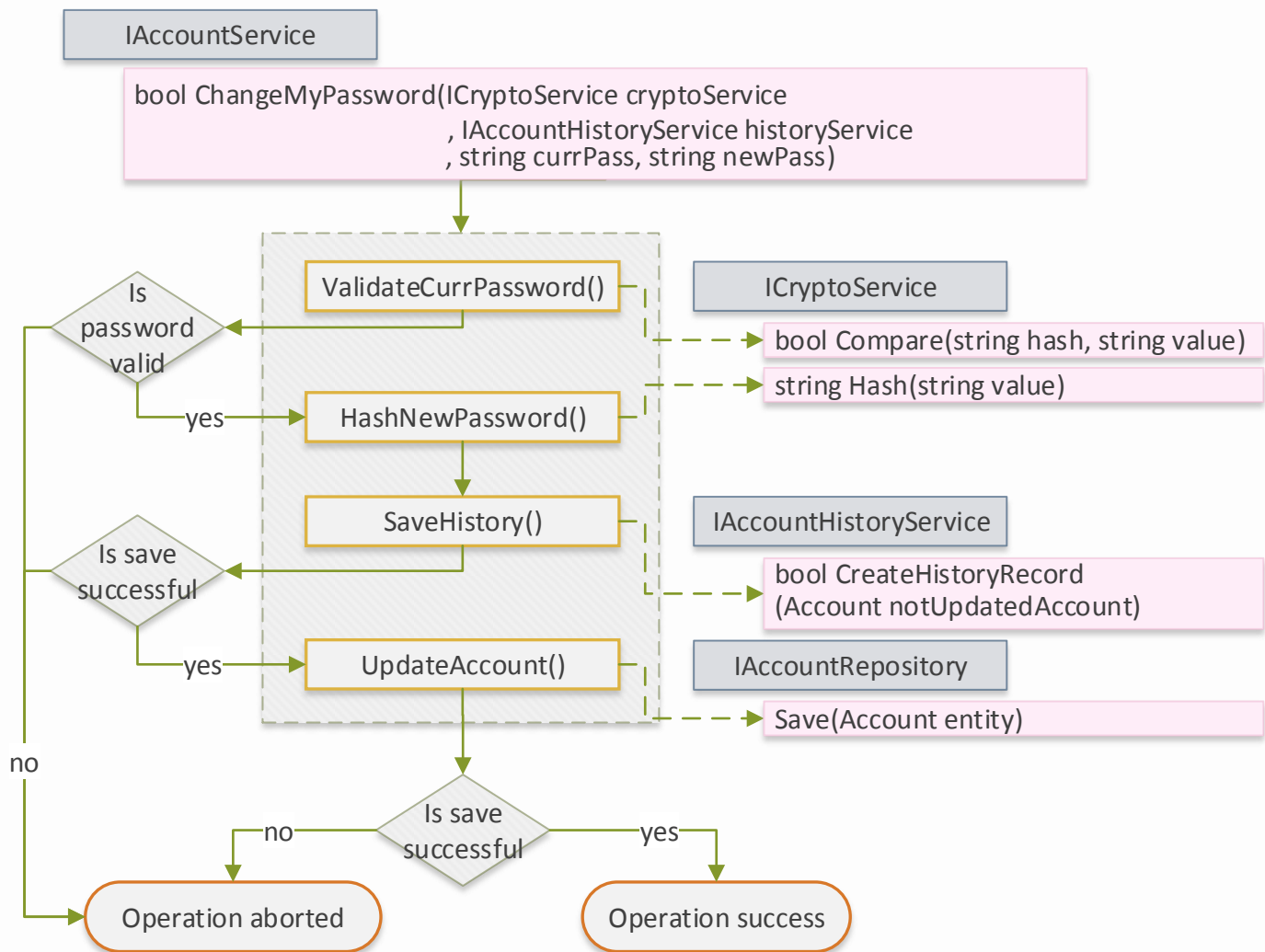
## Примерен сървиз



## Примерен интерфейс на сървиз



## Примерна операция на сървиз, която използва функционалност от други сървизи



Показаната операция е разделена на 4 основни функции, които използват 2 сървиза.

Чрез използването на функционалност от други сървизи се държи кода на AccountService-а конкретен, имплементиращ само функционалност около Account ентитето.

Реферирането на други сървизи позволява да се мокнат операциите свързани с тях в юнит тестовите на главната операция → по този начин кода на юнит тестовите не се натоварва с излишни задачи.

Пример: ICryptoService – изпълнява задачи в сферата на криптирането. Чрез написване на юнит тестове за този сървиз отпада нуждата в тестовите на IAccountService-а да се проверяват върнатите резултати от ICryptoService.

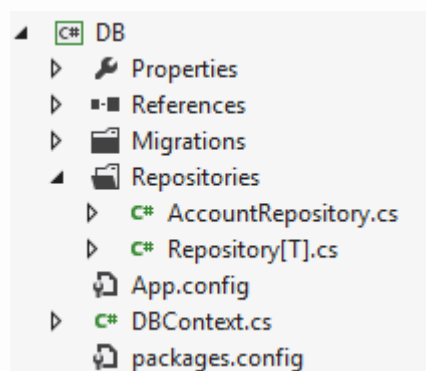
### 3. DB: Операции с база от данни

Сдържа логика за работа с база от данни, която бива достъпвана чрез имплементиране на репозитори интерфейсите в BL.

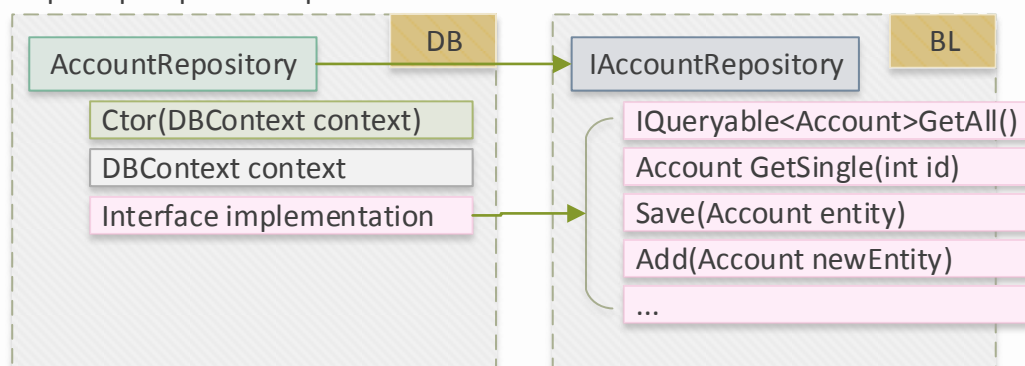
В кода за операции с базата се използва Entity Framework Code First за MSSQL, като може да бъде подменен с всяка друга технология и база от данни (ADO.NET, MongoDB, MySQL).

Отделянето в отделен проект е направено, за да може лесно да се добавят и премахват връзки към бази от данни. Удобно е когато може да се наложи да се подменя използваната база данни или когато част от информацията се намира в друга база от данни.

Стандартно случаят не е такъв (или не го позволява: Entity Framework Model First/Database First) и затова може логиката да се намира в BL\Repositories и BL\Entities.



Пример за репозитори





## Разширения

Показаната структура до момента се тества лесно и разделя кода на логически групи, които могат да бъдат подменяни и разширени без това да указва външно влияние.

Независимо от това, с разрастването на проекта работата с кода и поддръжката му е възможно да се усложнят поради:

- Струпване на код (валидно за съвизите и операциите в тях);

- Повтаряемост на код (еднакви дефиниции на методи в съвизите/репозиторията, дубликация на вмъкваните съвизи в контролерите и методите им);

- Множество входни параметри (в конструкторите на контролерите и съвизите, методите на съвизите).

Ще бъдат показани няколко структурни разширения, чрез които могат да бъдат избегнати част от тези ситуации.

### 4. Бърз/Лесен достъп до всеки съвиз

С разрастването на проекта се увеличават зависимостите между съвизите. За да се достъпва сервис от друг, той се инжектира или подава като параметър на операцията, за която е необходим.

И двата случая водят до неприятно натрупване на референции към други сервиси. Това може да бъде избегнато чрез създаване на обект, който съдържа референции към съвизите.

---

**BL.** Създаден е съвиз **IServiceFactory**, който ще връща инстанции на съвизи, с метод **GetService<T>**.

Имплементацията му е оставена на главния проект (Web), защото **GetService<T>** ще трябва да работи с **DependencyResolver**-а на MVC (в случай, че главният проект е от друг тип: с DI контейнера).

Създаден е **ServicesHolder**, който съдържа референции към интерфейсите на всички съвизи, представен от интерфейс **IServicesHolder**. В конструктора си приема инстанция на **IServiceFactory**, като { get; }-а на всяка референция към съвиз връща **IServiceFactory.GetService<IServiceInterface>**.

По този начин съвиз инстанция се създава само когато е необходима \* (осъществява се Lazy Loading).

Създаден е и абстрактен клас **ServicesContainer**, който съдържа пропърти от тип **IServicesHolder**, маркирано с **DIInjectAttribute**. Съвизите се достъпват през това поле.

**DIInjectAttribute** е празен атрибут, който се използва за маркиране на полета, които трябва да се инжектират. Това е необходимо, защото BL не работи с DI контейнер. DI контейнера в Web е настроен да инжектива полетата маркирани с него.

Съвизите получават достъп помежду си чрез наследяването на **ServicesContainer**. При добавянето на нов съвиз той трябва да бъде вкаран като пропърти в **IServicesHolder**, за да имат останалите съвизи достъп до него.

\*Lazy Loading-а е необходим, за да не се създават инстанции на всички съвизи при всеки Request, като реално малка част от тях ще бъдат използвани. Намалява се броя на създаваните обекти и времето за зареждане на страниците.

---

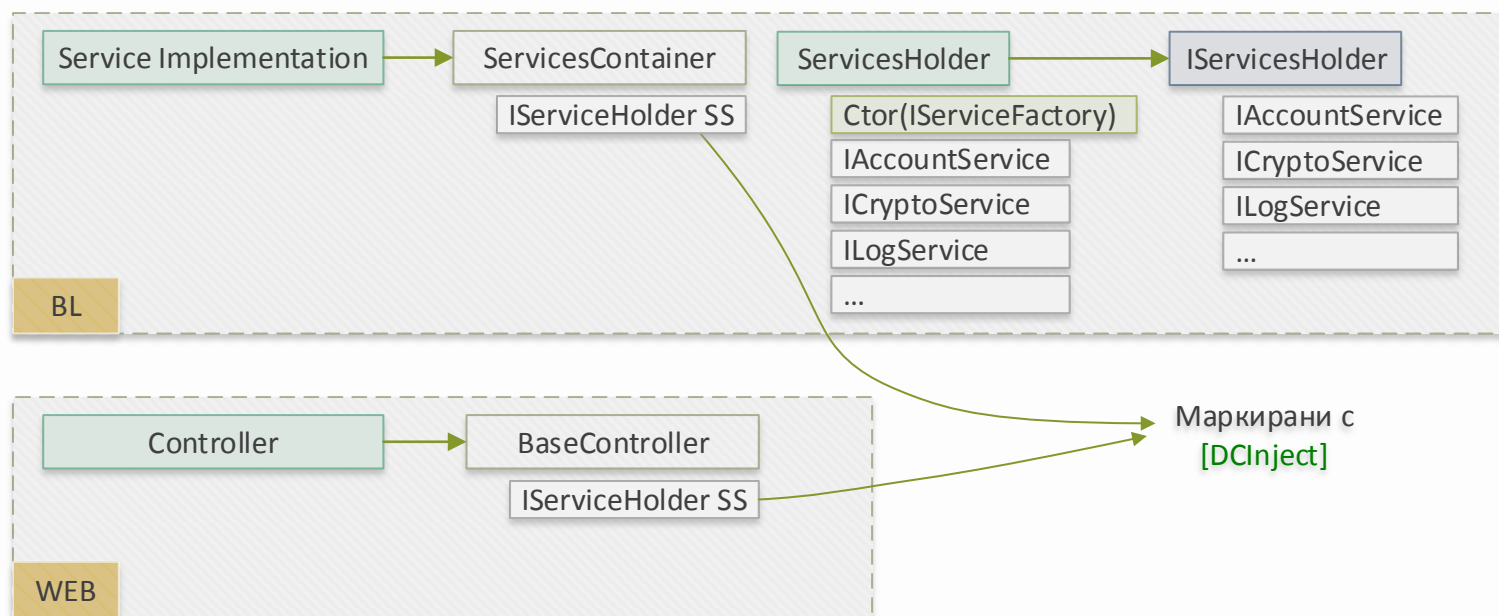
**Web.** При контролерите се реализира по подобен начин – пропърти от тип **IServicesHolder** маркирано с **DIInjectAttribute** се вкарва в **BaseController**. Всички контролери, които искат да има достъп до съвизите го наследяват.

Целта е да се избегне писането на код от този тип:

```
public AccountService(IUsersRepository usersRepository
, IAuthenticationService authService, ICryptoService cryptoService
, IConfigurationService configService, IEmailService emailService
, IAccountHistoryService accountHistoryService
, ILoggingService loggingService)
{
public class AccountService : IAccountsService
{
    [DCInject]
    public IAuthenticateService AuthenticateService { get; set; }
    [DCInject]
    public ICryptoService CryptoService { get; set; }
    [DCInject]
    public ILogService LogService { get; set; }

    public IOperationStatus ImportToLocation(ILocationsService locationsService
, IPartService partsService, ICsvService csvService, int locationId
, string importFilePath, bool skipNotFoundParts)
{
}
```

И вместо това да се осигори лесен достъп до тях чрез наследяването на сървиз контейнера:



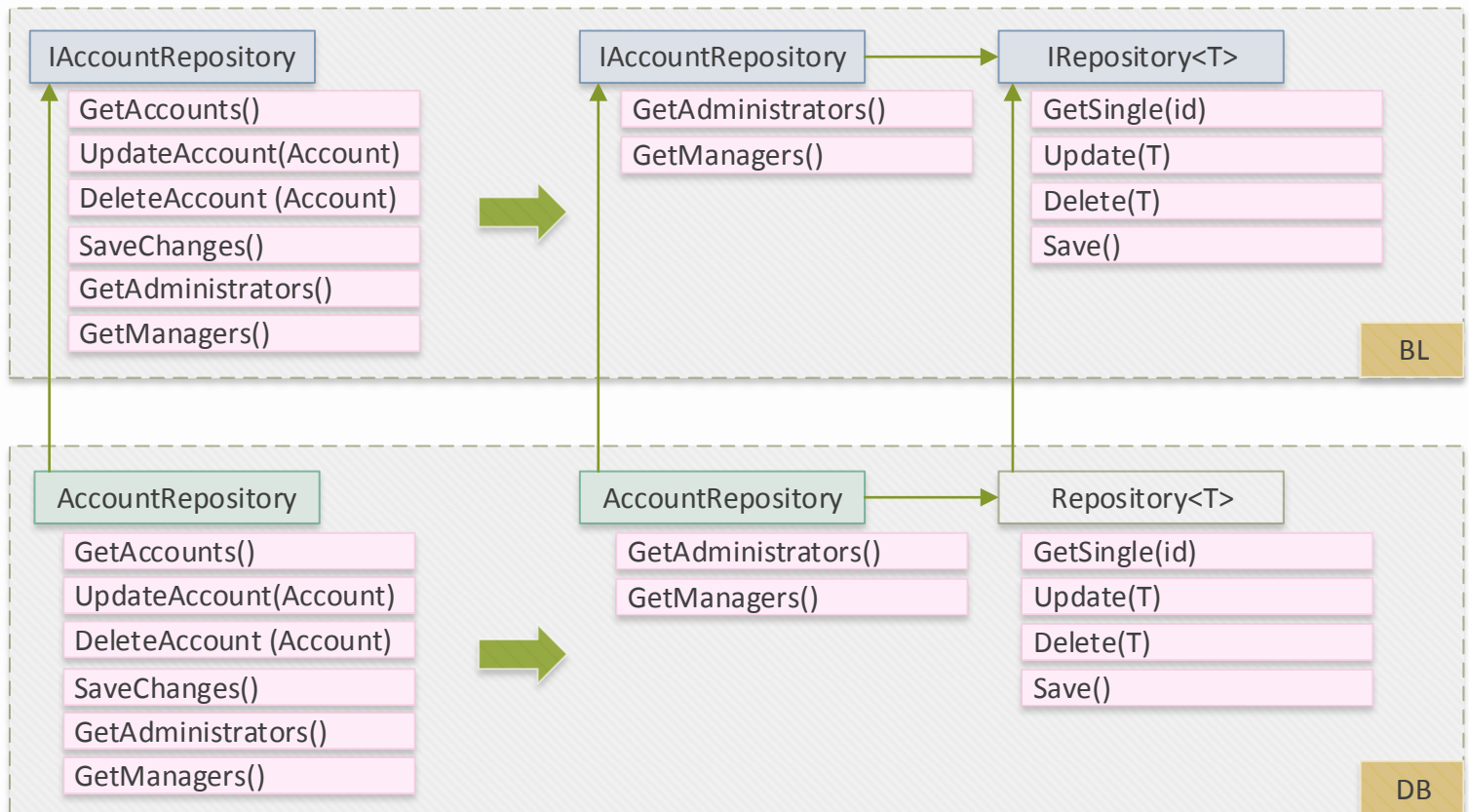
## 5. Унифициране на общи операции в репозиторията, BL: IRepository, DB: Repository

Стандартно част от методите в репозиторията могат да бъдат унифицирани, така че да важат за всички.

Пример: **GetAll**, **GetSingle**, **Update**, **Save**, **Add**. За да се избегне тяхното дефиниране във всяко репозитории те могат да бъдат изместени в отделен интерфейс, който ще се използва като тяхна база (**IRepository<T>**)

Наличието на IRepository намалява дефинирането на методи в интерфейсите, но не и тяхната имплементация в DB. За тази цел е удобно да се създаде и базов абстрактен клас имплементиращ методите на IRepository (**Repository<T>**).

Пример за унифициране на общите операции между репозиторията



## 6. Добавяне на базова история за ентитета, BL: IEntity, EntityBase, EntityServiceBase<T,R>

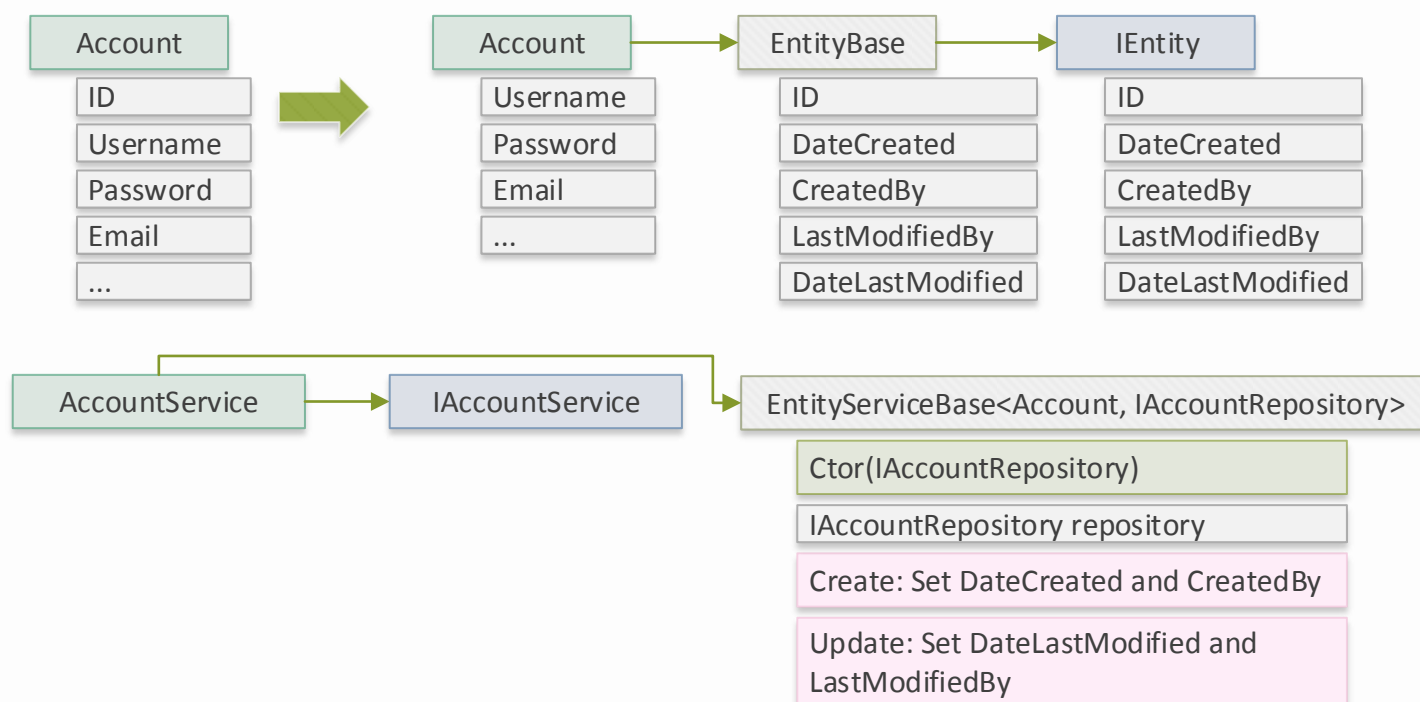
Добре е да има поне базова проследяемост на действията в базата от данни (кога и от кого даден ред/запис е създаден или променен). Това може да се постигне когато entity обектите бъдат разширени с унифицирани полета, които съдържат тази информация. За да могат да се обновяват по унифициран начин е нужно да са дефинирани в интерфейс.

Създаден е **IEntity**, в който са дефинирани **DateCreated/DateLastModified/CreatedBy/LastModifiedBy**. За да не се предефинират полетата във всяко ентити е създаден и абстрактен клас **EntityBase**, който дефинира полетата на IEntity. От своя страна EntityBase се наследява от всяко ентити.

Специфика на тези полета, е че трябва да се попълват при записване на ново ентити и при неговата промяна, тоест при Create/Update операциите на репозиторието. Биха могли да бъдат обновявани в репозиторието, но тъй като това е част от бизнес логиката, е по-добре това да е извън репозиторието. Единствен друг вариант са сървизите. За да се избегне повтаряемостта на кода, който ги обновява в сървизите, е най-добре да бъде изместен в отделен абстрактен клас, който надгражда методите на репозиторието (няма нужда от интерфейс, защото CRUD методите на репозиторието са за използване само вътрешно от сървиза).

Създаден е **EntityServiceBase<T, R>** работещ с обекти наследяващи IEntity и техните репозитори, като в него са разширени Create/Update методите на репозиторието, така че при тяхното извикване те да попълват полетата за проследяемост. От тук нататък трябва да се използват методите за обновяване на EntityServiceBase, за да има проследяемост на действията.

### Преработка на ентитета и сървизите за вкарване на проследяемост



7. Използване на универсални методи за достъп на информация, която трябва да е предварително филтрирана на база актуалния контекст, BL: IEntityService, EntityServiceBase<T,R>

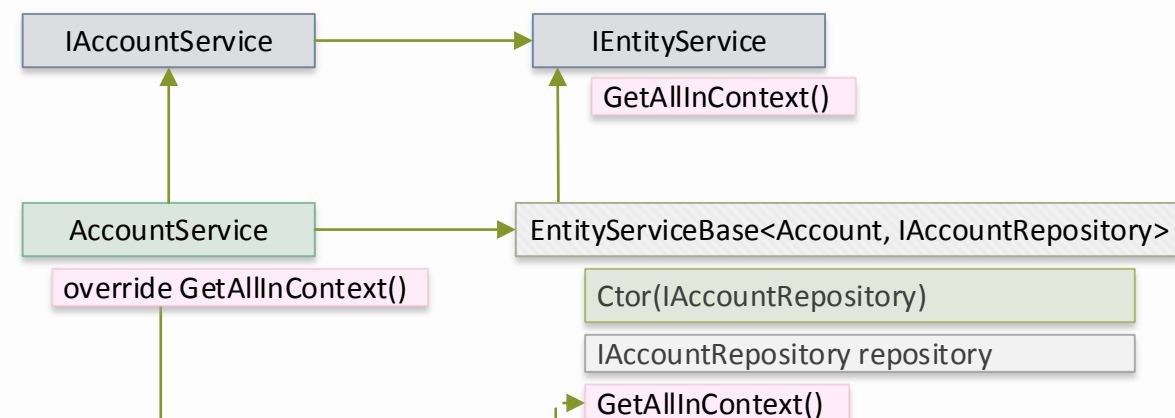
Обикновено проектите трябва да ограничават достъпа до информация в определени ситуации.

Пример: Сайт, в който потребителите са разделени на групи с права. В него показаната информация трябва да отговаря на правата на потребителя.

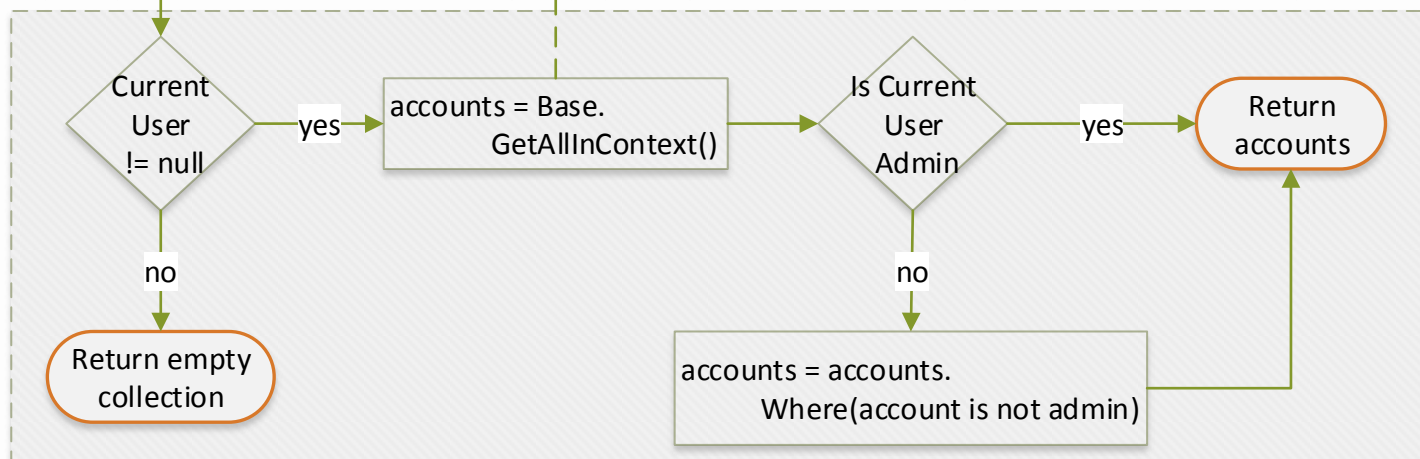
Филтрация обикновено е необходима на повече от 1 място (в сървизите, които работят с ентитета), затова е добре се абстрактне в метод с унифицирано име. Най-лесно е да се реализира като **GetAll()** с добавена филтрация според ситуацията (при необходимост). Метода трябва да бъде използван като база от останалите заявки. За да не се налага дефинирането му на няколко места, и за да има главния проект достъп до него – трябва да е дефиниран в интерфейс.

Създаден е **IEntityService** интерфейс, в който са дефинирани общи методи за всички сървизи, работещи с ентити репозитория. Трябва бъде наследяван от интерфейсите на ентити сървизите. Филтрацията е дефинирана в **GetAllInContext()**.

Създаден е и абстрактен **EntityServiceBase<T,R>** клас, имплементиращ **IEntityService** и наследяван от имплементациите на ентити сървизите (T: типа на ентитета, R: типа на репозитория за ентитета). В него **GetAllInContext()** е имплементиран като **Virtual**, за да се оверрайдва от всеки сервиз, който трябва да филтрира информация. Така всеки ентити сървиз получава **GetAllInContext()**, който е с добавена логика само на необходимите места.



Пример за **GetAllInContext()**, които филтрира потребителите на база ролята на актуалния потребител



## 8. Изместване на операциите в сървизите в отделни класове, BL: Services/Executors

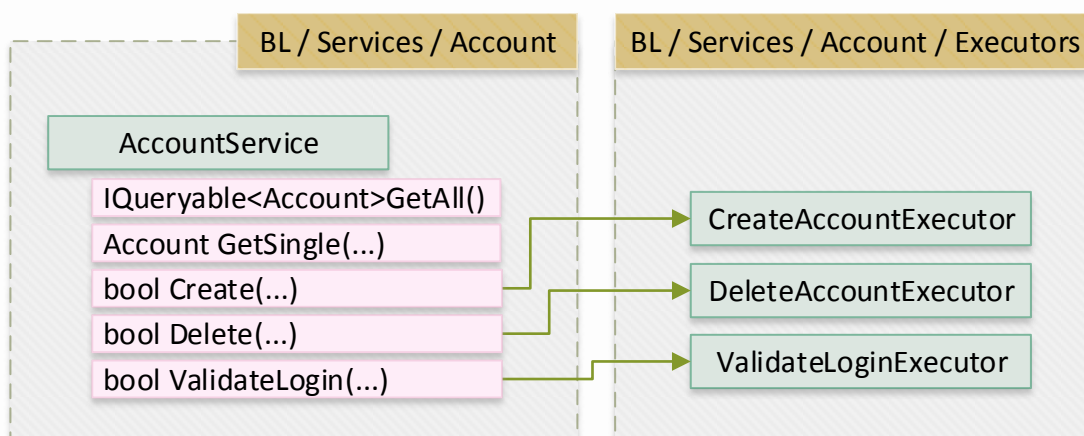
Често една от причините за затруднена работа/поддръжка на код е голямото му струпване на едно място (метод или клас). В показаната структура това най-вероятно ще се случи в сървизите. Ситуацията може да бъде избегната чрез изнасянето на операциите с повече код в отделни класове. В примера са наименувани Executors (намиращи се в поддиректория на сървиза).

Предимства:

- Сървизите се олекотяват от към редове код;
- Имплементациите на операциите могат да бъдат открити по файловата структура на проекта;
- Улеснява поддръжката и промяната на операциите;
- Интуитивно предразполагане към раздробяване на операциите на малки функции.

Executor-ите са част от имплементацията на сървиз, поради което те го използват директно (без интерфейси като посредници).

Изместване на операции с повече код в отделни класове



## Интересни места в кода

- **Първо включване на проекта:** Приложението използва Entity Framework Code First подход за работа с бази от данни. Базата ще бъде създадена при първо включване. За целта трябва да се настройт връзките към базата от данни в `Web\web.config` (DbContext и ElmahConnectionString трябва да сочат към една и съща база) (<connectionStrings> секцията)
- **Настройки на DI:**  
Всички настройки на `Ninject` се намират в `Web\App_Start\NinjectWebCommon.cs`.  
Необходими пакети: `Ninject.MVC3`.  
Указанията, на кой интерфейс каква инстанция да върне се намират в `RegisterServices()`.
  - `InRequestScope()` указва, че по време на рекуест, сървиз ще бъде създаден един път и инжектиран където е необходимо. Това е важно от гледна точка, че информацията се мени от рекуест на рекуест.
  - При добавяне на нов интерфейс за сървиз или репозитори ще трябва да се обнови `RegisterServices()`.`CustomPropertyInjectionHeuristic` съдържа custom логика, която отговаря дали поле (на вмъкван обект), трябва да се инжектира. Отговора е положителен ако е маркирано с `DIInject`.
- **Сървизи дефинирани в BL и имплементирани в Web:**
  - `BL\Services\Common\IConfigurationService.cs` -> `Web\Common\Services\WebConfigurationService.cs`, чрез него се вземат стойности на конфигурируеми настройки намиращи се в `Web.config`
  - `BL\Services\Security\IAuthenticateService.cs` -> `Web\Common\Security\WebAuthenticationService.cs`, методи за логване/разлогване и поле, което връща ID-то на актуалния потребител. ID-то на актуалния потребител се използва за да се изтегли информация за него от базата, която може да се използва всички останали сървизи (зареждането става в `AccountService.cs`, предоставя се за използване от `ServiceContainer.cs`).
- **Authorization в Web.** Намира се в `Web\Common\Security\WebAuthorizeAttribute.cs`. Състои се от два класа, `WebAuthorizeAttribute`, чрез които се маркират контролерите и екшъните, които са със ограничен достъп и `WebAuthorizeFilter`, в който се намира логиката. `NinjectWebCommon.cs` е настроен да вмъква `WebAuthorizeFilter` където се срещне `WebAuthorizeAttribute`. Това е необходимо, за да може да се инжектират необходимите сървиз инстанции в `WebAuthorizeFilter`.
- **Филтрация спрямо актуалният потребител** в сървизите. Осъществява се чрез оверрайждане на `GetAllInContext()` в сървизите, като се връща информация спрямо потребителя. Пример `AccountService.GetAllInContext()`. Всеки ентити сървиз има достъп до този метод и е препоръчително да се оверрайдва, когато трябва да се ограничи достъпа до информация.
- **Разширяване на репозитори операциите.** В `EntityServiceBase.cs` `Add()` и `Update()` попълват `DateCreated`, `CreatedBy`, `LastModified`, `LastModifiedBy` полетата на всички ентитета, чрез което се вкарва базова история. `Save()` пише в лога ако има грешка при запис и връща `false` за неуспешна операция, по този начин не стига грешката до потребителя, но се запазва за преглеждане.
- **Executors.** `Bl\Services\Accounts\Executors\CreateAccountExecutor.cs` е пример за функционалност с повече код, която е изнесена в отделен файл с цел: да не се натоварва сървиза; да добави известно капсулиране на данни; да е по-лесна за поддръжка.

- **Статистики.** В Web е инсталиран [Glimpse](#), който показва различни статистики при зареждане на страниците, включително времето за зареждане на отделните компоненти. Добавен е и екстенжън, който показва времената за изпълнение на заявките към базата. Включва се при достъп на (localhost url)/glimpse.axd, след което отдолу на страниците се показва туулбар с различни данни, при натискане върху логото той се разширява с подробна информация.  
Необходими пакети: Glimpse.Mv4, Glimpse.EF6
- **Логване.** Използва се [Elmah](#), която е инсталирана в Web и BL. В Web автоматично логва всички неприхванати грешки, които могат да се видят на (localhost url)/elmah. В BL се използва като имплементация на [BL\Services\Log\ILogService.cs](#).  
Необходими пакети: Elmah.MVC