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

Georgi Kolev

Contents:
- MVC solution structure based on loose coupling
- Inversion of Control pattern with Dependency Injection for components management
- Service – Repository pattern as Model from MVC
- Approaches for ease of extensibility and code maintenance

The purpose of this example is to show you structure, which is: scalable, extendable, has reusable components, is easy to test and maintain.

Note: The idea is shown via schemes, the code can be seen in the attached project.

First project start

A little bit for the used design practices:

Repository pattern – Separates the business logic from database operations. This way the business logic doesn't know and care how database information is retrieved and stored.
http://msdn.microsoft.com/en-us/library/ff649690.aspx

Service layer pattern – Business logic which is independent and hides how it works.
http://programmers.stackexchange.com/questions/162399/how-essential-is-it-to-make-a-service-layer

Dependency Injection (DI) – Injection of object's dependencies instead of creating them inside him. The project uses Ninject library for DI.
http://stackoverflow.com/questions/130794/what-is-dependency-injection

Loose coupling – Components interact through interfaces. This allows multiple implementations and functionality can be substituted without affecting the dependency.
http://en.wikipedia.org/wiki/Loose_coupling

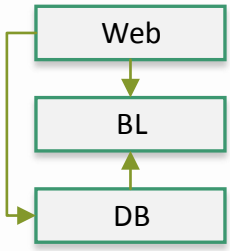
The following description assumes that you are familiar with the ideas of the described practices.

Structure

Usually projects are composed of 3 main components:
- User interface – with which the user interacts with the applications;
- Business layer – contains the logical part of the application;
- Database operations – which store and retrieve application information from database.

In the attached solution they are split in separate projects:
- Web – user interface;
- BL – business logic;
- DB – database connection.

References between projects



The following graphical objects will be used in the next schemes

AccountService — Class

BaseController — Abstract / Base class

IAccountRepository — Interface

ValidateCurrPassword() — Logical part of operation

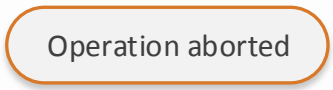
IQueryable<Account>GetAll() — Method

Ctor(IAccountRepository repository) — Constructor

IAccountRepository repository — Property

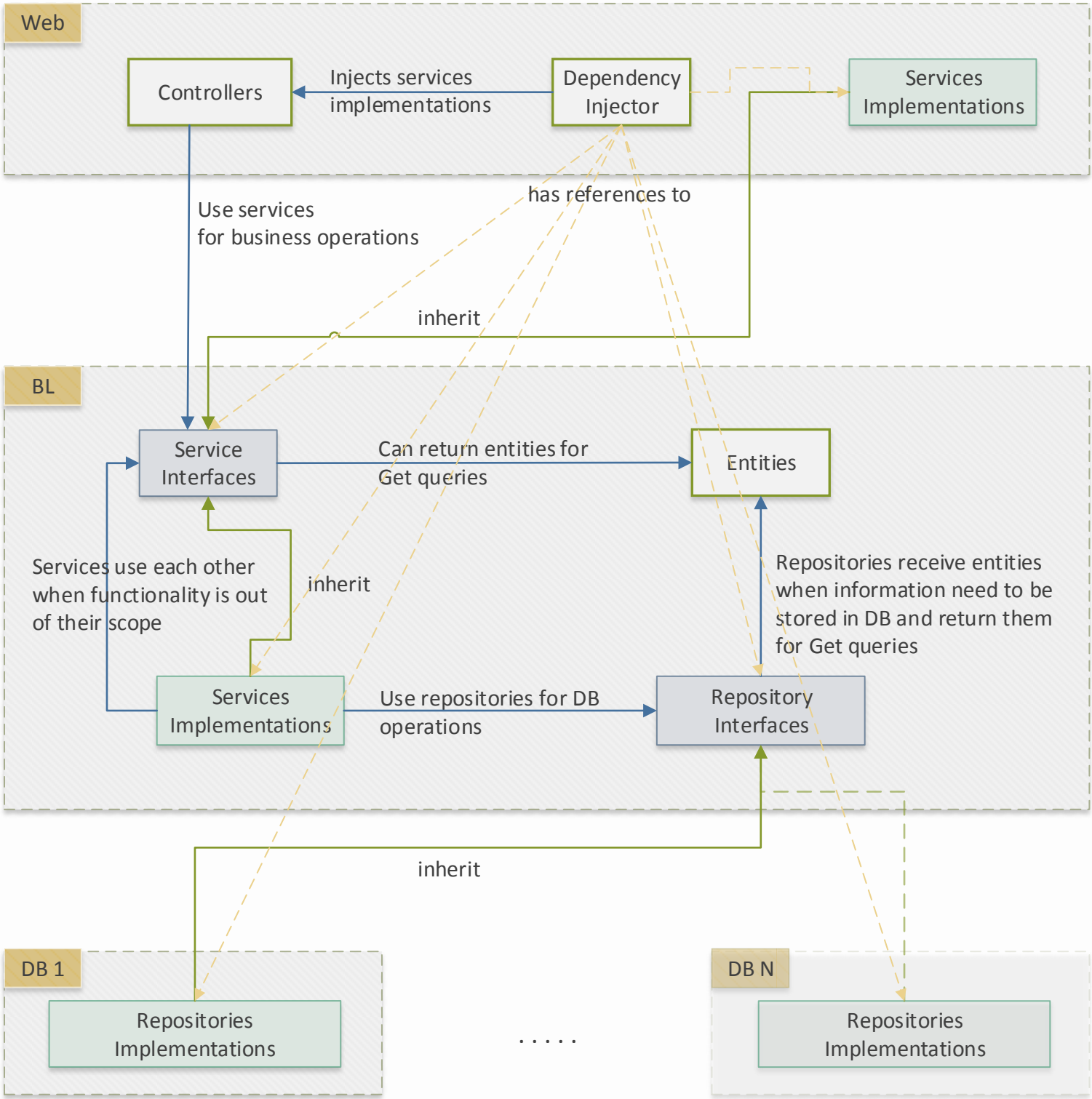
Update

logical IF block



End of block scheme

Components and their interactions. More for them on the following pages.



1. Web : User interface

User interface communicates with business logic via interfaces. The implementations of those interfaces are created and injected in the controllers and filters of Web via Dependency Injection (DI).

For this matter the DI object need to be configured, which classes are used for the interfaces, and objects lifetime (one instance per request, always same instance, every time new instance,...).

DI gives the following advances:

Centralized place, which control the used components in the application;

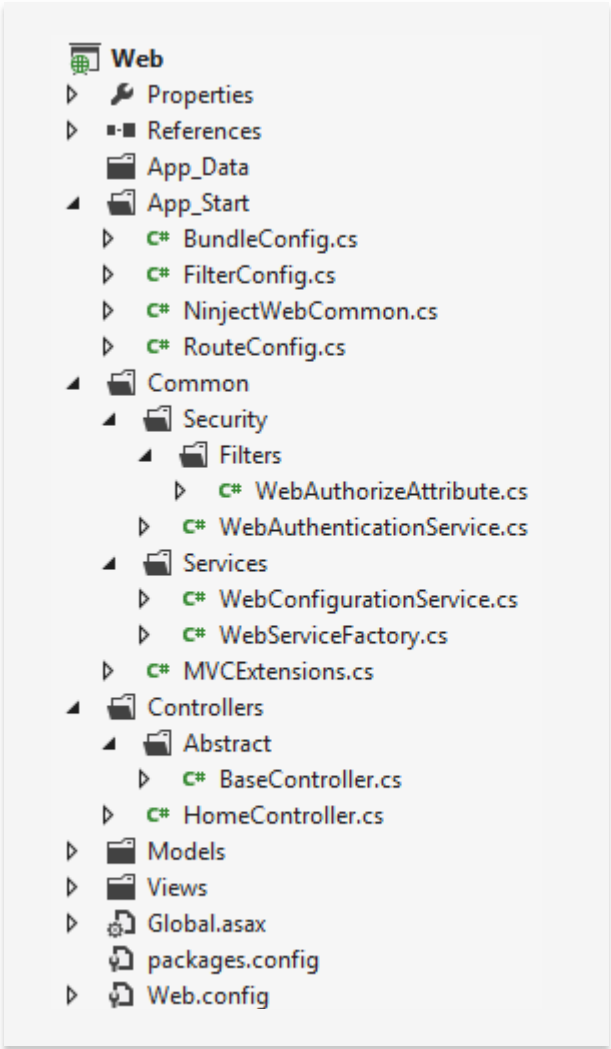
No need for creation of factory classes, which return interfaces implementations (as he is one);

Easy substitution of the injected implementations.

Important:

Avoid direct reference of the DI container outside of it's configuration.

Ninject library is used for DI in the included project. It's configuration is located in NinjectWebCommon.cs.



Example Ninject configuration

Injection of implementations for services interfaces. New implementation instance is created for each http request, which is injected thorough request's lifetime.

```
Bind<IAuthenticationService>().To<WebAuthentication
Service>().InRequestScope()
Bind<IAccountService>().To<AccountService().InReques
tScope()
```

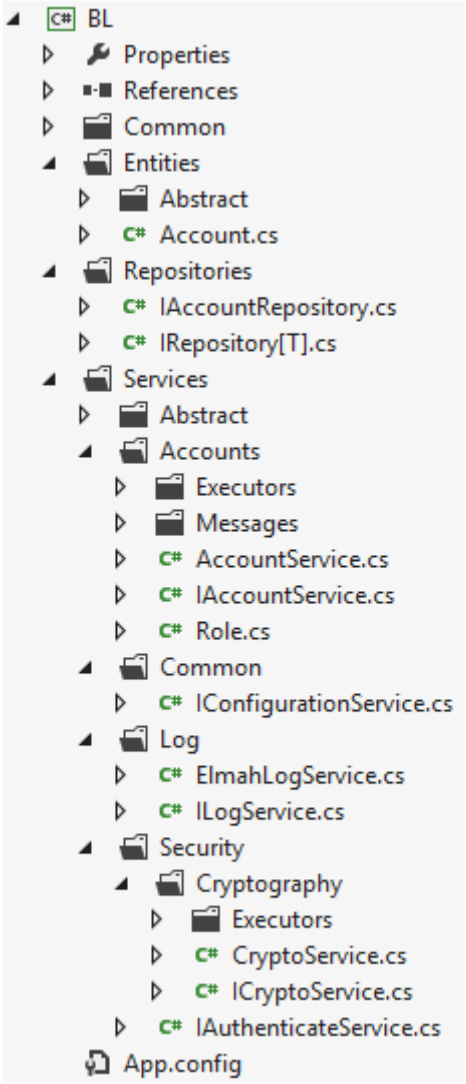
Injection of implementations for repository interfaces.

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

Injection of one instance of DBContext in each repository, per request lifetime. This realizes Unit of Work pattern.

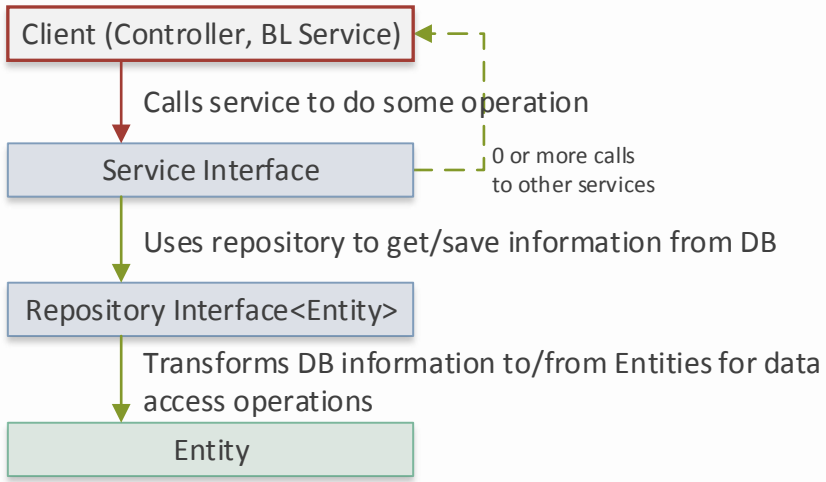
```
Bind<DbContext>().ToSelf().InRequestScope()
```

2. BL : Business logic



The business logic contains 3 main parts:
Entities – objects representing database information;
Repository interfaces – define database operations, which use entities;
Services – the business logic of the application.

Operation interactions



Entities – Objects reflecting database rows. Returned by repositories when information is retrieved, and handed to when information in database need to be saved.

Repositories – Repository interfaces, which work as middleman between database and business logic. Define Create/Read/Update/Delete (CRUD) methods, which work with Entity objects.

The interfaces purpose is to create loose coupling between database and business logic.
Also they allow substitution of database in unit tests, with in memory store (cache).
They are implemented in DB projects.

The DI container in Web need to be configured to inject implementations (located in DB projects) for repository interfaces.

Important:
Each repository works with one table
-> separation of concerns;
Repositories are internally used only by services;
It is recommended that repository accept/return only it's entity object;
If possible they should return IQueryable for Get queries;
They are used through their interfaces -> loose coupling.

Example repository interface

IAccountRepository

IQueryable<Account>GetAll()
Account GetSingle(int id)
Save(Account updatedEntity)
Add(Account newEntity)
...

Services – The core business logic of the application, represented by interfaces and their implementations. The interfaces create loose coupling between business logic and it's clients (UI).

The DI container in Web need to be configured to inject implementations for service interfaces.

Example interaction with service: Suppose the database has table for users, which can be added or updated from the UI. Since these are business operations, they have to be defined in service interface, and then implemented in class inheriting from it. The service interface is also added as parameter in the constructor of the controller, which exposes the operations to the UI, and the DI container is configured to inject the implementation.

Soon after an table for user addresses is added. This forces the operations for add/update of users to be updated to fill in address information. The operations for adding/updating users addresses are defined in new service interface, implemented by new class. The service for users will have to be updated to receive address information, and it's implementation will have to receive reference to addresses service, which will be used to execute the operations with addresses. Or in other words: the users service will use the addresses services for addresses operations.

In some cases the business logic is not aware of the implementation, because it is client (UI) specific. Then the implementation is created in the client project.

Example: IConfigurationService defines method for retrieval of configurable settings. Since the location of those settings is client specific – the implementation realization is left to him. In the included project, the implementation is located in Web as WebConfigurationService, which accesses Web.config for settings.

Services work with repositories though their interfaces, this creates loose coupling between business layer and the database. Repository is injected in service via constructor injection.

Important:

Each service is responsible for functionality in specific sphere (users / security / mails) and it is independent from the client (web / console / win forms / web services..) -> separation of concerns;

** Each service can use (refer) up to 1 repository, and 1 repository can't be used from more than one service -> separation of concerns;

Advisable to return IQueryable when possible for Get queries -> so that the interface can apply additional filtration if needed;

Used through their interfaces -> loose coupling.

**It is normal operation to read/write from couple of tables, which impose use of couple repositories. Often those repositories are injected in the service performing the operation.

Although this realization is easier, it has the following disadvantages:

1 Repository is used from more than 1 place -> table is modified from couple places;

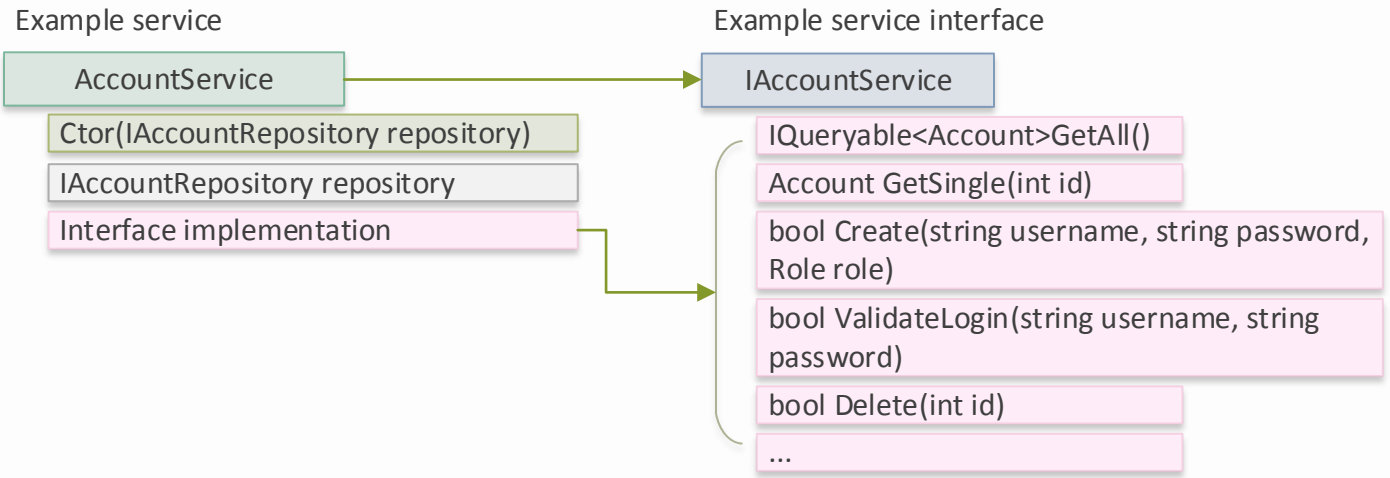
Some parts of the operation can't be mocked, so that the core can be tested;

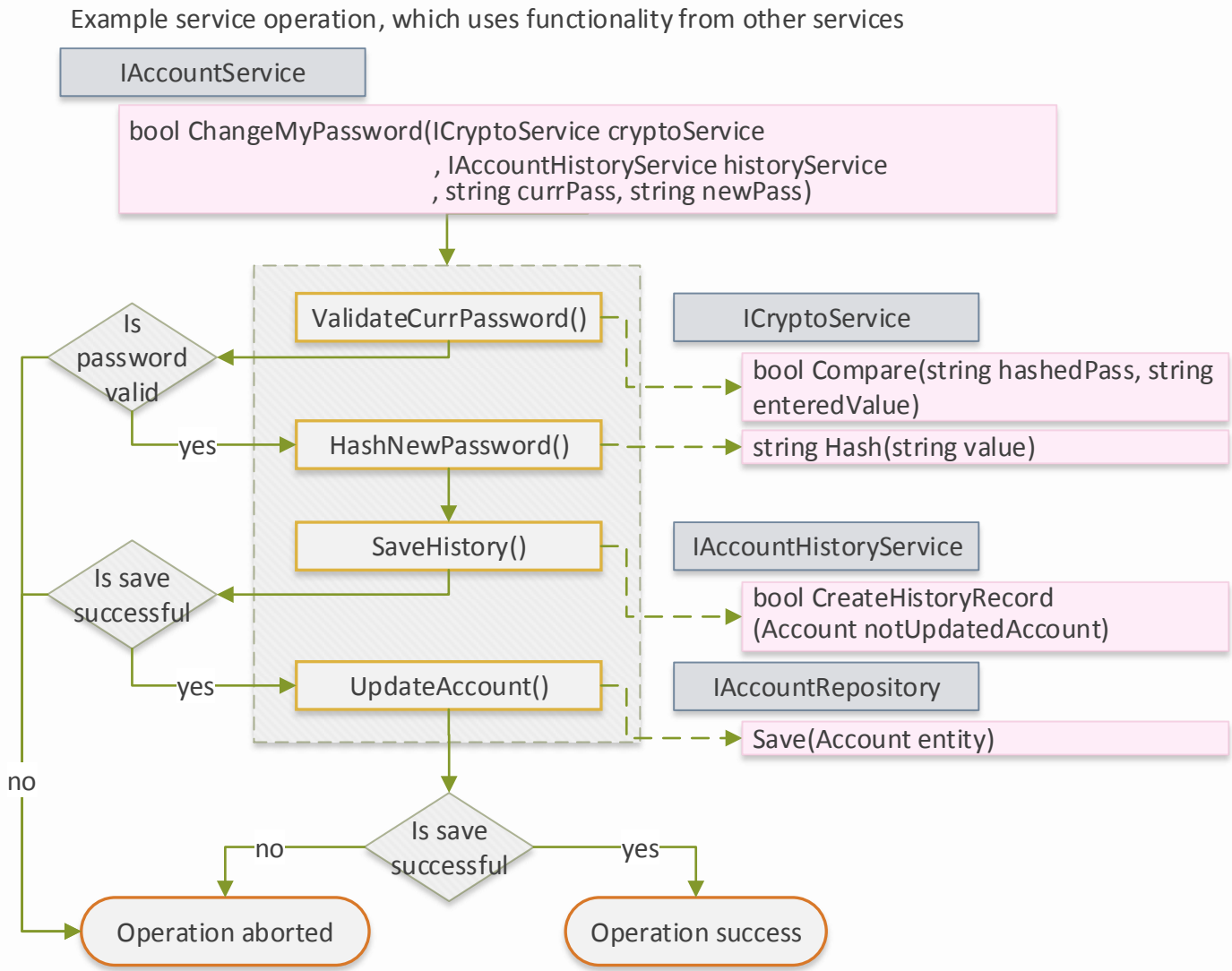
Services are filled with repositories and do not use each other;

Favors code duplication;

Increases service responsibilities.

The better option is to have services work with each other, for better separation of concerns, and so that operations parts can be tested individually. More for relations between services in point 4.





The shown operation is composed of 4 parts and uses 2 services.

By using functionality from other services, the code in AccountService stays specific for Account entity.

Sub operations executed in referenced services can be mocked up in unit tests for the main operation -> this way the unit tests code is easier to write and has less responsibilities.
Example: ICryptoService – executes cryptic operations. By having unit tests for this service, there is no need to check it's returned results in the tests for IAccountService.

3. DB: Database operations

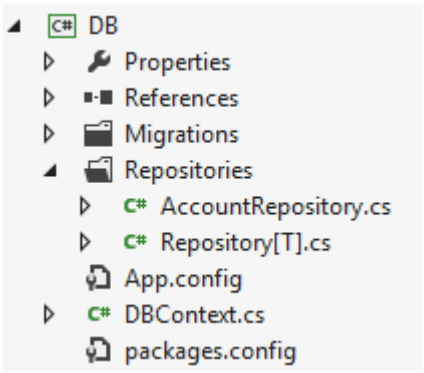
Contains logic for accessing database, which implements the repository interfaces in BL.

The included project uses Entity Framework Code First for MSSSQL, but it can be substituted with any other technology or database (ADO.NET, MongoDB, MySQL...).

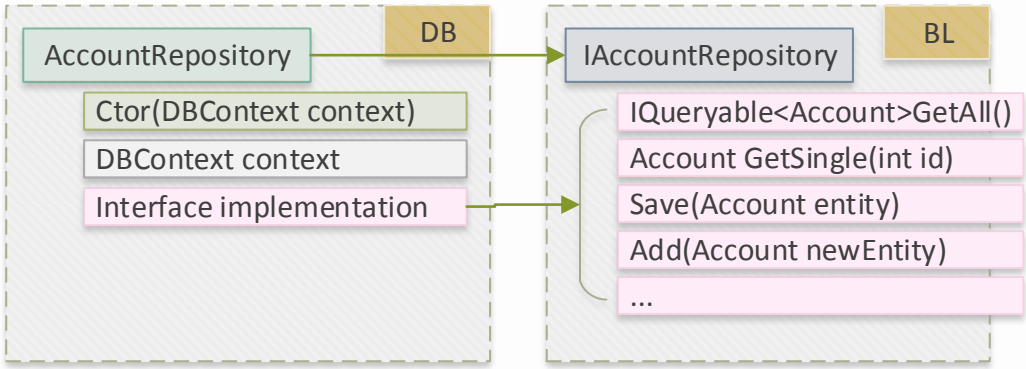
Database logic is in separate project so that database connections can be easily added or removed. This is handy when the used database might be changed or when more than one database is used.

If this is not the case (or it can't be realized: Entity Framework Model First/Database First) the logic can be put in BL\Repositories and BL\Entities

---



Example repository



Structure extensions

The shown structure is easily tested, and separates the code in logical groups, which can be replaced and extended without outside effect.

Nevertheless with project growth, his maintenance and ease of work can become harder because of:
Code concentration (valid for services and their operations);
Code duplication (repeated method definitions in services/repositories);
Many parameters (in controllers and services constructors, and services methods)

These situations can be avoided with couple structure extensions, which are described in the follow up.

4. Easy access to each service

The dependencies between services increase while the project is in development. The used service is either injected or handed as method parameter for the operation, to which is needed.
Both cases cause unpleasant pile of references to services. This can be avoided with creation of object, which holds references to all services.

BL. For this, an component which returns instances of other services is needed. It is abstracted by interface called IServiceFactory and has GetService<T> method. It's implementation is left to the UI project (Web), because GetService<T> will have to use the DI container (in MVC this will be the DependencyResolver).
The services will be accessed via object, which holds references to them. The component is presented via IServicesHolder and it's implementation (in BL). It receives IServiceFactory in his constructor and exposes each service with { get; } which returns IServiceFactory.GetService<IServiceInterface>.
This way an instance of service is created only when it is needed (lazy loading)*.
The last component is abstract class ServicesContainer, which has IServicesHolder property, marked with [DCInjectAttribute] attribute. From this point every service, which uses other services, can inherit ServiceContainer, and access them via IServicesHolder property.
DCInjectAttribute is empty attribute, used on properties for injection. It is needed, because BL does not work with DI container (nor have reference to it) and need a way to hint that property must be instantiated when object is created. The DI container in UI project (Web) must be configured to inject values on properties marked with it.

Each new service added to BL need to be referenced in IServicesHolder in order to be accessible by other services.

*Lazy loading saves creation of all services on each request, when only small part of them will be needed. This results in less created objects and decreased loading time.

Web. The realization is almost the same – property of type IServicesHolder marked with DCInjectAttribute is added to BaseController. And then it is inherited by all controllers, which need access to services.

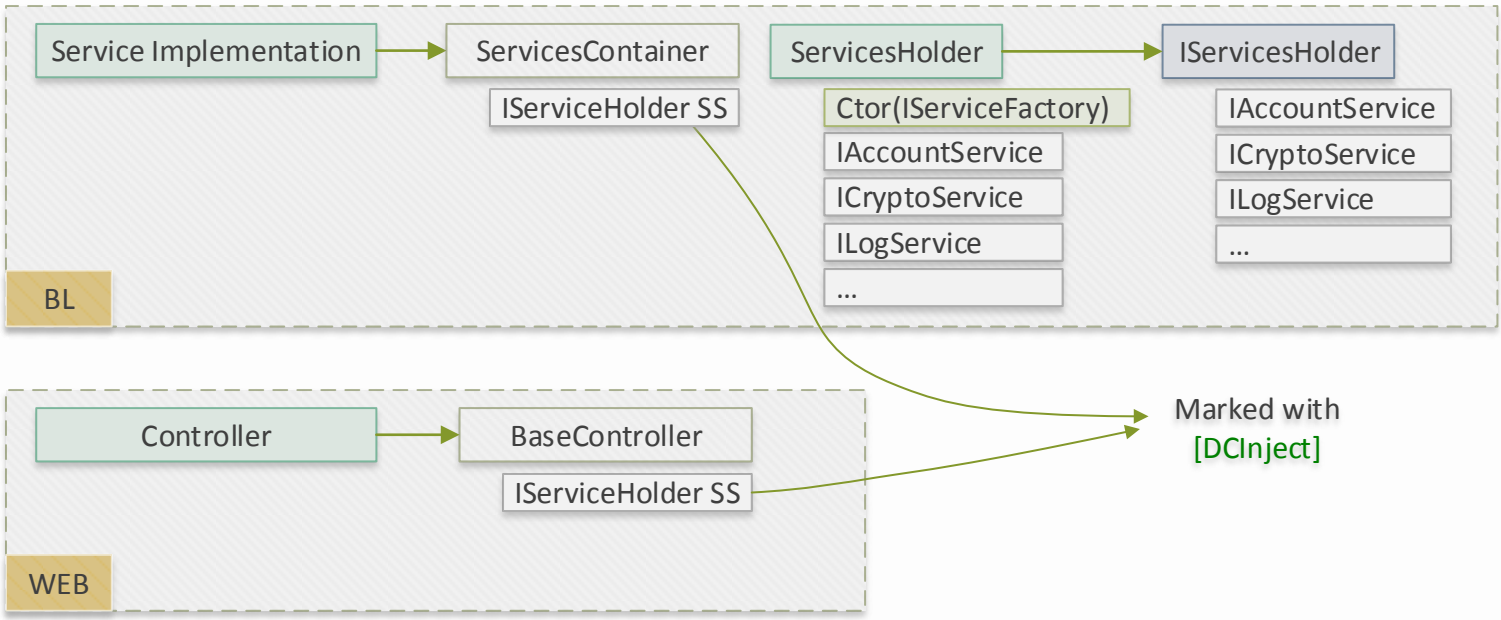
The idea is to prevent code like this:

```
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)
{
```

And instead have easy access by inheriting the service container:

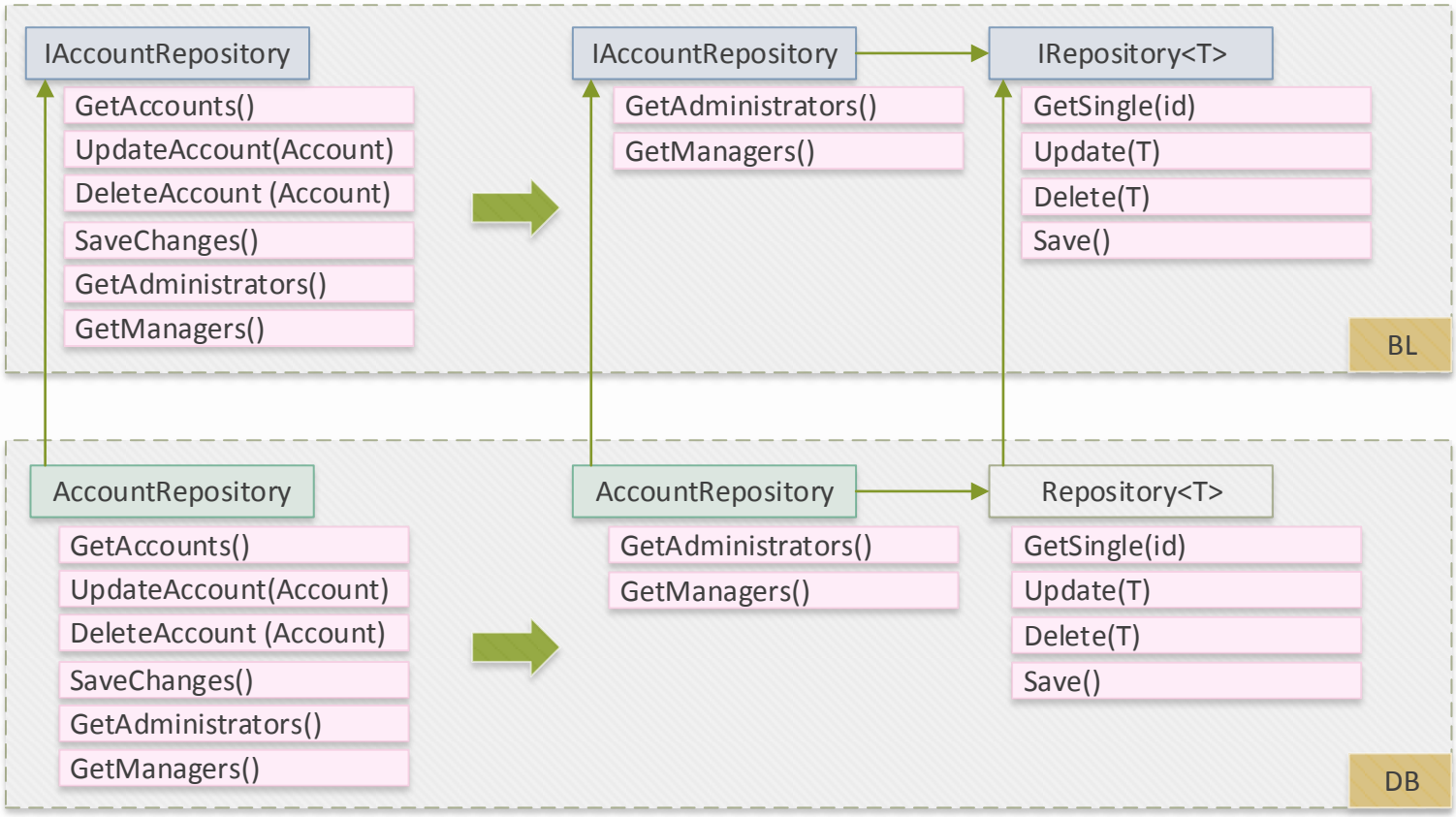


5. Universal repository operations, BL: IRepository, DB: Repository

Usually most operations in the repositories can be unified in a way that can be applied to all.

Example operations: GetAll, GetSingle, Update, Save, Add. They can be moved to separate interface (IRepository<T>), which will save their definition in each repository. To avoid writing code for those methods, the generic interface can be implemented in base abstract class (Repository<T>). From there it need to be inherited by the repositories.

Unification example



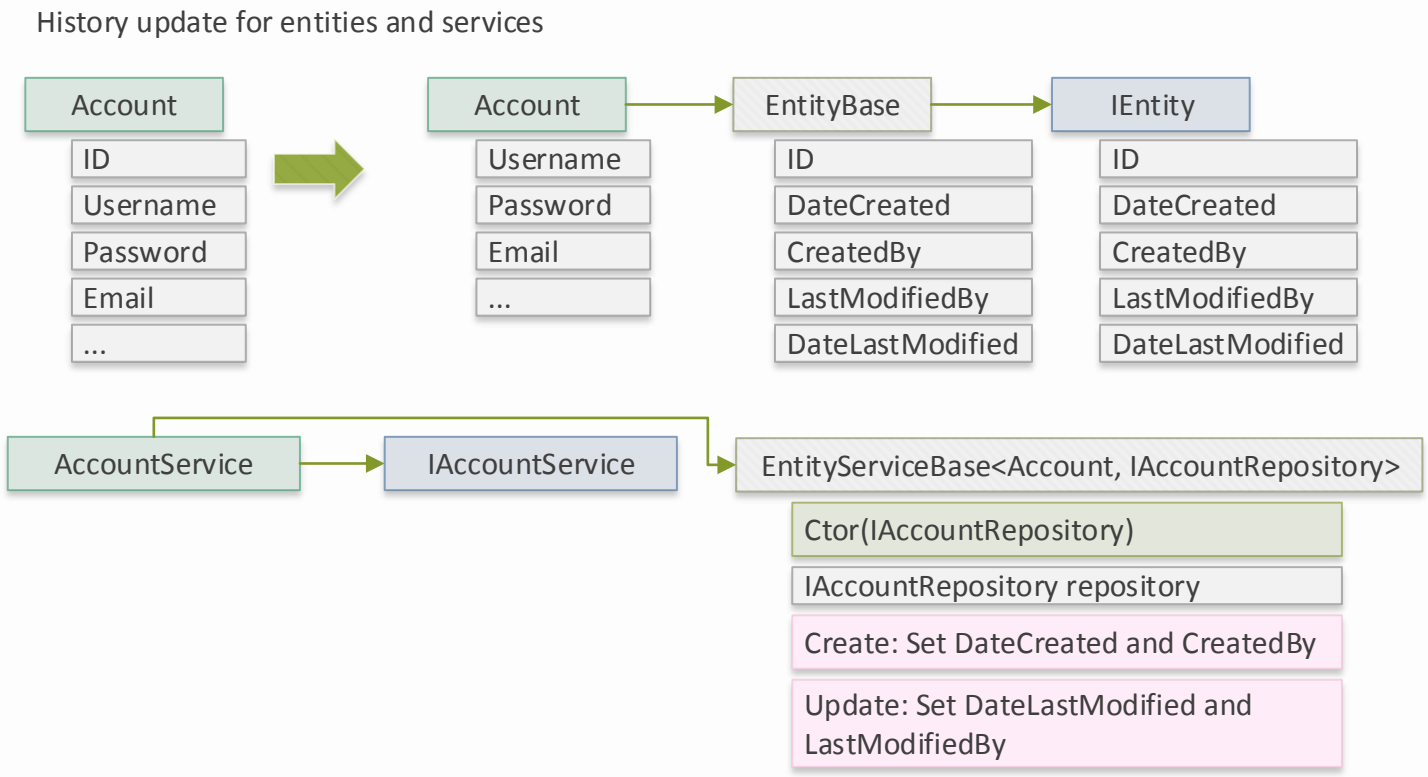
6. Basic history for entities, BL: IEntity, EntityBase, EntityServiceBase<T,R>

It is good to have some basic trace for operations in database (when and who added/updated/modified row). It can be achieved by extending entities with properties containing that information. In order to have that information updated in unified way, these properties need to be defined in interface.

This is presented with IEntity, which defines DateCreated/DateLastModified/CreatedBy/LastModifiedBy properties. They are implemented in abstract class EntityBase, which is inherited by each entity.

Those fields need to be set when entity is being added or updated (when Create/Update repository operations are used). However it should not happen inside the repositories, since it is part of the business logic. It is best to have them set up in abstract generic class, which works on top of the repository.

It is presented by EntityServiceBase<T, R>, which works with objects inheriting IEntity and their repositories. It extends Create/Update methods of the repository by filling Entity history information. From there on, the EntityServiceBase<T, R> Create/Update methods are used for entities, which require tracing.



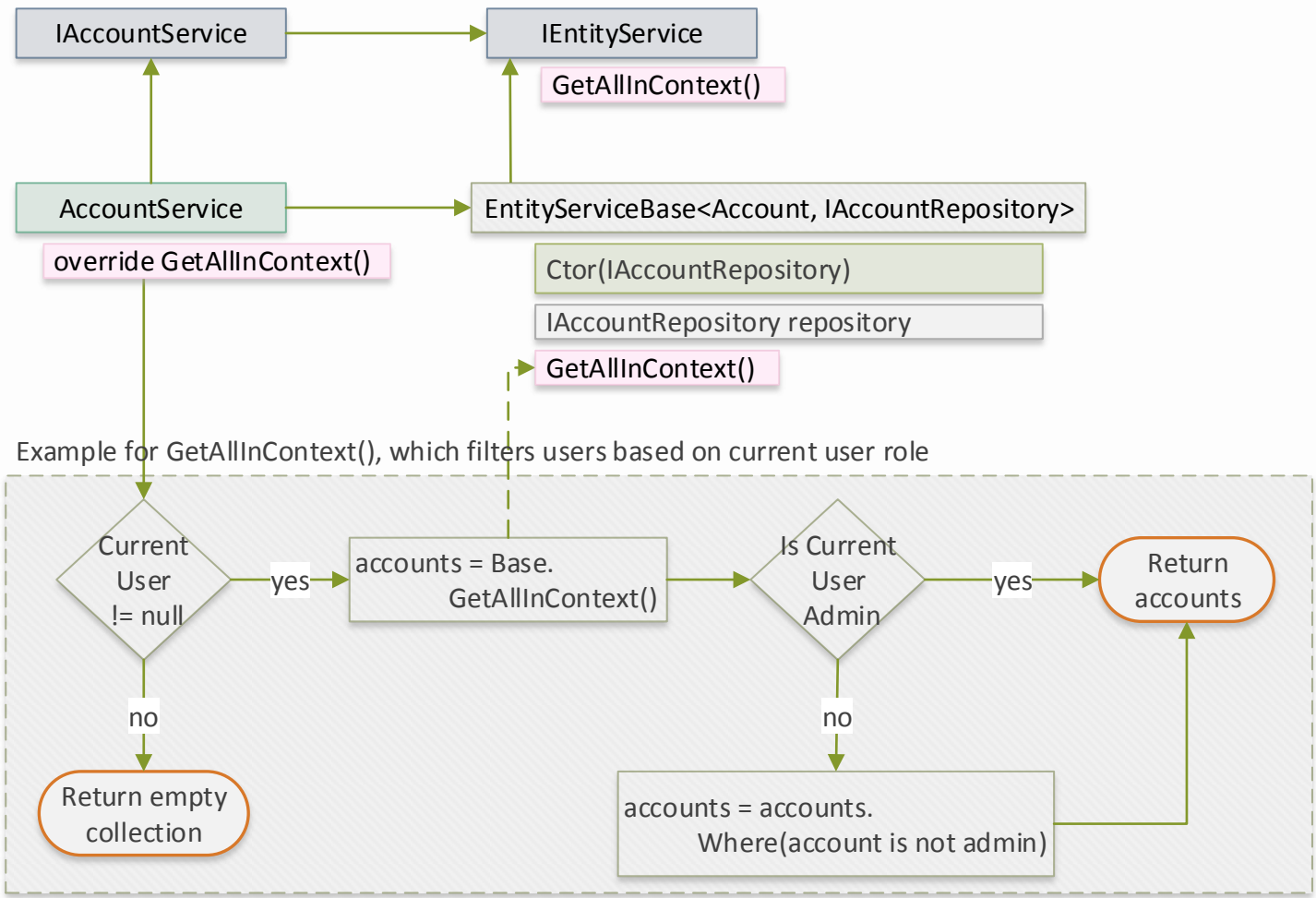
7. Use of universal methods for access of information, which need to be filtered based on the current context, BL: IEntityService, EntityServiceBase<T,R>

Usually assess to information has to be restricted in particular cases.
Example: Site, which groups users based on their roles. Shows information allowed by role.

Since filtration will be needed (most often) on more than one place, it should be abstracted in method with universal name. It should work on top of GetAll() with additional filtering based on the current situation. The method will be used as base from other queries. It need to be defined in interface so the UI project has access to it and to save it's definition in every service.

IEntityService interface is created, which defines common methods for services working with repositories. It is inherited from the interfaces of entity services. The filtration is defined in GetAllInContext() method.

EntityServiceBase<T,R> (T: entity type, R: entity repository type) abstract class is created, which implements IEntityService and is inherited by services implementations. In it GetAllInContext() is implemented as virtual, so it can be overridden in each service, which need to filter information. This way every service has GetAllInContext(), but the additional filtration is added only in the needed places.



8. Extraction of service operations in separate classes, BL: Services/Executors

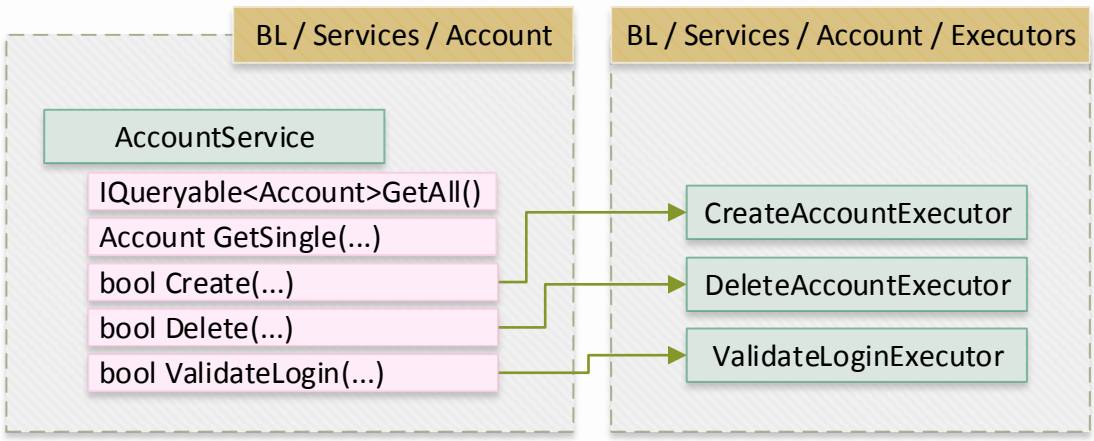
Code concentration (in file/method/class) often leads to difficult maintenance and work. In the shown structure this will happen in the services. It can be avoided by exporting the operations with more logic, in separate classes. They are called Executors in the provided project (put in service subdirectory).

Advantages:

- Less code in services – easier maintenance;
- Operations can be found from the solution explorer (file structure);
- Easier maintenance of operations;
- Favors splitting of operations on small functions.

Since executors are part of service implementation, they can use it directly (via class reference) and interfaces are not needed for them.

Exporting operations with more code in separate classes



In the applied project code

- **First start up**: The application uses MSSQL and Entity Framework Code First for database operations. The database will be created on first run, but before that the connection strings have to be configured to point to MSSQL server instance. This is done in Web\web.config (<connectionStrings> section) (DBContext and ElmahConnectionString need to point to same database)

- **Dependency Injector configuration**:
  The settings for Ninject are in Web\App_Start\NinjectWebCommon.cs.
  Package: Ninject.MVC3
  In RegisterServices() are specified the implementations returned for each interface.
  - .InRequestScope() specifies that interface implementation instance will be created only once and injected wherever is needed, during request lifetime. This is important, because information is changed between requests, and to prevent creation of new instance when that interface is needed for injection.
  - RegisterServices() have to be updated each time when new interface for service or repository is added.
  The logic inside CustomPropertyInjectionHeuristic class answers if property value (of injected object) have to be injected. It will return true is the property is marked with DCInject.

- **Services defined in BL and implemented in Web:**
  - BL\Services\Common\IConfigurationService.cs -> Web\Common\Services\WebConfigurationService.cs, it returns values of configurable settings in Web.config.
  - BL\Services\Security\IAuthenticateService.cs -> Web\Common\Security\WebAuthenticationService.cs, contains authentication methods and property, which returns the ID of the current user. That ID is used to retrieve user information from the database, which information can be used from other services (information retrieval is done in AccountService.cs and the user is accessible from ServiceContainer.cs).

- **Authorization in Web** is located in Web\Common\Security\WebAuthorizeAttribute.cs. Contains two classes: WebAuthorizeAttribute used to mark controllers and actions with restricted access; and WebAuthorizeFilter which contains the restriction logic. NinjectWebCommon.cs is configured to inject WebAuthorizeFilter where WebAuthorizeAttribute is found. This is done in order to be possible to inject the needed services in WebAuthorizeFilter.

- **Information filtration for the current user** in services. Implemented by overriding GetAllInContext() by returning information, which is available for the user. Example is AccountService.GetAllInContext(). Each entity service have access to that method, and should override it when information access is restricted.

- **Repository operations extension** in EntityServiceBase.cs:
  - Add() and Update() fill entities DateCreated, CreatedBy, LastModified, LastModifiedBy fields, this adds basic history.
  - Exceptions in Save() will be caught and logged, and the method will return false for unsuccessful operation. This way the error is not shown to the client, and is saved for inspection.

- **Executors**. Bl\Services\Accounts\Executors\CreateAccountExecutor.cs is example for functionality with more code, which is exported in separate file in order to: unload service from code; capsulate information; have easier maintenance.

- **Statistics**. Page loading and database execution times (and much more) can be seen in Web with the help of Glimpse plugin. The statistics can be turned on by navigating to (host url)/glimpse.axd, after that statistics toolbar will be shown on the bottom of the pages (the toolbar is expanded with information when logo is clicked).
  Packages: Glimpse.Mv4, Glimpse.EF6

- **Logging** is handled by Elmah, which is included in Web and BL. All exceptions in Web are logged automatically, and BL uses it as implementation of BL\Services\Log\ILogService.cs. The errors can be seen on (localhost url)/elmah.
  Packages: Elmah.MVC