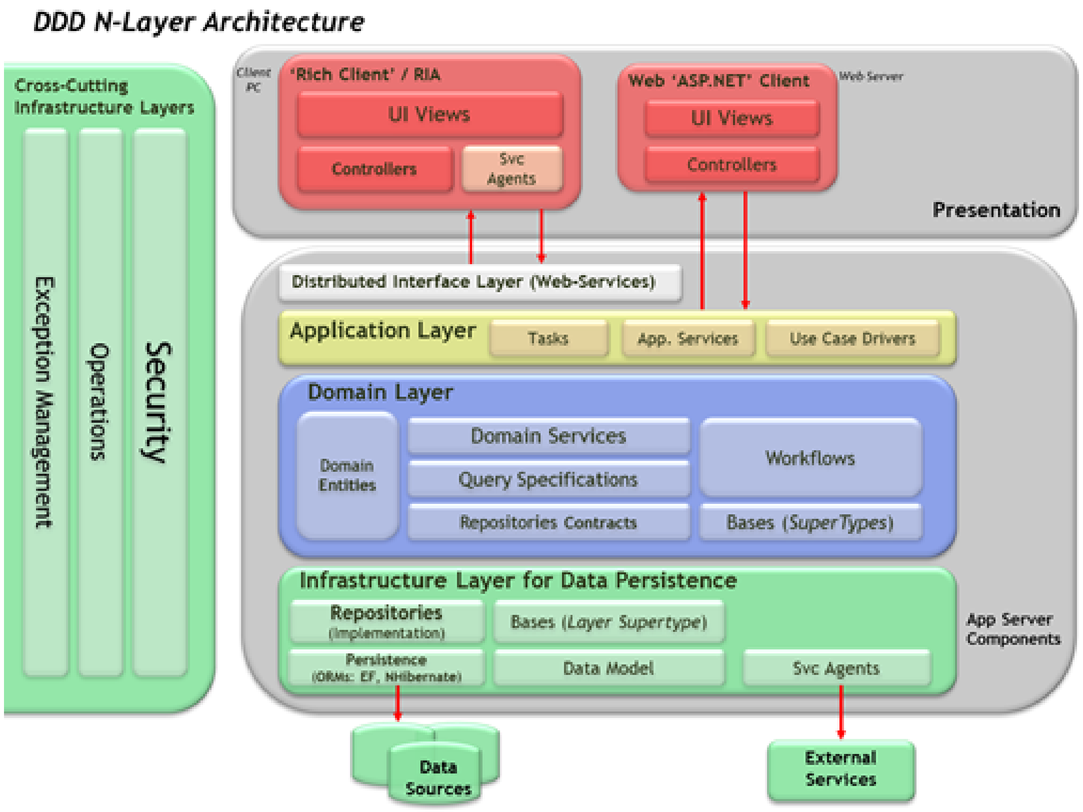
Software Architecture Applied

Robust and ready to grow

I used a combination of Frameworks and Patterns such as DDD, Ioc, TDD to create this solution. The result is a robutst, testable, scalable and ready to change and grow Solution that as a Team is much easier to work with since each layer has specific responsibilities and we can isolate the layers for Unit Tests and focus on the task we are doing instead of spend time to reproduce the environment.

DDD - Domain Driven

The Domain represents the Company Business. It contains all Entities, Specifications, Validations and all that is necessary for the business works. The Domain is agnostic to technogies and can't make a reference in any Layer, except the CrossCutting tier. This way we can share the Domain between any clients or UI or Services. If we force the Domain to make a reference on System.Web we bring unecessaries things for this layer are useless when we need to implement a desktop app for example.



Project Layers

1.Presentation

It's the UI layer where we can have Asp.Net MVC, WPF, Mobile, etc. It's responsible only for display data with rich and robust layout. They must make a reference to the Application Layer and never access direclty the Domain.

Basic flow MVC UI (Pipeline)

- The User Request a URL
- MVC Pipeline workflow starts with Router.
- Then extract the Route from URL
- Then it matches the extracted route with Mapped routes
- MVC starts the Controller Initialization with Controller Factory based on the Route
- The Action Execution Starts where the Model must be created and returned
- The Result Execution Starts and MVC analyzes the ActionResult
- If the Result is a View Result MVC starts the View Engine process (Razor or Aspx until MVC 4, Razor only after MVC 5)
- If the Resul is NOT a View, MVC just return the Result(It can be a Json result, File, RedirectResult, etc)
- The execution returns to IIS that returns the final Http to the Client/Browser/User

2.Services

Responsible for expose Services and APIs from our Application Layer. We can have here WebApi, WCF, etc. They must make a reference to the Application Layer and never access direclty the Domain. The WebApi is not finished but I included just to show how easy is to share functionalities from the Application layer. Please check the SecurityController in Folder 2.Services in the Solution

3.Application

It's the Middleware between UI and Domain. It's responsible for abstract the UI and pass information to the Domain. Also we can mantain the ViewModels and UI validations here to be shared. For example, once defined the ViewModels here they can be easly shared between MVC and WebApi. Also, instead of put all the validation rules in the Controller we give this responsability to the Application Service. So when we need to figure out which method we have to invoke according with the given parameter we transfer this responsibility to App layer and this method can be shared between UI avoid redunant code and make the maintenance easy. If I need to change the rules, I change only in my App layer, any of the UIs won't be changed.

4.Domain

The main layer. It's reponsible to represent the Business. This way we can speak the language of our Client. The Domain must be agnostic to technology (client, web, etc). And must be focus on the business rules and prepared to change and Grow. I expose the funcionalities of my Domain by Domain Services(notice that is not the same as WCF or Web Services). The ideal scenario is Isolate your Domain from other layes using a Midlleware Tier, for example on this Project is DomainDrivenDesign.Application layer.

5.Infrastructure

Contains the Data layer which is responsible for bring the data from Datasources using preferable ORM. I didn't implement the ORM layer to keep the solutin small and simple. The CrossCutting layer is prepared to be used across the Project as auxiliary.

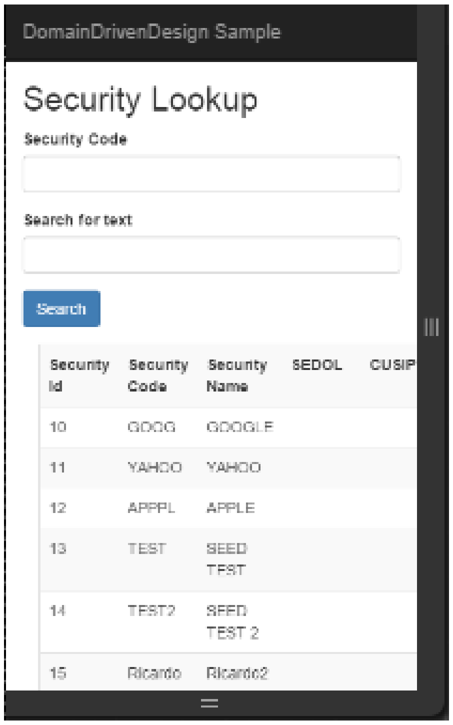
Asp.net MVC layer

Partial Views

I used the Partial views to update just a piece of the Webpage. Partial views are usally used to Render parts of view and can be shared between views. Example: Html.Partial("SearchResultGridPartial") I used this aproach to avoid writing big If and facilitating code readability and maintenance as well Another good thing is the partial view (grid/table) should be responsible to display the data and the rules or validations must be inside it. You could easily put the validation out of the view, but this is a bad practice and a error prone approach since you have to repeat the validation every time you need to use this view.

AntiForgeryToken

I used the ValidateAntiForgeryToken to avoid cross-site injections. Basically it generates a Token in the View and you decorate your Action with ValidateAntiForgeryToken attribute in order to validate the token.



DomainDrivenDesign Sample

Security Lookup

Security Code

Search for text

Search

Security Id	Security Code	Security Name	SEDOL	CUSIP
10	GOOG	GOOGLE		
11	YAHOO	YAHOO		
12	APPPL	APPLE		
13	TEST	SEED TEST		
14	TEST2	SEED TEST 2		
15	Ricardo	Ricardo2		

Mobile First

I used Bootstrap to ensure that the Website would be responsible for Mobile devices. Also we can hide some elements if we needed. But to make this Project simple I just implemented the width attribute such as: content="width=device-width, initial-scale=1.0" I used the table-responsive in the SearchResultGridPartial to make the table resizable to Mobile.

Frameworks and Patterns used

IoC - Inversion of Control & DI - Dependency Injection

Every layer is important but the Domain and Ioc tiers are crucial to have a system ready for accepts changes and support the Business The SimpleInjection is used as Ioc Container to do the Dependency Injections. Using this approach we can easly change any implementation of any Interface just changing the references in the Ioc project. Besides, that facilitates the tests and we can Inject Fakes or Mocks to any Layer or Tier and easly perform Unit Tests.

TDD

The main point here is start with tests and using the AAA concept: Arrange, Assert I developed first the tests and the return must be a fail/red. Then I implement the code and run the tests again, then the tests must be green/passed

BDD - Behavior Driven

The BDD relies on the tests should be created based on specifications of the System. You can create Stories for each behavior an so on. I didn't use for this project but I could use Jasmine framkework.

Mock

Since I'm using IoC in this project, is easly to Mock almost any layer and I use that to ISOLATE the layer I'm testing. For example I used Mocks for test the Domain layer to inject a ISecurityRepository with few values just to test the logic I need. So if there is problem with the method

SOLID

I used strong Object oriented principles which is defined in SOLID specifications. It's a bit work in the beginning but the result, maintenance and readability after is amazing.

SOLID relies on:

- Single-responsiblity principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency Inversion Principle

Single-responsiblity principle

A class should have one and only one reason to change, meaning that a class should have only one job. For example, the class Repository(5.Infrastructure=>5.1.Data=>Repository) is responsible for the data only. It doesn't send email or perform any other respon responsibility than take care of data.

Open-closed principle

Objects or entities should be open for extension, but closed for modification. For example, the class Repository(5.Infrastructure=>5.1.Data=>SecurityRepository) extends the Repository class which keeps the principle that the Objects sould be open to extend and close to modification. Another example is use of Extension methods. For example instead of alter a Class we can add Extension methods the expand its functionalites. I used that in static class StringExtensionMethods for String class but it can use for our classes as well. This avoid that you change complex classes or objects reducing the change to get a bug.

Liskov substitution principle

All this is stating is that every subclass/derived class should be substitutable for their base/parent class. I used that when I ensured that the class Repository(5.Infrastructure=>5.1.Data=>SecurityRepository) is a Repository and moreover it can act exacly as a Repository class. In other words, both Repository and SecurityRepository are responsible only for provide de Data to the Domain.

Interface segregation principle

A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use The class Repository is not using this principle for simplication purposes, but we could easly apply on that. For example, Instead of exists all 3 methods for Insert, update and Delete we could split those actions in separate Interfaces: IInsert, IUpdate, IDelete. And if you need to Delete a Entity in your class you can implement this contract. I used that principle on Citibank where some apps you could Insert and Search, but you couln't delete or update the data for compliance purposes So this way you can avoid uncessary throw new NotImplementedException(); throughout your classes.

Dependency Inversion Principle

Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions. The Application layer depends on Interfaces in the Domain Layer which means that doesn't matter what is the implementation, we always reference the Interfaces and never the implementation itself. That helps a lot future changes since we need to update only the IoC registers. This way, in the Application layer, we are passing control of funcionalities to the Domain:

SecurityAppService class

```
                              public class SecurityAppService: ISecurityAppService
                              {
                                  private readonly ISecurityService _securityDomainServ
ice;

                                  public SecurityAppService(ISecurityService securityDo
mainService)
                                  {
                                      _securityDomainService= securityDomainService;
                                  }

                                  ///..............................................
                              }
```

Benefits of SOLID

Code more robust, prepared and ready to receive changes. This is pretty good because we develop Apps and System to change and when one change is needed we can't say: "We have to re-write the hole system!" If the system is prepared to changed, flexible with each layer responsible for specific things the maintenance is much easier. The overall result is more productivity with quality.