Microsoft | Architecture

# N-Layered Domain-Oriented Architecture Guide with .NET 4.0

César de la Torre (Microsoft)
Unai Zorrilla (Plain Concepts)
Javier Calvarro (Microsoft)
Miguel Ángel Ramos (Microsoft)

1st EDITION


krasis PRESS

# Foreword

By **Diego Vega** (Program Manager, Microsoft Corp., Redmond, Seattle, U.S.)

By the time we released the first version of Entity Framework we were constantly getting feedback from the DDD Community about things that were missing in EF. The main issues were blockers for practicing DDD with EF, such as lack of Persistence Ignorance support, difficulties of testability and high friction in some areas of the API.

Members of the DDD Community and the EF team spent considerable time discussing and cross-educating each other these subjects and on the true potential of EF. This had a strong influence in the second version of the EF, called EF 4.0, and the improvements that later crystallized in EF 4.1, which included massive improvements intended to address many of those concerns.

EF is still going to evolve to improve the experience and to make it easier to fall into what many like to call the "Pit of Success" of software development. But in EF 4 we already reached an important turning point: When customers pick EF for using it in their applications, they often come to us to ask for best practices, e.g. how to implement things with less and more maintainable code. Many of these customers now learn about concepts like Persistence Ignorance and Testability for the first time in our forums, blogs and conference talks! Therefore we are always looking for ways to disseminate this information.

**This book is a necessary and great attempt to distill the existing body of best practices for doing DDD with EF.** I hope it will be very useful for those customers in need of such kind of guidance. Like EF, I hope this book will also evolve over time to accommodate new knowledge and scenarios. I am looking forward to seeing the impact of this initial work, as well as other things coming from the authors in the future.

### Target audience of the Guide

This guide is targeted to the people involved in the entire lifecycle of software products or corporate applications with custom development. Specially, the following roles are applicable:

- Software Architect
- Lead Developer and Developer


### Authors

César de la Torre - Microsoft
Unai Zorrilla (MVP) - Plain Concepts
Javier Calvarro - Microsoft
Miguel Ángel Ramos - Microsoft

### Partial Authors

Cristian Manteiga - Plain Concepts
Fernando Cortés - Plain Concepts
Israel García - Microsoft



ISBN 978-84-939036-2-6

Microsoft | Architecture

krasis PRESS



### Technical Contributors

Diego Vega - Microsoft Corp
Jonathan Aneja - Microsoft Corp
Eugenio Pace - Microsoft Corp
Diego Dagum - Microsoft Corp
Morteza Manavi - Senior Software Engineer, Canada
Pierre Milet Llobet - Microsoft Services
Ricardo Minguez Pablos 'Rido' - Microsoft Services
Hadi Hariri (MVP)- JetBrains
Roberto Gonzalez (MVP) - Renacimiento
Juan Cid - Avanade
Lalo Steinmann - Microsoft
Agustin Alcazar Navarro - Microsoft Premier Support
Rodrigo Bruno Cobo - Microsoft Premier Support
David Mangas Nuñez - Microsoft Services
Pedro Ley - Microsoft Services
Carlos Rodriguez - Indra
Julio César Sánchez Trejo - IT Smart Services
Kaj Wikman - Novia University of Applied Sciences, Finland

# N-Layered Domain-Oriented Architecture Guide with .NET 4.0


Cesar de la Torre - Microsoft
Unai Zorrilla - Plain Concepts
Miguel A. Ramos - Microsoft
Javier Calvarro - Microsoft

## Partial Authors

Cristian Manteiga - Plain Concepts
Fernando Cortés - Plain Concepts
Israel García - Microsoft

## Technical Contributors

Diego Vega - Microsoft Corp
Jonathan Aneja - Microsoft Corp
Eugenio Pace - Microsoft Corp
Diego Dagum - Microsoft Corp
Morteza Manavi - Senior Software Engineer, Canada
Pierre Milet Llobet - Microsoft Services
Ricardo Minguez Pablos 'Rido' - Microsoft Services
Hadi Hariri (MVP)- JetBrains
Roberto Gonzalez (MVP) - Renacimiento
Juan Cid - Avanade
Lalo Steinmann - Microsoft
Agustin Alcazar Navarro - Microsoft Premier Support
Rodrigo Bruno Cobo - Microsoft Premier Support
David Mangas Nuñez - Microsoft Services
Pedro Ley - Microsoft Services
Carlos Rodriguez - Indra
Julio César Sánchez Trejo - IT Smart Services
Kaj Wikman – Novia University of Applied Sciences, Finland


www.EBooksWorld.ir


N-LAYERED DOMAIN-ORIENTED ARCHITECTURE GUIDE WITH .NET 4.0

ALL RIGHTS RESERVED. NO PART OF THIS BOOK MAY BE REPRODUCED, IN ANY FORM OR BY ANY MEANS, WITHOUT PERMISSION IN WRITING FROM THE PUBLISHER.

© Microsoft 2011,

ISBN: 978-84-939036-2-6

www.EBooksWorld.ir

# This book is dedicated to…

**Author - César de la Torre**

"This book is especially dedicated to my family. They suffered my work on this book during many weekends. I also dedicate it to Microsoft and Microsoft Ibérica, because doing this work we joined forces from several complementary areas. One-Microsoft!

The following are comments from my family… ;-)

My wife, Martha:

Let's see if you finish this stuff so we can focus on our house or getting out, trips, etc…

My daughter, Erika (9 years-old):

Daddy, you work a lot but I don't understand anything reading this book…

My son, Adrian (6 years-old):

I don't know.., let's play to the XBOX!

"

**Author - Unai Zorrilla**

"To Lucia and Maria, my family, for their patience regarding my work, far away from home, traveling continuously…"

**Author - Javier Calvarro**

"To my grandmother Teresa. I dedicate to her all my effort put in these pages."

**Author - Miguel Ángel Ramos Barroso**

"For Rosario; my love, my friend, my partner and my life. Only 15 years together and still a lot to share."



www.EBooksWorld.ir

# Contents





www.EBooksWorld.ir

www.EBooksWorld.ir

www.EBooksWorld.ir

www.EBooksWorld.ir

www.EBooksWorld.ir

www.EBooksWorld.ir

www.EBooksWorld.ir

www.EBooksWorld.ir

# Prologues

**Prologue by Enrique Fernandez-Laguilhoat**
(*Director of Development and Platform Division in Microsoft Ibérica*)

It is not just a coincidence that the IT sector imitated the construction sector using the Architect and Architecture names. Like in the case of great construction works, to ensure success in the development of a software application, it is first necessary to have a good definition of the structure that will be followed, the different elements or modules to be built and how they interact with each other in a safe and effective manner. A poor architecture job leads in many cases to the project's failure, and on the contrary, if the software architecture does his job well, the resulting product will be strong, the time and effort to develop it will be lower, and something very important, widening or extending the development in the future will be much easier.

This guide covers a very important area in the development world. With a noticeable group of software professionals, and led by César de la Torre, one of the main Software Architects of Microsoft, a global and systematic vision is offered about how to approach layered development using the .Net technology. If developing with the .Net framework has always been easy and highly productive, the arrival of this guide offers also a highly structured help that enables definition of the architecture and modeling of the application.

It has been a pleasure to see for several months the illusion (and long work hours) that both César and the ones that helped with their contribution have invested in this guide. As far as I am concerned, I would like to thank his job and effort and recognize the high quality of the resulting product. And I am sure the reader will be able to thank that too, by taking advantage of this guide in the new development challenges.

www.EBooksWorld.ir

**Prologue by José Murillo**
(*Developer Solution Specialist, Microsoft DPE*)

The great business software projects usually fail. This is a harsh statement but let's face it, it is the awful truth all of us who have worked for years in the applications development world are familiar with.

The software development "industry" is only 60 years old. During that time we have been learning to go from sand to brick, from brick to pre-manufactured blocks, but all these perfectly valid construction techniques for houses are not enough and not useful for great buildings. If we try to apply them to these macro-projects, development time is exponentially increased or the building goes down at the first tremor or users load test.

What is failing? There is no doubt for me, the Development Lifecycle Management and **Applications Business Architecture**. In the same way that design, structures and load calculations are so important in traditional Architecture, in the world of software development, the Software and Systems Architecture are. It is the discipline that teaches us how we have to combine existing blocks and technologies to form solid and durable applications. This role, unfortunately, is not so present in the current companies, where any good programmer, as time goes by, and after he is recognized for his past merits, is promoted to "Project Manager". But what the hell does this have to do with anything?

This book offers precisely the guidelines, guide, recommendations and good practices for Software Architects to be able to design business applications without reinventing the wheel, using existing patterns and tested good practices. It can effectively place abstract concepts and a lot of the latest Microsoft technologies in concrete recommendations for these new .NET Architects.

These are the reasons of my acknowledgement, and I would like to thank my partner and friend César de la Torre for his work. I know perfectly well the great personal effort he has made to bring this project to life, and I am sure it will lead to improvement of quality of business applications started following his recommendations. Also, I thank the rest of the collaborators without whose help this book would have ended with Cesar.

www.EBooksWorld.ir

**Prologue by Aurelio Porras**
(*Developer Solution Specialist, Microsoft DPE*)

I had the opportunity to participate in the development of some applications of a certain relevance and I happily remember these meetings at the beginning of projects where we drafted with boxes and arrows the architecture skeleton, we detected patterns and labeled any element of the diagram with the latest technologies available that helped us implement in the best possible way the required functionality without having to reinvent the wheel. In these architecture discussions, the typical quarrels appeared about the level of complexity of the application architecture to be implemented; on one hand, the ones in favor of mounting a simpler architecture using code libraries and implementations of patterns already built, to produce business logics immediately and to present results as soon as possible, giving more freedom to the developer when using technologies; and on the other hand, the ones in favor of building a more complex architecture, building libraries and implementing patterns in accordance with the application to speed up the production of business logics after that, although the results appeared later, increasing the level of abstraction to prevent the developer from having to make technological decisions. It was interesting to see how the "simplistic group" rebuked the "complex group" the wasted effort in building unnecessary church arcs that the technological infrastructure manufacturers in the following version would make obsolete and tedium of the developer of business logics that sometimes stopped being a programmer and turned into a mere person in charge of setting up the architecture; and the "complex group" scolded the "simplistic group" for the amount of duplicated code they threw away to the garbage and the effort in coordination they wasted to avoid those functional duplicity problems by giving so much freedom to the developer. Yes, it sounds like grandpa's stories that are told over and over, but these were very amusing meetings.

The final result of these discussions was a series of architecture decisions determining the technological infrastructure to be used to build the application, the relationships with external systems, the organization of the code in layers, the libraries already available for use, and the ones to be developed customized, among other things. I remember in particular how we tried to de-couple parts of the application to enable its future evolution, up to the limit allowed by the state of the art technology at that time, to be able to modify or extend the business logics not having to touch all the modules or to be able to interchange one of the external systems, the applications server or the database with no problem.

But these architecture decisions were not only conditioned by technical factors such as technological infrastructures, programming language or development tools; but also by factors related to development of a software project such as budget and duration, milestones and deliverables, experience of the development team, business knowledge and all the things the projects have. At the end, architecture could suffer the unwanted cuts due to "project decisions".

So, unfortunately, I also had the chance to verify how certain architecture decisions can condemn the future of great applications. I know the case of a financial application that could adapt to business changes very quickly, thanks to the high level of

www.EBooksWorld.ir

abstraction its architecture provides; the user himself is capable of modifying the application logics through a visual tool and programming a pseudo-language of business; the problem is that capacity of integration in line with other systems is very limited because it is built on obsolete technologies and its coupling to them is such, that the cost of migrating to the latest technologies is too high and is not justified from the business point of view; especially if we consider that the application still works as a charm and following the rule of our industry, if it works, don't touch it. I also know other financial application that is well de-coupled from the applications server and the database and it is relatively simple to update it technologically, but it did not take care of the code organization and the business logics is so intricate in the different application layers that it is not quickly adapted to changes as it would be good for business, and streamlining it would involve rewriting most part of the application; this is unthinkable, almost a new project. Surely, the two applications were created this way due to the particular circumstances of the corresponding projects, but it is clear that architecture decisions made at that time negatively affected the evolution maintenance of those two applications, which, as it was expected from the beginning, would have long duration in the production environment.

This is the root cause of this guide. Naturally, the state of the art technology has changed a lot, the architecture trends, the capacities of modern technological infrastructure, the new things in programming language and new development tools help a lot to build weakly coupled architectures to enable evolution maintenance of the applications; but if, also we conceive application architecture considering first the importance of its future evolution, to adapt easily to the business changes and to add the latest technologies replacing the oldest ones, we will be close to building a great business application with healthy life guaranty.

And this is what the guide is about, building an architecture that de-couples the business logics of the technology used to build the application so that one can evolve independently from the other. And it does not only talk about birds and flowers, but it also goes to a level of technical detail that will show us in the latest .NET technologies and Microsoft development tools and its application in great business applications, indicating when to use which technology and why, and including also the sample application code following the precepts set forth throughout the whole guide.

I thank César for all this clarifying material, for his effort leading this initiative that will surely help architects and developers to approach application architectures with a more holistic vision, and I also thank the authors and collaborators that participated in its preparation. Congratulations on the result.

www.EBooksWorld.ir

**Prologue by Israel Garcia Mesa**
(*Consultant - Microsoft Services*)

We currently have a wide range of technological options we can use in our projects and that cover many needs being detected through the years. The experience we have in *Microsoft Ibérica* is that this variety of choices does not resolve all the problems of the projects in our customers. The analysis we made and we continue to make to be better every day has given us a series of conclusions we want to share in this guide.

**Some thoughts about Architecture**

The development of a software construction project is a process where many factors are involved, and that is why it is important to have the suitable tools. Currently there are many technological options available that help us compose our solutions, but, however, they do not mitigate the main problems of a project:

- Needs of adaptation to changes in projects (functional and technical requirements) that can have great impact on the effort.
- Uncertainty when choosing and using technology that best fits each scenario.
- Integration with legacy systems that do not have a clear alignment with the project requirements.

These and other situations can affect the development of projects and increase the possibility of new risks appearing that impact on the project. To mitigate these risks the following is recommended:

- The methodology of work must be adapted to our team, type of project and customer, since this will be our tactic to meet our goal and all the details must be considered. So it is important to choose a work method adapted to the context of the project where the type of solution and work team must be considered.
- Consider an architecture model that meets the known needs and with a low coupling level, which will enable its adaptability. In this point different options can be chosen when approaching the system, but following the model proposed by the Domain Directed Design (DDD) can help us follow the most suitable approach.

The design of the solution, besides being an incremental process, can be a process to be performed from different approaches until completion of the solution vision. From the experience gathered in the different projects we have developed we have considered useful some approaches we summarize below:

- Solutions, of whatever size, are born from a global design where technical aspects are not relevant (we could talk about conceptual design) and subsequently design the parts of the solution as we focus on each one of

www.EBooksWorld.ir

them. With this model, slowly, we will get closer to the details of implementation de-coupling the design, reducing complexity and the possibility that a technical problem can affect the rest of the solution.

- Also, it will be necessary to conjugate the design of the logical model with the physical model or models, ideally an approach would condition the other to the least extent possible. This type of approach enables reuse and adaptability of the solution to the different scenarios.

There will always be a desire to build a solution around the idea that technology will solve our problems, and we will think this is a shortcut to our goals. However, we can discover this is not the quickest path since, when a design cannot grow and/or evolve because it requires a lot of effort or we do not control the impact of such changes, this is when the technology does not bring value to the solution and can become a problem.

Additionally, there is a series of very useful tools to build a solution and that help us to face changes in implementation and its design:

- **Test Development**: having unit tests and automated functional tests will help us know the stability of our solution, and therefore determine if any change could affect the solution and to what extent.
- **Refactorization**: approaching and implementing changes in a solution through *refactoring* techniques is an efficient manner that helps us control their impact. Complementing refactoring with the use of tests helps reducing risks, so these are two perfectly complementary tools.
- **Communication**: a good communication inside the team reduces possibility of working inefficiently or even duplicating functionality. Besides it is a useful tool in our relationship with the customer, helping us place common expectations, detecting new requirements or possible risks, in quick and agile manner.

These conclusions that can seem logical and, however, so difficult to be carried out, are the reason why we want to share knowledge of this guide so that our experience can be useful in projects and so that technology becomes that **tool that makes our job easier.**

www.EBooksWorld.ir

CHAPTER 1

# .Net Architecture Guide Introduction

## 1.- INTRODUCTION

*Microsoft Spain (Microsoft Ibérica)* has noticed, in multiple customers and *partners*, the need to have a ".NET Base Architecture Guide" that can serve as an outline for designing and implementing complex and mission critical enterprise .NET applications with long term life and long evolution. This frame of common work defines a clear path to the design and implementation of business applications of great importance with a considerable volume of business logic. Following these guidelines offers important benefits regarding quality, stability, and especially, an improvement of future maintenance of the application, due to the loose-coupling between components, homogeneity, and similarities of the different developments that will be done based on these guidelines.

*Microsoft Spain* defines this '***Architecture Guide***' as a pattern and base model; however, this guide could be extended by any company so it fits specific requirements. This guide is only the first step of a staircase, an initial asset that must could be customized and modified by each organization that adopts it, focusing on the specific needs, adapting it and adding specific functionality according to the target needs.

### 1.1.- Target audience of the Guide

This guide is targeted to the people involved in the entire lifecycle of software products or corporate applications with custom development. Specially, the following roles are applicable:

- Software Architect
- Lead Developer and Developer



www.EBooksWorld.ir

## 1.2.- .NET Architecture's Guide Objectives

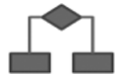
This document intends to describe an architecture on which products or custom applications can be developed, and provides a set of rules, best practices and development guides to use .NET properly and, specially, in a homogeneous manner.

***DISCLAIMER*:**
We would like to emphasize the following and remark that this proposal of '***N-Layered Domain-Oriented Architecture***' is not suitable for all types of applications, **but appropriate only for complex business applications with a relevant volume of business logic with a long term life and evolution**, where it is important to implement loose-coupling concepts and certain DDD patterns. For small or data oriented applications, probably, a simpler architecture approach implemented with RAD technologies is probably more suitable.
Also, this guide (and its associated sample application) is simply a proposal to be considered, evaluated and customized by organizations and companies at their own discretion. *Microsoft* is not liable for problems that could arise from this work.

## 1.3.- .NET Architecture's documentation levels

Documentation of this architecture is structured in two main levels:

- **Logical Level:** This is a Software Architectural level agnostic to technology where there is no implementation of a concrete .NET technology. To highlight this level, the following icon will be shown:



- **Implementation Level:** The second level is the specific implementation of a .NET Architecture, where possible technologies for each scenario will be listed with specific versions. Normally, an option will be chosen and its implementation will be explained. Moreover, the implementation of the architecture has a sample .NET application, with a very small functional scope, but it must implement each and every technological area of our architecture. To highlight this level, the .NET icon will appear at the beginning of the chapter:

www.EBooksWorld.ir

## 1.4.- Sample Application in CODEPLEX

It is essential to remark that, in parallel to the drafting of this book/architecture guide, we also developed a sample application that implements the patterns exposed in this guide with the latest Microsoft technologies ('Wave .NET 4.0').

Also, most code *snippets* shown in this book are precisely code extracts of this sample application.

This sample application is published on CODEPLEX with OSS code and can be downloaded from the following URL:

**Table 1.- CodePlex 'Open source Project Community**

| | |
|---|---|
| CodePlex Open Source Project Community | **http://microsoftnlayerapp.codeplex.com/** |

In CODEPLEX, we not only have the source code of the sample application but also certain documentation on requirements (necessary technologies, such as *Unity 2.0, PEX & MOLES, WPF Toolkit, Silverlight 4 Tools for Visual Studio 2010, Silverlight 4.0 Toolkit, AppFabric, Windows Azure*, etc., as well as links to download them.) There is also a **Discussions/Forum** page, to cooperate with the community, ask questions, provide ideas, evolution proposals, etc…

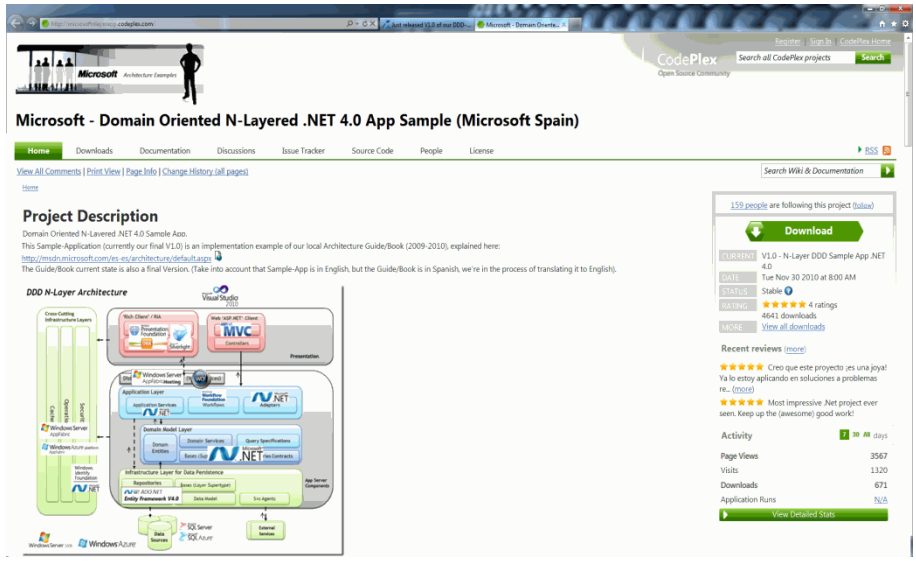



**Figure 1.- Domain oriented N-Layered .NET 4.0 App Sample**

www.EBooksWorld.ir

The sample application implements different DDD Architecture and Design patterns with the latest Microsoft technologies. It also contains several clients including WPF, Silverlight, ASP.NET MVC front-end and others to be added such as OBA and Windows Phone 7.0, etc.

It is important to emphasize that functionality of the sample application is obviously quite simple since Architecture is the main focus here, and not the implementation of a large volume of functionality that complicates Architecture tracking and comprehension.

The presentation layer and the different implementations are simply one more area of the architecture and not precisely the *core* of this reference guide, where our focus is on the layers related to the back-end (Domain layer, Application layer, Data access infrastructure layer, and their corresponding patterns). Furthermore, there is also a review of the different patterns in the presentation layer (MVC, M-V-VM, etc.) and how to implement them with different technologies.

Here we show some snapshots of the sample application:

## Silverlight 4.0 Customer

**Silverlight – Customers' List**


**Figure 2. - Silverlight – Customers' List**

www.EBooksWorld.ir

**Silverlight – Silverlight transition**

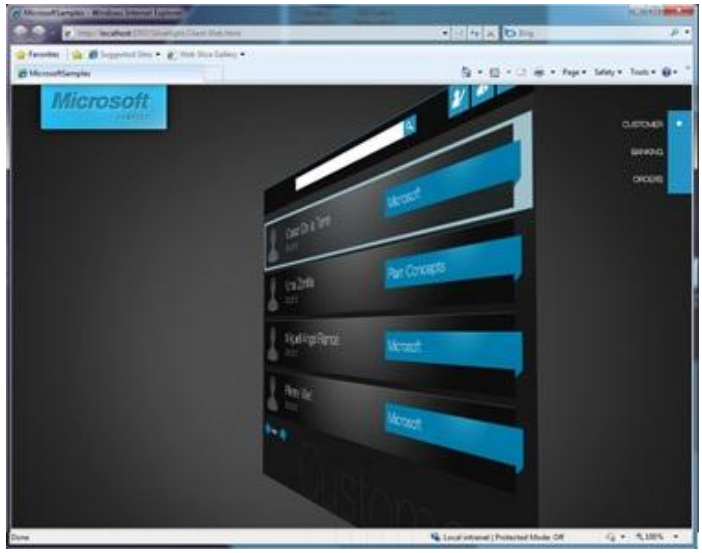



**Figure 3.- Silverlight – Silverlight transition**

**Silverlight – 'Customer View'**


**Figure 4.- Silverlight – 'Customer View'**

www.EBooksWorld.ir

## WPF 4.0 Customer

**WPF – View of 'Customers' list'**

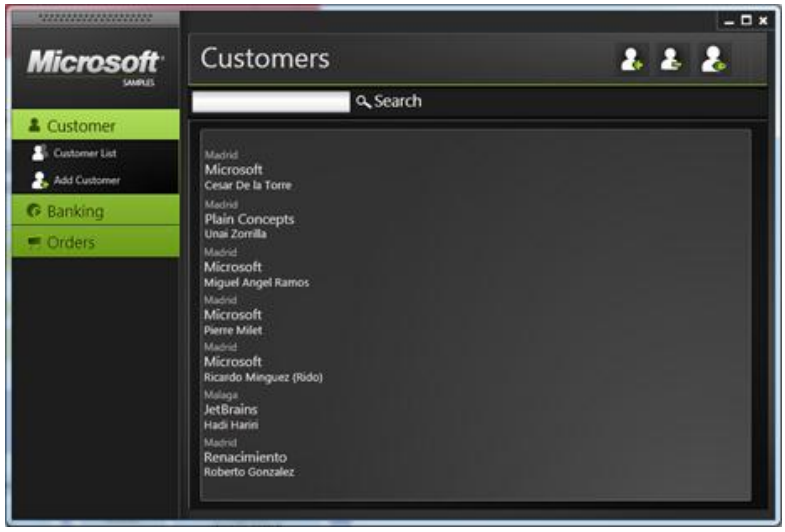



**Figure 5.- WPF – View of 'Customers' list'**

**WPF – Customers view**


**Figure 6.- WPF – Customers view**

www.EBooksWorld.ir

**WPF – 'Bank Transfers' (Domain Logic & Transaction sample)**


**Figure 7.- WPF – 'Bank Transfers'(Domain Logic & Transaction sample)**

## ASP.NET MVC Customer

**MVC – 'Bank Transfers'**

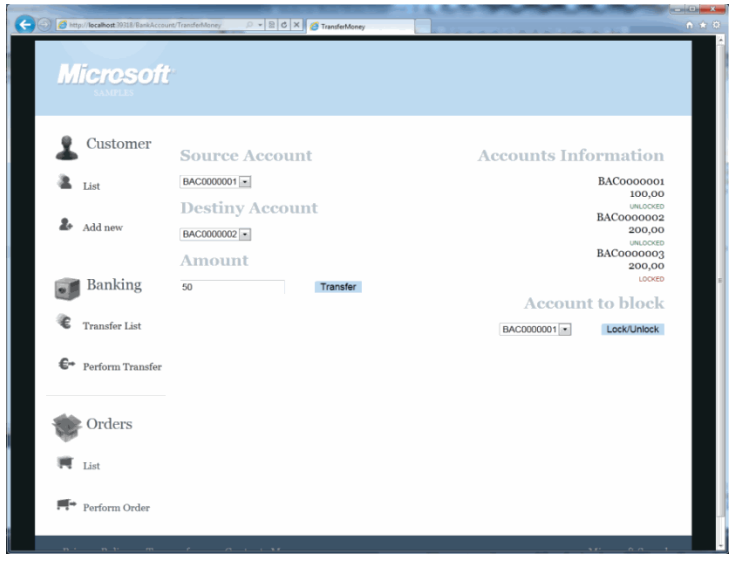



**Figure 8.- MVC – 'Bank Transfers'**

www.EBooksWorld.ir

**MVC– View of "Customers list"**

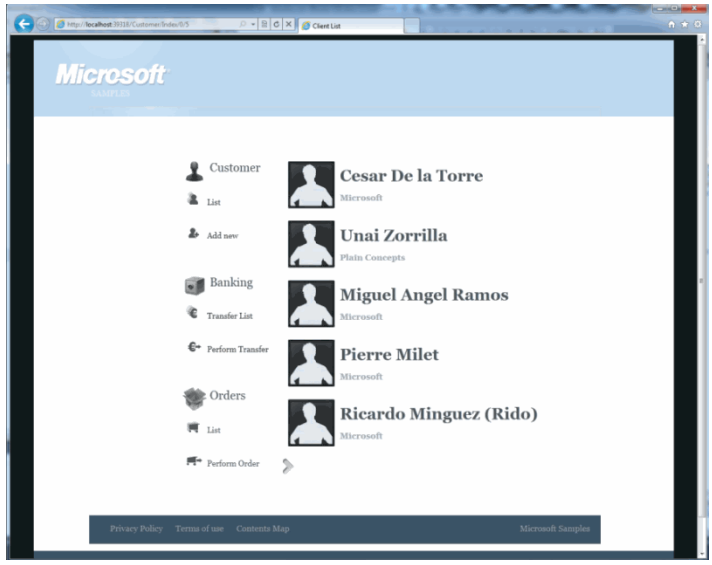



**Figure 9.- MVC– View of "Customers list"**

Finally, please note that we developed both the application interfaces and the entire source code of the solution in English so that it can be used by the entire community.

We recommend downloading this sample application from CODEPLEX and referencing it as you read this architectural guide/book, especially while going through sections on implementation marked with the following .NET logo:

www.EBooksWorld.ir

CHAPTER 2

# The Architecture Design Process

The architecture design process plays a very significant role in the scope of software engineering and ALM. The difference between a good architectural design process and a bad one can be the difference between failure and success of the project. In the architecture design, we address more important issues when defining our system; we create a basic model of our application. Inside the architecture design process, the following aspects are decided:

- Type of application to be built. (Web, RIA, RichClient…)
- Logical structure of the application (N-Layers, Components, etc…)
- Physical structure of the application (Client/Server, N-Tier, etc…)
- Risks to be faced, and how. (Security, Performance, Flexibility, etc…)
- Technologies to be used (WCF, WF, WPF, Silverlight, ASP.NET, Entity Framework, etc...)

To follow this entire process, we will start with the information generated by the requirement gathering process, specifically:

- User stories or use cases.
- Functional and non-functional requirements.
- Overall technological and design restrictions.
- Proposed deployment environment.



www.EBooksWorld.ir

Beginning with this information, we must generate the necessary devices so that the programmers can correctly implement the system. In the architecture design process, we must define a minimum of the following:

- Significant use cases to be implemented.
- Risks to be mitigated and how.
- Potential architectures to be implemented.

As mentioned above, the architecture design is an iterative and incremental process. In architecture design we repeat 5 steps until we complete the full system development. The steps we repeat and the clearest way to see them is this:

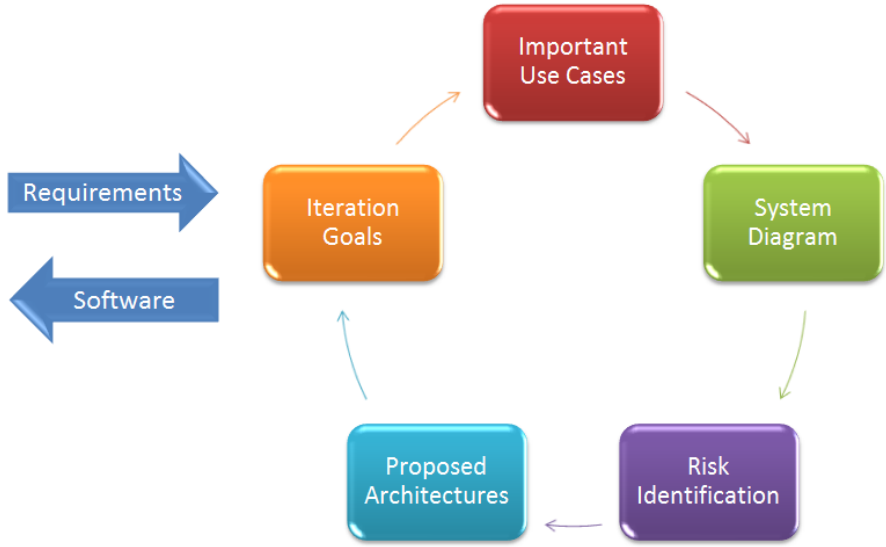



**Figure 1.- Architecture Design process**

Below we explain each step in detail to learn what should be defined and will try to make each one as clear as possible.

## 1.- IDENTIFYING THE PURPOSES OF ITERATION

The purposes of iteration are the first step in shaping the architecture of the system. At this point, the important thing is to analyze the restrictions of the system in terms of technologies, deployment topology, system use, etc…. In this stage, it is very important to define the purposes of the architecture; we must decide if we are building a prototype, a full design, or if we are testing possible architecture development manners. At this point we must also consider the individuals that make up our team. The type of

www.EBooksWorld.ir

documentation to be generated as well as the format will depend on whether we address other architects, developers, or people with no technical knowledge

The purpose of this stage of the architecture design process is to fully understand the environment of our system. This will allow us to decide where to focus our activity in the next stages of the design, and will determine the scope and time necessary to complete the development. At the end of this phase, we must have a list of the goals of iteration, preferably with plans to deal with them and metrics to determine time and effort required to complete them. After this phase, it is necessary to have an estimation of the required time to spend on the rest of the process.

## 2.- SELECTING ARCHITECTURALLY IMPORTANT USE CASES

The architecture design is a process driven by the client's needs, and by the risks to be faced; this means we will develop the use cases (functionality) with more value for the client and mitigate the most important risks faced by our architecture (quality requirements) first. The importance of a use case will be identified according to the following criteria:

- Importance of the use case inside the business logic: This will be given by the frequency of use of the case in the system in production, or the value provided by this functionality to the client.

- The development of the use case implies an important development of the architecture: If the use case affects all the tiers of the architecture, it is a strong candidate to be prioritized, since its development and implementation will allow the definition of all the tiers of architecture and hence, increase the stability of the architecture.

- The development of the use case implies addressing some quality requirements: If the use case requires addressing issues such as security, availability or system failure tolerance, it is an important use case since it allows us to address horizontal aspects of the system as functionality is developed.

- The way the use case meets the iteration goals: When selecting the use cases we will implement, we will have to consider the way they meet the goals we defined for the iteration. We will not choose use cases that highly develop the functionality of the system if the iteration goal is to reduce bugs or to mitigate any given risk.

It is important to clarify that the system architecture must not be designed in a single iteration. In this stage of the design process we analyze all use cases and select only a subset, the most architecturally important, and we proceed to its development. At this point, we only define the aspects of architecture for the use cases we selected, and we leave the rest of the aspects open for future iterations. It is worth noting that we

www.EBooksWorld.ir

may not define an aspect of the system completely in a single iteration, and we have to clearly understand that we must try to minimize the number of changes in the future iterations. This does not mean we must not "assume that the software evolves", but that, when developing an aspect of the system, we should not depend on a specific solution but rather search for a generic solution that allows us to deal with possible changes in future iterations. Ultimately, all this comes down to taking short but steady steps.

When developing the system it is interesting to consider the different stories of the user, system and business. The user, system and business stories are small phrases or paragraphs describing some aspects of the system from the point of view of the party involved. The user story defines how users will use the system, the system story defines the requirements that the system will need to meet and how it will be internally organized, and the business story will define how the system will meet the business restrictions.

Breaking down the use cases in several areas like user, system and business stories will allow us to more easily validate our architecture by making sure the user, system and business stories of the iteration are met.

## 3.- PREPARING A SYSTEM SCHEME

After the goals of the iteration and the functionality to be developed are clear, we can proceed to the design. At this point, the first step is to decide the kind of application to be developed. The type of application to be chosen will depend on the deployment and connectivity restrictions, the complexity of the user interface and restrictions of interoperability, flexibility and technologies imposed by the client. Each type of application offers a series of advantages and disadvantages. The architect has to choose the type of application that best fulfills the advantages he expects his system to have and presents the least inconvenience. The main application types we will develop are:

- **Applications for mobile devices:** Web applications with an interface adapted to mobile devices or user applications developed for the terminal.

- **Desktop applications:** Classic applications installed on the user machine.

- **RIA (Rich Internet Applications):** Applications executed inside the browser thanks to a plug-in and that offer a better response than web applications and an interface with similar quality to the user applications, with the advantage of not having to install them.

- **Services:** Applications showing certain functionality in the form of Web services for other applications to use.

- **Web applications:** Applications used through a browser and that offer a standard user interface, completely interoperable.

www.EBooksWorld.ir

For the purpose of summary and guidance, the following table gathers the main advantages and considerations to be taken into account for each type of application:

**Table 1.- Advantages and Considerations. Application Types**

| Application type | Advantages | Considerations |
|---|---|---|
| Applications for mobile devices | • They are good for no connection or limited connection scenarios.<br>• They can be carried in hand-held devices.<br>• They offer high availability and easy access for users outside their usual environment. | • Limitations when interacting with the application.<br>• Reduced size of the screen. |
| Desktop applications | • They better leverage the client's resources.<br>• They offer the best response to the interaction, a more powerful interface and a better user experience.<br>• They provide a very dynamic interaction.<br>• They support disconnected or limited connection scenarios. | • Complex deployment.<br>• Complicated versioning.<br>• Low interoperability. |
| RIA (Rich Internet Applications) | • They provide the same graphic power as desk applications.<br>• They offer support to view multimedia contents.<br>• Simple deployment and distribution. | • Slightly heavier than Web applications.<br>• Desktop applications leverage resources better.<br>• They require the installation of a plug-in to operate. |

www.EBooksWorld.ir

| | | |
|---|---|---|
| Service-oriented applications | • They provide a loosely-coupled interface between client and server.<br>• They can be used by several applications that are unrelated.<br>• They are highly interoperable. | • They do not have a graphic interface.<br>• They need Internet connection. |
| Web applications | • They reach any type of user and they have a standard and cross platform user interface.<br>• They are easy to deploy and update. | • They depend on network connectivity.<br>• They cannot offer complex user interfaces. |

After deciding the type of application we will develop, the next step is to design the infrastructure architecture, that is to say, the deployment topology. The deployment topology depends directly on restrictions imposed by the client, the system's security needs and infrastructure available to deploy the system. We define the infrastructure architecture at this point, to consider it when designing the logical architecture of our application. Given that layers are logical, we will fit the different logical layers inside the system tiers. In general, there are two possibilities, distributed deployment and non-distributed deployment.

Non-distributed deployment has the advantage of being more simple and efficient in communications since the calls are local. On the other hand, this makes it more difficult to allow several applications to use the same business logic at the same time. Also, in this type of deployment, the machine's resources are shared by all layers so if a layer uses more resources than the others, there will be a bottleneck.

Distributed deployment allows the separation of logical layers in different physical tiers. This way, the system can increase its capacity by adding servers wherever necessary and the load can be balanced to maximize efficiency. At the same time, by separating the layers in different tiers, we leverage the resources by balancing the number of servers by tier depending on the needs of the layers it holds. The downside of distributed architectures is that the cost of serialization of the information and its transportation through the network is not insignificant. In addition, distributed systems are more complex and expensive.

After deciding the type of application we will develop and its deployment topology, it's time to design the logical architecture of the application. To do so, and insofar as possible, we will use a set of known architecture styles. The architecture styles are "patterns" of an application design that define an aspect of the system we are building and represent a standard manner of definition or implementation of said aspect. The difference between an architecture style and a design pattern is the abstraction tier, that

www.EBooksWorld.ir

is, a design pattern provides a concrete specification on how to organize classes and interaction between objects, while an architecture style provides a series of indications on what is to be done and what is not to be done in a certain aspect of the system. The architecture styles can be grouped according to the aspect defined by them, as shown in the following table:

**Table 2.- Structural styles. Aspects**

| Aspect | Structural styles |
|---|---|
| **Communications** | SOA, Message Bus, Piping and Filters. |
| **Deployment** | Client/Server, 3-Tiers, N-Tiers. |
| **Domain** | Domain Model, Repository. |
| **Interaction** | Separate presentation. |
| **Structure** | Components, Object-oriented, Layered Architecture. |

As shown in this table, in an application we will use several architecture styles to shape the system. Therefore, an application will be the combination of many styles and our own solutions.

Now that we have decided the type of application, physical infrastructure and logical structure, we have a good idea of the system we will build. The next logical step is to start implementing the design, and to such effect, the first thing to do is to decide the technologies to be used. For the most part, these technologies will be determined by the architecture styles we use to shape our system, the type of application to be developed and the physical infrastructure. For example, to do a desktop application we will choose WPF or Silverlight (Out of the Browser), or if our application exposes its functionality as Web services, we will use WCF. Summing up, the questions we have to answer are the following:

- What technologies help implement the selected architecture styles?
- What technologies help implement the selected application type?
- What technologies help meet the specified non-functional requirements?

At the end of this stage the important thing is to be able to draw an architecture scheme that reflects the structure and the main restrictions and design decisions made. This will allow us to set up the framework for the system and discuss the proposed solution.

www.EBooksWorld.ir

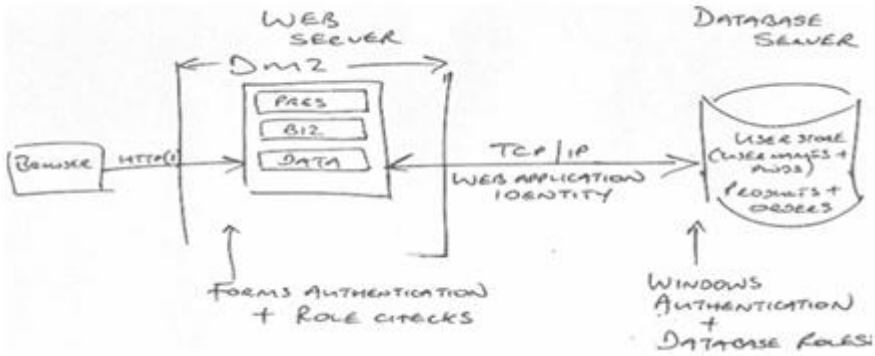



**Figure 2.- Example of an Architecture Scheme**

## 4.- IDENTIFYING THE MAIN RISKS AND DEFINING A SOLUTION

The architecture design process is directed by functionality, but also by risks to be mitigated. The sooner we minimize risks, the more likely we will have success in our architecture.

The first question is how to identify the risks of an architecture. To answer this question we must first have a clear idea of the non-functional requirements (aka quality requirements) that our application must meet. This information must have been defined after the inception phase, and should therefore be available when designing the architecture.

The non-functional requirements are the properties that our solution must have and that are not about functionality. For example: High availability, flexibility, interoperability, maintainability, manageability, performance, reliability, reusability, scalability, security, testability and usability.

It is worth mentioning that normally nobody (a normal client) would tell us "the solution has to ensure high availability", but these requirements will be given in the client's jargon, for example "I want the system to keep on working although a component fails", and it is the job of the architect to translate them, or to put them in any of the categories.

Now that we have clarified the non-functional requirements (and therefore the risks) to be considered, we can proceed to define a solution to mitigate each one of them. The non-functional requirements have an impact on how our architecture will address the certain "key points" such as: authentication and authorization, data cache and status maintenance, communications, composition, concurrency and transactions, setup management, coupling and cohesion, data access, exceptions handling, logging and system instrumentation, user experience, data validation and business processes flow. These key points will have an associated functionality in some cases or will determine how implementation of an aspect of the system is carried out.

www.EBooksWorld.ir

As mentioned above, **non-functional requirements are properties** of the solution **and not the functionality**, but they directly affect the **key points** of architecture which **are functionalities** of the system. We can say the non-functional requirements are the specification of the properties of our system and the key points are the implementation.

In general, a non-functional requirement has several associated key points that have a positive or negative effect on their achievement. Therefore, we will analyze each non-functional requirement based on the "key points" affected, and we will make the proper decisions. It is important to understand that the relationship between non-functional requirements and key points is many to many, which means that there will be situations when a key point will affect several non-functional requirements. When a key point is beneficial for meeting all the non-functional requirements, there will be no problem, but when it affects one in positive manner and another one in a negative manner, decisions will be made to establish a commitment between both requirements.

Each attribute is addressed in more detail in the chapter dedicated to the horizontal/transversal aspects of the architecture.

As mentioned, in this stage of the project we will mitigate the main risks by creating plans to solve them and contingency plans in the event that they cannot be solved. The design of a plan for quality requirement will be based on the key points affected by said requirement. The plan of a requirement will consist of a series of decisions about key points. Whenever possible, it is better to express these decisions graphically. For instance, in the case of security, indicating in the physical architecture diagram the type of security used in each zone or in the case of performance, where the data cache is made.

## 5.- CREATING CANDIDATE ARCHITECTURES

After following the steps above, we will have candidate architecture to evaluate. If we have several architectures that are candidates, we will assess each one and will implement the best valuated architecture. Any candidate architecture must answer the following questions:

- What functionality does it implement?
- What risks does it mitigate?
- Does it meet the restrictions imposed by the client?
- What issues does it leave unsolved?

Otherwise, the architecture is not yet well defined or we have not followed some of the steps above.

To value the candidate architecture we will analyze the functionality it implements and the risks it mitigates. Like in any other validation process we have to establish metrics to define the eligibility criteria. To such effect there are multiple systems but in general we will have 2 types of metrics: Qualitative and Quantitative.

www.EBooksWorld.ir

The quantitative metrics will assess each aspect of our candidate architecture and will provide a rate that we will compare with the rest of the candidate architectures as well as with a possible minimum required rate.

The qualitative metrics will determine whether the candidate architecture meets a certain functional requirement or a quality requirement. In general they will be assessed in binary manner as yes or no.

With these two metrics we can create combined metrics: for example, quantitative metrics that will only be assessed after going through the qualitative metrics.

As mentioned, there are multiple systems to evaluate the software architecture, but all of them are based on this type of metrics to some degree. The main software architecture evaluation systems are:

- *Software Architecture Analysis Method.*
- *Architecture Tradeoff Analysis Method.*
- *Active Design Review.*
- *Active Reviews of Intermediate Designs.*
- *Cost Benefit Analysis Method.*
- *Architecture Level Modifiability analysis.*
- *Family Architecture Assessment Method.*

All of the decisions regarding architecture must be documented so they can be understood by all of the development team members, as well as the rest of the project stakeholders, including clients. There are many ways of describing architecture, such as through ADL (*Architecture Description Language*), UML, Agile Modeling, IEEE 1471 or 4+1. As the proverb says, one image is worth a thousand words, so we prefer graphic methodologies such as 4+1. 4+1 describes software architecture through 4 different views of the system:

- **Logical view:** The system's logical view shows the functionality that the system provides to the end users. It does so by using class, communication and sequence diagrams.
- **Process view**: The system's process view shows how the runtime system behaves, the processes that are active and how they communicate. The process view solves issues such as concurrency, scalability, performance, and in general any dynamic feature of the system.
- **Physical view**: The physical view of the system shows how the different system's software components are distributed in the physical nodes of the infrastructure and how they communicate with each other. This is done by using deployment diagrams.

www.EBooksWorld.ir

- **Development view**: The system's development view shows the system from the programmers' point of view and is focused on software maintenance. This is done by using components and packages diagrams.

- **Scenarios:** The scenarios view completes the architecture description. Scenarios describe sequences of interactions between objects and processes and are used to identify architecture elements and to validate the design.

## 6.- DOMAIN DRIVEN DESIGN ASPECTS

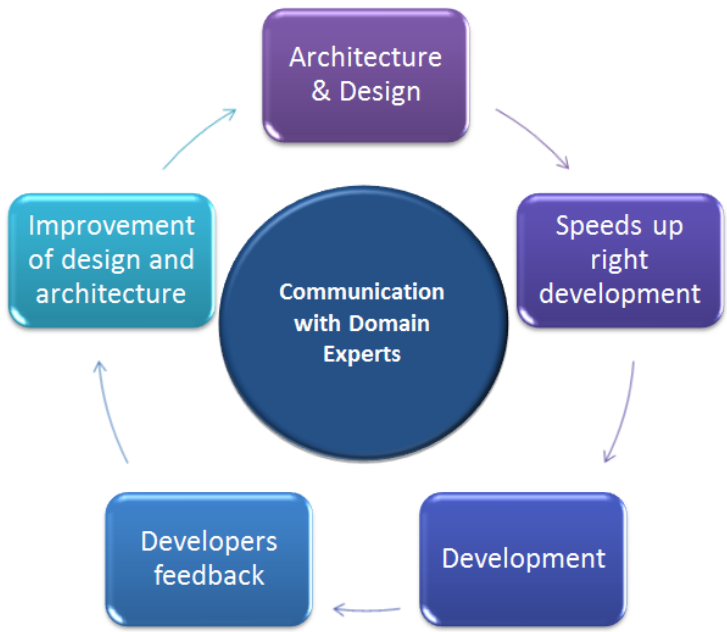



**Figure 3.- Communication with Domain Experts**

So far, we have discussed the architecture creation process, focusing on how to choose the use cases relevant to architecture, how to decide the type of application to be implemented and how to face the project risks inside the architecture. Below we will discuss the key items we have to consider to get the architecture to reflect our domain.

The purpose of an architecture based on Domain Driven Design is to get an object oriented model that reflects the knowledge of a certain domain and that is fully independent from any infrastructure and technology concept (data access technologies, graphic interface, etc.). What we want is to build a model through which we can solve problems by the cooperation of the set of objects. We must therefore have a clear idea of the following:

www.EBooksWorld.ir

- Any software project with complex logic and a complicated domain must have a model representing the aspects of the domain that allow us to implement use cases.

- The focus of attention in our architecture must be the domain model and its logic, and how to decouple the domain logic from infrastructure assets.

- The model we build must be closely related to the solution we deliver and therefore it must take into account the implementation considerations.

- The domain models represent accumulated knowledge and since that knowledge is acquired gradually and incrementally, the creation process of a model that thoroughly represents the domain concepts must be iterative as well.

## 6.1.- Ubiquitous language

One of the main reasons for the failure of software projects is the lack of communication between the domain experts and the developers in charge of building a system. The lack of a common language to communicate between domain experts and developers, as well as between the developers themselves, generates problems such as the different interpretation of concepts or multiple representation of a same concept. This leads to implementations not related to the domain that they are working on or where they try to solve problems. The implementations not related to the domain which they are working on features two clearly noticeable symptoms:

- The system does not solve a problem properly.

- The system does not solve the proper problem.

It is essential to know that any model we build must be deeply represented in the implementation of the system. This means that, instead of having an analysis model and an implementation model, we must have one single model, the "domain model".

Any model we build must explicitly represent the main concepts of the domain of knowledge that our system works with. We must promote building a common language to be used between domain experts and developers, and between the developers themselves, with the main concepts of the domain of knowledge that the system works with, and that is the language used to express how the different target problems of our system are solved. By using a common language to communicate among ourselves, we promote the transfer of knowledge from the domain experts to the developers, which allows them to implement a domain model that is much deeper. The good models are obtained when the developers have a deep knowledge of the domain they are modeling, and this knowledge is only acquired through time and communication with the domain experts. For this reason, the use of a common language is absolutely necessary.

www.EBooksWorld.ir

## 6.2.- Practices that help get a good domain model.

The key point for a successful project is the effective transfer of knowledge from the domain experts to the developers of the system. To enable this knowledge transfer we can use several known development techniques.

### 6.2.1.- Behavior Driven Development (BDD)

BDD is a practice that can be applied within any methodology, consisting of a description of the requirements as a set of tests that can be automatically executed. BDD helps transfer of knowledge by causing the main concepts of domain in the requirements to pass directly to the code, highlighting its importance inside the domain and creating a context or base for building of the model.

### 6.2.2.- Test Driven Development (TDD)

TDD is a practice that can be applied within any methodology, consisting in the development of a set of tests that serve to specify and justify the need to create a component to implement a given functionality. These tests allow us to define the shape of the component itself, and research the relationships of the component with other components of the domain, promoting model development.

www.EBooksWorld.ir

CHAPTER 3

# N-Layered Architecture

## 1.- N-LAYERED APPLICATIONS ARCHITECTURE

### 1.1.- Layers vs. Tiers

These terms are historically commonly used almost interchangeably in the industry but we want to adopt/propose this stronger distinction because we believe it is useful. From our point of view, it is important to distinguish between the concepts of Layers and Tiers.

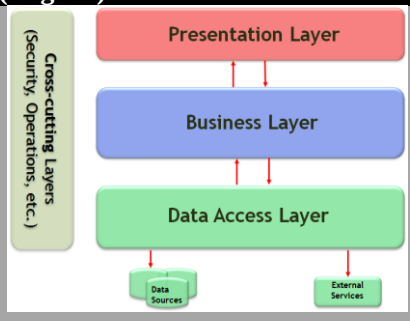
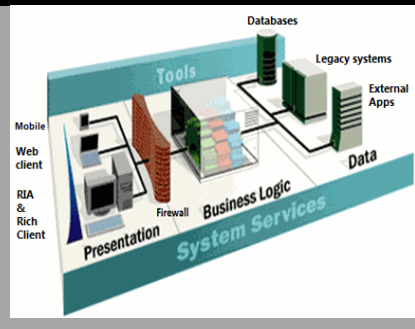
Layers refer to the logical division of components and functionality, and not to the physical location of components in different servers or places. Conversely, the term Tiers refers to the physical distribution of components and functionality in separate servers, including the network topology and remote locations. Although both Layers and Tiers use similar sets of names (presentation, services, business and data), it is important not to mistake one for the other. Remember that only the term Tiers implies a physical separation, and is usually used to refer to physical distribution patterns such as "2 Tier", "3 Tier" and "N-Tier".



www.EBooksWorld.ir

Below we show a **3-Tier** scheme and an **N-Layer** scheme where you can see the differences discussed (logic vs. physical location):

**Table 1.- N-Tier vs. N-Layer**

| Traditional N-Layer architecture (Logical) | 3-Tier Architecture (Physical) |
|---|---|
|  |  |
| **Figure 1.- Traditional N-Layer architecture (Logical)** | **Figure 2.- Tier architecture (Physical)** |





Finally, it is worth noting that all applications containing a certain level of complexity should implement a logical architecture of the N-Layer type, since it provides the correct logical structure; however, not all of these applications should be implemented in a *N-Tier* mode unless they require a physical separation of tiers, as is the case with many web applications.

Note that 3-Tier architectures are quite old nowadays, but still applicable. Of course, internal technologies have changed significantly. In fact, the 3-Tier image we show (Figure 2) is an old Microsoft diagram from 1998 approximately.

## 1.2.- Layers

### Context

The design of a complex business application consists of a considerable number of components at different levels of abstraction.

### Problem

The problem is how to structure an application to support complex operational requirements with good maintainability, reusability, scalability, strength and security.

www.EBooksWorld.ir

**Related aspects**

When structuring an application, the following "forces" should be reconciled within the context of the application's environment:

- Changes in one part of the solution should impact other parts minimally, reducing the work required to repair defects, enhancing application maintenance and improving the overall flexibility of the application.

- Separation of responsibilities/concerns between components (for example, separating the user interface from the business logic, and the business logic from the access to the database) also increases flexibility, maintainability and scalability.

- To ensure stability and quality, each layer must have its own unit testing.

- Certain components must be reusable in different modules of the application or even across different applications.

- Development teams should be able to work on different parts of the solution with minimal dependence on other teams working on other parts of the solution, and to that end, they should develop well defined counter interfaces.

- Individual components must be cohesive.

- Components that are not directly related must be loosely coupled.

- The different components in a solution must be able to be deployed independently, and maintained and updated at different times.

Layers are viewed as logical groups of software components, stacked horizontally, that compose an application or service. They help us differentiate between the various types of tasks performed by components, offering a design that maximizes reuse and enhances maintainability. In short, this is about applying the principle of SoC (*Separation of Concerns*) within the Architecture.

Each first-level logical layer may contain a number of components grouped together as sub-layers, where each sub-layer performs a specific task. The generic component types used in these layers form a recognizable pattern in most solutions. We can use these patterns as a model for our design.

Dividing an application into separate layers, with different roles and functionalities, improves maintenance. It also enables different types of deployment, while providing a clear delineation for the location of each type of functional component and technology.

www.EBooksWorld.ir

### Basic Design of Layers

First of all, bear in mind that when referring to a 'basic design of Layers', we are not speaking of a DDD N-Layered architecture. Rather, we are speaking of a traditional N-Layered architecture (which is simpler than a DDD N-Layered Architectural style).

As already stated, the components of each solution should be separated in layers. The components within each layer should be cohesive and have approximately the same level of abstraction. Each first-level layer should be loosely coupled with other first-level layers as follows:

Start with the lowest level of abstraction, such as "1-Layer". This is the base layer of the system. These abstract steps are followed by other layers (N-Layer, N-1 Layer) until the last level (N-Layer):

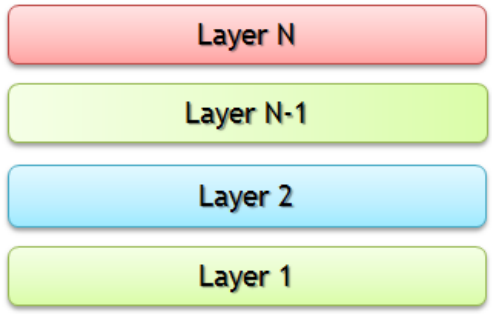



**Figure 3.- Basic design of layers**

The key to an N-layer application lies in management of dependencies. In a traditional N-layered architecture, the components of a layer can only interact with components of the same layer or lower layers. This helps to reduce dependencies between components in different levels. Normally there are **two approaches to the N-layer design: Strict and Flexible.**

A **'Strict layer design'** limits the components of one layer to communicate only with the components of this same layer, or the layer immediately below. In the figure above, if we use the strict system the N layer can only interact with the components of the N-1 layer, the N-1 layer only with the components of the N-2 layer, and so on.

A **'Flexible layer design'** allows the components of a layer to interact with any lower level layer. Using this approach, the N layer can interact with the N-1, N-2 and 3- layers.

The use of a flexible approach can improve performance because the system does not need to make redundant calls to other layers. However, the use of a flexible approach does not provide the same level of isolation between the different layers, and makes it more difficult to replace a lower level layer without affecting multiple higher level layers.

In large and complex solutions involving many software components, it is common to have a great number of components at the same level of abstraction (layer). These, however, are not cohesive. In this case, each layer must be separated into two or more

www.EBooksWorld.ir

cohesive sub-systems known as Modules, vertically grouped within each horizontal layer. The concept of a Module is explained later in this chapter as part of the proposed frame architecture.

The following UML diagram represents the layers composed, in turn, by multiple sub-systems:

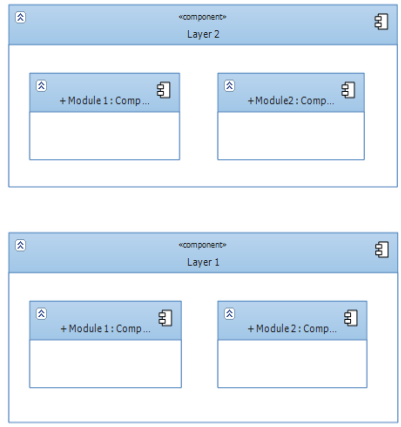



**Figure 4.- Multiple sub-systems in each layer**

### Testing Considerations

An N-Layered application considerably improves the ability to properly implement tests:

- Due to the fact that each layer interacts with other layers only through well-defined interfaces, it is easy to add alternative implementations to each layer (e.g. *Mock* and *Stubs*). This enables unit testing in one layer when the dependent layers are not completed, or when the intention is to execute a very large set of unit tests faster, and accessing dependent layers has dramatically reduced the speed of execution. Also, isolating layer components by using mocks and stubs, it limits the causes a given test will succeed or fail. Therefore we can really test our own logic without considering external factors. This is really Unit Testing. Other than that, we will be performing integration tests. This capability is enhanced if we use base classes ('*Layered Supertype*' pattern) and base interfaces ('*Abstract Interface*' pattern), because they further limit the dependencies between layers. It is especially important to use interfaces because they support more advanced de-coupling techniques, which will be introduced later.

- It is much easier to perform tests on individual components, since the components of high-level layers can only interact with the ones at a lower level. This helps to isolate individual components for proper testing, and makes it

www.EBooksWorld.ir

easier to change components in lower layers with little impact on the rest of the application (as long as the same interface requirements are met).

**The Benefits of using Layers**

- Maintaining a solution is much easier when functions are localized. Loosely-coupled layers with high internal cohesion make it easy to vary the implementations/combinations of layers.

- Other solutions are able to reuse functionality exposed by the different layers when they are designed in this manner.

- Distributed development is much easier to implement when the work is divided into logical layers.

- Distributing layers into different physical levels can, in some cases, improve scalability; however this step should be carefully evaluated because it can negatively impact performance.

**References**

***Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerland, Peter; and Stal, Michael***. *"Pattern-Oriented Software Architecture, Volume 1: A System of Patterns" Wiley & Sons, 1996.*

***Fowler, Martin***. *"Patterns of Application Architecture". Addison-Wesley, 2003.*

***Gamma, Eric; Helm, Richard; Johnson, Ralph; and Vlissides, John***. *"Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995.*

## 1.3.- Basic design principles to be followed

When designing a system, there is a set of fundamental design principles that will help you create an architecture that conforms to proven practices. The following key principles help minimize maintenance costs, maximize usability, and enhance extensibility.

www.EBooksWorld.ir

## 1.3.1.- 'SOLID' Design Principles

The acronym SOLID is derived from the following phrases/principles in English:


**Figure 5.- SOLID Design Principles**

We summarize the SOLID design principles below:

- **Single Responsibility Principle:** Each class should have a unique responsibility or main feature. In other words, one class should have only one reason that justifies performing changes on its source code. One consequence of this principle is that, in general, classes should have few dependencies on other classes/types.

- **Open Closed Principle:** A class must be open for extension but closed for modification. That is, the behavior of a class should be able to be extended without requiring modifications on its code.

- **Liskov Substitution Principle:** Sub-types must be able to be replaced by their base types (base class or interface). This stems from the fact that the behavior of a program that works with abstractions (base classes or interfaces) should not change because a specific implementation is replaced with another one. Programs should refer to abstractions, and not to implementations. We will see later that this principle is closely related to the Dependency Injection and substitution of some classes for others providing they meet the same interface.

- **Interface Segregation Principle:** The class interface implementers should not be obliged to implement methods that are not used. This means the class interfaces should be specific depending on who consumes them and should therefore be split into different interfaces, and no large interfaces should be created. Classes should expose separate interfaces for different clients/consumers with differing interface requirements.

- **Dependency Inversion Principle:** Abstractions should not depend on details; the details should depend on abstractions. Direct dependencies between classes

www.EBooksWorld.ir

should be replaced by abstractions (interfaces) to allow top-down designs without the need to design the lower levels first.

## 1.3.2.- Other key design principles

- **The component design must be highly cohesive:** do not overload the components by adding mixed or non-related functionality. For example, avoid mixing data access logic with business logic belonging to the Domain model. When functionality is cohesive, we can then create assemblies with more than one component and place them in the appropriate layers of the application. This principle is therefore closely related to the "N-layer" pattern and to the 'Single Responsibility Principle'.

- **Keep the cross-cutting code abstracted from the application-specific logic**: the cross-cutting code refers to horizontal aspect code, such as security, operations management, logging, instrumentation, etc. The combination of this type of code with specific implementation of the application can lead to designs that, in the future, are difficult to extend and maintain. The AOP (Aspect Oriented Programming) principle is related to this.

- **Separation of Concerns:** dividing the application in different parts and minimizing the overlapping functionalities between such parts. The key factor is to minimize the interaction points to achieve high cohesion and low coupling. However, separating functionality in the wrong boundaries can lead to a high degree of coupling and complexity among the characteristics of the system.

- **Don't Repeat Yourself (DRY)**: the "intention" must be specified in only one part of the system. For example, in terms of an application design, a specific functionality should be implemented in only one component; the same functionality should not be implemented in other components.

- **Minimize the top-down design (Upfront Design)**: design only what is necessary, do not perform "over engineering" and take into account **the agile YAGNI principle** (*'You Ain't Gonna Need It'*).

www.EBooksWorld.ir

## 1.4.- Orientation to DDD architecture trends (Domain Driven Design)

The purpose of this architectural framework is to provide a consolidated base and a set of guidelines for a specific type of application: **'Complex Business Applications'**. This type of application is characterized by having a relatively long life span and can absorb a considerable amount of evolutionary change. Ongoing maintenance is very important in these applications, including updating/replacing technologies and frameworks such as ORM (Object Relational Mapping) with more modern versions. The objective is that all of these changes can be performed with minimal impact to the rest of the application. Changes in infrastructure technology of an application should not affect high-level layers in the application. Specifically, the "Application Domain" layer should be affected to the least extent possible.

In complex applications, the behavior of business rules (Domain logic) is often subject to change, so it is important to retain the ability to modify, build and perform tests on domain logic layers in an easy and independent manner. Achieving this important goal requires minimal coupling between the Domain Model (logic and business rules) and the rest of the system layers (Presentation layers, Infrastructure layers, data persistence, etc.)

Trends in application architecture are increasingly oriented to achieve this de-coupling between layers, especially as regards the Domain Model layer. Domain Oriented N-Layered Architectures, as part of the DDD (*Domain Driven Design*), focus on this objective.

***DDD (Domain Driven Design) is, however, much more than just a proposed Architecture; it is also a way of dealing with projects, a way of working as the development team, the importance of identifying an "Ubiquitous language" whose planning is based on knowledge of domain experts (business experts), etc. However, these aspects of DDD are not addressed in this guide; our scope is limited to logical and technological Architecture. Please refer to DDD-related books and other sources of information for further details.***

**Reasons why you should NOT adopt Domain Oriented N-Layer Architectures**

If the application to be developed is relatively simple, you do not foresee changes in infrastructure technology during the life of the application and, above all else, the business rules automated in the application will change very little, then your solution probably shouldn't follow the type of architecture presented in this guide. Instead you should consider a RAD (Rapid Application Development) development/technology. Rapid implementation technologies can be very effective in building simple applications where the de-coupling between components and layers is not particularly

www.EBooksWorld.ir

relevant, and the emphasis is on productivity and *time to market*. Typically, these kinds of apps are Data Driven Applications, rather than Domain Driven Designs.

**Reasons to adopt a Domain Oriented N-Layered Architecture (DDD Architectural Styles)**

Whenever you expect the business behavior to evolve over time, you should strongly consider using the "DDD N-Layered Architecture style". In these cases a "Domain Model" will reduce the effort required for each change, and the TCO (Total Cost of Ownership) will be much lower than if the application had been developed in a more coupled manner. In short, encapsulating the business behavior in a single area of your software dramatically reduces the amount of time needed to modify the application. This is because changes will be performed in only one place, and can be conveniently tested in isolation. Being able to isolate the Domain Model code as much as possible reduces the chances of having to perform changes in other areas of the application (which can always be affected by new problems, regressions, etc.). This is vitally important if you want to reduce and improve stabilization cycles and solution commissioning.

**Scenarios where the Domain Model is effective**

Business rules indicating when certain actions may be performed are good candidates for implementation in a domain model.

For instance, in a business system, a rule specifying that a customer cannot have more than $2000 of outstanding payments should probably belong to the domain model. Implementations of rules like this usually involve one or more entities, and must be evaluated in the context of different use cases.

Thus, a domain model will have many of these business rules, including cases where some rules may replace others. For example, regarding the above rule, if the customer is a strategic account, this amount could be much higher, etc.

In short, the greater importance business rules and use cases have in an application is precisely the reason to orient the architecture towards the Domain Model and not simply define entity relationships as in a data oriented application.

Finally, in order for information to persist by converting sets of objects in memory (objects/entities graphs) to a relational database, we can use some data persistence technology of the ORM type (*Object-Relational Mapping*), such as *Entity Framework* or *NHibernate*. However, it is very important that these specific data persistence technologies (infrastructure technologies) are well separated and differentiated from the application business behavior, which is the responsibility of the Domain Model. This requires an N-Layered architecture, which should be integrated in a de-coupled manner, as we will see later.

www.EBooksWorld.ir

## 1.5.- DDDD (Distributed Domain Driven Design)

Four 'Ds'? Well, yes, it is clear that DDDD is an evolution/extension of DDD where distributed systems aspects are added. Eric Evans, in his book about DDD almost entirely avoids the issue on distributed technologies and systems (Web services, etc.) because it is mainly focused on the Domain. However, the distributed systems and remote Services are something we need in most scenarios.

Actually, this proposed N-Layered architecture is **DDDD**-based, since we consider the Distributed Services Layer from the very beginning, and we even map it afterwards for implementation with Microsoft technology.

Ultimately, this fourth D added to DDD brings us closer to distributed scenarios, great scalability and even scenarios that would normally be closer to '*Cloud-Computing*' by affinity.

# 2.- DDD N-LAYERED ARCHITECTURE STYLE

As stated, we want to make it clear that we are speaking of "Domain Oriented" architecture, and not about everything covered by DDD (*Domain Driven Design*). To talk about DDD we should be focused, not only on the architecture (the goal of this guide) but rather on the design process, on the way the development team works, the "ubiquitous language", etc. These aspects of the DDD will be briefly addressed herein. **The purpose of this guide is to focus exclusively on an *N-Layer Architecture* that fits with DDD, and how to "map it" later to the Microsoft technologies. We do not intend to expound and explain DDD, since there are great books dedicated to this.**

This section provides a global definition of our proposed Domain Oriented N-layered Architecture as well as certain patterns and techniques to consider for integration of such layers.

## 2.1.- Presentation, Application, Domain and Infrastructure Layers

At the highest and most abstract level, the view of the logical architecture of a system can be deemed as a set of related services which are grouped in several layers, similar to the following diagram (following the trends of DDD Architecture):

www.EBooksWorld.ir

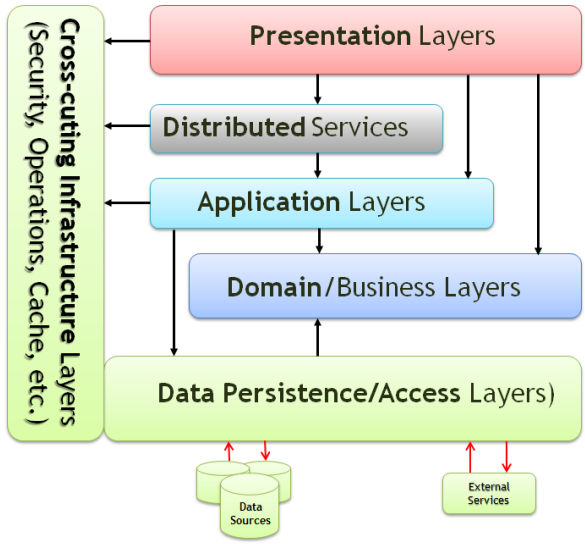



**Figure 6.- Simplified view of logical architecture of a DDD N-Layers system**

In "Domain oriented architectures" it is crucial to clearly delineate and separate the Domain Layer from the rest of the layers. It is really a pre-requisite for DDD. "*Everything must revolve around the Domain*".

Hence, a complex application should be partitioned into layers. A design should be developed within each layer. This design must be cohesive but clearly define the different layers between them, applying standard patterns of Architecture so that such dependencies are mostly based on abstractions and do not refer one layer directly to another. All the code related to the domain model should be focused on one layer, isolated from the rest of the codes of other layers (Presentation, Application, Infrastructure and Persistence, etc.). The Domain should not require having to show itself, persist/save itself, manage application tasks, etc. It can then focus exclusively on expressing the domain model. This allows a domain model to evolve and become rich and clear enough to represent the essential business knowledge and achieve the business requirements within the application.

Separating the domain layer from the rest of the layers also allows a much cleaner design for each layer. The isolated layers are much easier to maintain because they tend to evolve at different rates and respond to different needs. For example, the infrastructure layers will evolve when the technologies they are based on evolve. On the other hand, the Domain layer will evolve only when you want to make changes to the business logic of the particular domain.

Furthermore, *the separation of layers aids in the deployment of a distributed system, which allows different layers to be located flexibly on different servers or clients, so as to minimize the over-communication and improve performance (Quote of M. Fowler).*

**Loose-coupled integration between layers components is essential.** Each layer of the application will have a series of components implementing the functionality of that

www.EBooksWorld.ir

layer. These components should be internally cohesive (within the same first-level layer), but some layers (such as Infrastructure/Technology layers) should be loosely coupled with the rest of the layers in order to empower unit testing, *mocking* and reuse, to reduce impact on maintenance. The design and implementation that leads to this loose-coupling among the main layers will be explained in more detail later.

## 2.2.- Domain Oriented N-Layered Architecture

The aim of this architecture is to structure, clearly and simply, the complexity of a business application based on the different layers, following the N-Layered pattern and the trends in DDD architecture. The N-Layered pattern distinguishes different internal layers and sub-layers in an application.

Of course, this particular N-Layered architecture is customizable according to the needs of each project and/or preferences of Architecture. We simply propose following an architecture that serves as a baseline to be modified or adapted by architects according to their needs and requirements.

Specifically, the proposed layers and sub-layers for *"Domain oriented N-Layered"* applications are the following:

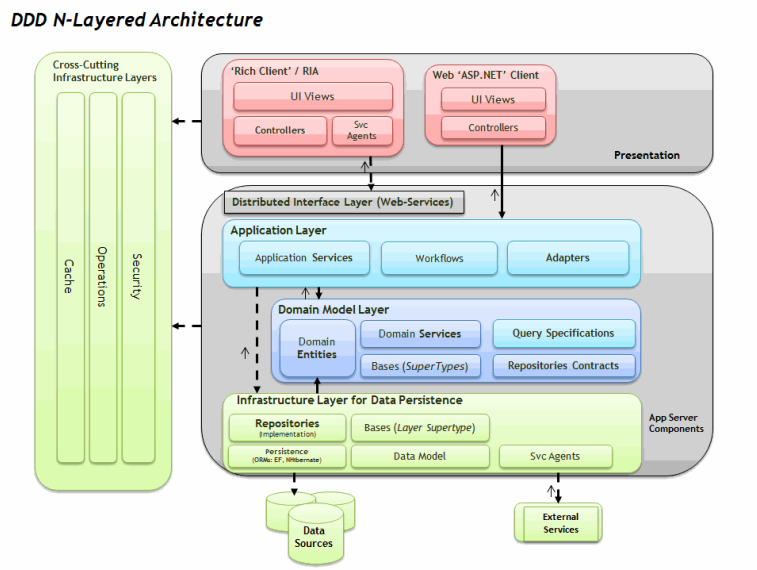



**Figure 7.- Domain Oriented N-Layered Architecture**

www.EBooksWorld.ir

- **Presentation layer**
  - Sub-layers of visual components (Views)
  - Sub-layers of user-interface logic (Controllers and the like)
- **Distributed services layer (Web services)**
  - Web services publishing the Domain and Application layers
- **Application layer**
  - Application services (Tasks and Use Case Coordinators)
  - Adapters (Format Converters, etc.)
  - *Workflows* sub-layer (Optional)
  - Application layer base classes (*Layer-Supertype Pattern*)
- **Domain model layer**
  - Domain entities
  - Domain services
  - Query specifications (Optional)
  - *Repository* Interfaces/Contracts
  - Domain Base Classes (*Layer-Supertype Pattern*)
- **Data persistence infrastructure layer**
  - Repository implementation
  - Logical Data Model
  - Base classes (*Layer-Supertype Pattern*)
  - ORM technology infrastructure
  - External Service Agents
- **Cross-cutting components/aspects of the Architecture**
  - Horizontal aspects of Security, Operations Management, Monitoring, Automated Email, etc.

www.EBooksWorld.ir

All these layers are briefly explained herein and an entire chapter will be dedicated to each later on. However, before that, it is interesting to know, from a high level perspective, what the interaction between such layers looks like, and why we have divided them this way.

One of the sources and main precursors of DDD is *Eric Evans*, who in his book "*Domain Driven Design – Tackling Complexity in the Heart of Software*" describes and explains the following high level diagram of the proposed N-Layered architecture:

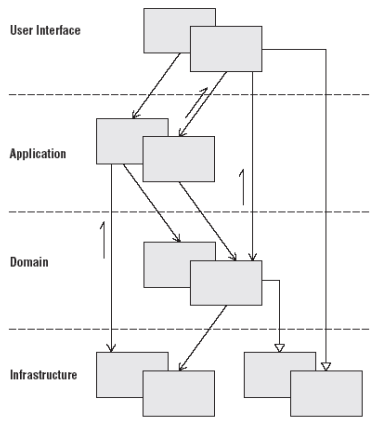



**Figure 8.- DDD N-Layered architecture diagram [Eric Evans version]**

It is worth noting that, in some cases, the access to other layers is straightforward. That is, there is no reason why there should be only one single path going from one layer to another, but that will depend on each case. To illustrate those cases, we show the previous diagram of Eric Evans below. The diagram has been modified and contains more details, whereby it is related to the lower level sub-layers and elements proposed in our Architecture:

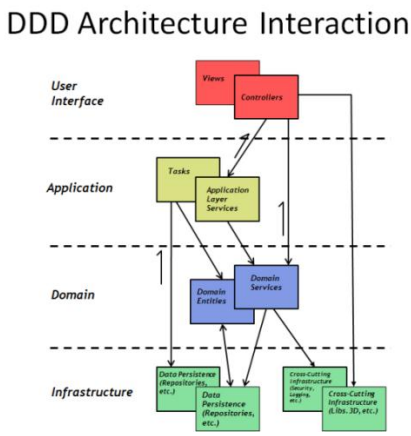



**Figure 9.- DDD Architecture interaction**

www.EBooksWorld.ir

First, we can see that the **Infrastructure Layer,** featured by a DDD architecture style, is very broad and provides for many different contexts, (Server and Client Contexts). The infrastructure layer will contain everything related to technology/infrastructure. There are fundamental concepts such as Data persistence (Repositories, etc.) included therein, as well as cross-cutting issues such as Security, Logging, Operations, etc. It could even include specific libraries of graphic capacities for UX (3D libraries, specific control libraries for a particular presentation technology, etc.). Due to these huge differences in context and the importance of data access, in our proposed architecture we will specifically separate the **Data Persistence Infrastructure Layer** from the rest of the infrastructure layers, normally **Cross-Cutting Infrastructure Layers,** which can be used in a horizontal/cross-cutting manner by any layer.

The other interesting aspect we anticipated in advance is that access to some layers is not only through a single path ordered by different layers. Specifically, we can directly access the Application, Domain and Cross-Cutting Infrastructure layers, as necessary. For example, we can access directly from a Web Presentation Layer (this does not require remote interfaces of the Service-Web type) to the lower layers we need (Application, Domain and certain aspects of Cross-Cutting Infrastructure). However, to reach the Data Persistence Layer and its Repository objects (in some respects this might remind you of the traditional Data Access Layer – DAL – but it is not the same), it is always recommended to access through coordination objects (Services) of the Application layer, since this is the module that coordinates them.

It is worth mentioning that implementation and use of all these layers should be somewhat flexible. There should probably be more combinations of arrows (shortcuts) in the diagram. Above all, it does not need to be used exactly in the same way in all applications.

Later in this chapter we will briefly describe each of the layers and sub layers mentioned earlier. We will also present some overall concepts on how to define and work with these layers (e.g. loose-coupling between layers, deployment in different physical levels, etc.).

Subsequently, in the following chapters, we will explain each high-level layer in detail (One chapter for each high-level layer).

**Presentation layer**

The purpose of this layer is to present information to the user and interpret his actions.

The components of the presentation layers implement the functionality required for users to interact with the application. It is generally recommended to divide these components into several sub-layers using patterns such as MVC, MVP or MVVM:

- **Visual components sub-layer (Views):** These components provide the basic mechanism for the end-user to use the application. These components format data in terms of visual controls and also obtain data provided by the user.

www.EBooksWorld.ir

- **Controller sub-layer:** It may be useful to conduct the process using separate components from the actual GUI components to help synchronize and direct user interactions. This prevents the process flow and state management logic from being programmed within the individual controls and visual forms and allows us to reuse the logic and patterns and isolate them from other interfaces or "views". It is also very useful for performing unit testing of the presentation logic. These controllers are typically implemented based on the MVC patterns and derivatives.

**Distributed services layer (Web services)**

When an application acts as a service provider for other remote applications or even when the presentation layer is also physically located in remote locations (*Rich-Client*, RIA, OBA applications, etc.), the business logic (internal business layers) is normally published through a *distributed services layer*. This distributed services layer (usually Web services) provides a means of remote access based on communication channels and data messages. It is important to remember that this layer should be as light as possible and should not include business logic.

**Application layer**

This layer is a part of the proposed Domain-Oriented Architecture. It defines tasks that the application itself must perform in order to coordinate use cases of the application, therefore, it coordinates domain and infrastructure objects (data persistence, etc.), which are those that should resolve problems internally.

Actually, this layer should not contain domain rules or business logic knowledge; it should simply perform coordination tasks of the technological features of the application that we would never explain to a domain expert or business user. Here we implement the coordination of the application "plumbing", such as transaction coordination, execution of units of work, and ultimately, calls to tasks necessary for the application (*software*). Other features to be implemented here can be optimizations of the application, data/format converters, etc., but we always refer to them as coordination. The final work will be subsequently delegated to objects of lower layers. This layer should not contain states reflecting the situation of the internal business logic either, but it may have a state reflecting the progress of an application task in order to show that progress to the user.

This layer is somewhat similar to a "Business facade" layer, since it will work as a facade for the Domain model. But it is not only in charge of simplifying access to the Domain, but also does other things. Features to be included in this layer are:

- Coordination of most calls to Repository objects of the Persistence layer and data access.

- Grouping/merging data of different entities to be sent more efficiently (minimizing remote calls) by a higher layer of web services. These objects to be sent are DTOs (*Data Transfer Objects*) and the code in the application layer to

www.EBooksWorld.ir

transform from domain entities to DTOs and vice versa are called *DTO-Adapters*.

- Actions that consolidate or group Domain operations *depending on the actions shown in the user interface*, associating such actions to the persistence operations and data access.

- Maintenance of states related to the application (not internal states of the Domain objects).

- Coordination of actions between the Domain and the infrastructure layers. For example, performing a bank transfer requires getting data from the data sources through the use of Repositories, then using domain objects with the bank-transfer business logic (payment and charge) and perhaps, eventually, sending an email to the stakeholders, which invokes another infrastructure object that sends the email.

- **Application services**: It is important to note that the concept of Service in domain-oriented N-Layered architectures does not have any connection with the Web services for remote access. First, the concept of DDD service exists in different layers: Application, Domain or even Infrastructure layers. **The concept of services is simply a set of classes where behaviors and action methods** that do not belong to a certain low level class (such as domain services and entities) are grouped. Therefore, the services will normally coordinate objects of the lower layers.
  "Application Services" in particular, are the services that normally coordinate the work of other lower layer services (Domain layer services or even cross-cutting infrastructure layer services). For example, an application service layer can call a domain service layer to perform the logic to create an order in entities in memory. Once such business operations are performed by the Domain layer (most of them are changes to the in memory objects), the application layer can call infrastructure Repositories delegating the task of persisting changes in the data sources. This is an example of lower layer services coordination.

- **Business workflow (Optional):** Some business processes consist of a certain number of steps that should be implemented in accordance with specific rules, depending on events that can occur in the system and normally with a **long total runtime** (indeterminate, in any event).Some steps interact with others through an orchestration that depends on such events. These types of business processes are naturally implemented as workflows through specific technologies and business process management tools specially designed for that purpose.

This Application layer can also be published through a higher layer of web services, so that it can be invoked remotely.

www.EBooksWorld.ir

**Domain layer**

This layer is responsible for representing business/domain concepts, information on the status of business processes, and implementation of domain rules. It should also contain the states reflecting the status of the business processes.

**This layer, the 'Domain', is the heart of the software.**

Hence, these components should implement the core domain-functionality of the system and encapsulate all the relevant business logic (generally known as Domain Logic according to the DDD terminology). Basically, they are usually classes that implement the domain logic within its methods. Following the Domain oriented N-Layer Architecture patterns, this layer should be completely unaware of the data persistence details. These persistence tasks should be performed by the infrastructure layers and coordinated by the Application layer.

Normally we can define the following elements within the Domain layer:

**Domain Entities**: These objects are disconnected entities (data + logic) and are used to host and transfer entity data between different layers. But in addition, a fundamental feature in DDD is that they also contain the domain logic related to each entity. For example, in the case of a bank account, the operation of adding an amount of money to the account balance should be performed with logic within the Bank Account entity itself. Other examples include data validation according to the business logic, calculated properties, associations with other entities, etc. Ultimately, these classes represent the real world business entities. On the other hand, the data entities used internally by the application are objects in memory with data and some related logic. If we use entities as "just data", with no logic of their own situated within them then we will fall into an anti-pattern called '*Anemic Domain Model'*, originally described by *Martin Fowler*. In addition, it is a recommended pattern that these classes should also be POCO entities (*Plain Old CLR Objects*), that is, classes independent from any specific data access technologies or frameworks. The ultimate goal of this design (*Persistence Ignorance*) is for domain classes to "know nothing" about the inside of the repositories or data access technologies.

The entity classes are located within the domain layer, since they are domain entities and are independent from any infrastructure technology (data persistence, ORM, etc). In any case, the entities are objects floating through all or most of the architecture.

Regarding DDD definitions, and in accordance with Eric Evans, "*An object that is primarily defined by its identity is called Entity*". Entities are fundamental concepts in the Domain model and must be carefully identified and designed. What may be an identity in some applications may not be an identity in others. For example, an "address" in some systems may not have any identity at all; they may represent only attributes of a person or company. In other systems, however, such as an application for an Electricity company, the customer's address can be very important and should be an entity, because billing needs to be directly connected to the address. In this case, an address might be classified as a Domain Entity. In other cases, such as an e-commerce

www.EBooksWorld.ir

application, the address can simply be an attribute of a person's profile. In this case, the address is not so important and should be classified as a 'Value-Object', as it is called in DDD patterns.

- **Domain Services:** In Domain layers, services are basically classes that group behaviors and/or methods of execution of the domain logic. These classes, in general, should not contain states related to the domain (they should be stateless classes). They will be classes that coordinate operations composed by the domain entities. A typical case of a Domain Service is that it is related to several entities at the same time. But we can also have a Service in charge that is interacting (retrieving, updating, etc.) through a single root entity (which can embed other related data/entities by following the Aggregate pattern which we will explain in following chapters). Regarding the use of Repositories, these will usually be invoked from the Application Services, especially when executing transactions and using a UoW (Unit of Work pattern, which will be explained in following chapters). But sometimes, from the Domain Services, we will need to get data from Repositories depending on certain Domain logic. In that case (queries, most of all), it is ok to use repositories from the Domain Services.

- **Repository contracts**: It is evident that implementation of the Repositories themselves will not take place in the Domain, since implementation of Repositories is not part of the Domain but part of the Infrastructure layers (the Repositories are linked to data persistence technology, such as an ORM). However, interfaces or 'contracts' of how such Repositories should interact, must be part of the Domain. Such contracts show what each Repository should offer in order to work with the specified Domain, no matter how they are internally implemented.
  These interfaces/contracts should be "agnostic" to the underlying technologies. On the other hand, the classes implementing those interfaces will work directly with certain technologies. It is therefore important that the Repository interfaces/contracts be defined within the Domain layers. This is one of the patterns recommended in Domain oriented architectures and is based on the '*Separated Interface Pattern*' defined by *Martin Fowler*.
  Logically, to be able to comply with this rule, '*Domain Entities*' and '*Value-Objects*' need to be POCO; that is, objects in charge of hosting entities and data should also be fully agnostic to the data access technologies. It must be considered that the domain entities are, ultimately, the "types" of parameters sent to and received by the Repositories.

**Data Persistence Infrastructure Layer**

This layer provides functionality in order to access data. It can be data from our system or data exposed by external systems (external Web Services, etc.). Thus, this data persistence layer exposes the data access to higher layers, normally application and domain layers. This exposure should be performed in a de-coupled manner.

www.EBooksWorld.ir

- **Implementation of 'Repositories':** In generic terms, the Repository "*Represents all the objects of a certain type as a conceptual group*" (Definition by *Eric Evans*). On a practical level, a Repository will normally be a class in charge of performing persistence and data access operations and is therefore related to a specific technology (e.g. linked to an ORM such as *Entity Framework*, *NHibernate*, or even just *ADO.NET* for a particular relational database provider). By doing this, we centralize the data access functionality, which makes maintenance and setup of the application easier and more direct.
  Normally, we must create a *Repository* for each 'Root Domain Entity.' It is almost the same as saying that the relationship between a *Repository* and a root entity is 1:1. Root entities can sometimes be isolated and other times they are the root of an '*Aggregate,'* which is a set of entities, '*Values-Objects*' as well as the root entity itself.
  The access to a Repository should be performed through a well-known interface, a contract that is "deposited" in the Domain, so that we can replace one Repository with another one that is implemented with other technologies and the Domain layer will not be affected as a result.
  The key point of the Repositories is that they should make it easier for the developer to keep the focus on the Domain model logic and therefore hide the data access "plumbing" through such repository "contracts". This concept is also known as '*PERSISTENCE IGNORANCE,'* which means that the Domain model fully ignores how data are persisted or queried against data sources (Databases or other types of storage).
  Finally, **it is essential to differentiate between a "Data Access" object (used in many traditional N-Layer architectures) and a Repository**. The main difference lies in the fact that Data Access objects directly perform persistence and data access operations against the storage (normally a database). However, a Repository marks/saves objects in the memory (a context) as well as the operations it intends to perform, but they will not be performed. Later, from the Application layer, these persistence/data access operations will actually be performed in one single action all at once. This is normally based on the "*Unit of Work*" pattern, which will be explained in detail in the "Application Layer" chapter. In many cases, this pattern or how to implement/carry out operations against storage can increase application performance. In many cases it can also reduce the possibilities of inconsistencies. It also reduces database blockages originated by transactions.

- **Base components (*Layer Supertype pattern*):** Most data access tasks require a certain common logic that can be extracted and implemented in a separate and reusable component. This helps to simplify the complexity of the data access components, and above all, minimizes the code volume to be maintained. These components can be implemented as base classes or utility classes (depending on the use case) and the code can be reused in different projects. This concept is actually a very well-known pattern called '*Layered Supertype Pattern'* defined by *Martin Fowler*, which basically says that "If behaviors and common actions of similar classes are grouped in a base class, this will eliminate many

www.EBooksWorld.ir

behaviors and code duplications". The use of this pattern is purely for the sake of convenience and does not distract attention from the Domain at all.
The '*Layer Supertype Pattern'* can be applied to any type of layer (Domain, Application, Infrastructure, etc.) and not only to Repositories.

- **Data model**: Normally ORMs (such as *Entity Framework)* have data model definition mechanisms like entity diagrams, even at a visual level. This sub-layer should contain these entity models, with visual interfaces and diagrams if possible.

- **Remote/external Service Agents:** Sometimes a business component needs to use functionality provided by external/remote services (Typically Web Services). In those scenarios, a component should be implemented to manage the communication semantics with that particular service or even to perform additional tasks such as mapping between the different data formats. The Service Agents isolate such idiosyncrasy so that by defining certain interfaces, it would be possible to replace the original external service with a different second service, and our core system would not be affected.

**Cross-Cutting Infrastructure Layers**

These provide generic technical capabilities used by other layers. Ultimately, they are "building blocks" related to particular technologies to leverage their functions.

There are many tasks implemented in the codes of an application that should be applied in different layers. These tasks or cross-cutting (Horizontal) aspects implement specific types of functionality that can be accessed / used from components of any layer. The most common cross-cutting aspects are: **Security** (Authentication, Authorization and Validation) and **Operation Management Tasks** (policies, *logging*, traces, monitoring, etc.). These aspects will be provided in detail in the following chapters.

- **Cross-Cutting Infrastructure Services:** The concept of Services also belongs to the cross-cutting infrastructure layers. These will be in charge of grouping infrastructure actions, such as sending emails, monitoring security issues, operations management, logging, etc. Thus, these **Services** are in charge of grouping any type of cross-cutting infrastructure activity related to specific technologies.

- **Cross-Cutting Infrastructure objects**: Depending on the type of cross-cutting infrastructure, we will need particular objects to be implemented, whether they are security issues, tracing, monitoring, sending of emails, etc.

These "Cross-Cutting Infrastructure" layers cover many different concepts, and many of them are related to the Quality of Service (QoS) and, actually, to any

www.EBooksWorld.ir

implementation related to a specific technology/infrastructure. This will be defined in more detail in a chapter dedicated to these cross-cutting aspects.

**'Services' as a generic concept available in the different Layers**

Since the SERVICES are present in different layers of a DDD Architecture, we have summarized the concept of SERVICE used in DDD in a special chart below:

**Table 2.- Services in DDD N-Layer Architectures**

| Services in Domain Oriented N-Layer Architectures |
|---|
| As we have seen in the different layers (APPLICATION, DOMAIN AND CROSS-CUTTING INFRASTRUCTURE) all of them can have a sub-layer called Services. Since this is a concept that appears in different areas, it is convenient to have an overall vision on what the Services are in DDD.<br>First, it is important to clarify that DDD-SERVICES are not WEB SERVICES used for remote invocations. WEB SERVICES can be in a higher-level layer called "Distributed Services Layer" and may, in turn, publish the lower layers allowing remote access to the DDD-SERVICES and also to other objects of the Application and Domain Layer.<br>The concept of DDD SERVICE, in the cleanest and most pragmatic designs include operations that do not conceptually belong to particular objects of each layer (e.g., operations that do not belong exclusively to an entity). In these cases we can include (group) such operations in explicit SERVICES.<br>These operations are by nature those activities that are not part of the characteristics of specific objects of each layer. But since our programming model is object oriented, we should group them in objects as well. These objects are what we call SERVICES.<br>The motivation behind this is that forcing such operations (normally high level operations that group other actions) to be part of the natural objects of the layer would distort the definition of real objects of that layer. For example, the logic of an entity should be related to the internal things such as validations with respect to the data in the memory, or calculated fields, etc., but not to the treatment of the entity itself as a whole. For example, an "engine" performs actions related to engine usage, not related to how said engine is manufactured. Likewise, logics belonging to an entity class should not be in charge of its own persistence and storage.<br>Furthermore, a SERVICE is an operation or set of operations offered as an interface. It must not encapsulate states (they must be stateless). This does not imply that the class implementing them must be static; it will usually be an instance class. The fact that a SERVICE is stateless means that a client program can use any instance of the service no matter what the individual object's state is. More to the point, the execution of a SERVICE could use information that is globally accessible and it can even change such information (normally it makes global changes). But the service does not contain states that can affect its own behavior, unlike entities, which do.<br>The word "Service" of the SERVICE pattern precisely emphasizes what it offers: "*What it can do and what operations are offered to the client that uses it and emphasizes the relation with other objects of each layer*".<br>Some SERVICES (mostly the highest level services in the Application layer and/or certain services of the Domain that coordinate the business logic) are usually named after the Action names, not after the object names. Therefore, they are related to the verbs of the analysis Use Cases and not to the nouns (objects), even when there is an abstract definition for a specific operation. (e.g., a "Transfer Service" related to the action/verb "Transfer Money from one bank account to another").<br>To clarify this point, how to partition different Services in different layers in a simplified banking scenario is shown below: |

www.EBooksWorld.ir

| | |
|---|---|
| APPLICATION | *Application service of 'BankingService' (Banking operations)*<br>• *It accepts and converts formats of input data (like XML data conversions)*<br>• *It provides banks-transfer data to the Domain layer so that the business logic is really processed there.*<br>• *It coordinates/invokes persistence objects (Repositories) of the infrastructure layer, to persist changes made by the domain layer on the entities and bank accounts.*<br>• *It decides if the notifications should be sent (email to the user) using the cross-cutting infrastructure services.*<br>• *Ultimately, it implements all the "coordination of technical plumbing" (like using a Unit of Work and transactions) so that the Domain Layer is as clean as possible and expresses its logic better and very clearly.* |
| DOMAIN | ***Domain** service of 'Bank Transfer (Verb Transfer funds)*<br>• *It coordinates the usage of entity objects such as "Bank Account" and other objects of the Banking Domain.*<br>• *It provides confirmation of the result of business operations.* |
| CROSS-CUTTING INFRA-STRUCTURE | ***Cross-cutting Infrastructure** service such as "Sending Notifications" (Verb: Send/Notify)*<br>• *It sends an email, SMS or other types of notifications required by the application.* |

From all the explanations in this chapter so far, you can deduce what could be the first rule to follow in business application development (based on this Architecture guide):

www.EBooksWorld.ir

**Table 3.- D1 Design Rule**

| | |
|---|---|
|  **Rule #: D1.** | **The internal architecture of an application (logic architecture) will be designed based on the N-Layered application architecture model with Domain orientation and DDD trends and patterns (*Domain Driven Design*)** |

**Rules**

- In general, this rule should be applied in almost 100% of the complex business applications that have a certain volume of Domain logic.

**When TO IMPLEMENT a Domain Oriented N-Layered Architecture**

- It should be implemented in complex business applications with a business logic subject to multiple changes where the application goes through changes and subsequent maintenance during a relatively long application life cycle.

**When NOT TO IMPLEMENT a DDD N-Layered Architecture**

- In small applications that, once completed, are expected to have few changes. These types of applications have a relatively short life cycle and the development speed prevails. In these cases implementing the application with RAD technologies (such as 'Visual Studio LightSwitch' and 'Microsoft RIA Services') is recommended. However, this will have the disadvantage of implementing more strongly coupled components, which will result in an application with relatively poor quality. Therefore, future maintenance costs will probably be higher depending on whether the application continues to have a large volume of changes or not.

**Advantages of using N-Layered Architecture**

- Structured, homogeneous and similar development of the different applications within an organization.
- Easy application maintenance because different types of tasks are always situated in the same areas of the architecture.
- Easy change of the topology in the physical deployment of an application (2-Tier, 3-Tier, etc.), since the different layers can be physically separated more easily.

**Disadvantages of using N-Layered Architecture**

- In the case of very small applications, we add excessive complexity (layers,

www.EBooksWorld.ir

loose-coupling, etc.). In this case it might be over-engineered. But this case is very unlikely in business applications with a certain level of complexity.

**References**

***Eric Evans**: Book "Domain-Driven Design: Tackling Complexity in the Heart of Software"*

***Martin Fowler**: Definition of 'Domain Model Pattern' and book "Patterns of Enterprise Application Architecture"*

**Jimmy Nilson**: *Book "Applying Domain-Driven-Design and Patterns with examples in C# and .NET"*

*SoC - Separation of Concerns principle:*
***http://en.wikipedia.org/wiki/Separation_of_concerns***

*EDA - Event-Driven Architecture: SOA Through the Looking Glass – "The Architecture Journal"*

*EDA - Using Events in Highly Distributed Architectures – "The Architecture Journal"*

Although these layers are initially meant to cover a large percentage of **N-Layered** applications architecture, the base architecture is open to introducing new layers and customization necessary for a given application (for example, EAI layer for integration with external applications, etc.).

Likewise, the full implementation of the proposed layers is not mandatory either. For example, in some cases the Web-Services layer may not be implemented because you might not need remote accesses, etc.

## 2.3.- De-coupling between Components

It is essential to note that the components of an application should not only be defined between the different layers; we should also pay special attention to how some components/objects interact with each other, that is, how they are consumed and especially how some objects are instantiated from others.

In general, this de-coupling should be done between all the objects (with dependency and execution logic) belonging to different layers, since there are certain layers that we really want to integrate in the application in a de-coupled manner. This is the case in most of the infrastructure layers (related to some specific technologies), such as the data persistence layer, that we may have linked to a particular ORM solution, or even to a specific external backend (e.g., linked access to a Host, ERP or

www.EBooksWorld.ir

any other business backend). In short, to be able to integrate this layer in a de-coupled manner, we should not directly instantiate its objects (e.g., not directly instantiating Repository objects or any other object related to a specific technology in the infrastructure layers).

This point is essentially about de-coupling between any type/set of objects, whether they are sets of different objects within the Domain (e.g., for a country, client or specific topology, being able to inject some specific classes of business logic), or in the Presentation layer components, being able to simulate functionality/data gathered from Web-Services, or in the Persistence layer, also being able to simulate other external Web-Services, etc. In all these cases, it should be performed in a de-coupled manner in order to replace real implementation with a simulation or another implementation, with the least impact. In all these examples, de-coupling is a very sensible approach.

Finally, we are dealing with achieving the "*state of the art*" in our application's internal design: *"To have the entire application Architecture structured in de-coupled manner which enables us to add functionality to any area or group of objects at any time. This is not necessarily just between different layers".*

Simply "de-coupling between layers" is probably not the best approach. The example of sets of different objects to be added within the Domain itself, which is a single layer (e.g., for a country, client or topology in particular, including a vertical/functional module), clarifies a lot.

In the sample application that comes with this Architecture Guide we have opted to perform de-coupling between most objects of the application's internal layers. Thus, the mechanism is fully available.

The de-coupling techniques are based on the **Principle of Dependency Inversion**, which sets forth a special manner of de-coupling, where the traditional dependency relationship used in object orientation ('high-level layers should depend on lower-level layers'), is inverted. The goal is to have high-level layers independent from the implementation and specific details of the lower level layers, and therefore, independent from the underlying technologies as well.

The **Principle of Dependency Inversion states the following**:

A. High-level layers should not depend on low-level layers. Both layers should depend on abstractions (Interfaces)

B. Abstractions should not depend on details. The Details (Implementation) should depend on abstractions (Interfaces).

The purpose of this principle is to de-couple the high-level components from the low-level components so that it is possible to reuse the same high-level components with different implementations of low-level components. For example, being able to reuse the same Domain layer with different Infrastructure layers using different technologies but implementing the same interfaces (abstractions) defined in the Domain layer.

The contracts/interfaces define the required behavior of low-level components. These interfaces should exist in the high-level assemblies.

www.EBooksWorld.ir

When the low-level components implement interfaces (that are in the high-level layers), this means the low-level components/layers are the ones that depend on the high-level components. Thus, the traditional dependency relationship is inverted and the reason why it is called “Dependency inversion".

There are several techniques and patterns used for this purpose, such as *Plugin, Service Locator, Dependency Injection and IoC (Inversion of Control)*.

Basically, the main techniques we propose to enable de-coupling between components are:

- Inversion of control (IoC)
- Dependency injection (DI)
- Interfaces of Distributed Services (for consuming/remote access to layers)

The proper use of these techniques, thanks to the de-coupling they provide, enables the following:

- The possibility of replacing the current layers/modules with different ones (implementing the same interfaces) at runtime without affecting the application. For example, at runtime a database access module can be replaced with another that accesses a HOST type external system or any other type of system, as long as they meet the same interfaces. In order to add a new module we won’t need to specify direct references or re-compile the layer that consumes it.

- The possibility of using **STUBS/MOLES and MOCKS** in tests: This is really a concrete scenario of ‘switching from one module to another’. For example, replacing a real data access module with by a module with similar interfaces which fakes that it accesses data sources. The dependency injection allows this change to be made even during runtime without having to re-compile the solution.

## 2.4.- Dependency Injection and Inversion of control

**Inversion of Control pattern (IoC):** This delegates the duty of selecting a concrete implementation of our class dependencies to an external component or source. In short, this pattern describes techniques to support a “plug-in” type architecture, where the objects can search for instances of other objects they require and on which they depend.

**Dependency Injection pattern:** This is actually a special case of IoC. It is a pattern where objects/dependencies are provided to a class instead of the class itselfcreating the objects/dependencies needed. The term was first coined by *Martin Fowler*.

We should not explicitly instantiate the dependencies between different layers. To achieve this, a base class or interface can be used (it seems preferable to use interfaces)

www.EBooksWorld.ir

to define a common abstraction to inject instances of objects to components that interact with this shared interface.

Initially, this object injection can use an "Object Factory" (Factory Pattern) that creates instances of our dependencies and provides them to our objects during the creation of the object and/or initialization. However, the most powerful way to implement this pattern is through a "DI Container" (instead of an Object Factory" created by us). The DI Container injects each object with the dependencies or necessary objects according to the object's relationships. Required dependencies can be registered by code or in the XML configuration files of the "DI Container".

Typically, the application is provided with this DI container by an external framework (such as *Unity*, *MEF*, *Castle-Windsor*, *Spring.NET*, etc.). Therefore, it is the IoC container in the application that instantiates classes from our layers.

The developers will work with an interface related to the implementation classes and will use a container that injects instances of objects that the class is dependent on. The object instance injection techniques are "interface injection", "*Constructor Injection*", "*Property (Setter) Injection*" and "*Method Call Injection*".

When the "Dependency injection" technique is used to de-couple objects of our layers, the resulting design will apply the "Principle of Dependency Inversion".

One interesting de-coupling scenario with IoC is within the Presentation Layer, in order to perform a *mock* or *stub/mole* of the components in an isolated and configurable manner. For example, in an MVC or MVVM presentation layer we may want to *simulate* Web Service consumption for a quick execution of unit testing.

And of course, **the most powerful option related to de-coupling is using the IoC and DI between virtually all the objects belonging to the architecture layers. This will allow us to inject different simulations of the behavior or actual implementations at any point during runtime and/or setup.**

In short, **IoC containers and Dependency Injection add flexibility,** comprehension and maintainability to the project **and will result in "touching" the least possible code as the project goes forward.**

**Table 4.- Dependency Injection (DI) and De-coupling between objects as a "Better Practice"**

| **Dependency Injection (DI) and De-coupling between objects as a "Best Practice"** |
|---|
| The **Single Responsibility principle** states that each object should have a unique responsibility.<br><br>The concept was introduced by *Robert C. Martin*. It establishes that one responsibility is one reason to change and concludes by saying that a class must have one and only one reason to change.<br><br>This principle is widely accepted by the industry and favors designing and developing small classes with only one responsibility. This is directly connected to the number of dependencies, that is, objects which each class depends on. If one class has one responsibility, its methods will normally have few dependencies with other objects in its execution. If there is one class with many dependencies (let's say 15 dependencies), this would indicate what is commonly known as "*bad smells*" of the |

www.EBooksWorld.ir

code. In fact, by doing Dependency Injection in the constructor, we are forced to declare all the object dependencies in the constructor. In the example, we would clearly see that this class in particular does not seem to follow the Single Responsibility principle, since it is unusual for a class with one single responsibility to declare 15 dependencies in the constructor. Therefore, DI also serves as a guide for us to achieve good designs and implementations and offers a de-coupling that we can use to inject different implementations clearly.

It is also worth mentioning that it is possible to design and implement a Domain Oriented Architecture (following patterns with DDD trends) without using de-coupling techniques (i.e. without IoC or DI). It is not mandatory, but it greatly favors isolation of the Domain with respect to the rest of the layers, which is a primary goal in DDD. The reverse is also true: it is certainly possible to use de-coupling techniques in Architectures that are not DDD style. Finally, if IoC and DI are used, it is a design and development philosophy that helps us create a better designed code, and it favors the Single Responsibility principle, as we said at the beginning.

The IoC container and the dependency injection greatly enhance and facilitate successful completion of Unit Testing and Mocking. Designing an application that can be effectively unit tested forces us to do "a good design job".

Interfaces and dependency injection help us make an application extensible (pluggable) and this in turn helps testing. We can say that this testing facility is a nice consequence but it is not the most important one provided by IoC and DI.

**Table 5.- IoC and DI are not only to favor Unit Testing**

| **IoC and DI are not only to favor Unit Testing!!** |
|---|
| This is essential. The Dependency Injection and Inversion of Control containers are not only to promote unit testing and integration! Saying this would be like saying that the main goal of interfaces is to enable testing.<br>DI and IoC are about de-coupling, more flexibility and having a central place that enables maintainability of our applications. Testing is important, but not the first reason or the most important reason to use the Dependency Injection or IoC. |

Another point that needs to be clarified is that DI and IoC containers are not the same thing.

www.EBooksWorld.ir

**Table 6.- Difference between DI and IoC**

| **DI and IoC are different things** |
|---|
| Bear in mind that DI and IoC are different things.<br>DI (Dependency injection through constructors) can surely help testing but its main useful aspect is that it leans the application towards the **Single Responsibility principle** and the **Separation of Concerns principle**. Therefore, DI is a highly recommended technique and a best practice in software design and development.<br>Since implementing DI on our own (for example, with Factory classes) could be quite cumbersome, we use IoC containers, which provide flexibility to the object dependency graph management. |

**Table 7.- Design Rule N° D2**

|  **Rule # D2.** | **Consumption and communication between the different objects belonging to the layers should be de-coupled, implementing patterns of Dependency injection (DI) and Inversion of Control (IoC).** |
|---|---|

○ **Rules**

- In general, this rule should be applied to all N-Layered architectures of medium to large applications. *Of course, it should be done between objects where execution of logic (of any type) is the main duty and that have dependencies with other objects. Clear examples are Services and Repositories, etc. On the other hand, it does not make any sense to do so with the Entity classes themselves.*

✓ **When TO IMPLEMENT Dependency Injection and Inversion of Control**

- These should be implemented in virtually all the medium to large N-Layered business applications. They are particularly useful in Domain and Infrastructure layers as well as in presentation layers with MVC and MVVM patterns.

✘ **When NOT TO IMPLEMENT Dependency Injection and Inversion of Control**

- At the solution level, normally, DI and IoC cannot be used in applications developed with RAD technologies (*Rapid Application Development*). These types of applications do not really implement a flexible N-Layered architecture and there is no possibility of introducing this type of de-coupling. This usually happens in small applications.

www.EBooksWorld.ir

- **At object level, using IoC in the classes that have no dependencies (such as ENTITIES) does not make any sense.**

**Advantages of using Dependency Injection and Inversion of Control**

- Possibility of replacing the Layers/Blocks at runtime.
- Ease of use of STUBS/MOCKS/MOLES for *Testing*.
- Additional flexibility resulting in "touching" the least possible code as the project goes forward.
- Additional project understanding and maintainability**.**

**Disadvantages of using Dependency Injection and Inversion of Control**

- If the IoC and DI techniques are not very well known by every developer then certain initial complexity would be added to the development of the application. However, once the concepts are understood, it is really worthwhile in most applications as it adds greater flexibility and will ultimately result in high quality software.

 References

*Dependency Injection:*
***MSDN - http://msdn.microsoft.com/enus/library/cc707845.aspx***

*Inversion of Control:*
***MSDN - http://msdn.microsoft.com/en-us/library/cc707904.aspx***

*Inversion of Control Containers and the Dependency Injection pattern* ***(By Martin Fowler) – http://martinfowler.com/articles/injection.html***

www.EBooksWorld.ir

## 2.5.- Modules

The Domain model tends to grow dramatically in large and complex applications. The model will reach a point where it is difficult to talk about it as a "whole" and it could be very difficult to fully understand all the relationships and interactions between its areas. Therefore, it is necessary to organize and *partition* the model into different modules. The modules are used as a method of organizing concepts and related tasks (normally different business units). This enables us to reduce complexity from an external point of view.

The concept of a module is actually something used in software development from its inception. It is a lot easier to see the overall picture of a complete system if we subdivide it into different vertical modules and later in the relations between these modules. Once the interactions between the modules are understood, it is also easier to focus on each of them in more detail. It is an efficient way to manage complexity as well. "*Divide and Conquer*" is the phrase that best defines it.

A good example of division in modules is most ERPs. These are normally divided into vertical modules, each of which is responsible for a specific business area. Some examples of ERP modules are Payroll, Human Resource Management, Billing, Warehouse, etc.

Another reason to use modules is related to the quality of the code. It is an industry-accepted principle that the **code should have high cohesion and low coupling.** While cohesion starts at the level of classes and methods, it can also be applied at a module level. Therefore, it is recommended to group related classes in modules to achieve the maximum cohesion possible. There are several types of cohesion. Two of the most used are "*Communication Cohesion*" and "*Functional Cohesion*". Communication cohesion is related to those parts of a module that operate on the same data sets. Grouping it makes perfect sense, because there is a strong connection between those parts of the code. On the other hand, functional cohesion is achieved when all parts of the module perform a task or set of well-defined functional tasks. This is the best type of cohesion.

Thus, using modules in a design is a good way to increase cohesion and decrease coupling. Usually, modules will be divided and will share the different functional areas that do not have a strong relation/dependency on each other. However, in normal conditions there should be some type of communication between the different modules; therefore, we should also define interfaces for communication between them. Instead of calling five objects of a module, it is probably better to call an interface (a DDD service, for instance) of another module that adds/groups a set of functionality. This also reduces coupling.

Low coupling between modules reduces complexity and substantially improves the maintainability of the application. It is also a lot easier to understand how a full system works when we have fewer connections between modules that perform well defined tasks. By the same token, if we have many connections between the modules, it is

www.EBooksWorld.ir

much more difficult to understand the system and if we need to have it this way, there should probably be only one module. Modules should be quite independent from each other.

The name of each module should be part of the DDD "*Ubiquitous Language*", as well as the names for entities, classes, etc. For more details on what the DDD Ubiquitous Language is, read the documentation about DDD in Eric Evan's book *Domain-Driven Design*.

The scheme of the proposed architecture is shown below, taking into consideration the different possible modules of an application:

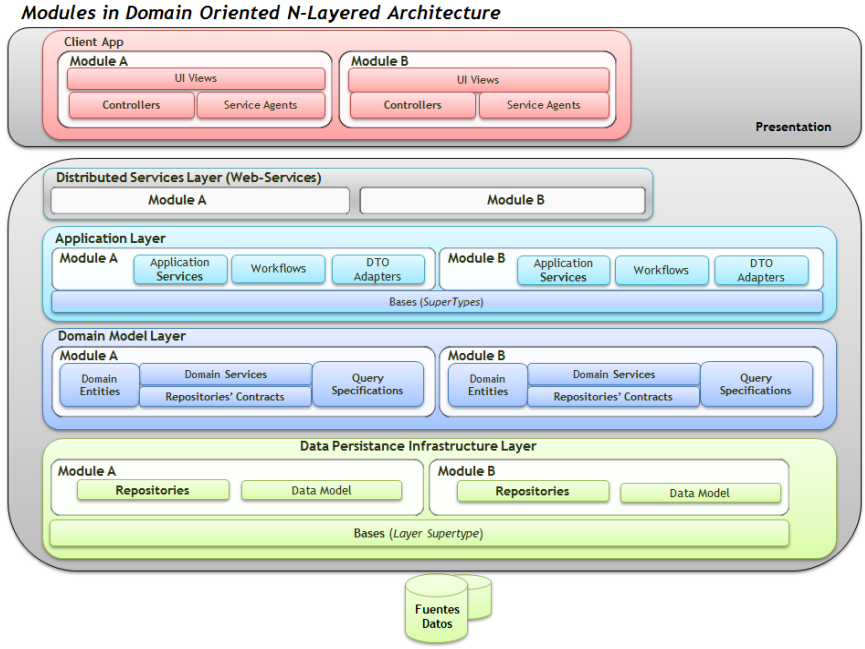



**Figure 10.- Modules in Domain Oriented N-layered Architecture**

A problem arises at the user interface level when different development teams work on each different module. In this case, there is normally only one presentation layer (client application) in the end and changes made to it by some teams can negatively affect changes made by other teams.

As a result, modules are closely related to the concept of composite applications, where different development teams may be working independently on different modules of the same application. However, eventually everything has to be integrated in the same user interface. For this integration to be much less problematic, we recommend using the concepts found in *'Composite Applications'*.

That is, to define specific interfaces to be implemented by each visual module (menus, content areas, loading visual modules (like using MEF) from a configurable

www.EBooksWorld.ir

point in the application, etc.) so that integration is highly automated and not something painful when integrating the different modules in a single client application.

**Table 8.- Design Rule # D3**

**Rule # D3.** **Definition and Design of Application Modules Lead to Differentiated Functional Areas.**

**Rules**

- In general, this rule must be applied in most applications of a certain volume and differentiated functional areas.

✓ **WHEN TO DESIGN and implement the Modules**

- It must be implemented in virtually all business applications with medium to large volume and mostly where we can identify different functional areas that are independent from each other.

✘ **WHEN NOT TO DESIGN and implement Modules**

- In applications where there is a single functional area which is very tight and when it is very difficult to separate it into independent and de-coupled functional modules.

**Advantages of using Modules**

- Using modules in a design is a good way to increase cohesion and decrease coupling.
- A loose coupling between modules reduces complexity and substantially improves the maintainability of the application.

**Disadvantages of using Modules**

- If the entities of a hypothetical module have many relationships with other entities of other module(s), it should probably be a single module.
- Additional investment of initial design time is required to define communication interfaces between the modules. However, as long as definition and separation of the modules fits well (there are differentiated functional areas), it will be very beneficial for the project.

www.EBooksWorld.ir

**References**

*Modules: DDD book – Eric Evans*

*Microsoft - Composite Client Application Library****: http://msdn.microsoft.com/en-us/library/cc707819.aspx***

## 2.6.- Model Subdivision and Work Context

In this section we will see how to work with large models; we will present techniques to maintain coherence of the models by dividing a large model into several smaller models with well-defined borders. In this section we will focus on bounded contexts. It is vital to clarify that a bounded context is not the same as the context of an ORM such as *Entity Framework* or sessions *in NHibernate*, but represents a totally different concept. Here, we are talking about a work context of a development group, as we will see below.

## 2.7.- Bounded Contexts

In large, highly complex applications, our models grow quickly in terms of the number of elements and relations between them. Maintaining coherence in large models is very complicated because of their size and the number of people working on them at the same time. It is very common for two people to have different interpretations regarding the same concept or for a concept to be replicated in another object because they did not know that this concept was already implemented in an object. To solve these problems we should place a limit on the model size by defining a context within which these models are valid.

The idea of having one model for the entire system is tempting, but not feasible, because maintaining coherence inside such a large model is almost impossible and it is not worthwhile in terms of cost. In fact, the first question we should ask when facing the development of a large model is 'Do we need total integration between all the features of our system?' The answer to this question will be negative in 90% of the cases.

Therefore, large models can be split into several smaller models. It will be established that any given element of our system only makes sense within the context (or sub model) where it is defined. We will focus on maintaining coherence within these contexts and will deal with the relations between contexts separately. Contexts are partitions of the model aimed at maintaining coherence, not just simple functional

www.EBooksWorld.ir

partitions. The strategies to define context can be multiple, such as the division into contexts by work teams (the idea is to encourage communication and seamless integration within the context), or by high-level functionalities of the system (one or more functional modules), etc. For instance, in a project where we are constructing a new system that should work in parallel with a maintenance system, it is clear that the older system has its own context, and we do not want our new system to be in the same context, since this would affect our new system's design. Another example is the existence of an optimized algorithm for any calculation where a specific model is used, such as any type of complex math calculation we want to perform on the elements of that specific model.

Nonetheless, establishing contexts within the system has a disadvantage: we lose the global view of it, and this happens when two contexts should communicate to implement functionality and they tend to get mixed up. Therefore, it is essential to create a context map simultaneously, where the different contexts of the system and the relations between them will be clearly identified. This achieves the coherence and cohesion advantages offered by the contexts and preserves the global view of the system by clearly establishing the relationships between the contexts.

## 2.8.- Relations between Contexts

The different relationships that exist between two or more contexts depend greatly on the degree of communication between the various teams in each context and the degree of control we have over them. For example, we may not be able to perform changes in a context, such as in the case of a system in production or a discontinued system, or our system may require support by other systems in order to work. Below, we will see some relationships that typically exist between contexts, but it is important to understand that we should not force these relationships between contexts in our system unless they occur naturally.

### 2.8.1.- Shared Kernel

When we have two or more contexts where the teams who are working on them can communicate fluently, it is interesting to establish a shared responsibility for the objects that both contexts use to interact with the other context. These objects become what is known as a *shared kernel* of both contexts. In order to make a change on any object of the shared kernel, it is necessary to have the approval of the teams of all the contexts involved. It is recommended to jointly create a set of unit tests for each object of the shared kernel so that the shared kernel behavior is fully defined. Promoting communication between the different teams is critical; therefore, a good practice is to

www.EBooksWorld.ir

make some members of each team circulate around the teams of the other contexts, so that the accumulated knowledge in a context is conveyed to the rest.

## 2.8.2.- Customer/Supplier

It is quite common to realize that we are developing a system that depends on other systems to do its job. For instance, it could be an analysis system or a decision-making system. In these types of systems there are usually two contexts, in one context our system uses the system on which it depends and that is located in the other context.

Dependencies between the two contexts are one-way, from the "customer" context to the "supplier" context, or from the dependent system to the system on which it depends.

In this type of relationship the customer might be limited by the supplier functionality. At the same time, the supplier context could be inhibited from making changes due to fear of causing the appearance of bugs in the customer context or contexts. Communication between the different context teams is the key to solving this type of problem. The members of the customer context team should participate as customers in the supplier planning meetings to prioritize the user's stories of the supplier system. In addition, a set of acceptance tests should be jointly created for the supplier system, so that the interface expected by the customer contexts is fully defined and the supplier context can make changes without fear of accidentally changing the interface expected by the customer contexts.

## 2.8.3.- Conformist

The customer/supplier relationship requires collaboration between the teams of the different contexts. This situation is often quite ideal, and in most cases the supplier context has its own priorities and is not set up to meet the needs of the customer context. In this type of situation, where our context depends on another context over which we do not have any control (we cannot make changes or request functionalities), and with which we don't have a close relationship (the cost of translating communications from one context to another is high) we use a *conformist* approach. This involves adapting our model to the one shown by the other context. This limits our model to performing simple additions to the model of the other context and limits the shape our model can take. However, it is not a crazy idea since the other model can add the accumulated knowledge to the development of the context. The decision of whether to follow a conformist relationship greatly depends on the model quality of the other context. If it is not appropriate, a more defensive approach should be followed, such as Anti-corruption layer or Separate ways as we will see below.

www.EBooksWorld.ir

## 2.8.4.- Anti-Corruption Layer

All the relationships we have seen so far assume the existence of good communication between the teams of the different contexts as well as a well-designed context model that can be adopted by others. But what happens when a context is poorly designed and we do not want this to affect our context? For this type of situation we can implement an *Anti-Corruption* layer, which is an intermediate layer between the contexts that performs the translation between our context and the context that we want to communicate with. In general, this communication will be started by us, although it is not mandatory.

An anti-corruption layer consists of three types of components: *adapters*, *translators* and *facades*. First, a facade is designed to simplify the communication with the other context and that exposes only the functionality our context will use. It is important to understand that the facade should be defined in terms of the other context's model elements; otherwise we would be mixing the translation with the access to the other system. After the facade, an adapter is placed to modify the interface of the other context and adapt it to the interface expected by our context. Finally, we use a translator to map the elements of our context that are expected by the facade of the other context.

## 2.8.5.- Separate ways

Integration is overrated, and often not worth the cost involved. Thus, two groups of functionalities with no connection between them can be developed in different contexts without any communication between them. If we have functionalities that need to use both contexts, we can always perform this operation at a higher level.

## 2.8.6.- Open Host

When we develop a system and decide to split it into contexts, a common way to do so is by creating an intermediate translation layer between the contexts. When the number of contexts is high the creation of these translation layers involves a considerable extra workload. When we create a context, it usually has high cohesion and, features offered can be seen as a set of services (We are not referring to Web services but just services.).

In these situations, it is best to create a set of services that define a common communication protocol for other contexts to be able to use the context's functionality. This service should maintain compatibility between versions, but it can gradually

www.EBooksWorld.ir

increase the functionalities offered. Functionalities exposed should be general and if another context needs a specific functionality then it should be created within a separate translation layer so that the protocol of our context is not polluted.

## 2.9.- Implementation of Bounded Contexts in .NET

As we have stated at the beginning of this section, bounded contexts are organizational units designed to maintain coherence of large models. For this reason, a bounded context can represent one area of the system's functionality to an external system or represent a set of components designed to optimally perform a task. There is no general rule to implement a bounded context, but here we will address the most important aspects and give some examples of typical situations.

In our architecture, we divide domain and functionalities into large size modules. Each module is logically assigned to a different working team, and it features a set of very cohesive functionalities that can be displayed as a set of services. The most logical thing to do when we have to deal with several modules is to use a "separate ways" relationship between them. Each module in turn will be an "open host" that will offer a set of functionalities in the manner of services. Thus, any functionality that involves several modules will be implemented from a higher level. Each module will be in charge of its own object model, and of managing its persistence. When using Entity Framework we will have a 1 to 1 correspondence between the module and entity framework contexts.

There will probably be enough complexity within each module for us to continue partitioning the system into smaller contexts. However, these work contexts are more related and will feature a communication relationship based on a "shared kernel" or "customer/supplier". In these cases, the context is more an organizational unit than a functional one. The different contexts will share the same entity framework model, but the modification of certain key objects will be subject to the agreement between the two teams of the different contexts.

Finally, we should address a specific aspect of our system, which is its relationship with external systems or third-party components. These are clearly different bounded contexts. Here, the approach could be to accept the external system model, adopting a "conformist" approach, or we can protect our domain through an "anti-corruption layer" that translates our concepts into the concepts of the other context. The decision on whether to follow a conformist approach or choose an anti-corruption layer depends on the model quality of the other context and on the cost of translation from our context to the other context.

www.EBooksWorld.ir

## 2.9.1.- How to Partition an Entity Framework Model?

A very effective way of partitioning an EF model is by finding the entities that are the most interconnected of all and either remove from the model or remove all the associations and just leave the FKs. Often times the most interconnected entities are those that contribute the less semantic value to the model, e.g. it could represent a cross-cutting concern such as User that will be associated to each other Entity through the "LastModifiedBy" property. It is not always required to have a relationship between entities, and we will see why by analyzing a relationship in detail. What is the point of having a relationship between two entities? Typically one of these entities uses the functionality of the other one to implement its own functionality. For example, consider the case of an Account entity and a Customer entity, where the assets of a Customer are calculated by aggregating the balance of all his accounts and properties.

In general, a relationship between two entities can be replaced by a query in the repository of one of them. This query represents the relationship. In the methods of the other entity we can add an extra parameter containing the information of the association as the result of the query being made to the repository and it can operate as if the relationship existed.

Interaction between the two entities is implemented at the service level, since this type of interaction is not very common and the logic is not complex. If an association requires modification (by adding or eliminating an element), we will have query methods in such entities that will return Boolean values indicating if such an action should be carried out or not, instead of having methods to modify the association we deleted. Continuing with our example of Accounts and Customers, let's suppose we want to calculate the interests to be paid to a certain customer, which will vary depending on the customer's features. This service should also keep interests in a new account if they are exceeded by a certain amount, depending on the seniority of the customer. (We know that this is not the way it is done, but this is just an illustrative case). In this case, we would have a service with the following interface:

```
public  interface IInterestRatingService
{
void RateInterests(intclientId);
}
```

```
public class InterestRatingService : IInterestRatingService
{
   public InterestRatingService(IClientService clients,
   IBankAccountService accounts)
   {
     …
   }
   public void RateInterests(intclientId)
   {
     Client client = _clients.GetById(clientId);
     IEnumerable<BankAccount>clientAccounts =
```

www.EBooksWorld.ir

```
                                    accounts.GetByClientId(clientId);
        double interests = 0;
        foreach(var account in clientAccounts)
        {
           interests += account.calculateRate(client);
        }
        if(client.ShouldPlaceInterestsInaNewAccount(interests))
        {
           BankAccountnewAccount = new Account(interests);
           accounts.Add(newAccount);
        }
        else
        {
           clientAccounts.First().Charge(interests);
        }
      }
      }
```

## 2.9.2.- Connection between Bounded Contexts and Assemblies

The existence of a bounded context does not directly imply the creation of a specific assembly. However, depending on the relationships in the context map, some bounded contexts will go in the same assembly while others will be separated. Usually, when two bounded contexts have a strong relationship, such as the one determined by a shared kernel or customer/supplier, such contexts are placed within the same assembly. In weaker relationships such as interaction between modules, there are two approaches.

One approach is having all the modules in one assembly and using assemblies only for the division of layers. This will facilitate the interaction between modules, because they are able to host references to elements of any other module. There is also the advantage of having our entire domain in a single assembly; this simplifies the deployment and reuse of the domain in other applications. It is worth noting that the fact that all the modules are in the same assembly does not mean they share the same context of Entity Framework. This is the approach we have followed in the example of the interaction between modules.

The other approach is having each module in a different assembly. By doing this, we not only improve but ensure the isolation between the modules. However, communications between the modules become more complicated. Each module should define its own abstractions of the entities of another module it needs (which should be minimized), and create an adapter of the entities of the other module to the abstractions defined in the module through an anti-corruption layer at a higher level.

www.EBooksWorld.ir

## 2.10.- Mapping technologies in N-Layered Architecture

Before analyzing how to define the structure of our Visual Studio solution in detail, it is convenient to have a high-level view where the different mentioned layers are mapped with their corresponding technologies:

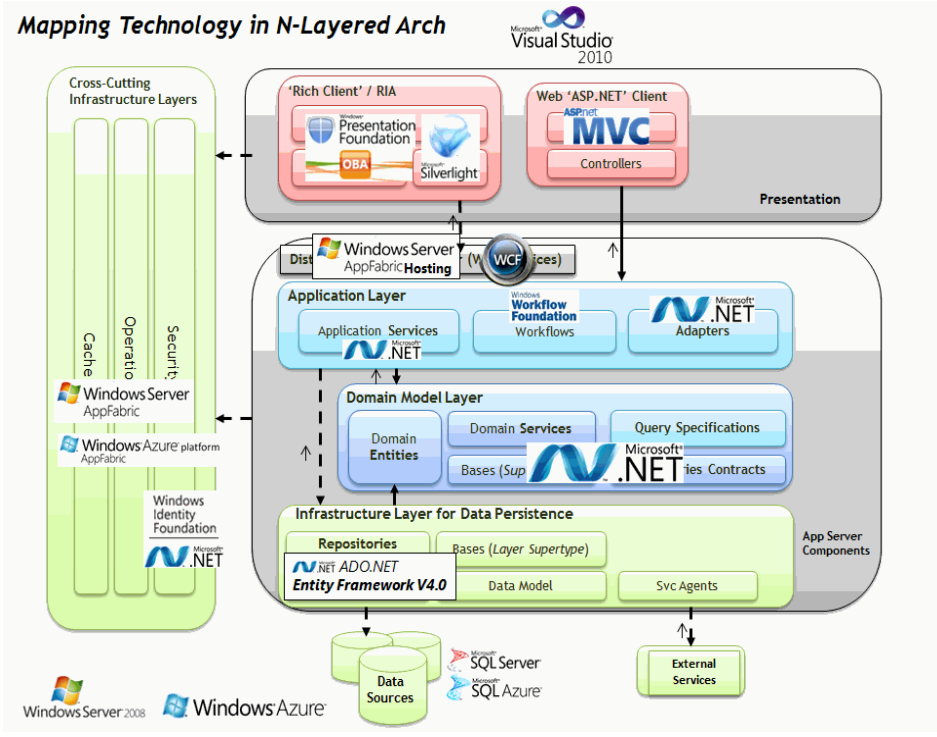



**Figure 11.- Mapping Technology in N-Layered Arch**

In the following chapters we will see in detail how to implement the different patterns of the architecture with each of the technologies placed in the chart.

www.EBooksWorld.ir

## 2.11.-Implementing a Layered Architecture in Visual Studio 2010

In order to implement a Layered Architecture (according to our DDD N-Layered Architecture style) a series of steps should be taken:

1.- Visual Studio solution should be organized and should clearly show where the implementation of each layer and sub-layer is located.

2.- Each layer should be correctly designed and include the design patterns and technologies of the layer.

3.- There will be cross-cutting layer patterns and technologies to be used throughout the whole application, such as implementation of the technology chosen for IoC, or security aspects, etc. These cross-cutting layers (DDD Cross-Cutting Infrastructure) will be quite reusable in different projects undertaken in the future. This is a mini-framework, or better known as *seedwork*. Ultimately, it becomes a source code that will be reused in other future projects, as well as certain base classes (*Core*) of the Domain and Data Persistence layers.

## 2.12.- Sample application of N-Layer DDD with .NET 4.0

Virtually all the code examples and the solution structure shown herein belong to the sample application developed to accompany this book. We strongly recommend downloading the source code from the Internet and reviewing it as explained in the book, since more details can be seen directly in the actual code.

The sample application is published on CODEPLEX, licensed by OPEN SOURCE, at this URL:

**http://microsoftnlayerapp.codeplex.com/**

## 2.13.- Visual Studio Solution Design

Once we have a Visual Studio 'Solution', we will start by creating the logical folder structure to host and distribute the various projects. In most cases we will create a project (DLL) for each layer or sub-layer to provide higher flexibility and make it

www.EBooksWorld.ir

easier to provide the best de-coupling possible. However, this generates a considerable number of projects, so it is essential to sort/rank them in hierarchies by logical folders in Visual Studio.

The initial hierarchy would be something similar to the following:

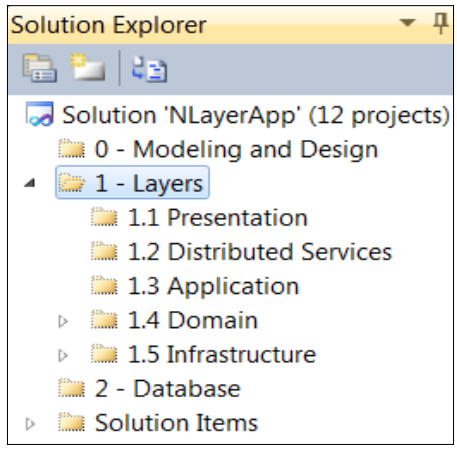



**Figure 12.- Hierarchy of Folders in Visual Studio Solution**

Starting from the top, the first folder (‘*0 –Modeling & Design*’) will contain the different diagrams of Architecture and Design made with VS2010, such as Layer diagram of the Architecture, and the different UML diagrams of internal design. These diagrams will be used to represent the implementations we make.

Layer numbering is simply to make them appear in an appropriate order aligned with the natural order of the architecture and also to make it easier to look for each layer within the Visual Studio *solution*.

The next folder, ‘1 Layers’, will contain the different layers of the N-Layer Architecture, as shown in the hierarchy above.

### Presentation layer

The first layer, Presentation, will contain different types of presentation projects such as RIA (Silverlight), Web (ASP.NET), Windows Phone or Windows-Rich (WPF, WinForms, OBA) projects, etc.:

www.EBooksWorld.ir

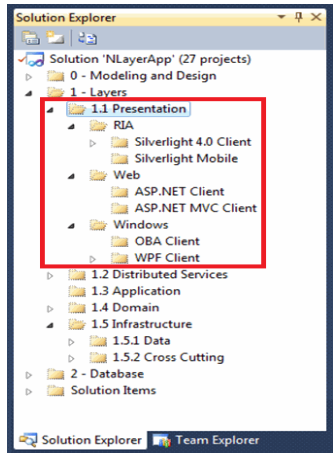



**Figure 13.- Presentation Layers**

Subsequently, we have the component layers that are normally situated in an application server (we would be referring to deployment, which can vary, but now at the organization level in VS, we do not specify details of the deployment). In short, we will have different main Layers of N-*Layered* Domain oriented architecture, with different projects for each sub-layer:

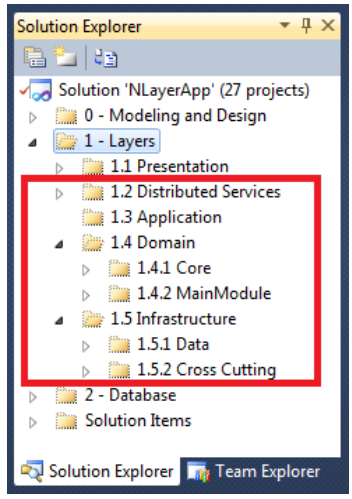



**Figure 14.- Layers of the Application Server**

Within each of these folders, we will add the necessary projects according to the typical elements of each layer. This is also determined by the patterns to be implemented (explained later at logical and implementation levels in this guide).

www.EBooksWorld.ir

**Distributed Services Layer (WCF Services)**

This Layer is where we will implement the WCF services (Normally, Web Services) in order to be able to remotely access components of the Applications Server. It is worth noting that this Distributed Services layer is optional, since in some cases (such as ASP.NET Web presentation layer) it is possible to directly access the components of APPLICATION and DOMAIN providing that the ASP.NET Web server is at the same server level as the business components.

If we are using the distributed services for remote access, the structure may be something similar to the following:

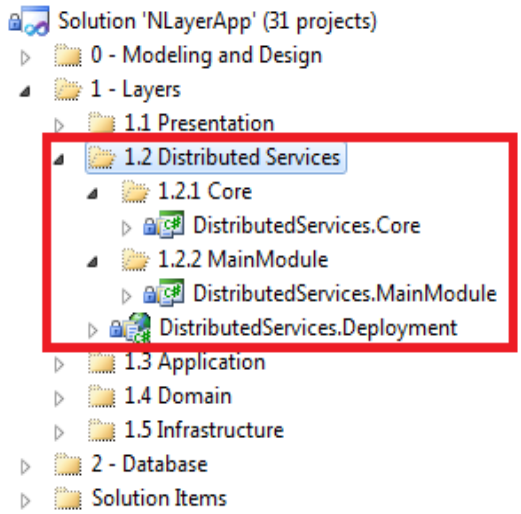



**Figure 15.- Use of distributed services**

We would need a project to *Host* our WCF service, that is, the process under which the WCF service is run and published. This project/process can be a *WebSite* in IIS (or *Cassini* during development), a Windows service, or any type of process.

But the functionality of the Web-Service is really in the Service that exposes the logic of each module. We will have **a WCF Service project (DLL) for each functional MODULE of the application**. In our example, we have only a single module called '**Main Module**'.

In case of *hosting* it in a Web Server, a .SVC file will be internally added for each application MODULE.

In addition, we will also need to have a project for Unit Testing within this layer.

For a WCF service in production, deployment is recommended in an IIS WebSite (If possible, using an **IIS 7.x**, which enables us to use TPC *bindings* such as *NetTCP* instead of using HTTP based bindings), or even going for the best deployment scenario with IIS plus Windows Server AppFabric where we have WCF services monitoring and instrumentation capabilities provided by AppFabric.

www.EBooksWorld.ir

### Application layer

As explained earlier in the Logical Architecture part of this guide, this layer should not actually contain the domain rules or business logic knowledge. It should just perform coordination tasks of the technological aspects of the application that we would never explain to a domain expert or business user. Here we implement the coordination of the application "plumbing", such as transaction coordination, Unit of Works, Repositories' coordination and calls to objects of the Domain.

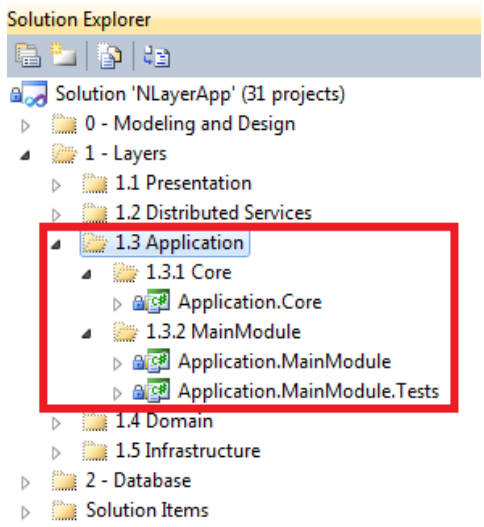



**Figure 16.- Application Sub-Layers**

In turn, each layer with logical classes will have a Unit Testing Project.

### Domain Layer

This layer is the most important from a business/domain point of view, since this is where we implement all the domain logic, domain entities, etc.
Internally, this Layer has several sub-layers or types of components. It is recommended to consolidate the number of projects required within this layer insofar as possible. However, it is still a good practice to have a specific assembly/project for the entities, so they are not coupled to the Domain Services:

www.EBooksWorld.ir

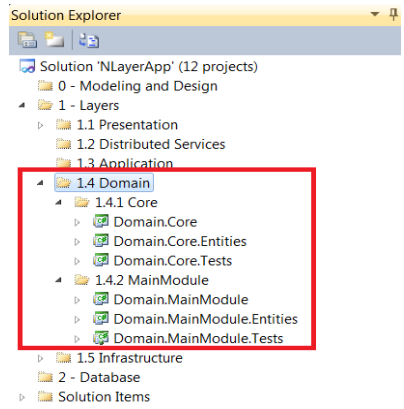



**Figure 17.- Domain Sub-Layers**

Generally speaking, we can have a '*Core*' project of base classes and other reusable classes in a horizontal manner in all the Domain's functional modules.

For each functional MODULE of the application (in this case, the so called '*Main Module'*), we will implement the entire module logic (Services, Specifications and Repository Contracts) within a single project (in this case Domain.MainModule), but we need an isolated project for the '**Domain Entities'**, in each MODULE, where Entity Framework generates our POCO or Self-Tracking entity classes.
This is the content of the *Domain* projects for one of the modules (MainModule):

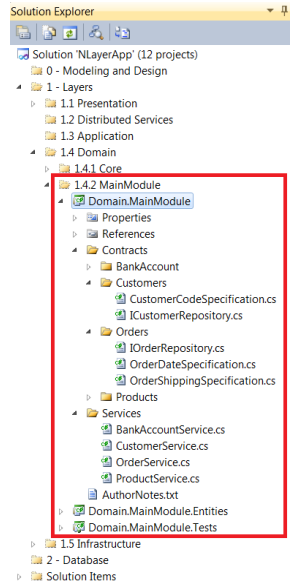



**Figure 18.- Contents of the Domain projects**

www.EBooksWorld.ir

Again, each project with logical classes will also have a Unit Testing project and we could also have other integration and functional testing projects.

This Domain layer will be explained at both logical and implementation levels in a full chapter of this guide.

### Data Persistence Infrastructure Layer

The most characteristic part of this layer is the implementation of REPOSITORIES for data access and persistence. In this module we will also implement everything related to the model plus links/actions to the database in Entity Framework.

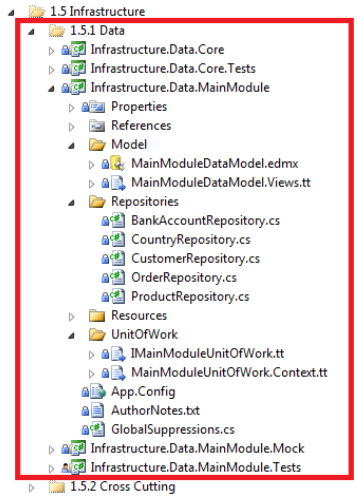



**Figure 19.- Data Persistence Infrastructure Layer**

At the level of each functional MODULE (in this case, *Main Module*) we will have a single project with the following elements:

- **'Entity Model'**: This will contain the *Entity Data Model*,although classes generated by Entity Framework (*Context* and POCO entities) will be extracted and put into other projects in order to de-couple them according to the DDD Domain design. Therefore, only the data model will appear here (in our case, *MainModuleDataModel.edmx*). However, if we choose to follow a *Code First* approach then we will just have the POCO classes and the EDMX file will be gone.

- '**Context'/Unit of Work:** This implements an abstraction of the Entity Framework's context, in order to replace it with a *fake/mock* context to perform unit testing.

- **Repositories**: Classes in charge of the data persistence logic.

www.EBooksWorld.ir

We will also have another project for testing the entire module.

The projects of the ***Core*** type are used for the implementation of base classes and extensions. These projects are valid to be reused in a horizontal manner for the implementation of Persistence layer of all the application's functional modules.

This 'Data persistence' will be explained at both logical and implementation levels in a full chapter of this guide.

## 2.14.- Application Architecture with Layer Diagram of VS.2010

In order to better understand the design of the Architecture, we can use a layer diagram in VS2010 to visualize the N-Layered Architecture. In addition, this allows us to map the layers we visually draw with their real logical *namespaces* and/or *assemblies* in the solution. As a result, this enables the validation of the architecture against the actual source code, so that it can be evidenced if accesses/dependencies between layers, not permitted by the architecture, are being made in the code, and even by connecting these validations to the source code control process in TFS.

This is the diagram of **N-Layer Architecture for** our sample application:

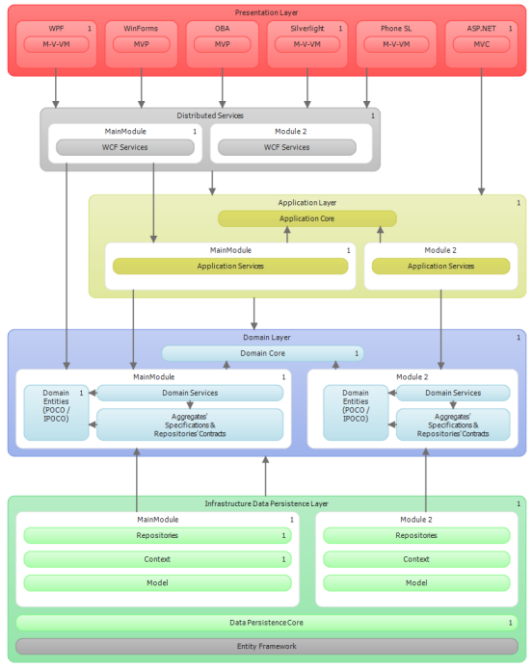



**Figure 20.- DDD N-Layer Architecture in VS.2010**

www.EBooksWorld.ir

We analyze the logic and implementation of each one of these layers and sub-layers in the following chapters. Here, we will only point out some global aspects.

As we can see in the Architecture diagram, the core layer which the entire Architecture is based on is the Domain layer. This is also significant at the dependency level. Most dependencies end in the Domain layer (e.g., dependencies with the Domain Entities). The Domain layer, in turn, has minimum dependencies to other layers (Infrastructure, Data Persistence), in which case they are "de-coupled dependencies", that is, they are based on abstractions (interfaces) through IoC containers. That is why the dependencies do not appear directly as "arrows" in the diagram.

Another aspect to be mentioned is that the "Remote Services" or "Distributed Services" layer (WCF services, in .NET), is an optional layer depending on the type of Presentation layer to be used. If the presentation layer is run on a client machine (Silverlight, WPF, WinForms or OBA) then it is evident that it will be necessary. However, for example in the case of a Web client (ASP.NET or ASP.NET MVC), there is a common possibility that the Web server of the presentation layer is at the same physical server as the business components. In this case, there is no point in using WCF services, since this would unnecessarily impact application performance.

Regarding the "Application layer", it will normally be our "Facade" layer, where the Application Services that coordinate tasks and actions to be performed against the Domain and persistence are presented.

## 2.15.- Implementation of the Dependencies Injection and IoC with UNITY

In this section, we will explain the techniques and technologies used to implement a specific de-coupling between the Architecture layers. Specifically, this is to explain **DI (Dependency Injection)** and **IoC (Inversion of Control)** with a specific technology of **Microsoft Pattern & Practices**, called ***Unity***.

**DI** and **IoC** can be implemented with different technologies and frameworks from different vendors, such as:

**Table 9.- Implementations of IoC containers**

| Framework | Implementer | Information |
|---|---|---|
| **Unity**<br>http://msdn.microsoft.com/en-us/library/dd203101.aspx<br>http://unity.codeplex.com/ | Microsoft Pattern & Practices | This is currently the most complete light-weight Microsoft *framework* to implement IoC and DI.<br>It is an *Open Source* project with licensing of Microsoft Public License (Ms-PL) type. |

www.EBooksWorld.ir

| | | |
|---|---|---|
| **Castle Project (Castle Windsor)**<br>http://www.castleproject.org/ | CastleStronghold | Castle is an Open Source project. It is one of the best frameworks for IoC and DI. |
| **MEF (Microsoft Extensibility Framework)**<br>http://code.msdn.microsoft.com/mef<br>http://www.codeplex.com/MEF | Microsoft<br>(Part of .NET 4.0) | This is currently a *framework* for automatic extensibility of tools and applications, not so concerned with de-coupling between Architecture layers using IoC and DI.<br>In the future it could replace Unity. |
| **Spring.NET**<br>http://www.springframework.net/ | SpringSource | Spring.NET is an Open Source project. It is one of the best AOP (Aspect Oriented Programming) frameworks also offering IoC capacities. |
| **StructureMap**<br>http://structuremap.sourceforge.net/Default.htm | Several developers of the .NET community | Open Source project. |
| **Autofac**<br>http://code.google.com/p/autofac/ | Several developers of the .NET community | Open Source project. |
| **LinFu**<br>http://code.google.com/p/linfu/downloads/list<br>http://www.codeproject.com/KB/cs/LinFuPart1.aspx | Several developers of the .NET community | Open Source project.<br>It provides IoC, AOP and other features. |

For our example of N-layered application Architecture, we have chosen UNITY because currently it is the most complete IoC and DI framework offered by Microsoft. But of course, in a business framework architecture, any IoC framework can be used (whether listed or not in the table above).

www.EBooksWorld.ir

## 2.15.1.- Introduction to Unity

The *Application Block* called **Unity** (implemented by *Microsoft Patterns & Practices*), is an extensible and light-weight dependency injection container. It supports constructor injection, properties injection, injection in method calls and nested containers.

Basically, *Unity* is a container where we can register types (classes, interfaces) and also mappings between such types (like a relation between an interface and a class that implements that interface). The *Unity* container can instantiate specific types upon request as well.

Unity is available as a free public *download* from the Microsoft website and it is also included in the Enterprise Library 4.0/5.0 and in PRISM *(Composite Applications Framework)*, which makes extensive use of *Unity*.

To use *Unity*, we normally record types and mappings in a container and specify dependencies between interfaces, base classes and specific types of objects. We can define these records and mappings directly by code (e.g. C# code), or we can use XML configuration files if we want to change things during runtime. Also, dependency injection can be specified in our own classes by using attributes that designate properties and methods requiring other objects to be injected, as well as objects specified as the parameters of a constructor which are injected automatically.

We can even use container extensions that support other things such as the "*EventBroker*" extension, which implements a publication/subscription mechanism based on attributes that we can use in our applications. We can even build our own container extensions.

*Unity* provides the following advantages for application development:

- It supports requirement abstraction; this allows developers to specify dependencies during runtime or during setup and simplifies the management of horizontal aspects (*cross cutting concerns*), such as performing unit testing against *mocks* and *stubs*, or against real objects in the application.

- It simplifies the creation of objects, especially those with complex hierarchical structures and dependencies, which ultimately simplifies the application code.

- It increases flexibility by moving the configuration of components to the IoC container.

- It provides service location capacity; this allows clients to save or cache the container. This is particularly useful in ASP.NET Web applications where developers can persist the container in the session or in the ASP.NET application.

www.EBooksWorld.ir

## 2.15.2.- Usual scenarios with Unity

*Unity* solves typical development problems in component-based applications. Modern business applications consist of business objects and components that perform generic or specific tasks within the application; in addition, we often have components in charge of horizontal aspects of the application architecture, such as trace, logging, authentication, authorization, cache and exceptions management.

**The key to successfully build these business N-layered applications is to achieve a de-coupled design** *(by de-coupled we mean very loosely coupled rather than de-coupled as such)*. De-coupled applications are more flexible, easily maintained and, what is more important, they are easier to test during the development (Unit Testing). **Mocks** (simulations) of objects with strong concrete dependencies (such as database connections, network connections, connections to external applications such as ERP, etc.) can be made. As a result, unit testing can be done against mocks or against real objects by changing it dynamically or based on configuration.

Dependency injection is a fundamental technique to build de-coupled applications. It provides ways to manage dependencies between objects. For example, an object that processes customer information may depend on other objects that access the database, validate information and verify if the user is authorized to perform updates. The dependency injection techniques can ensure that the Customer class instantiates and correctly executes such objects on which it depends, especially when dependencies are abstract.

## 2.15.3.- Main Patterns

The following design patterns define architecture and development approaches that simplify the process:

**Inversion of Control pattern (IoC).** This generic pattern describes techniques to support a "plug-in" architecture where objects can search for instances of other objects they require.

**Dependency Injection pattern (DI)**. This is really a special case of IoC. It is a programming technique based on altering the behavior of a class without changing the internal code of the class. The object instance injection techniques are "*interface injection*", "*constructor injection*", "*property injection (setter)*", and "*injection of method calls*".

**Interception pattern**. This pattern introduces another level of indirection. This technique places an object (a proxy) between the client and the real object. The behavior of the client is the same as if it interacted directly with the real object, but the

www.EBooksWorld.ir

proxy intercepts it and solves its execution by collaborating with the real object and other objects as required.

## 2.15.4.- Main methods

*Unity* exposes two methods for registering types and mappings in the container:

**RegisterType():**This method registers a type in the container. At the appropriate time, the container builds an instance of the specified type. This may happen in response to a dependency injection initiated through the class attributes or when the *Resolve* method is called. The object's lifetime is specified by one of the method's parameters. If no value is passed as lifetime, the type will be recorded as a transient, which means the container creates a new instance every time Resolve method is called**.**

**RegisterInstance():**This method registers an existing instance of the specified type with a specified lifetime in the container. The container returns the existing instance during this lifetime. If no value is assigned to lifetime, the instance has a lifetime controlled by the container.

## 2.15.5.- Registering Types in the Unity Container

As an example of the **RegisterType** and **Resolve** methods usage, in the code below, we record a mapping of an interface called *ICustomerService* and we specify that the container should return an instance of *CustomerService* class (which implements *ICustomerService* interface).

```
C#
//Register of types in container of UNITY
IUnityContainer container = new UnityContainer();
container.RegisterType<ICustomerManagementService,
CustomerManagementService>();
...
...
//Resolution of type from interface
   ICustomerManagementServicecustomerSrv =
container.Resolve<IICustomerManagementService> ();
```

Normally, in the final version of the application the registration of types and mappings in the container will be done by the XML configuration files. However, as shown in the code above, during the development it is probably more convenient to do it "*Hard-coded*," so that typographical errors will be detected during compilation time instead of during runtime (as in the case of XML configuration files).

Regarding the code above, the line that will always be in the application code would be the one that resolves the class that should be used by the container, that is, the call to

www.EBooksWorld.ir

the **Resolve()** method (Regardless of whether the type registration is made through XML or is hard coded).

## 2.15.6.- Dependency Injection in the Constructor

To understand injection in the constructor, consider a scenario where we instantiate a class using the Resolve() method of Unity container, this class having a constructor with one or more parameters (dependencies to other classes). As a result, Unity container will automatically create instances of the required objects specified in the constructor.

As an example, we have a code that does not use Dependency Injection or Unity and we want to change this implementation so that it is de-coupled, using IoC through Unity. This code uses a business class called **CustomerManagementService** with simple instantiation and usage:

```
C#
…
{
CustomerManagementServicecustService =
new CustomerManagementService();
custService.SaveData("0001", "Microsoft", "Madrid");
}
```

**It is important to consider that this would be the code to be implemented at the beginning of the action.** For example, it would be the code to be implemented in a WCF Web Service method.

Below we have the implementation of this **initial Service class** (**CustomerManagementService**) **without dependency injection**, which in turn uses a data access layer class called **CustomerRepository**:

```
C#

public class CustomerManagementService
{

    private ICustomerRepository _custRepository;

    public CustomerManagementService()
    {
        _custRepository = new CustomerRepository();
    }

    public SaveData()
    {
        _custRepository.SaveData("0001", "Microsoft", "Madrid");
    }
}
```

www.EBooksWorld.ir

So far, in the code above we have nothing of IoC or DI and there is no dependency injection or use of Unity. In other words, this is all traditional object oriented code. Now we are going to modify CustomerManagementService class so that the creation of the class on which we depend (CustomerRepository) is not explicitly made by us and instead this object is instantiated automatically by the Unity container. This means we will have a code with dependency injection in the constructor.

```
C#

   public class CustomerManagementService
   {
       //Members
       private ICustomerRepository _custRepository;
       //Constructor
       public CustomerManagementService (ICustomerRepository
   customerRepository)
       {
           _custRepository = customerRepository;
       }
       public SaveData()
       {
           _custRepository.SaveData("0001", "Microsoft", "Madrid");
       }
   }
```

It is important to remember that, as shown above, we have not made any explicit call to the constructor of CustomerRepository class (i.e. there is no *new*). Unity is the container that will automatically create the CustomerRepository object and will provide it to us as an input parameter to the constructor. **This is precisely the dependency injection in the constructor.**

During runtime, the instantiation of *CustomerManagementService class* would be made using the Resolve() method of the Unity container, which initiates the instantiation generated by the Unity framework for the *CustomerRepository* class within the scope of the *CustomerManagementService* class.

The following code is what we implement in the first-level layer that consumes Domain objects. That is, it would probably be Distributed Services layer (WCF) or even the Web presentation layer running in the same application server (ASP.NET):

```
C# (In WCF service layer or Application Layer or in ASP.NET application)
…
{
   IUnityContainer container = new UnityContainer;
   CustomerManagementServicecustService =
   container.Resolve<ICustomerManagementService>();
   custService.SaveData("0001", "Microsoft", "Madrid");
   }
```

As can be seen in the use of **Resolve(),** we never created an instance of the class on which we depend (e.g. CustomerRepository). Therefore, we did not explicitly pass a CustomerRepository object to the constructor of CustomerManagementService class, and yet – when the service class (CustomerManagementService) was instantiated – we

www.EBooksWorld.ir

automatically received a new CustomerRepository instance in the constructor. This has been done precisely by Unity container when detecting the dependency. This is dependency injection and provides us with the flexibility of changing dependency during setup time and/or runtime. For example, if in the configuration file we had specified that Mock objects should be created instead of real data access objects (Repository) then the class to be instantiated would have been **CustomerMockRepository** instead of **CustomerRepository** (both would implement the same **ICustomerRepository** interface).

## 2.15.7.- Property Injection (Property Setter)

To understand the injection of properties, consider a class called *ProductService* that has a reference to another class as a property, called *Supplier*. To force the injection of a dependent object, we must place the **Dependency** attribute on the property, as shown in the following code:

```
C#

public class ProductService
{
    private Supplier supplier;
    [Dependency]
    public Supplier SupplierDetails
    {
        get { return supplier; }
set { supplier = value; }
    }
}
```

So, by creating an instance of ProductService class through Unity container, an instance of ***Supplier*** class will automatically be generated and set as the value of SupplierDetails property of the ProductService class.

To see more examples of dependency injection with Unity, check out the following labs and documentation:

*Unity 1.2 Hands On Labs*
**http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=93a5e18f-3211-44ef-b785-c59bcec4cd6f**

*Webcast Demos*
**http://unity.codeplex.com/Wiki/View.aspx?title=Webcast%20demos**

*MSDN Technical Article &Sample Code*
**http://msdn.microsoft.com/en-us/library/cc816062.aspx**

www.EBooksWorld.ir

## 2.15.8.- Summary of the Main Features of Unity

Unity provides the following noteworthy features:

- Unity provides a mechanism to build (or assemble) object instances, which may contain other object instances.
- Unity exposes "RegisterType()" methods that allows us to configure the container with type and object mapping (including singleton instances) and "Resolve()" methods that return built object instances that can contain dependent objects.
- Unity provides **inversion of control (IoC) that** allows preconfigured object injection in the classes built by *application block*. We can specify an interface or class in the constructor (constructor injection) or we can apply attributes to properties and methods to start property injection or injection of method calls.
- Container hierarchy is supported. A container can have child containers, which allow object localization queries to be passed from the child containers to the parent containers.
- Standard configuration files (.config) can be used to configure the container.
- No specific definitions in classes are required. There are no requirements to be applied to the classes (like attributes), except when method call injection or property injection are used.
- Unity allows extension of container functionalities; for example, we can implement methods to enable additional container features, like cache.

## 2.15.9.- When to use Unity

Dependency injection provides opportunities to simplify the code, abstract dependencies between objects and generate instances of dependent objects in an automated manner. However, the process may have a slight impact on performance (this impact is normally insignificant when at the same time we have dependencies to external resources like databases or distributed services, which actually create bottlenecks in most applications). On the other hand, when working only with objects in the memory (Games, Intensive Graphics App, etc.), this could significantly impact performance.

www.EBooksWorld.ir

Also, complexity will increase a bit where there are only direct dependencies.

In general:

***Unity* should be used in the following situations:**

- Your objects and classes have dependencies on other objects and classes
- Your dependencies are complex and require abstraction
- The intention is to use injection features in the constructor, method or property.
- The intention is to manage the lifetime of object instances.
- The intention is to be able to configure and change dependencies at runtime.
- The intention is to perform unit testing with mocks/stubs
- The intention is to cache or persist dependencies throughout *postbacks* in a Web application.

**There is no need to use Unity in the following situations:**

- Your objects and classes do not have dependencies on other objects and classes
- Your dependencies are very simple or do not require abstraction.

## 3.- EDA (EVENT DRIVEN ARCHITECTURE)

**EDA (*Event-Driven Architecture*)** is a software architecture pattern that essentially promotes use of events (generation, detection, usage and event reaction) as the main principle for the execution of certain Domain logic. It is a generic type of architecture, so it can be implemented with any development platform and it is not necessary or mandatory to use special technologies (although technologies especially designed to implement workflows and implementations of business processes contribute significantly to this architecture trend).

In this architecture guide, EDA will be included as an additional possibility, not as something mandatory to be designed and implemented, since suitability of a strong event orientation largely depends on the type of application to be developed.

An *event* can be defined as a "significant change of state". For example, a vacation request can be in a "standby" or "approved" state. A system implementing this logic could deal with this change of state as an event that can be produced, detected and used by several components within the architecture.

www.EBooksWorld.ir

The EDA architecture pattern can be applied in design and implementation of applications that transmit events throughout different objects (components and services that are loosely coupled, if possible). An event driven system will normally have event broadcasters (also called *Agents*) and event consumers (also known as *sink*). *Sinks* are in charge of performing a reaction as soon as the event occurs. This reaction may or may not be fully provided by the sink itself. For example, the *sink* can be in charge of filtering, transforming and sending the event to another component or it can provide a reaction to the event itself.

The applications and systems built around the concept of event orientation enables them to react much more naturally and closer to the real world, because the event oriented systems are, by design, more oriented to asynchronous and unpredictable environments (the typical example are *Workflows*, but we should not limit EDA to *Workflows only*).

**EDA** can perfectly complement a *DDD N-Layer* architecture and service oriented architectures (SOA) because the logic of domain and web services can be activated by triggers related to input events. This paradigm is particularly useful when the *sink* itself does not provide the feedback/reaction expected.

This 'intelligence' based on events facilitates design and implementation of business automated processes as well as user oriented workflows (*Human Workflows*); it is even useful for machinery processes (devices such as sensors, actuators, controllers, etc.) that can detect changes in objects or conditions to create events that can be then processed by a service or system.

Therefore, **EDA can be implemented in any event oriented area, be it *Workflows*, Domain rules processes, or even presentation layers based on events (such as MVP and MVVM), etc**.

EDA is also closely related to the **CQRS** (*Command and Query Responsibility Segregation*) pattern, which will be introduced later.

Finally, note that in this Architecture proposal, as well as in our sample application published on CODEPLEX, we are not using EDA, we simply introduce it here as an aspect of architecture for advanced scenarios to which we can evolve. It is also possible that in the next versions we will evolve this architecture towards EDA.

## 4.- DUAL ACCESS TO DATA SOURCES

In most systems, users need to see data and perform many types of searches, ordering and filtering, regardless of the transactional and/or update operations.

To perform such queries with the purpose of visualizing (reports, queries, etc.) only, we could use the same classes of domain logic and related data access repositories that we used for transactional operations (in many applications we will do it this way). However, if the goal is to reach the highest optimization and performance, this is probably not the best option.

In short, showing information to the user is not linked to most behaviors of the domain (business rules), or to the problems of concurrent updates (Optimistic

www.EBooksWorld.ir

Concurrency Management or its exception management). Therefore, it is not linked to *self-tracking* disconnected entities, which are necessary for optimistic concurrency management. All these issues finally impact on pure performance of queries and the only thing we really want to do in this case is to perform queries with very good performance. Even if we have security or other types of requirements that are also related to pure data queries (reports, listings, etc.), this can be also implemented somewhere else.

Of course, if only one model and data source access can be used, then escalating and optimizing performance will not be achieved. In the end, *"a knife is made for meat and a spoon for soup".* With a lot of effort we can cut meat with a spoon, but it is not the best option.

It is quite normal for Software Architects and Developers to be inflexible in the definition of certain requirements which are sometimes unnecessary, or not required from a business standpoint. This is probably one of those cases. The decision to use the domain model entities just to show information (only for visualization, reports, listings, etc.) is really something self-imposed by the developers or architects, but it does not have to be this way.

Another different aspect is that, in many multi-user systems, changes do not need to be visible immediately for the rest of the users. If so, why use the same domain, repositories and transactional data sources to show them? If the behavior of such domains is not necessary, why go through them? For example, it is quite possible that queries (for reports and display only) perform much better in many cases if a second database, based on cubes (BI) is used (e.g. SQL Server OLAP, etc.) and then to access it with an easier and lighter mechanism (e.g., using one simple data access library; the best way to get the highest performance is probably not to use an ORM).

In conclusion, in some systems the best architecture could be based on two internal foundations of data access:

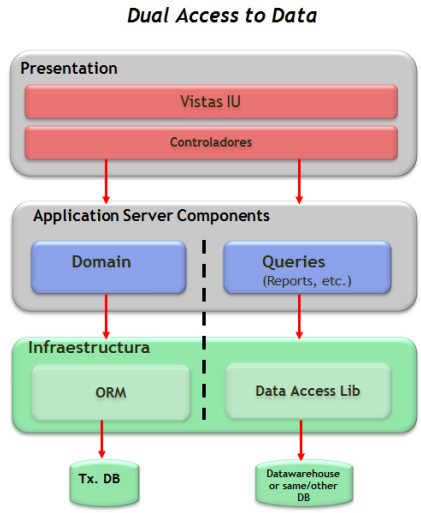



**Figure 21.- Dual Data Access**

www.EBooksWorld.ir

It is important to note that in this model/architecture, the column on the right is used only for pure queries (reports, listings, visualizations). In contrast, the column on the left (Domain and ORM) will continue making queries for cases when such returned data can be modified by the user (e.g., by using *databinding*, etc.).

Also, the feasibility of having a different database (even of a different type, such as relational vs. cubes) largely depends on the nature of the application. However, if feasible, it is the best option, since queries for "read only" data will never be interfered with. This ultimately maximizes scalability and performance of each type of operation. However, in this case, some type of data synchronization between the different databases will be required.

In sum, the final goal is to *"place all the code on the relevant part of the system, in a granular, focused manner that can be tested in an automated way."*

# 5.- PHYSICAL TIERS (TIERS) DEPLOYMENT

Tiers represent physical separations of presentation, business and data functionalities in various machines, such as servers (for business logic and database) and other systems (PCs for remote presentation layers, etc.) The common design patterns based on tiers are "*2-Tier", "3-Tier" and ultimately "N-Tier"*.

### 2-Tier

This pattern represents a basic structure with two main tiers, a client tier and a database server. In a typical web scenario, the client is the presentation layer and the business logic normally coexists in the same server, which in turn accesses the database server. So in Web scenarios, the client tier usually contains both the presentation layer and the business logic layer, and it is important, for the sake of maintenance, that such logical layers be present within that single client application.

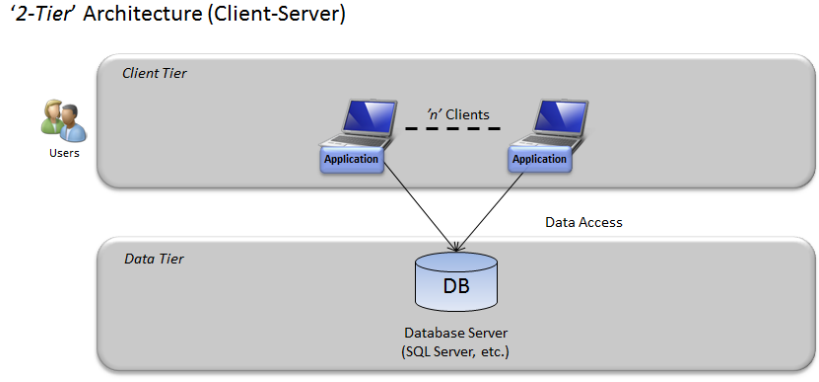



**Figure 22.- Client-Server Architecture**

www.EBooksWorld.ir

### 3-Tier

In a "3-Tier" design pattern, the user interacts with a client application physically deployed on his machine (usually a PC). This client application communicates with an application server (Web/App Tier) that will have the business logic and data access logical layers embedded. Finally, this application server accesses a third tier (Data tier) which is the database server. This pattern is very common in all Rich-Client, RIA and OBA applications as well as in web scenarios where the client is a simple web browser.

The following graph illustrates this "**3-Tier"** deployment model:

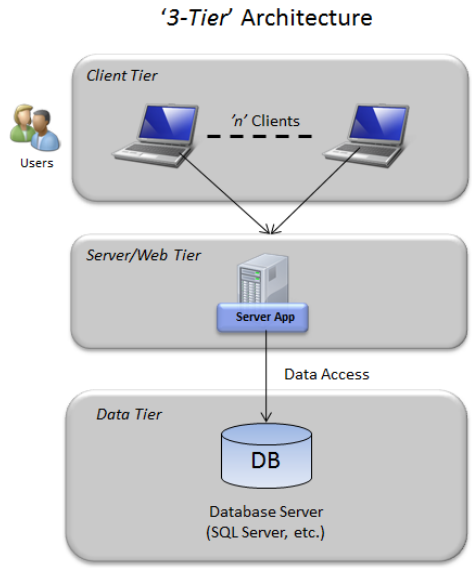



**Figure 23.- '3-tier' pattern**

### N-Tier

In this scenario, the Web server (which contains the presentation layer) is physically separated from the application server that exclusively contains business logic and data access layers. For reasons of network security policies, this separation is usually done in a normal way, where the web server is deployed in a perimeter network and accesses the application server situated on a different subnet through a firewall. It is also common for there to be a second firewall between the client tier and the Web tier.

www.EBooksWorld.ir

The following illustrates the **"N-Tier"** deployment pattern**:**

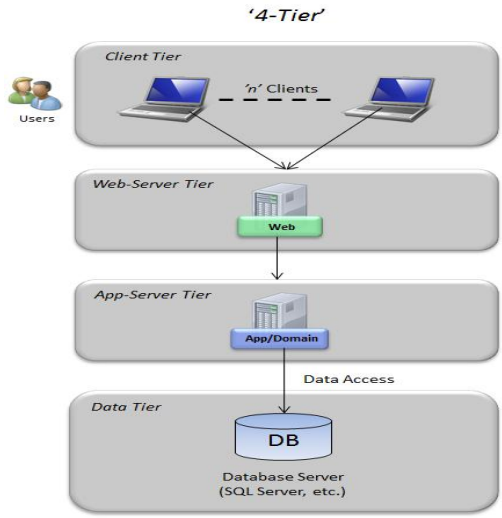



**Figure 24.- '4-Tier' pattern**

**Choosing tiers in architecture**

Applying tiers that physically separate our logical layers impacts the application's performance (because of the latency of the remote communications between the different tiers), although it can be beneficial for scalability by distributing the load among the different servers. Separating the most sensitive components of the application to different networks can also improve security. However, it should be considered that adding tiers increases deployment complexity and sometimes impacts performance, so no more tiers than necessary should be added.

In most cases, all the code of the application should be located on the same server or same tier of balanced servers. Whenever you use remote communications, performance will be affected by the latency of communications as well as by the fact that data should be serialized to be transmitted across the network. However, in some cases, we may need to divide functionality into different server tiers because of security restrictions or scalability requirements. In those cases, it is always better to choose communication protocols that are optimized to improve performance (TCP vs. HTTP, etc.).

**Consider "2-Tier" pattern if:**

- **Web application**: The intention is to develop a typical Web application, with the highest performance and no network security restrictions. If the intention is to increase scalability, the Web server should be cloned in multiple balanced servers.

www.EBooksWorld.ir

- **Client-server application**. The intention is to develop a client-server application that directly accesses a database server. This scenario is very different, since all the logical layers would be located at a client tier that in this case would be the client PC. This architecture is useful when there is a need for higher performance and quick access to the database, however, client-server architectures have many scalability, maintenance and troubleshooting problems, since the entire business logic and data access are moved to one single tier which is the user's PC, and they are left at the mercy of the different configurations of each end user. This case is not recommended in most situations.

**Consider "3-Tier" pattern if:**

- The intention is to develop a "3-Tier" application with remote client running in the user client machine ("Rich-Client", RIA, OBA, etc.) and an application server with web services publishing the business logic.
- All application servers can be located in the same network.
- An application called "intranet" is being developed where the security requirements do not require the separation of the presentation layer from the business layers and data access.
- The intention is to develop a typical Web application with the maximum performance

**Consider "N-Tier" pattern if:**

- There are security requirement demands where the business logic cannot be deployed in the perimeter network where the presentation layer servers are located.
- There is a very heavy application code (it uses the server resources intensively) and to improve the scalability, such business component functionality is separated at other server levels.

www.EBooksWorld.ir

CHAPTER 4

# Data Persistence Infrastructure Layer

## 1.- DATA PERSISTENCE INFRASTRUCTURE LAYER

This section describes the architecture of the data persistence layer. Following the trends of DDD architecture, the **Data Persistence Layer** is actually part of the '*Infrastructure layer'* (as defined in the DDD architecture proposed by Eric Evans); because it is related to specific technologies (data persistence technologies, in this case). However, due to the importance data persistence has in an application and to a certain parallelism and relationship with the Domain Layer, our proposal in this Architecture guide is that it should be predominant and have its own identity regarding the rest of the infrastructure aspects (also associated with specific technologies), which we call "Cross-Cutting Infrastructure" and will be explained in another chapter in detail. As a result, we are also aligned with traditional N-Layer architectures where the *"Data Access Layer"* is considered as an item/layer with its own identity (although it is not exactly the same layer concept).

Therefore, this chapter describes key guidance in order to design an application Data **persistence layer**. We will discuss how this layer fits and is placed into the proposed N-layer Domain Oriented Architecture as well as the usual patterns, components and problems to be considered when designing this layer. Finally, in the last section of this chapter we discuss the technical options and proposed implementation using .Net technologies.

The data persistence components provide access to the data hosted within the boundaries of our system (e.g., our main database), and also to the data exposed outside the boundaries of our system, such as Web services of external systems. Therefore, it has components of the "Repository" type that provide functionality to access the data hosted within the boundaries of our system, or "Service Agents" that will use Web



www.EBooksWorld.ir

Services exposed by other external backend systems. In addition, this layer will usually have base classes/components with reusable code for all the repository classes.

## 2.- LOGICAL DESIGN AND ARCHITECTURE OF THE DATA PERSISTENCE LAYER

The following diagram shows how the Data Persistence layer typically fits within our *N-Layer Domain-Oriented* architecture:

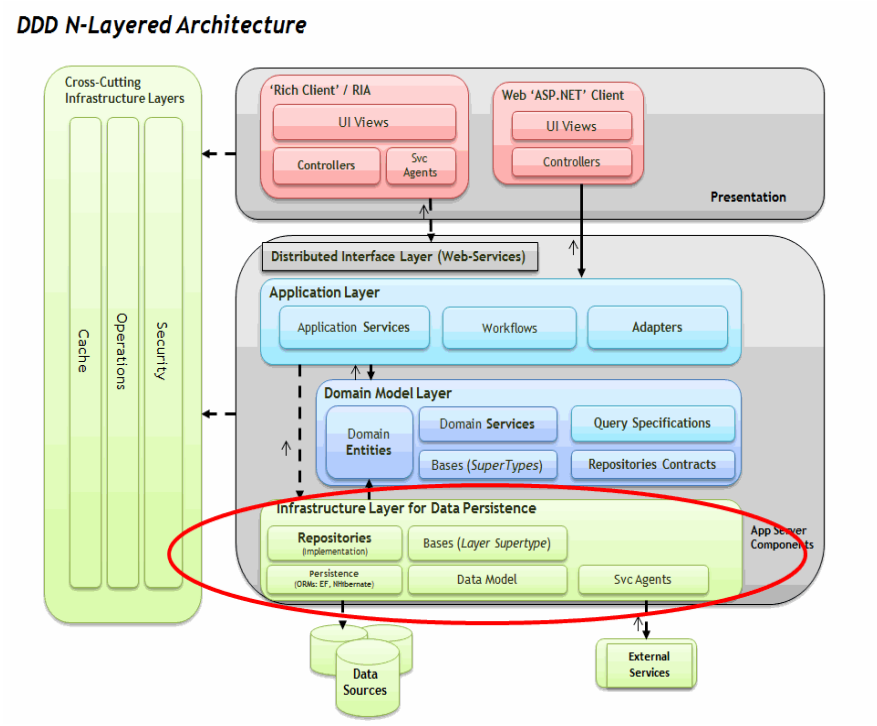



**Figure 1.- Data Persistence Layer location within the N-Layered Architecture**

## 2.1.- Data Persistence Layer Elements

The Data Persistence Layer usually includes different types of components. The following section briefly explains the responsibilities of each type of element proposed for this layer.

www.EBooksWorld.ir

## 2.1.1.- Repositories (Repository Pattern)

In some aspects, these components are very similar to the Data Access (DAL) components of traditional N-*Layer* architecture. Basically, Repositories are classes/components that encapsulate the logic required to access the application data sources. Therefore, they centralize common data access functionality so the application can have a better maintainability and de-coupling between technology and logic owned by the "Application" and "Domain" layers. If we use base technologies such as an O/RM (*Object/Relational Mapping Frameworks*), the code to be implemented is much more simplified, so we can focus on the *data access logic* rather than on *data access plumbing* (database connections, SQL statements, etc.). On the other hand, if we use lower level data access components (e.g. typical ADO.Net classes), it is usually necessary to have reusable utility classes that help to build our data access components.

**It is essential to differentiate between a '*Data Access*' object (used in many traditional N-layered architectures) and a Repository.** A Data Access object directly performs *data access* and *persistence operations* against the storage (usually a relational database). However, a repository "records/marks" the data that it works with in the memory as well as the operations it intends to perform against the storage (but these will not be performed immediately). These persistence operations will be really performed at a later time from the Application layer in a single action, all at once. The decision about "Applying changes" in memory into the real storage is usually based on the "Unit of Work" pattern, which is explained in the "Application layer" chapter in detail. In many cases, this pattern or way of applying operations against the storage can increase the application performance and reduce the possibility of inconsistencies. Also, it reduces transaction blocking in the database tables because all the intended operations will be committed as part of one transaction which will be more efficiently run in comparison to a regular data access class that does not group actions against the storage. Therefore, the selected O/RM will be given the possibility of optimizing the execution against the database (e.g., grouping several update actions) as opposed to many small separate executions.

**Repository Pattern**

'*Repository*' is one of the well documented ways of working with a data source. *Martin Fowler* in his PoEAA book describes a repository as follows:

"*A repository performs the tasks of an intermediary between the domain model layers and data mapping, acting in a similar way to a set of domain objects in memory. Client objects declaratively build queries and send them to the repositories for answers. Conceptually, a repository encapsulates a set of objects stored in the database and operations that can be performed on them, providing a way that is closer to the persistence layer. Repositories, also, support the purpose of separating, clearly*

www.EBooksWorld.ir

*and in one direction, the dependency between the work domain and the data allocation or mapping*".

This is currently one of the most common patterns, especially in *Domain Driven Design*, because it allows us to easily make our data layers "testable", and to achieve object orientation more symmetrically with our relational models. *Microsoft Patterns & Practices* has an implementation of this pattern called *Repository Factory*, available for download in CodePlex (we only recommend it for analysis, and not to actually use it because it makes use of technologies and frameworks that are somewhat outdated.).

**Hence, for each type of class that needs global access (usually for each Aggregate's Root Entity) we should create a Repository class that provides access to a set of objects in memory of all such classes. Access should be made through a well-known interface, and it should have methods in place in order to query, add, modify and remove objects which will actually encapsulate the insertion or removal of data in the data storage. Regarding queries, it should provide methods that select and gather objects based on certain selection criteria, and those returned objects are usually Domain Entities.**

**It is important to re-emphasize that REPOSITORIES should only be defined for the main logical entities (in DDD that means we have to create one Repository for each AGGREGATE *root or isolated ENTITY)*. In other words, we will not create a Repository for each table in a data source.**

**All these recommendations help the development of the higher layers (such as the Domain Layer) to focus on the model, and on the other hand, all data access and object persistence is delegated to the REPOSITORIES.**

**Table 1.- Repository Rule**

| Rule # D4. | **Design and implement Repository classes within the data persistence layer** |
|---|---|

**Rules**

- To encapsulate the data persistence logic, you should design and implement Repository classes. Repositories are usually supported by ORMs.

**Advantages of using Repositories**

- The developers of the domain and application layers will deal with a much simpler model in order to retrieve "persisted objects/entities" and to manage their object's life cycle.
- It de-couples the APPLICATION and DOMAIN layer from the persistence technology, multiple-database strategies, or even multiple data sources.

www.EBooksWorld.ir

- The can be easily replaced by fake data access implementations which are particularly useful in *testing* (especially unit-testing) of the domain logic. As a result, during the unit-testing time, access to the real database could be dynamically replaced by access to collections in memory which are '*hard-coded'* data. This is good for 'Unit Testing' as all data would be always the same when testing, so changes within the database will not impact our tests.

**References**

*'Repository' Pattern. By Martin Fowler.*
***http://martinfowler.com/eaaCatalog/repository.html***
*'Repository' Pattern. By Eric Evans in his DDD book.*

As shown in the following figure, we have a *Repository* class for each 'main entity' (also called *AGGREGATE's root entity* in DDD terminology), that can be persisted/represented in the database by one or more tables.

In other words, only one type of "object" within an Aggregate will be the root which data access will be channeled through:

***Relationship between Repositories and Entities***

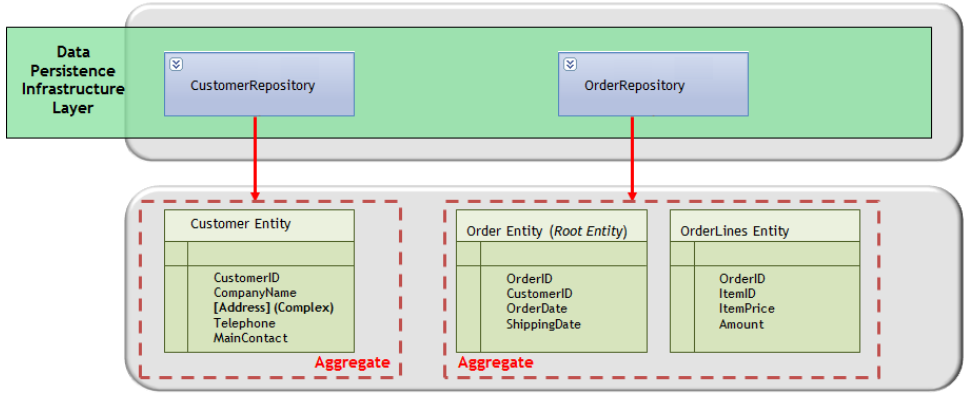



**Figure 2.- Relationship between Repository Classes and Entity Classes**

For example, in the diagram above, one of the root objects would be '*Order*' for the Order and OrderLines entities aggregation.

www.EBooksWorld.ir

**Table 2.- Repositories as a unique channel to access data storages**

**Rule # D5.** **Repository classes (data access and persistence classes) as a unique channel for data storage access**

○ **Rule**

- **In a project with an DDD architecture, the only interlocutors with data storages (typically relational databases or other type of storages) will be the Repositories.**
  This does not mean that in systems outside the *Domain Oriented* architecture, there cannot be parallel access to these storages through other paths.
  For example, when integrating a transactional database with a BI system or when generating reports with reporting tools, there is going to be another path providing direct access to data storages and that path could have nothing to do with our Repositories. Another example regarding parallel paths to access data could be CQRS architectures for very high scalable applications.

**Table 3.- Layer Supertype Rule**

**Rule # D6.** **Implement a "Layer Supertype" pattern for the Repositories Sub-layer**

○ **Recommendations**

- It is common and very useful to have "base classes" for every layer to group and reuse common behaviors that otherwise would be duplicated in different parts of the system. This simple pattern is called "*Layer SuperType*".

- It is particularly useful when we have similar data access code for different domain entities.

**References**

*'Layer Supertype' Pattern by* ***Martin Fowler***.
***http://martinfowler.com/eaaCatalog/layerSupertype.html***

www.EBooksWorld.ir

**Relationship between Query Specifications and Repositories**

Query specifications are an open and extensible way of defining query criteria. They are defined within the Domain layer; however, they are applied and used when coordinating Repositories in the Application Layer. This will be further explained in the Domain layer chapter because they are defined in the *Domain Layer* and used/coordinated within the *Application Layer*.

## 2.1.2.- Data Model

This concept, provided by some O/RM solutions, is used when implementing the Data Persistence Layer to define and sometimes visualize the "entity-relation" data model of the application.

As said before, this concept is provided by some O/RM solutions, so it is fully related to a specific infrastructure technology (e.g., *Entity Framework* provides a way to create an entity data model or even to create it from an existing database).

## 2.1.3.- Persistence Technology (O/RM, etc.)

The repository and UnitOfWork classes use the chosen data persistent technology internally like an O/RM such as *Entity Framework* or *NHibernate*, or simply the lower level technologies such as ADO.NET basic classes to access the database.

The details on how to implement the Data Persistence Layer in a specific technology is explained in the section "Data Persistence Layer Implementation" at the end of this chapter.

## 2.1.4.- External Distributed Services Agents

When a domain component should access data provided by an external distributed service (e.g. a Web Service), we should implement code that manages the communication semantics of this service in particular. These Service Agents implement external data access components that encapsulate and isolate the requirements of the distributed Services and can even support additional aspects such as cache, offline support and basic mapping between the data format exposed in the external distributed Services and the data format required/used by our application.

www.EBooksWorld.ir

## 2.2.- Other Data Access Patterns

The patterns we explain below helps to understand the different possibilities of data access strategies and are therefore useful for a better understanding of the options chosen by this architecture and design guide.

Although it might seem odd after so many years of technological advances, accessing data is still an important and extremely delicate issue within our developments. It is so delicate that it could "blow up" an entire project. The large amount of current techniques and patterns regarding data access only increase the level of confusion of many developers. Of course, each possible technique adds favorable elements which others do not, so selecting a suitable option is important for the project's lifetime.

It is always good to remember some well-known and well-documented patterns; these will undoubtedly help us to understand the philosophy of this Architecture and Design guide.

### 2.2.1.- Active Record

Active Record is one of the most used and best known patterns and, as sometimes happens with patterns, we do not know their given name although we have used them many times. In his book "*Patterns Of Enterprise Application Architecture: PoEAA*", *Martin Fowler* defines an 'Active Record' object as an object transporting not only data but also behavior, that is, an Active Record keeps the logic of its persistence within the object's domain.

This design pattern is put into practice for many implementations of dynamic languages such as *Ruby* and nowadays it is widely used by the developers' community. Currently in .NET, there are many implementations such as *Castle Active Record*, *.NetTiersApplication Framework* or *LLBLGenPro*.

However, one of the most important inconveniences of this pattern comes from its own definition, as it does not conceptually separate the data transportation from its persistence mechanisms. If we think about service oriented architectures where the separation between data contracts and operations is one of the main pillars, we will see that a solution like *Active Record* is not suitable, and is, quite often, extremely hard to implement and maintain. Another example of a solution based on '*Active Record*' which would not be a good choice, is one without a 1:1 relationship between the database tables and Active Record objects in the domain models, since the logic that these objects need to carry would be a bit complex.

www.EBooksWorld.ir

## 2.2.2.- Table Data Gateway

This pattern, also perfectly documented in **PoEAA [M. Fowler],** could be seen as an improvement of Active Record mentioned above, which tries to separate the data transport from the persistence operations. For many developers this is an improvement because it delegates the entire database interaction job to some intermediary, or *gateway* object. Like Active Record, this pattern works well when our entities are mapped to the database tables 1:1; however, when our domain model involves more complicated elements such as inheritance, complex or associated types, this pattern loses its strength and, in many cases, does not make any sense.

## 2.2.3.- Data Mapper

If we think about the above patterns, we will see that both domain entities are tightly coupled to the data model. In fact, object models and data models have different mechanisms to structure data. Sometimes, they prevent developers from being able to leverage the entire knowledge of object orientation when working with databases or restrict development because of a certain relational model.

There are many differences between relational models and object oriented models, generally known as the '*impedance mismatch'*. A good example of this impedance mismatch is how the relationships are implemented in both worlds. In relational models, relationships are established through data duplication in different tables. For instance, if we want to relate a tuple of Table B with a tuple of Table A, we would create a column in Table B with a value that allows us to identify the tuple of Table A that we want to have a relationship with. However, in object oriented programming languages there is no need to duplicate data to create relationships; object B can simply hold a reference to object A to set the desired relationship, which is known as an association in the Object-Oriented world.

The main purpose of the *Data Mapper* pattern is to separate the object model structures from the relational model structures and then to perform data mapping between them.

When using a *Data Mapper*, objects that consume 'Data Mapper' components ignore the present database schema, and of course, they do not need to make use of the SQL statements.

www.EBooksWorld.ir

## 2.2.4.- List of Patterns for the Data Persistence Layer

In the following table we list possible patterns for the data persistence layer:

**Table 4.- Categories/Patterns**

| Patterns |
| --- |
| • Active Record<br>• Data Mapper<br>• Query Object<br>• Repository<br>• Row Data Gateway<br>• Table Data Gateway<br>• Table Module |

**Additional references**

For information on Domain Model, Table Module, Coarse-Grained Lock, Implicit Lock, Transaction Script, Active Record, Data Mapper, Optimistic Offline Locking, Pessimistic Offline Locking, Query Object, Repository, Row Data Gateway, and Table Data Gateway patterns, see:

*"Patterns of Enterprise Application Architecture (P of EAA)" in* ***http://martinfowler.com/eaaCatalog/***

# 3.- TESTING IN THE DATA PERSISTENCE INFRASTRUCTURE LAYER

Like most elements of a solution, our Data Persistence layer is another area that should be covered by unit testing. It should, of course, meet the same requirements demanded from the rest of the layers or parts of the project. The implication of an external dependency such as a database has special considerations. These should be treated

www.EBooksWorld.ir

carefully so as not to fall into certain common anti-patterns when designing unit tests. In particular, the following defects in the created tests should be avoided.

**Anti-patterns to avoid:**

- ***Erratic Tests*.** One or more tests are behaving erratically; sometimes they pass and sometimes they fail. The main impact of this type of behavior comes from the treatment they are given, since they are usually ignored and they could hide some code failure internally that is not dealt with.

- ***Slow tests*.** The tests take too long to run. This symptom usually prevents developers from running system tests when one or more changes are made. This, in turn, reduces the code quality because it is exempted from continuous testing on it, while the productivity of the people in charge of keeping and running those tests is also reduced.

- ***Obscure Test.*** The real behavior of the test is obscured very frequently due to certain elements of test initialization and cleaning processes or initial data reset processes, and it cannot be understood at a glance.

- ***Unrepeatable Test*:** A test behaves differently the first time it is run than how it behaves on subsequent test runs.

Some **usual solutions** to perform tests where a database is involved can be seen in the following items, although, of course, they are not exclusive:

- **Database isolation:** a different database, separated from the rest, is provided or used for each developer or tester running tests involving the data infrastructure layer.

- **Undoing changes upon the completion of every test (Roll-back):** Undoing changes made in the process of running each test. When working with databases we can achieve this goal (undoing operations) using transactions (performing roll-back after the completion of each test). The problem is that this alternative impacts the speed of test executions.

- **Redoing the set of data upon completion of each test:** this consists of redoing a set of data to its initial state, in order to immediately repeat it.

www.EBooksWorld.ir

**Table 5.- Repository Rule**

**Rule # D7.** **Unit Testing the data persistence infrastructure layer**

**Recommendations**

- Enable the persistence infrastructure layer to inject dependencies regarding what component performs the operations on the database. This will allow simulation of a fake data storage access and injecting it dynamically. Thus, a big set of unit tests could be run quickly and reliably. This would also allow to not using a real database when performing unit testing to upper layers, in a dynamic way. On the other hand we could also choose to perform integration tests against a real database, again, in a dynamic way like changing a single application setting. This capability (dynamic change based on appsettings) is implemented in the Sample Application.

- If the persistence infrastructure layer introduces a Layer Supertype for common methods, make use of test inheritance (if the framework used allows it), to improve productivity at their creation.

- Implement a mechanism that allows the developer or tester to easily make changes if the set of tests is run with simulated objects or against a real database.

- When tests are executed against a real database we should ensure we are not falling into the **Unrepeatable Test** or the **Erratic Test anti-patterns.**

**References**

*MSDN UnitTesting*
**http://msdn.microsoft.com/en-us/magazine/cc163665.aspx**
*Unit Testing Patterns*
**http://xunitpatterns.com/**

www.EBooksWorld.ir

## 4.- DATA ACCESS DESIGN CONSIDERATIONS

The data access and Persistence Layer should meet the requirements for performance, security and maintainability and should also support changes in requirements. When designing the Data Persistence Layer, the following design guidelines should be considered:

- **Select a proper data access technology.** Choice of technology depends on the data type to be managed and on how you want to handle it within the application. Certain technologies are better indicated for certain areas. For example, although O/RM is recommended for most data access scenarios in DDD architecture, in some cases (*Business Intelligence*, reporting/queries, etc.), it might not be the best option. In these cases, the use of other technologies should be taken into account.

- **Use abstraction to de-couple the Persistence Layer components from other components.** This can be done by extracting interfaces (contracts) from all Repositories and implementing those interfaces. We must take into account that these interfaces/contracts must not be defined within the persistence layer (infrastructure) but in the Domain layer (as they are contracts proposed by the Domain). In short, the contract is what the Domain requires from a Repository so that it can work in the application. The Repository is the implementation of said contract. Using this approach we can really leverage the power of interfaces if we use IoC containers and Dependency Injection to instantiate Repositories from the application layer.

- **Decide how to manage and protect database connections information**. As a general rule, the Data Persistence Layer will be in charge of managing all the connections to data sources required by the application. Proper steps to keep and protect the connection information should be chosen. For example, by encrypting the configuration file sections, etc.

- **Determine how to manage data exceptions.** This Data Persistence Layer should catch and (at least initially) handle all the exceptions related to data sources and CRUD (*Create, Read, Update and Delete*) operations. The exceptions related to the data and "timeouts" errors of the data sources should be managed in this layer and transferred to other layers only if the failure affects functionality and response of the application. A summary of possible exceptions to take into account are the following:

    - Transient infrastructure errors that can be resolved by retrying and will not affect the application: those can be handled by this layer transparently.

www.EBooksWorld.ir

  - 'Data' errors that might be handled here, in upper application layers, by the user or even not handled at all, such as concurrency violations, validation errors, etc.

  - Invalid operations that are really code defects that the developer will need to fix, and hence shouldn't be handled at all.

- **Consider security risks**. This layer should protect against attacks attempting to steal or corrupt data, as well as protect mechanisms used to access the data sources. For example, care must be taken not to return confidential information on errors/exceptions related to data access, as well as to access data sources with the lowest possible credentials (not using 'database administrator' users). Additionally, data access should be performed through parameterized queries (ORMs do this by default) and should never form SQL statements through string concatenation, to avoid SQL Injection attacks.

- **Consider scalability and performance goals**. These goals should be kept in mind during the application design. For example, if an e-commerce application must be designed to be used by Internet users, data access may become a bottleneck. For all cases where performance and scalability is critical, consider strategies based on Cache, as long as the business logic allows it. Also, perform query analysis through profiling tools to be able to determine possible improvement points. Other considerations about performance are the following:

  - Use the Connection Pool, for which the number of credentials accessing the database server should be minimized.

  - In some cases, consider *batch* commands (several operations in the same SQL statement execution).

  - Consider using the optimistic concurrency control with non-volatile data to mitigate the data block cost in the database. This avoids having too many locks in the database, including database connections which should be kept open during locks (for instance, when using pessimistic concurrency control).

- **Mapping objects to relational data**. In a DDD approach usually based on Domain Entities, O/RMs may significantly reduce the amount of code to implement the data persistence layer. For more information on DDD, read the initial chapter of Architecture. Consider the following items when using frameworks and O/RM tools:

  - O/RM tools may allow design of an entity-relation model and generate a real database schema (this approach is called '*Model First*' in EF)

www.EBooksWorld.ir

while establishing the mapping between objects/entities of the domain and database.

- If the database already exists, the O/RM tools usually also allow generation of the entity-relation model of data from this existing database and then mapping of the objects/entities of the domain and database.

- A third approach can be '*Code First*' which consists in coding entity classes and generating the database schema from those entity classes. Using this approach there will be no visual entity model, just entity classes. This is probably the purest DDD approach

- **Stored procedures**. In the past, the stored procedures in some DBMS (Database Management Systems) provided an improvement in performance when compared to the dynamic SQL statements (because the stored procedures were compiled in a certain way, and the dynamic SQL statements were not). But in current DBMS, performance of the dynamic SQL statements and stored procedures is similar. There are several reasons for using stored procedures. For example, to separate data access from development so that a database expert can tune the stored procedures without needing to have the development 'know how' or without touching the application code base. However, the disadvantage of using stored procedures is that they completely depend on the chosen DBMS, with specific stored procedures code for each DBMS. On the other hand, some O/RMs are capable of generating native 'Ad-Hoc' SQL statements for different DBMS they support, so the application portability of one DBMS to another would be practically immediate.

  Another relevant reason to use stored procedures is that you can code data processing logic that require multiple steps and still have a single round-trip to the database. For this reason the performance of dynamic SQL against stored procedure is only comparable when the stored procedure in question is simple (i.e. it only contains one SQL statement).

  - Some O/RMs support the use of stored procedures. Logically, however, the portability to different DBMS is lost by doing so.

  - For the sake of security, typed parameters should be used when using stored procedures to avoid SQL injections. We also need to guard against having code inside the stored procedure that takes one of the input string parameters and uses it as part of a dynamic SQL statement that it later executes.

  - *Debugging* of queries based on dynamic SQL and O/RM is easier than doing so with stored procedures.

www.EBooksWorld.ir

- In general, whether stored procedures are used or not largely depend on the company policy as well. However, if there are no such policies, the general recommendation is to use O/RMs and stored procedures for particular cases of very complex and heavy queries that are meant to be highly controlled and that can be improved in the future by DBA/SQL experts.

- **Data Validation**. Most data validations should be performed in the Application and Domain Layer, since data validations related to business rules must be performed in those layers. However there are some types of data validations exclusively related to the Persistence Layer, such as:

  - Validating input parameters to correctly manage the NULL values and filter invalid characters.

  - Validating input parameters by examining characters or patterns that can attempt SQL injection attacks.

  - Returning informative error messages if validation fails, but hiding confidential information that can be generated in the exceptions.

- **Deployment considerations**. In the deployment design, the purpose of the architecture consists in balancing performance, scalability and security aspects of the application in the production environment, depending on the requirements and priorities of the application. The following guidelines should be considered:

  - Place the Data Persistence Infrastructure layer (components) in the same physical level as the Domain and Application Layer to maximize the application performance. The contrary is advisable only in the event of security restrictions and/or certain scalability cases that are not very common. However, if there are no restrictions, the Domain Layer, Application layer and the data access or persistence layer should usually be physically within the same application servers.

  - As far as possible, place the Data persistence infrastructure layer in servers different from the Database server. If it is placed in the same server, the DBMS will be constantly competing with the application itself to obtain server resources (processor and memory), which will harm application performance.

www.EBooksWorld.ir

## 4.1.- General References

".*NET Data Access Architecture Guide*"
**http://msdn.microsoft.com/en-us/library/ms978510.aspx**.

*"Concurrency Control*"
**http://msdn.microsoft.com/en-us/library/ms978457.aspx**.

"*Data Patterns*"
**http://msdn.microsoft.com/en-us/library/ms998446.aspx**.

"*Designing Data Tier Components and Passing Data Through Tiers*"
**http://msdn.microsoft.com/en-us/library/ms978496.aspx**.

"*Typing, storage, reading, and writing BLOBs*"
**http://msdn.microsoft.com/en-us/library/ms978510.aspx#daag_handlingblobs**.

"*Using stored procedures instead of SQL statements*"
**http://msdn.microsoft.com/en-us/library/ms978510.aspx**.

"*NHibernate Forge*" community site
**http://nhforge.org/Default.aspx**.

*ADO.NET Entity Framework*
**http://msdn.microsoft.com**

www.EBooksWorld.ir

# 5.- IMPLEMENTING DATA PERSISTENCE LAYER WITH .NET 4.0 AND ENTITY FRAMEWORK 4.0

The explanation and logical definition of this layer (design and patterns) is given in the first half of this chapter. Therefore, this section will not deal with the logical concepts of data persistence or Repository pattern, etc. The purpose of this chapter is to show the different technological options available to implement the Data Persistence Layer and, of course, to explain the technical option chosen by default in our .NET 4.0 Architecture.

The following diagram highlights the Data Persistence Layer location within a Visual Studio 2010 'Layer diagram':

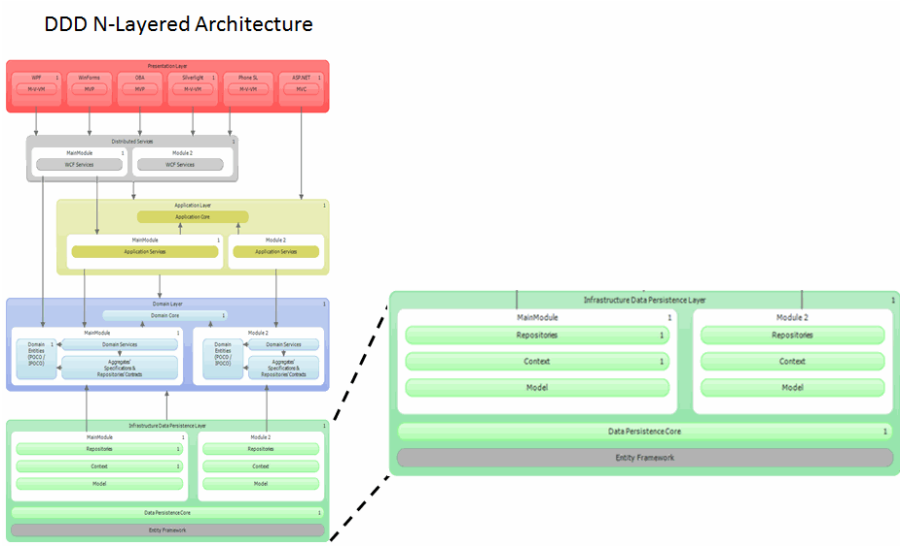



**Figure 1.- Data Persistence Layer Diagram in VS2010**

**Steps to be followed:**

1.- The first step will be to identify the limits of the data we want to access. This will help us to choose between the different technologies available for the implementation of the 'Repositories'. Are we dealing with relational databases? Which DBMS specifically? Or are we dealing with another type of data source?

2.- The next step is to choose the strategy required to convert domain objects to persisted data (usually databases) and to determine the data access approach. This includes identifying the business entities to be used and the format of

www.EBooksWorld.ir

those entities. Business entities are really Domain entities and have to be defined within the Domain layer and not in the Data persistence layer. However, we are actually analyzing the relationship of such Domain Entities with the Data Persistence layer. Many decisions regarding domain entities should be taken at this point (Persistence Layer implementation) because, depending on the technology we are using, they will have to be generated or developed on one way or another

3.- Finally, we should determine the error handling strategy to use in the management of exceptions and errors related to data sources.

## 5.1.- Technology Options for the Data Persistence Layer

### 5.1.1.- Selecting a Data Access Technology

The selection of a proper technology to access data should consider the type of data source we will have to work with and how we want to handle the data within the application. Some technologies are better adapted to certain scenarios. Some (mainly) Microsoft technologies and characteristics to be considered are:

- ***Entity Framework*:** Based on the ADO.NET platform, this option should be kept in mind if you want to create an entity model mapped to a relational database. At a higher level, one entity class is usually mapped to multiple tables that comprise a complex entity. The most outstanding advantage of EF is that the database it works with will be transparent in many ways. This is because the EF model generates native SQL statements required for each DBMS, so it would be transparent whether we are working against SQL Server, Oracle, DB2 or MySQL, etc. We simply need to change the EF provider related to each DBMS (In most cases, this involves nothing more than changing a connection string and regenerating the EF model). So, EF is suitable when the intention is to use an O/RM development model based on an object model mapped to a relational model through a flexible scheme. If you use EF, you will also probably use the following technology:
  - ***LINQ to Entities***: Consider '*LINQ to Entities*' if the intention is to execute strongly-typed queries against entities using an object-oriented syntax such as LINQ.
- ***ADO.NET*:** Consider using ADO.NET base classes if access to a lower API level is required. This will provide complete control over it (SQL statements, data connections, etc.) but relinquish the transparency provided by EF. You may also need to use ADO.NET if you need to reuse existing inversions

www.EBooksWorld.ir

(reusable services or existing Data Access Building Blocks) implemented using ADO.NET, naturally.

- ***Microsoft P&P Enterprise Library Data Access Building Block***: This data access library is based on ADO.NET. However, if possible, we recommend using *Entity Framework* instead, since EF is a full supported Microsoft technology and the P&P '*Building Block*' is technology older than EF. The 'Microsoft P&P' product team itself recommends EF, if possible, instead of this library.

- ***ADO.NET Sync Framework*:** Consider this technology if you are designing an application that should support scenarios occasionally disconnected/connected or that require cooperation between the different databases.

- ***LINQ to XML***: Consider this technology when there is an extensive use of XML documents within your application and you want to query them through LINQ syntax.

- ***Third party technologies***: There are many other good technologies (ORMs like NHibernate, etc.) which are not provided and supported by Microsoft.

## 5.1.2.- Other technical considerations

- If low level support is required for queries and parameters, use the plain ADO.NET objects.

- If you are using ASP.NET as a presentation layer to simply show read-only data (reports, lists, etc.) and when maximum performance is required, consider using ***Data Readers*** in order to maximize rendering performance. The ***Data Reader*** is ideal for 'read-only' and '*forward-only*' accesses where each row is processed very quickly. However, it doesn't fit at all within a DDD N-Layered Architecture style where we are de-coupling presentation layers from application layers and persistence layers.

- If you simply use ADO.NET and your database is SQL Server, use the SQL Client *provider* to maximize performance.

- If you use SQL Server 2008 or a higher version, consider using FILESTREAM to obtain higher flexibility in storage and access to BLOB type data.

- If you are designing a data persistence layer following **DDD** (*Domain Driven Design*) architectural style, the most recommended option is an O/RM framework **O/RM** such as **Entity Framework or NHibernate**.

www.EBooksWorld.ir

**Table 6.- Data persistence layer default technology**

**Rule # I1.**

**When using relational databases, the preferred default technology for implementing Repositories and Data Persistence Layer should be an ORM such as Microsoft ADO.NET Entity Framework.**

**Rule**

- According to previous considerations, it is convenient to use an O/RM in DDD Architectures. Since we are dealing with Microsoft technologies, the selected technology for data persistence will be ENTITY FRAMEWORK. Implementing *Repositories* and *Unit Of Work* with EF 4.0 is much more straightforward and easier than *'reinventing the wheel'* using plain ADO.NET classes.
- Another viable option could be to use any other third-party O/RM, such as NHibernate or similar.
- However, you should be open to using other technologies, (ADO.NET, *Reporting* technologies, etc.) for collateral aspects not related to the Domain logic and engine, such as *Business Intelligence*, or for read-only queries for reports and/or listings that should support the highest performance.

**Entity Framework Advantages**

- Database Engine Independence. An application database based on a specific DBMS can by transparently swapped by another database based on a different DBMS (SQL Server, Oracle, DB2, MySQL, etc.)
- Strongly Typed and object orientated programming model using 'LINQ to Entities'.
- The ability to have a "Convention over Configuration" programming model by choosing a "Code First" approach. O/RM wise, this is a totally unique feature built into Entity Framework.

**References**

**http://msdn.microsoft.com/en-us/data/aa937723.aspx**

www.EBooksWorld.ir

## 5.1.3.- How to get and persist objects in the Data storage

Once data source requirements are identified, the next step is to choose a strategy for data access and conversion to/from objects (domain entities). Likewise, we need to set our strategy regarding the transformation of these objects (probably modified) into data.

There is usually a typical impedance mismatch between the object oriented entity model and the relational data model. This sometimes makes "mapping" difficult. This mismatch can be addressed in several ways, but these differ depending on the type of data, structure, transactional techniques and how the data is handled. The best, most common approach is to use O/RM *frameworks*. Consider the following guidelines when choosing how to retrieve and persist business entities/objects to the data storage:

- Consider using an O/RM that performs mapping between domain entities and database objects. Additionally, if you are creating an application and a data store "from scratch", you can usually use the O/RM to generate the database schema from the logical entity-model defined in the application. (e.g., using the *EF 4.0 Model First approach*). On the other hand, if the database already exists, O/RM tools could be used for mapping the domain data model and the relational model.

- A common pattern associated to DDD is modeling domain entities with domain classes/objects. This has been logically explained in previous chapters.

- Make sure that entities are correctly grouped to achieve the highest level of cohesion. This means that you should group entities in *Aggregates* according to DDD patterns. This grouping must be part of your own logic. The current EF version does not provide the concept of *Aggregate* and *aggregate-root*.

- When working with Web applications or Web Services, sometimes entities should be grouped (e.g., in DTOs) so you can return entities with just required data and in a 'single shot'. This minimizes the use of resources by avoiding a 'chatty' model that calls to remote services too often (too many round-trips). This increases the application performance regarding communications.

## 5.2.- Entity Framework Possibilities in the Persistence Layer

As previously mentioned, **the technology selected** in this guide to implement the data persistence layer and therefore the Repositories in our DDD N-Layer architecture, i**s ENTITY FRAMEWORK**.

www.EBooksWorld.ir

## 5.2.1.- What does Entity Framework 4.0 provide?

As we have already discussed regarding data persistence, there are many different alternatives available. Each, of course, has advantages and disadvantages. One of the priorities regarding *Entity Framework* development has always been to recognize the importance of the latest programming trends and different developer profiles. From developers who like and feel comfortable and productive using wizards within the IDE, to those who prefer to have complete control over a code and their work.

One of the most important steps taken by EF 4.0 is providing the option of using your preferred domain entity type. Using EF 1.0 we could only use prescriptive entities which were tightly coupled to EF infrastructure. However, EF 4.0 offers the possibility of implementing our Domain Entities by POCOs or STE ("*Self Tracking Entities"*).

**Important**:
Before we can implement REPOSITORIES, we need to define the types/entities to be used.In the case of N-Layer Domain Oriented architectures, as mentioned; the business Entities should be located within the Domain Layer. However, when using the 'Model First' or 'Database First' approaches, the creation of such entities takes place during EF entity model creation which is defined in the data persistence infrastructure layer. But, before choosing how to create the Data persistence layer, we should choose what type of EF domain entities will be used (Prescriptive, POCO or STE). This analysis is explained in the Domain Layer chapter, so we recommend that the reader refer to this chapter and learns about the pros and cons of each type of possible EF entity type before moving forward in the current chapter.

## 5.3.- Domain Entity options using Entity Framework

EF 4.0 (and future versions) provides the following options regarding what kind of entities' implementation we can use:

- If following '***Model First***' or '***Database First***' approaches:
  - *Prescriptive Entities* (coupled to EF base classes and *EntityObject-based* template)
    - Need to use partial classes to add entity logic
  - *Self-Tracking Entities* (Using the STE T4 Template)

www.EBooksWorld.ir

        - Need to use partial classes to add entity logic
    - *POCO Entities* (Using the POCO T4 Template)
        - Need to use partial classes to add entity logic
- If following a **'*Code First*'** approach**:**
    - *POCO Entities* (Using your own POCO classes)
    - Directly mix data attributes and entity domain logic within your own POCO entity class.
    - At the time of this writing and *NLayerSampleApp* development, '*Code First*' was in CTP state (not released yet). This is why we did not use it as a viable option for our current sample application implementation.

We strongly encourage evaluating the last option (Code-First and Domain POCO entities plus DTOs for Distributed services) if you are looking for the purest DDD approach using .NET. It de-couples development between presentation layer data and domain entities, but all DTOs to Domain entities mapping, etc. have to be handled manually.

On the other hand, the *Self-Tracking Entities* approach is a more balanced approach which provides more initial productive development because of its self-tracking data capabilities.

Please refer to the chapter on Distributed Services to analyze aspects regarding DTOs vs. STE for N-Tier applications.

## 5.4.- Creation of the Entity Data Model

We chose *EF Self Tracking Entities* (STE) to implement our Domain Entities because it is probably the most balanced solution and especially designed for N-Tier scenarios (involving distributed services). Additionally, at the time of this writing 'Code First approach' was not available as released version.

All approaches except 'Code First' require an entity data model which can be created using EF modeler and Visual Studio. The creation and development procedure is quite similar when using the POCO templates or the STE templates, so if you choose to use the POCO templates, it will be quite similar. Therefore, the procedure requires the following steps.

First, note that this guide is not intended to teach the use of Visual Studio or .NET 4.0 step by step (like a *Walkthrough*). The great volume of Microsoft documentation or related books will provide that information. Therefore, this guide does not completely explain all the "details". Instead, we do intend to show the mapping between the technology and N-Layer Domain oriented architecture.

www.EBooksWorld.ir

However, we will deal with POCO/STE entities in EF 4.0 step by step because they are new in VS2010 and EF since EF 4.0 version.

To create the model, we will start with a class library project type. This assembly/project will contain everything associated with the data model and connection/access to the database, for a specific functional module of our application.

In our example, the project will be called:

*"Infrastructure.Data.MainModule"*

Take into account that its default namespace is longer (it can be changed within the project's properties):

*"Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule"*

Please note that, in this case, the vertical/functional module is simply called '*MainModule'*. We could have other modules like 'RRHH', 'CRM', or any other functional concept.

Then we add an EF data model called '**MainModuleDataModel**' to the project/assembly:

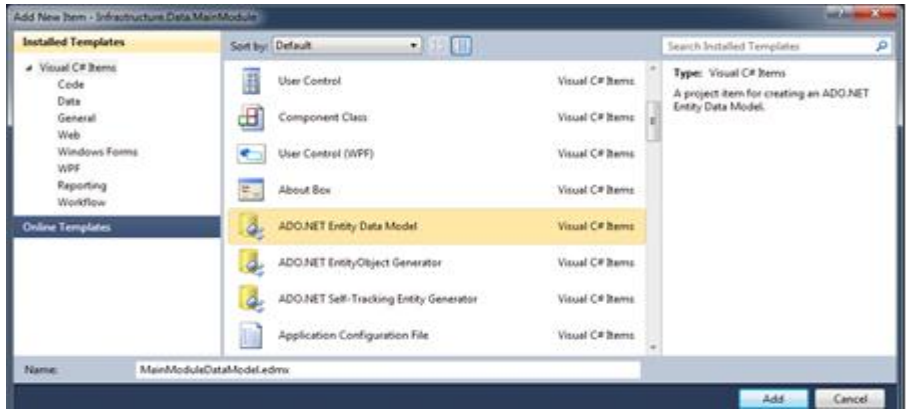



**Figure 4.- Creation of EF Data Entities Model**

If the model is going to be created from an existing database (that is the case now), we will have to provide the information of the database connection. It is very important to give the EF Context (Unit of Work pattern) and connection string a meaningful name. In our example we will call it **MainModuleUnitOfWork** (Context/UoW for our main Module):

www.EBooksWorld.ir

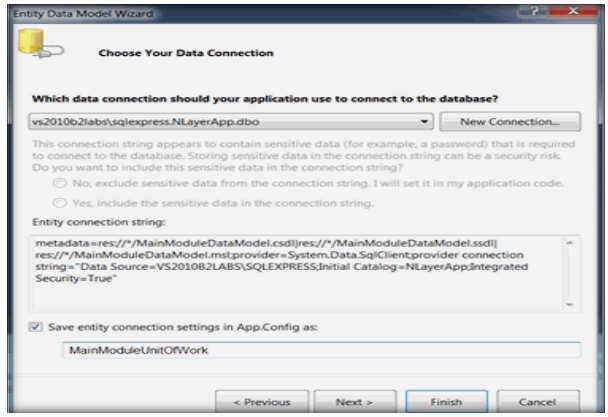



**Figure 5.- Wizard for Connection string and Context**

When adding tables (or creating a new model from scratch) we should select a name for the *namespace*. This is related to our vertical/functional module, for example **NLayerApp.MainModule.**

**At this point it is also very important to check that we included the '*foreign key*' columns. Also, if our tables are named in English and in singular, it is useful to specify that we want the object names to be in plural or in singular.**

We show this step below:

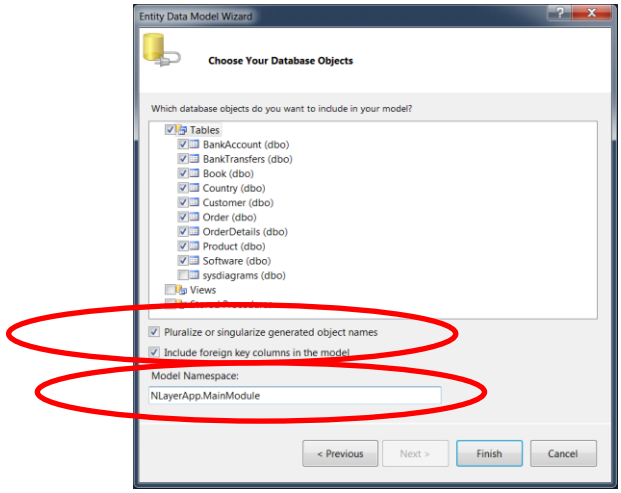



**Figure 6.- NameSpace of the EF Model: NLayerApp.MainModule**

This gives us the following model (matching with the data model of our sample Architecture application):

www.EBooksWorld.ir

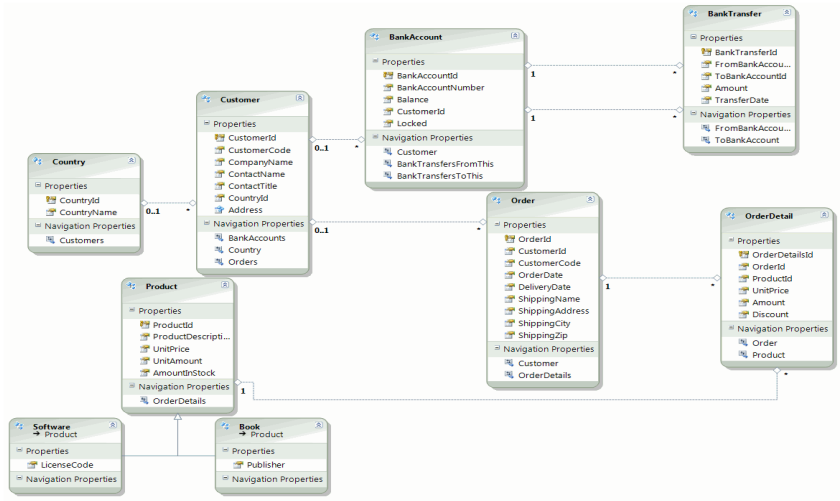



**Figure 7.- Domain Entities Model**

The following will be displayed in the '*Model Browser*' view:

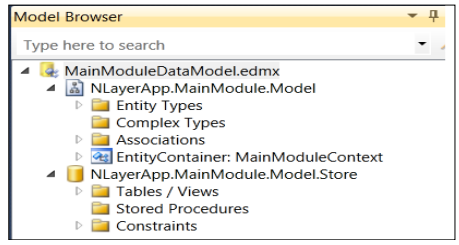



**Figure 8.- 'Model Browser' view**

## 5.5.- T4 Templates of POCO/Self-Tracking Entities generation

In *Visual Studio 2010* there are T4 templates for code generation. In EF there are currently two T4 templates that generate POCO or *Self-Tracking* entities from an entity data model.

We should usually have an EDM data model for each application functional module. But this point depends on the design decisions, the number of entities we have per each module, etc.

T4 is a code generating tool included in Visual Studio. The T4 templates can be modified to produce different code patterns based on certain criteria.

**Adding T4 templates**

www.EBooksWorld.ir

From any blank area within the EDM Designer, you can right-click and select "**AddCodeGenerationItem…**". A menu similar to the following will be displayed:

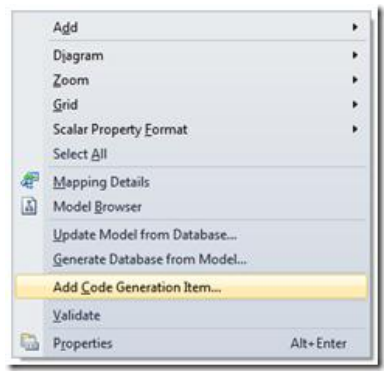



**Figure 9.- 'Add Code Generation Item' wizard**

This shows an "**Add New Item**" option. Select the type "**ADO.NET Self-Tracking Entity Generator**" and specify, for example, "**MainModuleUnitOfWork.tt**" as it is named in our sample at CODEPLEX:

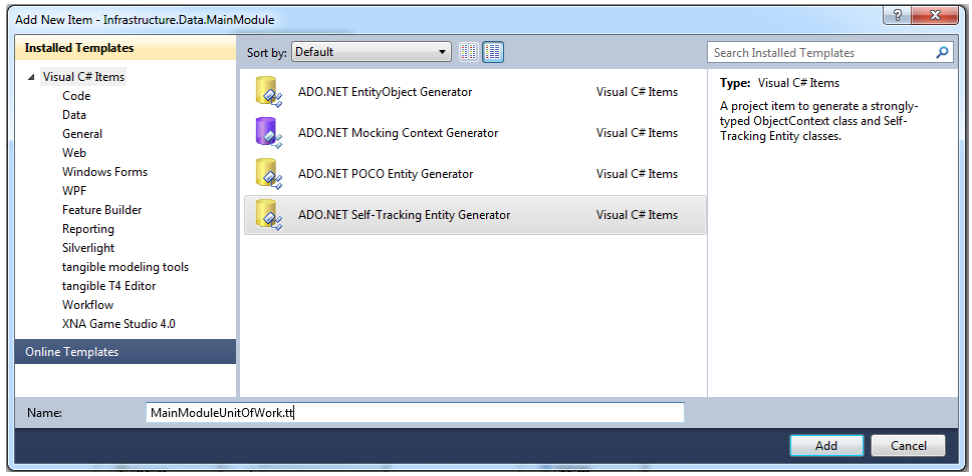



**Figure 10.- Creation of T4 templates for 'Self-Tracking' entities**

Actually, this step has not generated a single T4 file with the name we provided, but two T4 template files. The first template is suitable for generating the Entity classes (in this case, it is named **MainModuleUnitOfWork.tt** and will generate the Self Tracking Entities but in our sample application it is named **MainModuleUnitOfWork.Types.tt**). The second T4 template will generate classes

www.EBooksWorld.ir

related to database connections and EF infrastructure (in this case, **MainModuleUnitOfWork.Context.tt**).

This wizard (STE or POCO template adding) basically disables the default generation of prescriptive EF classes which are tight to EF (they have a direct dependency to EF infrastructure). On the other hand, in the future our T4 templates will be the ones generating these classes, in this case STE.

If we open any of these T4 templates, we will discover that there is a path file pointing to the EF model, such as the following:

```
        stringinputFile = @"MainModuleDataModel.edmx";
```

Whenever any of these T4 templates are saved, all related classes will be generated for us (and all the entity classes, EF Context, etc. will be overwritten).

At this point, we should have something similar to the following:

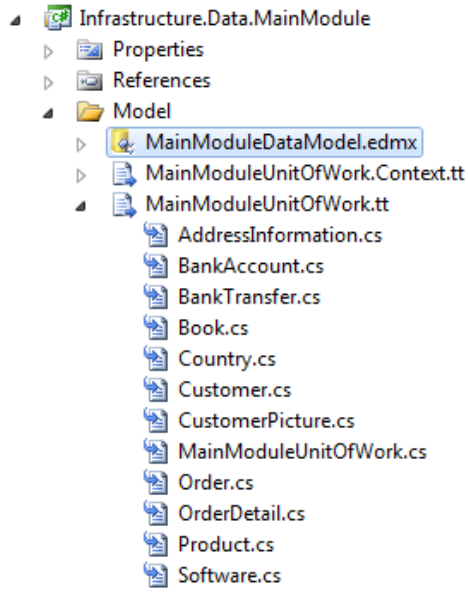



**Figure 11.- TT templates and generated classes**

As shown above, the name of the T4 templates depends on how it was named when we added the STE template ('**MainModuleUnitOfWork.tt**') If any of the team developers modifies the EF model (EDMX), if the template files are in the same project than the .ADMX, changes are propagated automatically on save. But because we will move the template file to a separate project (as Domain Entities must be part of the Domain Layer), when we want to propagate these changes to our classes, we will have to select the '*Run Custom Tool*' option from the pop-up menu by right-clicking on the .tt files, as follows:

www.EBooksWorld.ir

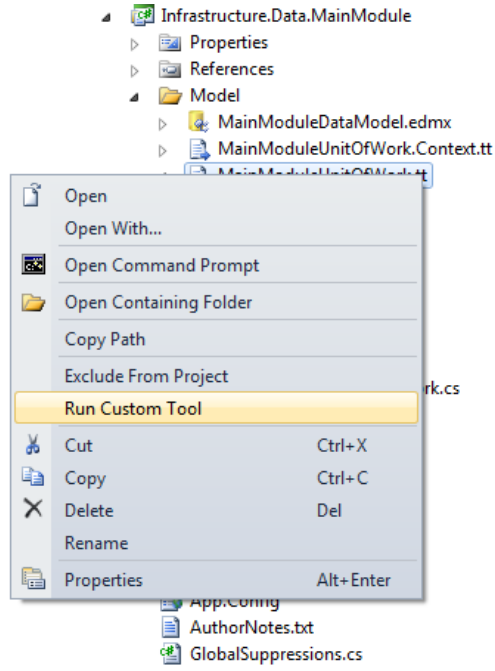



**Figure 12.- 'Run Custom Tool' option**

## 5.6.- EF 'Self-Tracking Entities'

Although the code generated for *Self-Tracking Entities (STE)* and POCO entities are somewhat similar to the internal code used for EF prescriptive entities, in this case we are leveraging the new support to the PI principle (*Persistence Ignorance*). Therefore, the code generated by the POCO and STE templates does not have any type directly related to EF. (Being frank, the PI principle is really applied when using POCO entities. When using STE, we are EF ignorant, but we are not completely persistence ignorant. STE approach is a more lax way regarding the PI principle).

Thanks to this feature (the generated code is our own code), the *self-tracking* and POCO entities can also be used smoothly in Silverlight, (When using STEwe just need to recompile the entities' assembly for Silverlight).

The code generated can be analyzed in any of the generated classes (e.g. in our sample case, "Customer.cs"):

www.EBooksWorld.ir

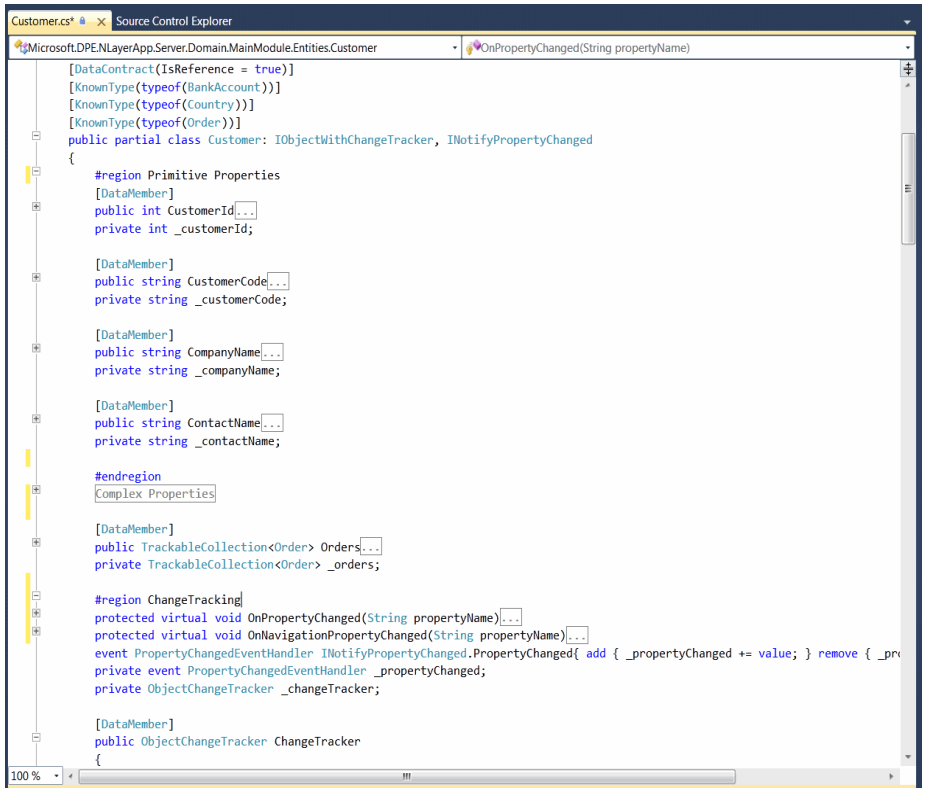



**Figure 13.- Example of Customer.cs Entity Class**

Highlights of a '*Self-Tracking*' entity:

1. Each entity type has a '**DataContract**' attribute with the property *IsReference = true* and all the public properties are marked as **DataMember**. This allows WCF to serialize bidirectional entity graphs.

2. **TrackableCollection is** a type of collection based on **ObservableCollection** that is also included in the generated code and has the ability to notify each individual change made in the collection (at the same time, it derives from the .NET ***Collection* class**). The Self-Tracking entities use this *type* to implement collection navigation properties. The notification is used for the sake of 'change tracking' but also to align several elements representing the same relation when one of them changes. For example, when an "Order" is added to the Customer's order collection, the reference to the Order's owner (a customer) is also updated so it points to the right Customer and the foreign key property (OwnerID) is updated with the owner's ID.

3. The **ChangeTracker** property provides access to the **ObjectChangeTracker** class which holds and controls each entity's 'change tracking' information.

www.EBooksWorld.ir

This will be used internally when we use the Optimistic Concurrence Exception Handling.

In order to obtain self-tracking entities on the client side (like Silverlight or WPF) when using STE, we will have to share the entity types code (in short, the client layer needs to have a reference to the DLL where the entities are implemented). Because of the STE handling in the client side, it is not possible to simply perform an 'AddService Reference' from the Client tier.

**Therefore, self-tracking entities are suitable for N-Tier applications where we control its development from end to end.**

**On the other hand, STE are not suitable for applications where there is no intention of sharing the entity types between the client and the application server.** For example, pure SOA applications, where only one development end is controlled, do not fit with STE. In these latter cases, where there is no intention or possibility of sharing data types between client and server tiers, we recommended using plain DTOs (Data Transfer Objects). This will be discussed further in the chapter about Distributed Services**.**

## 5.7.- Moving Entities to the Domain Layer

Based on the explanations of the previous chapters about Domain's independency regarding technology and infrastructure aspects (DDD concepts), it is important to place entities as elements within the Domain Layer. At the end of the day, they are "Domain Entities". In order to do that, we should move the generated code (T4 and sub-files called **MainModuleUnitOfWork.tt** in our example) to the Domain project where we intend to host the entities. In this case, that project is named:

'**Domain.MainModule.Entities**'

Instead of physically moving files, another option is to create a Visual Studio link to these files. That is, we could continue placing physical files in the *Data Model* project where they were created by Visual Studio, but creating links from the entity project. This will cause the real entity classes to be compiled where we want, in the '**Domain.MainModule.Entities**' domain entity assembly, without having to physically move the files from the physical location where they were placed by Visual Studio and the EF wizard and without having to edit the template file. However, this approach (links usage) causes some problems. Hence, we chose to physically move the entities T4 template to the '**Domain.MainModule.Entities**' assembly (Domain Entities assembly).

The first thing to do is "to clean" the T4 we are about to move. So, first disable the code generation from the T4 template **"MainModuleUnitOfWork.tt" (or MainModuleUnitOfWork.Types.tt)**. Select the file in the '*Solution Explorer*' and see its properties. Delete the value of the '**Custom Tool**' property and leave it blank.

www.EBooksWorld.ir

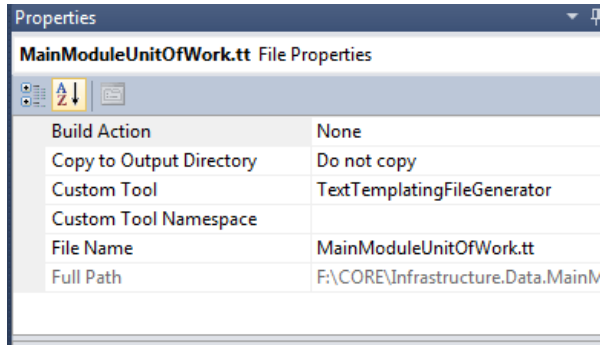



**Figure 14.- Custom Tool Property**

The files displayed in the template (.cs files of the generated classes) should also be deleted, because from this moment on they should not be generated within this project:

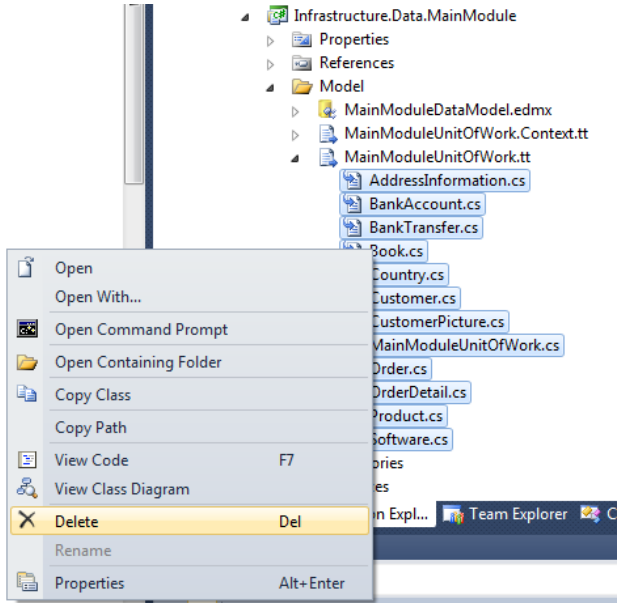



**Figure 15.- Deletion of .cs files of the generated entity classes**

So, simply exclude the file "**"MainModuleUnitOfWork.tt"**" from the current project (persistence layer assembly with the EF EDMX model) and physically copy this T4 file to the folder of a new assembly (within the Domain layer) created to exclusively contain the domain entities. In our case, this is done in the project called "**Domain.MainModule.Entities**." Logically, after copying it, we should add it as part of the Visual Studio project.

www.EBooksWorld.ir

**Important:**
Once the T4 file is copied to the new domain entity project, modify the path to the entity model (.EDMX) in the TT template. As a result the path should look something like this:

```
//(CDLTLL) Changed path to edmx file correct location
   stringinputFile =
@"..\Infrastructure.Data.MainModule\Model\MainModuleDataModel.edmx";
```

Finally, once the entity T4 (TT) file is in its final project and having modified the path so that it directs to the EF .EDMX model, we can test and generate the entity classes, by right-clicking and selecting the '*RunCustomTool*' option:

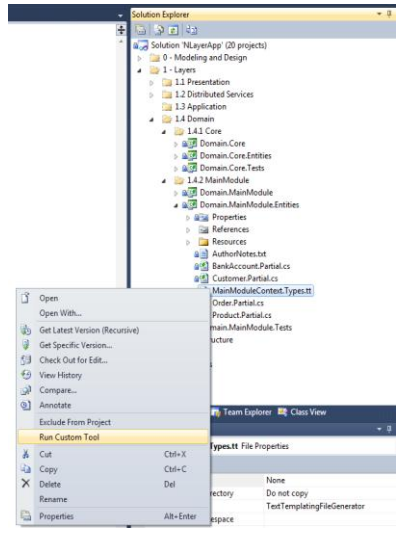



**Figure 16.- Generating Entity Classes with 'RunCustomTool'**

This will generate all the entity classes with the correct *namespace* (Domain *assembly namespace*), etc.:

www.EBooksWorld.ir

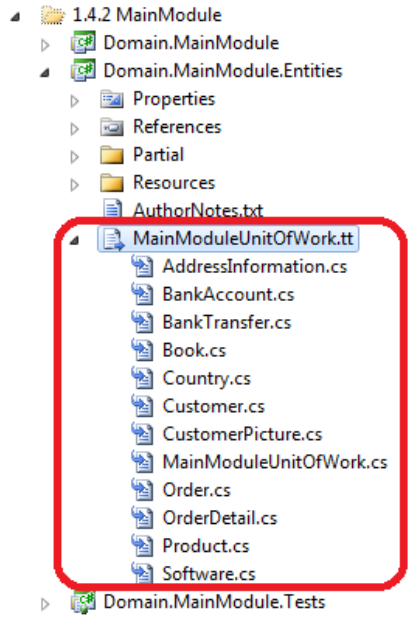



**Figure 17.- Entity Classes in the Domain**

These classes are therefore generated code and should not be directly modified in their class files, because the next time Visual Studio generates the entity code, the code written there will have been lost (overwritten).

However, as we will see in the Domain Model Layer chapter, we should always add Domain logic to the entity classes through partial classes that can be added later.

## 5.7.1.- Separation of T4 STE templates 'Core'

Two templates are generated by generating T4 STE templates of VS2010; one for disconnected entities and another template containing objects with connections against the database (context, etc.). There can be several modules in our application architecture, each of which should have its own entity model (T4 templates). However, there is a common part ('Core') within these templates generated for each module. This should be extracted to a third T4 template so that we do not have redundant code in the different modules. We have called this template **ObjectChangeTracker.Types.tt**' (you could remove the word 'Types' or call it as you want) and it is the code in charge of following up (tracking) the entity changes.

So, since this code is within the *Domain.Core.Entities*, it will be reused from the different modules (e.g., from the *Domain.MainModule.Entities* module and other additional modules, if there are any). There is no need to duplicate this code in each module and data model.

www.EBooksWorld.ir

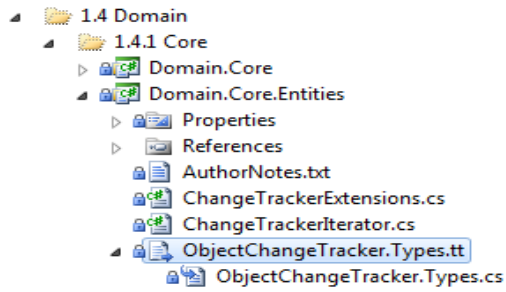



**Figure 18.- 'Core' ObjectChangeTracker.Types.tt template**

This isolated code must also be in a different assembly because we will need to refer to it from the Client agents (WPF, Silverlight, etc.) and be able to use the STE in the presentation layer. The latter case should only be considered if we decide to propagate the domain entities to the presentation layer by using the STE. On the other hand, if we decide to use DTOs for the presentation layer and domain entities just for the domain and application layer, then, logically, there will be no reference to this assembly from the client.

Finally we also added some extensions and Iterators implemented in the '**ChangeTrackerExtension.cs**' and '**ChangeTrackerIterator.cs**'files in our sample application.

## 5.8.- Data Persistence T4 Templates and Data Source Connection

Simultaneously to the T4 template generation for entities mentioned above a T4 template has also been generated to make the data persistence in the database. In our example, this is called '*MainModule**UnitOfWork.Context**.tt*' (or MainModuleModel.Contex.tt depending on how you named it) and placed within the Data Persistence Layer. That template is comprised by a context class with connection to the database, so this is a class completely associated with the *Entity Framework*. Precisely for this reason, it should be in a layer/sub-layer belonging to the '*Data Persistence Infrastructure Layer*'.

In our example, we leave it in the original '**Infrastructure.Data.MainModule**' project, although it can also be moved to a project other than the EDMX model, as we did with the entity T4 template.

The Context class generated by this T4 template will be the one subsequently used to develop our data access and persistence REPOSITORY classes.

www.EBooksWorld.ir

## 5.9.- Implementing Repositories using Entity Framework and LINQ to Entities

As discussed in the chapter about the design of this layer, these components are in some aspects similar to the "Data Access" (DAL) components of N-layer traditional architectures, but different in many others. Basically, they are classes/components that encapsulate the logic required to access the data sources required by the application. Therefore, they centralize common functionality of the data access so that the application has better maintainability and de-coupling between the technologies with respect to the Domain logic. If you use base O/RM technologies as we will do with the ENTITY FRAMEWORK, the code to be implemented is much more simplified and the development can be exclusively focused on data access and not so much on data access plumbing (connections to databases, SQL statements, etc.), which is much clearer in ENTITY FRAMEWORK.

A Repository “records” the data that it is working with in memory (a storage context). It even “records” operations it wants to perform against the storage (usually, database) although these will not be performed until those persistence/access “n” operations are meant to be performed from the Application layer in a single action, all at once. This decision of ‘Applying Changes’ in memory on the real storage with persistence is usually based on the Unit of Work pattern defined and used in the Application layer.

As a general rule, we will implement the Repositories with Entity Framework for N-Layer DDD applications.

**Table 7.- Frame Architecture Guide**

**Rule # I2.** **Implementing Repositories and Base Classes with Entity Framework.**

**Rule**

- It is important to locate the entire persistence and data access logic in well-known points (Repositories). There should be a Repository for each domain root entity (either simple or AGGREGATE ENTITIES). As a general rule and for our sample Architecture we will implement the repositories with Entity Framework.

www.EBooksWorld.ir

**References**

Using Repository and Unit of Work patterns with Entity Framework 4.0
**http://blogs.msdn.com/adonet/archive/2009/06/16/using-repository-and-unit-of-work-patterns-with-entity-framework-4-0.aspx**

## 5.10.- Repository Pattern Implementation

At the implementation level, a repository is simply a class with data access code, which can be the following simple class:

```
C#

Public class CustomerRepository
{
   …
    // Data Access and Persistence Methods
   …
}
```

So far, there is nothing special in this class. It will be a normal class and we will implement methods like "`Customer→GetCustomerById(int customerId)`" by using the 'LINQ to Entities' and POCO or *STE* for domain entities.

In this regard, the persistence and data access methods should be placed in the proper Repositories, usually based on the data or entity type that will be returned by a method, i.e. following this rule:

**Table 8.- Frame Architecture Guide**

**Rule # I3.**

**Placing the methods in Repository classes depending on the entity type returned or updated by these methods.**

○ **Rule**

- If, for example, a specific method defined with the phrase "Retrieve Company Customers" returns a specific entity type (in this case Customer), the method should be placed in the repository class related to this type /entity (in this case, Customer Repository. It would not be Company Repository).

- If these are sub-entities within an AGREGATE, the method should be placed

www.EBooksWorld.ir

in the Repository of the root entity class. For example, if we want to return all the detail lines for an order, we should place this method in the Repository of the aggregate root entity class, which is 'Order Repository'.

- In update methods, the same rule should be followed but depending on the main updated entity.

## 5.10.1.- Base Class for Repositories ('Layer Supertype' Pattern)

Before seeing how to develop each specific method in .NET and EF 4.0, we will implement a base for all the Repository classes. Finally, most Repository classes require a very similar number of methods, like "FindAll", "Modify", "Remove", "Add" etc., but each for a different entity type. Therefore, we can implement a base class for all Repositories (this is an implementation of the *Layer Super type* pattern for this sub-layer of Repositories) and reuse these common methods. However, if it were simply a base class and we derived directly from it, the problem is that we would inherit and use exactly the same base class methods, with a specific data/entity type. In other words, something like this would not make sense:

```
C#

//Base Class or Layered-Supertype of Repositories
Public class Repository
{
     //Base methods for all Repositories
//Add(), FindAll(), Add(), Modify(), etc…
}


Public class CustomerRepository : Repository
{
   …
    // Specific Methods of Data Access and Persistence
   …
}
```

The reason this would not make sense is because the methods we could reuse would be something unrelated to any domain entity type. We cannot use a specific entity class such as Products in Repository base class methods, because after that we may want to inherit the "Customer Repository" class which was not initially related to Products.

www.EBooksWorld.ir

## 5.10.2.- Using 'Generics' for Repositories' Base Class implementation

However, thanks to the *Generics* feature in .NET, we can make use of a base class in which the data types to be used have been established upon using this base class, through *generics*. In other words, the following would be very useful:

```
C#

//Base class or Layered-Supertype of Repositories
public class Repository<TEntity> : where TEntity : class, new()
{
     //Base methods for all Repositories
//Add(), FindAll(), Add(), Modify(), etc…
}


Public class CustomerRepository : Repository
{
   …
    // Specific methods of Data Access and Persistence
   …
}
```

'TEntity' will be replaced by the entity to be used in each case, that is, "Products", "Customers", etc. Thus, we can implement common methods only once and, in each case, they will work against a different specific entity. Below we partially explain the base class "Repository" we used in the N-layer application example:

```
C#

//Base Class or Layered-Supertype of Repositories
public class Repository<TEntity> : IRepository<TEntity>
    where TEntity : class,IObjectWithChangeTracker, new()
{
    private IQueryableContext _context;

    //Constructor with Dependencies
    public Repository(IQueryableContext context)
    {
        //…
        //set internal values
        _context = context;
    }

    public IContextStoreContext
    {
        get
        {
            return _context as IContext;
        }
```

www.EBooksWorld.ir

```
        }

        public void Add(TEntity item)
        {
            //…
            //add object to IObjectSet for this type
            (_context.CreateObjectSet<TEntity>()).AddObject(item);
        }

        public void Remove(TEntity item)
        {
            //…

            //Attach object to context and delete this
            // this is valid only if T is a type in model
            (_context).Attach(item);

            //delete object to IObjectSet for this type
            (_context.CreateObjectSet<TEntity>()).DeleteObject(item);

        }

        public void Attach(TEntity item)
        {
            (_context).Attach(item);
        }

        public void Modify(TEntity item)
        {
            //…

            //Set modifed state if change tracker is enabled
            if (item.ChangeTracker != null)
   item.MarkAsModified();

            //apply changes for item object
            _context.SetChanges(item);
        }

        public void Modify(ICollection<TEntity> items)
        {
            //for each element in collection apply changes
   foreach (TEntity item in items)
            {
                if (item != null)
                    _context.SetChanges(item);
            }
        }

        public IEnumerable<TEntity>GetAll()
        {
            //Create IObjectSet and perform query
            return
( context.CreateObjectSet<TEntity>()).AsEnumerable<TEntity>();
        }


        public IEnumerable<TEntity>GetBySpec(ISpecification<TEntity>
specification)
        {
            if (specification == (ISpecification<TEntity>)null)
```

www.EBooksWorld.ir

```
                throw new ArgumentNullException("specification");

            return (_context.CreateObjectSet<TEntity>()
                            .Where(specification.SatisfiedBy())
                            .AsEnumerable<TEntity>());
        }

        public IEnumerable<TEntity>GetPagedElements<S>(intpageIndex,
    intpageCount, System.Linq.Expressions.Expression<Func<TEntity,
    S>>orderByExpression, bool ascending)
        {
            //checking arguments for this query
            if (pageIndex< 0)
                throw new
    ArgumentException(Resources.Messages.exception_InvalidPageIndex,
    "pageIndex");

       if (pageCount<= 0)
                throw new
    ArgumentException(Resources.Messages.exception_InvalidPageCount,
    "pageCount");

            if (orderByExpression == (Expression<Func<TEntity, S>>)null)
                throw new ArgumentNullException("orderByExpression",
    Resources.Messages.exception_OrderByExpressionCannotBeNull);

            //Create associated IObjectSet and perform query

       IObjectSet<TEntity>objectSet = _context.CreateObjectSet<TEntity>();

            return (ascending)
                                    ?
       objectSet.OrderBy(orderByExpression)
                                        .Skip(pageIndex * pageCount)
                                        .Take(pageCount)
                                        .ToList()
                                    :
       objectSet.OrderByDescending(orderByExpression)
                                        .Skip(pageIndex * pageCount)
       .Take(pageCount)
                                        .ToList();
        }

       }
```

This illustrates how to define certain common methods that will be reused by different *Repositories* of different domain entities. A Repository class can be very easy at the beginning, with no direct implementation; however, it would already inherit the real implementation of such methods from the Repository base class.

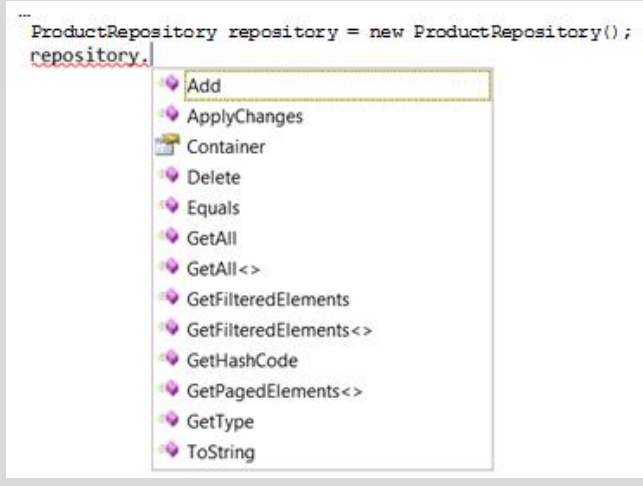
For example, the initial implementation of *ProductRepository*' could be as easy as this:

www.EBooksWorld.ir

```
C#

//Class Repository for Product entity
public class ProductRepository : Repository<Product>,
IProductRepository
{
public ProductRepository(IMainModuleContainer container)
         :base(container)
{
}
}
```

As you can see, we have not implemented any direct method in the 'ProductRepository' class, however, if we instantiate an object of this class, the following would be the methods that we could implement "without doing anything".

```
C#
…
ProductRepository repository = new ProductRepository()
```




We would therefore have basic query, addition, and deletion methods for the specific 'Product' entity without having implemented them specifically for this entity.

In addition, we can add new exclusive methods for the Product entity within the 'ProductRepository' class itself.

**The case of the Repository classes** in our sample Architecture application will appear in the following *namespace*, **within the "Data Persistence Infrastructure" layer and for a vertical/functional module in particular (in this case, the main module called *MainModule*)**:

*Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule.Repositories*

www.EBooksWorld.ir

**Implementing specific methods in Repositories (additional to base class methods)**

An example of specific implementation of a particular Repository method would be the following:

```
   C#
//Class OrderRepository with specific methods
public class OrderRepository
          : Repository<Order>,IOrderRepository
{
          public OrderRepository(IMainModuleContext context) :
base(context) { }


          public IEnumerable<Order>FindOrdersByCustomerCode(string
customerCode)
          {
              //… Parameters Validations, etc. …

   IMainModuleContextactualContext = base.StoreContext as
IMainModuleContext;

              //LINQ TO ENTITIES SENTENCE
              return  (from order
                           in actualContext.Orders
                       where
   order.Customer.CustomerCode == customerCode
   select
   order).AsEnumerable();
          }
}
```

## 5.10.3.- Repository Interfaces and the Importance of Decoupling Layers Components

Although so far we have only introduced the implementation of Repository classes for a correct de-coupled design, the use of Interface-based abstractions will be essential. So, for each Repository we define, we should also implement its interface. As we explained in the theoretical DDD design chapters regarding Repositories, these interfaces will be the only knowledge that the Domain/Application Layers have. Also, the instantiation of Repository classes will be performed by the chosen IoC container (in our case, Unity). This way, the data persistence infrastructure layer will be completely de-coupled from the Domain and Application layers.

Therefore, these abstractions (interfaces) will be defined in our example within the **Domain** layer project, usually in a folder that will group the contracts related to each Aggregate-Root entity.

www.EBooksWorld.ir

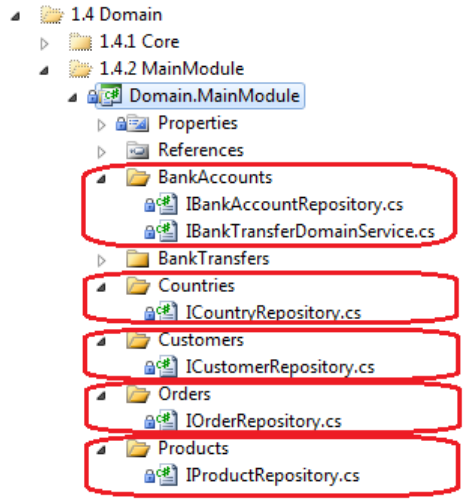



**Figura 19.- Folder group the contracts to each Aggregate-root entity**

This would allow us to fully replace the data persistence infrastructure layer, or repositories through asbtraction/interfaces without impacting on the Domain and Application layers, and without having to change dependencies or performing re-compilation.

Another reason why this de-coupling is so important is because it enables *mocking* of the repositories, so the domain business classes dynamically instantiate "fake" (*stubs* or *mocks*) classes without having to change code or dependencies. They simply specify the IoC container that, when prompted to instantiate an object for a given interface, instantiates a specific class or a fake one (depending on the mapping, but logically, both meeting the same interface).

This Repository de-coupled instantiation system through IoC containers such as Unity is further explained in the Application and Distributed Services Layers' Implementation chapters, because it is there where the instantiations should be performed.

Now, the only important thing to emphasize is that we should have interfaces defined for each Repository class, and that the location of these repository interfaces will be within the Domain layer, for the aforementioned reasons.

www.EBooksWorld.ir

At the interface implementation level, the following would be an example for **ICustomerRepository**:

```
C#

namespace Microsoft.Samples.NLayerApp.Domain.MainModule.Contracts
…
…
//Interface/Contrat ICustomerRepository
public interface ICustomerRepository : IRepository<Customer>
{
Customer GetSingleCustomerByIdWithOrders(intcustomerId);
Customer GetSingleCustomerByCustomerCodeWithOrders(string
customerCode);
}
```

Note that in the case of repository interfaces we are inheriting a "base interface" (IRepository) that gathers common methods from the repositories (Add(), Delete(), GetAll(), etc.).Therefore, in the previous interface we only define other new/exclusive methods of the repository for 'Customer' entity.

The IRepository base interface would be something like this:

```
C#

namespace Microsoft.Samples.NLayerApp.Domain.Core
…
…
public interface IRepository<TEntity>
        where TEntity : class, new()
{
IContainer Container { get; }
void Add(TEntity item);
void Delete(TEntity item);
void Modify(TEntity item);
void Modify (List<TEntity> items);
IEnumerable<TEntity>GetAll();
IEnumerable<K>GetAll<K>() where K : TEntity, new();
IEnumerable<TEntity>GetPagedElements<S>(intpageIndex, intpageCount,
Expression<Func<TEntity, S>>orderByExpression, bool ascending = true);
IEnumerable<TEntity>GetFilteredElements(Expression<Func<TEntity,
bool>> filter);
…
…
}
```

Therefore, all these derived methods are 'added' to our ICustomerRepository.

As discussed above, at this implementation level (Repositories) we simply came to this point. However, we should know how to use these repositories properly, that is, by using abstractions (interfaces) and indirect instantiations through IoC containers. All this is explained in the chapter on Domain Layer Implementation, which is where the Repositories are mostly used.

www.EBooksWorld.ir

## 5.11.- Unit Testing and Repository Integration Implementation

The repository testing implementation could be divided into several items. On one hand this would include the implementation of common elements in our repositories, basically all the methods included in our interface *IRepository<TEntity>*, and on the other hand the testing of the specific methods of each repository.

In the first case, the use of legacy unit testing was preferred in order to make our development productive. This functionality is offered by most testing frameworks and also of course by the Visual Studio Unit Testing Framework.

To perform this task, we created the *RepositoryTestBase* testing base class that implements all the generic methods coming from *Repository* and therefore the common methods for all repositories.

```
C#
[TestClass()]
public abstract class RepositoryTestsBase<TEntity>
whereTEntity : class,IObjectWithChangeTracker, new()
{
            ...
}
```

Some testing examples we can find in this base class of tests are as follows:

```
C#
[TestMethod()]
public virtual void AddTest()
 {
    //Arrange
IQueryableContext context = GetContext();

//Act
Repository<TEntity> repository = new Repository<TEntity>(context);

    TEntity item = new TEntity();
repository.Add(item);
}

[TestMethod()]
  [ExpectedException(typeof(ArgumentNullException))]
public virtual void AddWithNullTest()
{
      //Arrange
IQueryableContext context = GetContext();

      //Act
Repository<TEntity> repository = new Repository<TEntity>(context);
repository.Add(null);

}
```

www.EBooksWorld.ir

In the code above we can see how the 'generics' feature is leveraged within the language and how the dependency of our repository classes with the **IContext** interface is solved through a method called *GetContext*. This method enables running repository tests with a simulated object of the work context. In our case, this object is "**Entity Framework**" which makes tests run faster and keeps them isolated from this dependency which, after all, is external for the repositories.

```
C#
publicIMainModuleContextGetContext(boolinitializeContainer = true)
{
// Get context specified in unity configuration
// Set active context for
    // testing with fake or real context in application configuration
// "defaultIoCContainer" setting

   IMainModuleContext context =
IoCFactory.Resolve<IMainModuleContext>();

        returncontext;
}
```

**Tabla 9.- Unit-Testing for Repositories**

**Rule # I4.** **Implement Unit-Testing for Repositories**

**Recommendations**

- To have a test base class if the repositories use a common type with generic functionality in order to gain productivity when running tests.
- Injecting dependencies with a dependency container in the repository tests allows us to replace real tests against a database and perform them with some fake object.

If we want to run tests on a certain repository, such as **ICustomerRepository** after obtaining our test bases class, all we have to do is to create a test class inherited from **RepositoryTestsBase**.

```
C#

[TestClass()]
public class RepositoryTestsBase<Customer>
{
}
```

These classes also include tests for the specific repository methods that the tests are being made for.

www.EBooksWorld.ir

```
C#
[TestClass()]
public class CustomerRepositoryTests
     : RepositoryTestsBase<Customer>
{
         [TestMethod()]
         [ExpectedException(typeof(ArgumentNullException))]
public void
FindCustomer_Invoke_NullSpecThrowNewArgumentNullException_Test()
         {
             //Arrange
IMainModuleContext context = GetContext();
ICustomerRepository repository = new CustomerRepository(context);

//Act
repository.FindCustomer(null);
         }
     ...
     ...
}
```

The solution we adopted for simulation implementation was to create a simulated object in a new project called **Infraestructure.Data.MainModule.Mock**. The main reason for this decision was that we needed to replace the real repository dependency with EF in other layers of the solution so this component could be reusable.

The mechanism used to perform simulation of the *IContext* interface is based on the capacity of *Microsoft PEX/MOLES* to generate class 'stubs' and interfaces of our code. Once the "moles" assembly is added to the project that will host our simulated object, a stub of the ***IContext*** interface becomes available. This is specifically **IMainModuleContext,** for the case of the main module**.** Although we could use this stub directly, it would need a prior setup process, allocation of the delegates to specify behaviors in each one of its uses, etc. Therefore, in this implementation we decided to create a class that inherited the created stub and completely specified its behaviors. In the main module this class is called **MainModuleFakeContext**, a part of which is shown below:

```
C#
publicclassMainModuleFakeContext
         :
Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule.Context.Moles
.SIMainModuleContext
{
             private void InitiateFakeData()
     {

     //configure country
     this.CountriesGet = () =>CreateCountryObjectSet();
     this.CreateObjectSet<Entities.Country>(()=>
                    CreateCountryObjectSet());
                    ...
             }
             ...
             ...
}
```

www.EBooksWorld.ir

Observation of the simulated data initialization method shows that for each *IObjectSet<TEntity>* property defined within the *IMainModuleContext* interface we should specify the delegate that allows it to obtain its result. In short, these are the elements that can be queried by the repositories, and from which it can obtain data collections, filters, etc. The creation of *IObjectSet* type objects is essential for simulation configuration; therefore, there is an *InMemoryObjectSet* class within the **Infraestructure.Data.Core** project that allows the creation of *IObjectSet* elements from simple object collections.

```
C#

public sealed class InMemoryObjectSet<TEntity> : IObjectSet<TEntity>
whereTEntity : class
    {
            ...

            ...
}
```

```
C#

IObjectSet<Entities.Country>CreateCountryObjectSet()
{
return _Countries.ToInMemoryObjectSet();
        }
```

## 5.12.- Data Source Connections

It is essential to be aware of the existence of connections to data sources (especially databases). The connections to databases are limited resources both in this data persistence layer and in the data source physical level. Please take into account the following guidelines, although many of these items are already considered when using an O/RM:

- Open the connections against the data source as late as possible and close such connections as soon as possible. This will ensure that the limited resources are blocked for the shortest period of time possible and are available sooner e for other consumers/processes. If nonvolatile data are used, the recommendation is to use optimistic concurrency to decrease the chance of blockage on the database. This avoids record blocking overload. In addition, an open connection with the database would also be necessary during this time and it should be blocked from the point of view of other data source consumers.

- Insofar as possible, perform transactions in only one connection. This allows the transaction to be local (much faster) instead of a transaction promoted to

www.EBooksWorld.ir

distributed transaction when using several connections to the database (slower transactions due to the inter-process communication with DTC).

- Use "Connection pooling" to maximize performance and scalability. This requires the credentials and the rest of data of the "connection string" to be the same. Therefore, it is not recommended to use the integrated security with impersonation of different users accessing the database server if you want highest performance and scalability when accessing the database server. To maximize performance and scalability, it is always recommended to use only one identity to access the database server (only several types of credentials if you want to limit the database access by areas). This makes it possible to use the different available connections in the "Connections pool".

- For security reasons, do not use 'System' or DSN (Data Source Name) to save information of connections.

Regarding security and database access, it is important to define how the components will authenticate and access the database and what the authorization requirements will be. The following guidelines may be useful:

- Regarding the SQL Server, as a general rule it is better to use the Windows built-in authentication instead of the SQL Server standard authentication. Usually the best model is the Windows authentication based on the "trusted sub-system" (instead of customization and access with the users of the application, but access to the SQL Server with special/trusted accounts). Windows authentication is safer because, among other advantages, it does not need a password in the connection string.

- If you use SQL Server standard authentication, you should use specific accounts (never 'sa') with complex/strong passwords, limiting the permit of each account through database roles of the SQL Server and ACLs assigned in the files used to save connection strings, and encrypt such connection string in the configuration files being used.

- Use accounts with minimum privilege over the database.

- Require by program that original users propagate their identity information to the Domain/Business layers and even to the Persistence and Data Access layer. This will achieve a system of mass-granularized authorization, as well as the capacity to perform audits at components level.

- Protect confidential data sent through the network to or from the database server. Consider that Windows authentication only protects credentials, but not application data. Use the IPSec or SSL to protect data of the internal network.

- If you are using **SQL Azure** for an application deployed in Microsoft's PaaS Cloud (Windows Azure) there is currently a problem regarding **SQL Azure**

www.EBooksWorld.ir

**connections** you have to deal with. Check the following info to correctly deal with SQL Azure connections:

*Handling SQL Azure Connections issues using Entity Framework 4.0*
**http://blogs.msdn.com/b/cesardelatorre/archive/2010/12/20/handling-sql-azure-connections-issues-using-entity-framework-4-0.aspx**

## 5.12.1.- Data Source Connection 'Pool'

The '*Connection Pooling*' allows applications to reuse a connection already established against the database server, or to create a new connection and add it to the pool if there is no proper connection in the pool. When an application closes a connection, the pool is released, but the internal connection remains open. This means that ADO.NET does not require the complete creation of a new connection and opening it each time for each access, which would be a very expensive process. So, suitable reuse of the connection pooling reduces delays in accessing the database server and therefore increases application performance.

For a connection to be appropriate, it has to meet the following parameters: Server Name, Database Name and access credentials. If the access credentials do not match and there is no similar connection, a new connection will be created. Therefore, when there is Windows security reaching SQL Server and it is also impersonated/propagated from original users, the reuse of connections in the pool is very low. So, as a general rule (except in cases requiring specific security and if performance and scalability are not a priority), it is recommended to follow the "Trusted sub-system" access type, that is, accessing to the database server with only a few types of credentials. Minimizing the number of credentials increases the possibility that a similar connection will be available when there is a request of connection to the pool.

The following image shows a diagram representing the "Trusted Sub-System":

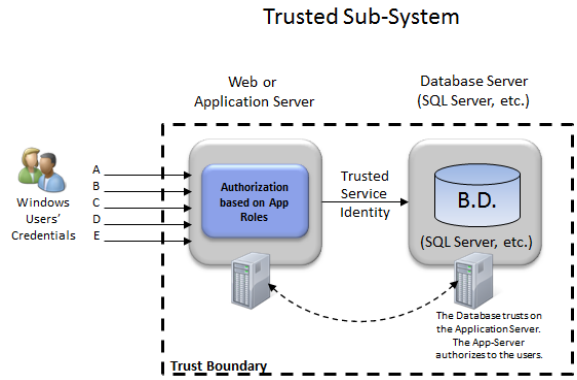



**Figure 20.- "Trusted Sub-System" diagram**

www.EBooksWorld.ir

This sub-system model is quite flexible, because it enables many options for authorization control in the components server (Application Server), as well as auditing accesses in the application server. At the same time, it allows suitable use of the "connection pool" by using default accounts to access the database server and properly reuse the available connections of the connections pool.

On the other hand and finally, certain data access objects have a very high performance (such as DataReaders); however, they may offer very poor scalability if they are not properly used. Due to the fact that a DataReader keeps the connection open during a relatively long period of time (since they require an open connection to access the data) scalability might be impacted. If there are few users, the performance will be very good, but if the number of concurrent users is high, this may cause bottleneck problems because of the number of open connections being used against the database at the same time.

## 5.13.- Strategies for Data Source Error Handling

It is convenient to have a homogeneous system and an exception management strategy. This topic is usually a Cross-Cutting aspect of the application, so having reusable components to manage exceptions in all layers homogeneously should be considered. These reusable components can be simple components/classes, but if the requirements are more complex (publication of exceptions in different destinations, such as Event Log and traps SNMP, etc.), we recommend using the ***Microsoft Enterprise Library Exceptions Management Building Block*** '(v5.0 for .NET 4.0).

However, having a library or reusable classes does not cover everything needed to implement exception management in the different layers. A specific strategy must be implemented in each layer. For example, the following decisions should be made:

- Determine the type of exceptions to be propagated to upper levels (usually most of them) and which ones will be intercepted and managed in one layer only. In the case of the Data Access and Persistence Infrastructure layer, we would usually have to specifically manage aspects such as interblockage, problems of connection to the database, some aspects of optimistic concurrency exceptions, etc.

- How to handle exceptions that we do not specifically manage.

- Consider the implementation of retry processes for operations where there may be '*timeouts'*. However, do this only if it is actually feasible. This should be analyzed on a case-by-case basis.

- Design a proper exception propagation strategy. For example, allow exceptions to be uploaded to the upper layers where they will be logged and/or transformed if necessary before transferring them to the next level.

www.EBooksWorld.ir

- Design and implement a logging system and error notification system for critical errors and exceptions that do not show confidential information.

## 5.14.- External Service Agents (Optional)

The "Service Agents" are objects that manage the specific semantics of communication with external services (usually, Web Services).They isolate our application from idiosyncrasies of calling different services and providing additional services, such as basic mapping, between the format exposed by the data types expected by the external services and the format of the data we used in our application.

In addition, the cache systems may be implemented here, as well as offline scenarios support, or those with intermittent connections, etc.

In large applications it is usual for the service agents to act as an abstraction level between our Domain layer (business logic) and remote services. This enables a homogeneous and consistent interface regardless of the final data formats.

In smaller applications, the presentation layer can usually access the Service Agents directly, not going through the Domain layer and Application layer components.

These external service agents are perfect components to be de-coupled with IoC and therefore, to simulate such Web services with fakes for the development time and to perform unit testing of those agents.

## 5.15.- References of Data Access Technologies

*"T4 and code generation"*
***http://msdn.microsoft.com/en-us/library/bb126445(VS.100).aspx***

*N-Tier Applications With Entity Framework*
***http://msdn.microsoft.com/en-us/library/bb896304(VS.100).aspx***

*".NET Data Access Architecture Guide"*
***http://msdn.microsoft.com/en-us/library/ms978510.aspx***

*"Data Patterns"*
***http://msdn.microsoft.com/en-us/library/ms998446.aspx***

*"Designing Data Tier Components and Passing Data Through Tiers"*
***http://msdn.microsoft.com/en-us/library/ms978496.aspx***

www.EBooksWorld.ir

CHAPTER 5

# The Domain Model Layer

## 1.- THE DOMAIN

This section describes the domain logic layer architecture (business rules) and contains key guidelines to be considered when designing these layers.

This layer should be responsible for representing business concepts, information on the status of the business processes and implementation of domain rules. It should also contain states reflecting the status of business processes, even when the technical storage details are delegated to the lower layers of the infrastructure (Repositories, etc.)

**The 'Domain Model' Layer is the Heart of the Software.**

The components of this layer implement the system's core functionality and encapsulate all the relevant business logic (generally called Domain logic according to DDD terminology). Basically, this layer usually contains classes which implement the domain logic within their methods, although it can also be implemented using a dynamic business rule system, etc.

Following the architecture patterns in DDD, this layer must completely ignore the data persistence details. These persistence tasks should be performed by the infrastructure layer.

The main reason for implementing the domain logic layer (business) is to differentiate and clearly separate the behavior of the domain rules (business rules that are the responsibility of the domain model) and the infrastructure implementation details (like data access and specific repositories linked to a particular technology such as O/RMs, or simply data access libraries or even cross-cutting aspects of the architecture**).** Thus, (by isolating the application Domain) we will drastically increase



www.EBooksWorld.ir

the maintainability of our system and we could even replace the lower layers (data access, O/RMs, and databases) with low impact to the rest of the application.

In each chapter of this guide, the intention is to show the approach on two separate levels. A first logical level (Logical architecture, as in this chapter) that could be implemented with any technology and language (any .NET version or even other non-Microsoft platforms) and subsequently a second level of technology implementation, where we will show how to develop this layer, particularly with .NET 4.0 technologies.

## 2.- DOMAIN LAYER: LOGICAL DESIGN AND ARCHITECTURE

This chapter is organized into sections that include the domain logic layer design as well as the implementation of the functionalities that are typical of this layer, such as **decoupling** from the data access infrastructure layer using **IoC** and **DI.** It also shows typical concerns within this layer regarding security, cache, exceptions handling, *logging* and validation concepts.

In this diagram we show how this Domain model layer typically fits into our '*N-Layer Domain Oriented'* architecture.

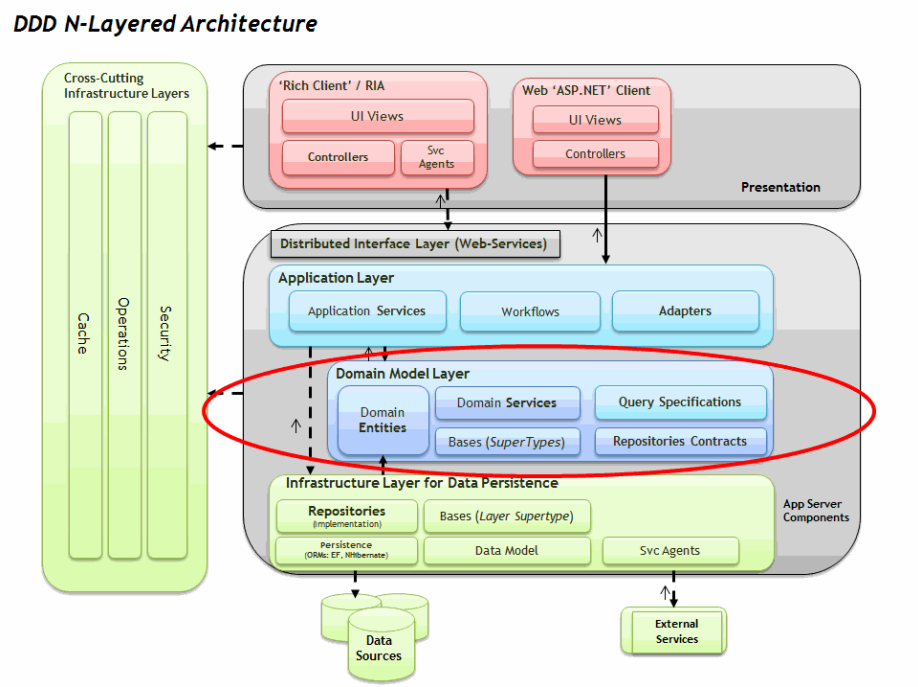



**Figure 1.- Position of the Domain Layer within the *DDD N-Layered Architecture***

www.EBooksWorld.ir

## 2.1.- Sample Application: Business Requirements of a Sample Domain Model to be Designed

Before proceeding with the details of each layer and how to design each one internally, we want to introduce "*StoryScript"* as a '*Sample Domain Model*' which will be designed following the domain oriented design patterns, in layers, and will be implemented at a later time (See our sample application at CODEPLEX.).

**Note:**
We have defined very simple business requirements below. Their functionality is intentionally very simple, especially in the areas related to banking. This is because the main purpose of the sample application is to highlight aspects of the architecture and design and not to design and implement a real and functionally complete application.

The initial details of domain requirements/problems, at a functional level have been obtained through discussions with domain experts (end users with–expertise in a particular functional area) and are as follows:

1.- A **Customer** and **Order** management application is required. There should also be a Bank module related to the company's Bank in order to make **transfers** and other bank operations for the customers.

2.- 'Customer lists' with flexible filters are required. The operators managing the customers need to be able to perform customer searches in a flexible manner; being able to search by a part/initial of a name which could be extended in the future to allow searches by other different attributes (Country, Province, etc.). It would also be very useful to have queries for customers whose orders are in a certain state (e.g. "unpaid."). The result of these searches is simply a customer list showing their primary data (ID, name, location, etc.).

3.- An 'Order list' owned by a specific customer is required. The total value of each order should be seen on the list, as well as the date of the order and the name of the parties involved.

4.- An order can have an unlimited number of detail lines (order items).

Each order line describes an order item which consists of a product and the required quantity of that product.

www.EBooksWorld.ir

5.- It is important to detect concurrency conflicts.

    IT Department told us that it is acceptable and even advisable to use "Optimistic Concurrency Control". For example, consider a scenario where a user tries to update data on a data set he retrieved earlier, but another user has modified the original data in the database after the first user retrieved it and just before he saves his changes back to the database. In this case, when the first user tries to update the data, this conflict is detected (original data has been modified and now there's a possibility of losing those updated data if we save new data over it). Only conflicts causing actual inconsistencies will be considered.

6.- An order cannot have a total value less than 6 $ nor more than 1 million $.

7.- Each order and each customer should have a user friendly number/code. This code should be legible, writable and easily remembered, as well as having the ability to search for a customer/order using the codes. If necessary, the application could manage more complex IDs but should be transparent to the end user.

8.- An order always belongs to a customer; **an order** line always belongs to an order. There cannot be orders without **a** specific customer nor there can be order lines without an order.

9.- Bank operations may be independent from the customers and orders module. They should provide a basic view, such as a list of existing accounts with their relevant data (e.g.. balance, account number, etc.) as well as the ability to perform simplified bank transfers between these accounts (source account and destination account).

10.- The effective implementation of a bank transfer (in this case, persisting timely changes on the account balance stored in the database) should be accomplished in an atomic operation ('all or nothing'). Therefore, it should be an atomic transaction.

11.- The accounts will have a blocked/unblocked state at the business level. The application manager should be able to unblock/block any chosen account.

12.- If an account is blocked, no operations can be performed against it (no transfers or any other type of operations). If any operation is attempted against a blocked account, the application should detect it and show a business exception to the application user, informing him of the reason why he cannot perform such an operation (for instance, because a specific account is blocked at the business level).

13.- (SIMPLIFICATION OF THE SAMPLE) The intention is to have the simplest sample possible at the functional and data design levels, in order

www.EBooksWorld.ir

to clearly show and understand the architecture, so the priority should be simplicity in logical entities and database designs. For example, a customer, organization and address all merged into the same logical entity and even into the same database table is not the best design at all. However, in this case (sample application) the goal is to end up with a design that maximizes the simplification of the application functionality. In fact, this sample application intends to show the best practices in Architecture, not in logical and database design for application-specific functionality. So, in the unreal world of this application, these characteristics must be considered when simplifying the design:

- A Customer/Company will only have one contact person (Although this is not true in the real world).
- A Customer/Company will only have one address (Although this is usually not the case in the real world because they may have several addresses for different purposes, etc.)

Based on these specifications, we will identify the specific elements of the sample application such as Entities, Repositories, Services, etc., as we go over the different elements of the Architecture.

## 2.2.- Domain Layer Elements

Below, we briefly explain the responsibilities of each type of element proposed for the Domain Model.

### 2.2.1.- Domain Entities

This concept represents the implementation of the *ENTITY* pattern.

**ENTITIES represent domain objects and are primarily defined by their identity and continuity over time and not only by the attributes that comprise them.**

Entities normally have a direct relationship to the main business/domain objects, such as customers, employees, orders, etc. Therefore, it is quite common to persist such entities in databases, although this depends entirely on each specific application. It is not mandatory but the aspect of "continuity" is usually strongly related to the storage in databases. Continuity means that the entity should be able to “survive” the execution cycles of the application. Each time the application is restarted, it should be possible to reconstruct these entities in memory.

www.EBooksWorld.ir

In order to distinguish one entity from another, the concept of identity that uniquely identifies entities is essential, especially when two entities have the same values/data in their attributes. Identity in data is a fundamental aspect in applications. A case of wrong identity in one application can lead to data corruption problems or program mistakes. Many items in the real domain (the business reality) or in the application domain model (business abstraction) are defined by their identity and not by their attributes. A good example of an entity is a person. The entity's attributes, such as address, financial data or even its name may change throughout its lifetime; however, the entity's identity will remain the same, the same person, in this case. Therefore, the essential concept of an ENTITY is a continuous abstract life that can evolve to different states and shapes, but that will always have the same identity.

**Some objects are not defined primarily by their attributes; they represent a thread of identity with a specific life and often with different representations. An entity should be able to be distinguished from other different entities even when they have the same descriptive attributes (e.g., there can be two people with the same first and last names).**

With regard to DDD, and according to Eric Evans' definition, "*An object primarily defined by its identity is called ENTITY.*" Entities are very important in the Domain model and they should be carefully identified and designed. What may be an ENTITY in some applications might not be one in other applications. For example, an "address" in some systems may not have an identity at all, since it may only represent the attributes of a person or company. However, in other systems such as an application for a Power Utility company, the customer's addresses could be very important and therefore the address must have an identity because the billing system can be directly linked to the addresses. In this case, an address should be classified as a Domain ENTITY. In other cases, such as in an e-commerce application, the address may simply be an attribute of the person's profile. In this last case, the address is not so important and should be classified as a VALUE-OBJECT, (as it is called in DDD and we will explain later on.)

An ENTITY can be of many types, it can be a person, car, bank transaction, etc., but the important point is that whether it is an entity or not depends on the specific domain model that we are dealing with. A particular object does not need to be an ENTITY in all application domain models. Also, not all the objects in the domain model are an Entity.

For example, in a bank transaction scenario, two identical incoming amounts on the same day are considered to be different bank transactions, so they have an identity and usually are ENTITIES. Even if the attributes of both entities (in this case, amount and time) were exactly the same, they would still be known as different ENTITIES.

### Entity Implementation Design

Regarding **design and implementation**, these entities are disconnected objects (with their own data and logic) and are used to obtain and transfer entity data between different layers. These objects represent real world business entities, such as products or orders. On the other hand, the entities the application uses internally are data

www.EBooksWorld.ir

structures in memory, such as the classes (entity data **plus entity logic**). Furthermore, if these entities depend on a particular data persistence technology (e.g., prescriptive *Entity Framework entities*), then these classes should be located inside the data persistence infrastructure layer because they are related to a specific technology. **On the other hand, if we follow the patterns recommended by DDD and use POCOs (*Plain Old CLR Objects*), they are plain classes (our own code) which are not tight to any particular technology. Therefore, these ENTITIES should be located as elements within the Domain Layer, since they are Domain entities and independent from any infrastructure technology (ORMs, etc.)**.

**Table 1.- Principle of Persistence Technology Ignorance**

| **PI Principle (Persistence Ignorance), POCO and STE** |
|---|
| This concept, which recommends **POCO (Plain Old CLR Objects) for** domain entity implementation, is probably the most important point to consider when implementing entities according to a Domain Oriented Architecture. It is fully supported by the principle that all components of the Domain layer must completely ignore technologies that the Data Persistence Infrastructure Layer is based on, such as O/RMs.<br>The way these entity objects are implemented is especially important for many designs. In many designs (such as in DDD), it is vital to isolate these elements from any knowledge on the data access base technologies, so that they really know nothing about the underlying technology to be used for their persistence. In other words, entities that do not inherit from any base class and do not implement any interface related to the underlying technologies are called **POCO** in **.NET**, or **POJO *(Plain Old Java Object*)** in the **Java** world.<br>On the contrary, objects that do inherit/implement a certain base class or interface related to the underlying technologies are known as *"Prescriptive Classes"*. The decision to choose one or the other is not trivial and must be carefully considered. On one hand, using POCO give us a great degree of freedom with respect to the persistence model we choose. On the other hand, however,<br>it brings restrictions and/or overloads associated with the "degree of ignorance" the persistence engine will have regarding these entities and their correlation to the relational model (this usually involves a higher degree of development efforts). POCO classes have a higher initial implementation cost, unless the O/RM we are using helps us in POCO class generation from a Data Model (as in the case of Entity Framework 4.0).<br>The concept of **STE** (*Self Tracking Entities*) is a bit laxer. That is to say, the data classes defined by the entities are not entirely "plain" but rather depend on **implementing one or more interfaces** that specify the minimum implementation to be provided. In this case, it does not completely meet the PI (Persistence Ignorance) principle but it is important for this interface to be under our control (our own code). In other words, the interface must not be part of any external infrastructure technology. Otherwise, our entities would stop being "agnostic" regarding the Infrastructure layers and external technologies and would become "Prescriptive Classes". |

www.EBooksWorld.ir

In any case, ENTITIES are objects that float throughout the whole architecture or at least part of it. The latter case is when we use DTOs (*Data Transfer Objects*) for remote communications between *Tiers*, where the domain model's internal entities would not flow to the presentation layer or any other point beyond the internal layers of the Service. DTO objects would be those provided to the presentation layer in a remote location. The analysis of DTOs versus Entities is covered in the Distributed Services chapter, since these concepts are related to the N-Tier applications and distributed development.

Finally, we must consider the serialization of class requirements that can exist when dealing with remote communications. Passing entities from one layer to another (e.g., from the Distributed Services layer to the Presentation Layer) will require such entities to be serialized; they will have to support certain serialization mechanisms, such as XML format or binary format. To this effect, it is important to confirm that the chosen entity type effectively supports serialization. Another option is, as we said, conversion to and/or aggregation of DTOs in the Distributed Services layer.

**Entity Logic contained within the Entity itself**

It is essential that the ENTITY objects themselves possess certain logic related to the entity's data (data in the memory). For example, we can have business logic in a "*BankAccount*" entity, which is executed when money is added or when a payment is made, or even when the amount to be paid is checked (logically, this amount must be greater than zero). Calculated field logic could be another example, and ultimately, any logic related to the internal part of such entity.

Of course, we could also have entities that do not have their own logic, but this case will occur only if we really don't have any related internal entity logic, because if all of our entities had a complete lack of logic, we would be falling into the '*Anemic Domain Model*' anti-pattern introduced by Martin Fowler. See '***AnemicDomainModel***' by Martin Fowler:

**http://www.martinfowler.com/bliki/AnemicDomainModel.html**

The '*Anemic-Domain-Model*' anti-pattern occurs when there are only data entities such as classes that merely have fields and properties and the domain logic belonging to such entities **is mixed with higher level classes (Domain Services or even worse, Application Services).** It is important to note that, under normal circumstances, Domain Services should not contain any internal entities logic, but a coordination logic that considers such entities as a whole unit or even sets of such units.

If the SERVICES (Domain Services) possessed 100% of the ENTITIES logic, this mixture of domain logic belonging to different entities would be dangerous. That would be a sign of '*Transaction Script*' implementation, as opposed to the 'Domain Model' or domain orientation.

In addition, the **logic related to using/invoking Repositories** is the logic that should normally be situated within the Application Layer SERVICES. Unless we really

www.EBooksWorld.ir

need it, it will not be within the Domain Services. An object (ENTITY) does not need to know how to save/build itself, like an engine in real life that provides engine capacity, but does not manufacture itself, or a book does not "know" how to store itself on a bookshelf.

**The logic we should include within the Domain Layer should be the logic that we could be speaking about with a Domain/Business Expert. In this case, we usually do not talk about anything related to Repositories or transactions with a Domain/Business Expert.**

Regarding **using Repositories from Domain Services**, there are, of course, **exceptions**. That is the case when we need to obtain data depending on Domain logic states. In that case we will need to invoke Repositories from the Domain Services. But **that use will usually be just for querying data**. All the transactions, UoW management, etc. should usually be placed within Applications Services.

**Table 2.- Framework Architecture Guide rules**

**Identifying ENTITIES Based on Identity**

**Rule # D8.**

**Rule**

- When an object is distinguished by its identity and not by its attributes, this object should be fundamental in defining the Domain model. It should be an ENTITY. It should maintain a simple class definition and focus on continuity of the life cycle and identity. It should distinguishable in some way, even when it changes attributes or even in form or history. In connection to this ENTITY, there should be an operation ensuring that we arrive at a single result for each object, possibly selecting a unique identifier. The model must define what it means to be the same ENTITY object.

**References**

*'ENTITY pattern' in the book 'Domain Driven Design'* by **Eric Evans**.

*The Entity Design Pattern*

**http://www.codeproject.com/KB/architecture/entitydesignpattern.aspx**

www.EBooksWorld.ir

**Table 3.- Framework Architecture Guide Rules**

**Rule # D9.** **ENTITIES in a Domain Oriented Architecture should be either POCO or STE.**

**Rule**

- In order to meet the **PI (Persistence Ignorance)** principle and not to have direct dependencies to infrastructure technologies, it is important that our entities being implemented as POCO or STE. (Productive way for N-Tier apps but a more lax way regarding PI)

**When to Use STE**

- Some O/RM frameworks allow the use of POCO and STE. However, they allow us to implement advanced aspects with less effort normally using STE. An example of STE (*Self-Tracking-Entities*) which are very useful and productive in N-Tier scenarios and Optimistic Concurrency Management. So, **for N-Tier application scenarios,** it is convenient to use STE because it offers a greater potential and less manual work to be done by us.

**When to Use POCO**

- In purely SOA scenarios, where interoperability is critical, or when we want our presentation layers to be developed/ changed at a different pace from the Domain layer, and those changes in Domain entities affect presentation layers to a lesser degree, it is better to use DTOs that are specifically created for distributed services and used in the presentation layers. If we use DTOs, we obviously cannot use the advanced aspects of the *STEs*. Therefore, it is recommended to use POCO domain entities, which offer full independence from the persistence layer (fulfilling the PI principle). The use of DTOs is purer DDD (thanks to the decoupling between Domain entities and DTOs, which will ultimately be the presentation layer entities). However, it has a much higher cost and development complexity due to data conversions required in both directions, from domain entities to DTOs and vice versa. The use of STEs directly in the presentation layers is a more productive approach, although as a result the presentation layer is tightly coupled to the Domain Layer (regarding entities). This decision (STE versus DTO) is a design/architecture decision that greatly depends on the magnitude of the application. If there are several development teams working on the same application, the DTOs decoupling will probably be more beneficial.

www.EBooksWorld.ir

- The last option is a "mixed bag" which is, using STEs for N-Tier applications (to be used by the presentation layer, etc.) and simultaneously having a SOA layer specially designed for external integration and interoperability. Therefore, this SOA layer will be offered to other applications/external services that use web-service integration with DTOs, which is more simplified.

**References**

'*ENTITY pattern' in the book 'Domain Driven Design' by* **Eric Evans**. *The Entity Design Pattern*

**http://www.codeproject.com/KB/architecture/entitydesignpattern.aspx**

## 2.2.2.- Value-Object Pattern

*"Many objects do not have conceptual identity. These objects describe certain characteristics of a thing."*

As we have seen before, tracking of the entity's identity is crucial; however, there are many objects and data in a system that do not require such an identity and monitoring. In fact, in many cases this should not be done because it can impair the overall system performance in an aspect that, in many cases, is not necessary. Software design is an ongoing struggle with complexity, and if possible, this complexity should always be minimized. Therefore, we must make distinctions so that a special management is applied only when absolutely necessary.

The definition of VALUE-OBJECT is: *Objects that describe things*; to be more accurate, *an object with no conceptual identity that describes a domain aspect.* In short, these are objects that we instantiate to represent design elements which only concern us temporarily. We care about *what* they are, not *who* they are. Basic examples are numbers, strings, etc. but they also exist in higher level concepts. For example, an "Address" in a system could be an ENTITY because in that system an address is important as an identity. But in a different system, the "Address" can be simply a VALUE-OBJECT, a descriptive attribute of a company or person.

A VALUE-OBJECT can also be a set of other values or even references to other entities. For example, in an application that generates a Route to get from one point to another, that route would be a VALUE-OBJECT (it would be a "snapshot" of points to go through this route, but this route does not have an identity or the requirement to be persisted, etc.) even though internally it is referring to different Entities (Cities, Roads, etc.).

www.EBooksWorld.ir

At the implementation level, a VALUE-OBJECT will normally pass through and/or return as parameters in messages between objects, and as previously mentioned, they will have a short life without identity tracking.

Also, an entity is usually composed of different attributes. For example, a person can be modeled as an Entity with an identity and composed internally by a set of attributes such as name, surname, address, etc., which are simply *Values*. From these values, those that are important to us as a set (like address) must be treated as VALUE-OBJECTS.

The following example shows a diagram of specific application classes where we emphasize what could be an ENTITY and what could subsequently be a VALUE-OBJECT within an ENTITY:

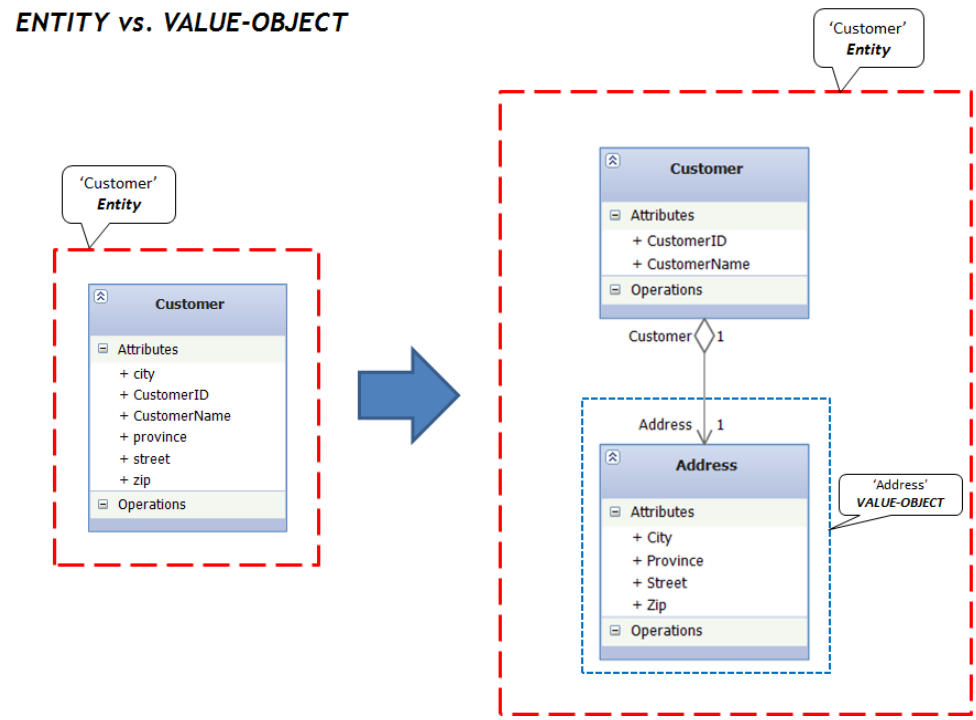



**Figure 2.- Entities vs. Value-Object**

www.EBooksWorld.ir

**Table 4.- Framework Architecture Guide Rules**

**Rule # D10.** **Identifying and Implementing the VALUE-OBJECT Pattern when required**

**Recommendations**

- When certain attributes of a model's element are important to us as a group, but the object must not have a traceable identity, we must classify them as VALUE-OBJECTS. The meaning of these attributes should be expressed and they should have a related functionality. We must also treat the VALUE-OBJECT as immutable information throughout its life, from the moment it is created to the moment it is destroyed.

**References**

**'***VALUE-OBJECT' Pattern. By* **Martin Fowler.**Book 'Patterns of Enterprise Application Architecture':"A small simple object, like money or a date range whose equality isn't based on identity."

*'VALUE-OBJECT' Pattern. Book 'Domain Driven Design' By* **Eric Evans.**

The attributes that comprise a VALUE-OBJECT should form a "conceptual whole". For example, street, city and zip code should not normally be separated into simple attributes within a Customer object (depending on the application domain, of course). In fact, those attributes are part of an address VALUE-OBJECT, which simplifies the Customer object.

**Design of VALUE-OBJECTS**

Due to the lack of restrictions on VALUE-OBJECTS, these can be designed in different ways, always favoring the most simplified design or what best optimizes the system's performance. One of the restrictions of VALUE-OBJECTS is that their values must be immutable from their inception. Therefore, at their creation (construction) we must provide values and not allow them to change during the object's lifetime.

Regarding performance, VALUE-OBJECTS allow us to perform certain "tricks", thanks to their immutable nature. This is especially true in systems where there may be thousands of VALUE-OBJECT instances with many coincidences of the same values. Their immutable nature would allow us to reuse them; they would be "interchangeable" objects, since their values are the same and they have no identity. This type of

www.EBooksWorld.ir

optimization can sometimes make a difference between software that runs slowly and another with good performance. Of course, all these recommendations depend on the application environment and deployment context. Sharing objects can sometimes provide better performance but in certain contexts (a distributed application, for example) it may be less scalable when having copies, because accessing a central point of shared reusable objects can cause a bottleneck in communications.

## 2.2.3.- AGGREGATE Pattern

An aggregate is a domain pattern used to define ownership and boundaries of the domain model objects.

A model can have any number of objects (entities and value-objects) and most of them will normally be related to many others. We will have, therefore, different types of associations. Most associations between objects must be reflected in the code and even in the database. For example, a one to one association between an employee and a company will be reflected as a reference between two objects and will probably imply a relationship between two database tables. If we talk about one to many relationships, the context is much more complicated. But there may be many relationships that are not essential to the particular Domain in which we are working. In short, it is hard to ensure consistency in changes of a model that has many complex associations.

Thus, **one of the goals we have to consider is to simplify the relationships in a domain entity model as much as possible.** This is where the **AGGREGATE pattern** appears. An aggregate is a group/set of associated objects that are considered as a whole unit with regard to data changes. The aggregate is delimited by a boundary that separates the internal objects from the external objects. Each aggregate has a root object (called root entity) and initially it will be the only accessible object from the outside. The root entity object has references to all of the objects that comprise the aggregate, but an external object can only have references to the root entity-object. If there are other entities (these could also be value-objects) within the aggregate's boundary, the identity of these entity-objects is only local and they only make sense if they belong to the aggregate. They do not make sense if they are isolated.

This single point of access to the aggregate (root entity) is precisely what ensures data integrity. From outside the aggregate, there is no access and no change of data to the aggregate's secondary objects, only through the root, which implies a very important control level. If the root entity is erased, the rest of the aggregate's objects must also be erased.

If the aggregate's objects need to be persisted in the database, then they should only be accessed through the root entity. The secondary objects must be obtained through associations. **This implies that only the root entities of aggregates may have associated REPOSITORIES. The same happens at a higher level with the SERVICES. We can have SERVICES directly related to the AGGREGATE root entity, but never directly related to a secondary object of an aggregate.**

www.EBooksWorld.ir

The internal objects of an aggregate, however, should be allowed to have references to root entities of other aggregates (or simple entities that do not belong to any complex aggregate).

In the following diagram we show an example of an aggregate:

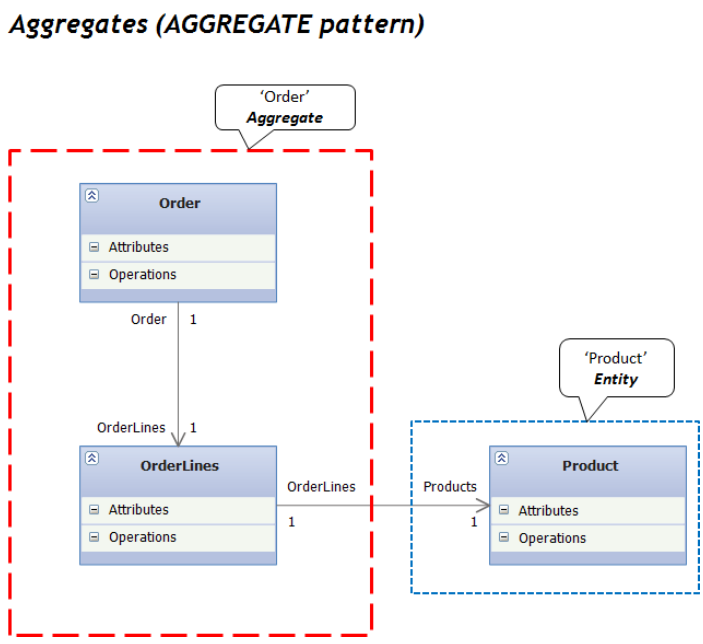



**Figure 3.- Aggregates (AGGREGATE Pattern)**

**Table 5.- Aggregates identification rule**

| | |
|---|---|
| **Rule # D11.** | **Identify and Implement the AGGREGATE Pattern in the necessary cases to simplify the relationships between the model objects as much as possible** |

**Recommendations**

- **One of the objectives we have to consider is the ability to simplify the relationships in the domain entity model as much as possible. This is where the AGGREGATE pattern appears.** An aggregate is a group/set of associated objects that are considered as a whole unit with regard to data changes.

www.EBooksWorld.ir

- Keep in mind that this implies that **only the aggregate root entities (or even simple entities) can have associated REPOSITORIES. The same happens on a higher level with SERVICES. We can have SERVICES directly related to the AGGREGATE root entity but never directly related to only a secondary object of an aggregate.**

**References**

*'AGGREGATE' pattern. Book 'Domain Driven Design' By* **Eric Evans.**

## 2.2.4.- Repository Contracts/Interfaces situated within the Domain Layer

The implementation of Repositories (Repository classes) is not part of the Domain but part of the infrastructure layers (since Repositories are linked to data persistence technology, such as an O/RM like *Entity Framework*); however, the contract referring to what such Repositories should look like (Interfaces to be implemented by these Repositories), should be a part of the Domain. That is why we include it here. The contract specifies what the Repository should offer, regardless of its internal implementation. These interfaces are agnostic to technology. Therefore, the Repository interfaces should be defined within the Domain layers. This point is highly recommended in DDD architectures and is based on the '*Separated Interface Pattern*' defined by Martin Fowler.

Logically, in order to comply with this pattern, Domain Entities and Value-Objects need to be **POCO/STE.** In other words, they should be completely agnostic to the data access technologies. We must keep in mind that, in the end, domain entities are the actual "object types" of the parameters sent to and returned by the Repositories.

In conclusion, the goal of this design (*Persistence Ignorance*) is for the domain classes "to know nothing at all" about the repositories. When we work in domain layers, we must ignore how the repositories are being implemented.

www.EBooksWorld.ir

**Table 6.- Framework Architecture Guide rules**

**Rule # D12.**

**Define Repository interfaces within the Domain Layer following the SEPARATED INTERFACE PATTERN**

**Recommendations**

- From the point of view of decoupling between the Domain Layer and the Data Access Infrastructure Layer, we recommend defining the Repository interfaces within the domain layer and the implementation of such interfaces within the Data persistence infrastructure layer. Thus, a Domain Model class may use a Repository interface as needed, without having to know the current Repository implementation, which has been implemented in the Infrastructure layer.
- This rule fits perfectly with decoupling techniques based on IoC containers.

**References**

'*Separated Interface'* pattern, by **Martin Fowler**.
*"Use Separated Interface to define an interface in one package but implement it in another. This way a client that needs the dependency to the interface can be completely unaware of the implementation."*

**http://www.martinfowler.com/eaaCatalog/separatedInterface.html**

## 2.2.5.-Domain Model SERVICES

In most cases, our designs include operations that do not conceptually belong to Domain ENTITY objects. In these cases we can include/group such operations in explicit Domain Model SERVICES.

www.EBooksWorld.ir

**Note:**

It is important to point out that the SERVICE concept in **N-layer DDD** is not a DISTRIBUTED SERVICE (typically Web Services) for remote access. A Web Service may "wrap" and publish it for remote accesses to the implementation of the Domain Service, but it is also possible for a Web application to have domain services and no Web Services.

Those operations that do not specifically belong to Domain ENTITIES are inherently activities or operations, not internal characteristics of Domain Entities. But since our programming model is object oriented, we should also group them in objects. These objects are what we call SERVICES.

Forcing those Domain operations (in many cases, these are high level operations and group other actions) to be a part of the ENTITY objects would distort the definition of domain model and would make the ENTITIES appear artificial.

A SERVICE is an operation or set of operations offered simply as an interface that is available in the model.

The word "Service" in SERVICE pattern precisely emphasizes what it offers: "What it can do and the actions it offers to the client that uses it, and highlights the relationship with other Domain objects (Covering several Entities in many cases)."

High level SERVICES (related to several entities) are usually called by activity names. In these cases, they are related to verbs associated with Use Case analysis, not to nouns, even when they may have an abstract definition of a Domain's business operation (for example, a Service-Transfer related to the action/verb "Transfer Money from one bank account to another.")

The SERVICE operation names should come from the UBIQUITOUS LANGUAGE of the Domain. The parameters and return values should be Domain objects (ENTITIES or VALUE OBJECTS).

**The SERVICE classes are also domain components, but in this case they are the highest level of objects within the Domain Layer. In most of the cases Domain Services cover different concepts and coordinate several related ENTITIES within business scenarios and use cases.**

When a Domain operation is recognized as an important Domain concept, it should normally be included in a Domain SERVICE.

A Service should be stateless. This does not mean that the class implementing it should be static; it may well be an instantiable class (in fact, it has to be non-static if we want to use decoupling techniques between layers, such as IoC containers). A SERVICE being stateless means that a client program can use any instance of a service regardless of its internal state as an object.

Additionally, the execution of a SERVICE may use globally accessible information and may even change such global information (that is, it can have side effects). But the service should not have states that can affect its own behavior, like most domain objects.

www.EBooksWorld.ir

As for the type of rules to be included in the Domain SERVICES, a clear example would be in a banking application, making a transfer from one account to another, because it needs to coordinate business rules for an "Account" type with "Payment" and "Charge" type operations. In addition, the action/verb "Transfer" is a typical operation of a banking Domain. In this case, the SERVICE itself does not do a lot of work, it just coordinates the Charge() and Pay() method calls which are part of an Entity class, such as "BankAccount". On the other hand, placing the Transfer() method in "Account" class would be a mistake in the first place (of course, this depends on the particular Domain) because the operation involves two "Accounts" and possibly other business rules to be considered.

Exceptions handling and business exceptions throwing should be implemented in both the Domain Services and the internal logic of entity classes.

**From a perspective outside the domain, the SERVICES will usually be those that should be visible in order to perform relevant tasks/operations of each layer.** In our example above (Bank Transfer), the SERVICE is precisely the backbone of the bank domain business rules.

**Table 7.- Framework Architecture Guide**

**Rule # D13.** **Design and Implement Domain SERVICES to Coordinate the Business Logic**

○ **Recommendations**

It is important to have these components in order to coordinate domain entities logic, and not to mix the domain logic (Business rules) with the application and data access logic (data persistence is coupled to a technology).

A good SERVICE usually has this characteristic:

The operation is associated with a Domain concept that is not a natural part of an ENTITY's internal logic.

**References**

SERVICE Pattern - Book '*Domain Driven Design*' - **Eric Evans**.

*SERVICE LAYER Pattern* – By **Martin Fowler**. Book 'Patterns of Enterprise Application Architecture: "*Layer of services that establishes a set of available operations and coordinates the application response in each main operation*."

www.EBooksWorld.ir

Another rule to consider when dealing with the definition of data entities, and even classes and methods, is to define what we are really going to use. We should not define entities and methods because they seem logical, since in the end many of them will probably not be used in the application. In short, we should follow a useful recommendation from Agile-Methdologies called 'YAGNI' (*You Ain't Gonna Need It*), already mentioned at the beginning of this guide.

We should also define Domain Services only when we have to, like when there is really a need for entity domain logic coordination.

As shown in the following figure, we can have a domain service (in this case, the *BankTransferService* class) coordinating actions of the BankAccount's business logic:

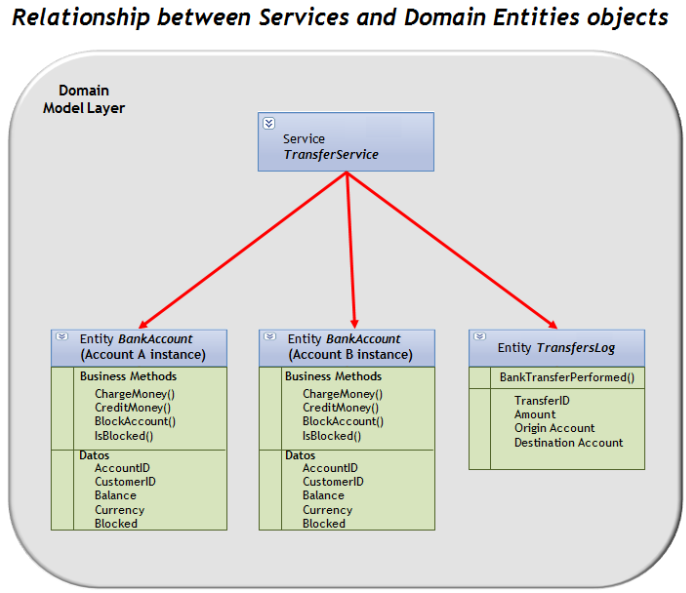



**Figure 4.- Possible relationship between Entity and Service objects**

A simplified UML sequence diagram (without considering transfer records) would have the following interactions. Basically, what we are pointing out is that calls between methods in this layer would be exclusively to execute the Domain logic whose flow or interaction could be discussed with a Domain expert or the end user:

www.EBooksWorld.ir

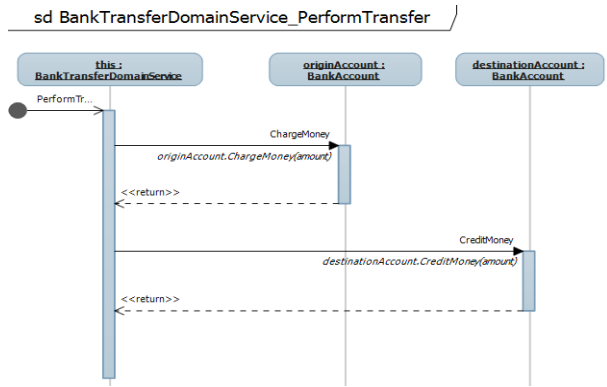



**Figure 5.- Sequence Diagram for a simplified Bank transfer**

Three objects appear in the sequence diagram. The first object (BankTransferDomainService) is a Domain Service that acts as the origin of the sequence and the other two (originAccount and destinationAccount, both instances of the BankAccount class) are "Domain Entity" objects, which would also have domain methods/logic (e.g. ChargeMoney and CreditMoney methods) that modify the "in memory data" of each domain entity object.

**Table 8.- Domain Services should govern/coordinate the Business Logic**

| **Rule # D14.** | **The Domain SERVICE classes must also govern/coordinate the Domain's main processes** |
|---|---|

**Rule**

- As a general rule, all complex business operations (**requiring more than a single operation unit)** related to different Domain Entities should be implemented within the 'Domain SERVICE' classes.
- Ultimately, this is about implementing the coordination of the whole use cases business logic.

**References**

*SERVICE Pattern - 'Domain Driven Design'* - **Eric Evans**.
*SERVICE LAYER Pattern* – By **Martin Fowler**. Book '*Patterns of Enterprise Application Architecture*': "*Layer of services that establishes a set of available operations and coordinates the application response in each main operation.*"

www.EBooksWorld.ir

**Table 9.- Implementing only Domain Logic Coordination**

**Rule # D15.**

**Implementing only Domain logic coordination in the Domain Services**

**Recommendation**

- Domain Service logic must be implemented with a very clean and clear code. Therefore, we must only implement the calls to the lower level components (usually entity class logic), that is, only those actions we would explain to a Domain/Business expert. Usually (with some exceptions), coordination of the application/infrastructure actions, such as calls to Repositories, creation of transactions, use of UoW objects, etc. should not be implemented here. These other actions for coordinating our application"plumbing" should be implemented within the Application Layer Services.

- This is a recommendation to make Domain classes much cleaner. However, mixing persistence coordination code, UoW and transactions with business logic code in Domain Services could be perfectly viable (many N-layered architectures, including DDD samples, do it this way).

- Implement Domain Services only if they are necessary (YAGNI).

## 2.2.6.-SPECIFICATION Pattern

The **SPECIFICATION** pattern deals with **separating the decision as to which object types should be selected in a query from the object that makes the selection.** The Specification object will have a clear and limited responsibility that will be separated and decoupled from the Domain object that uses it.

This pattern is explained at the logical level and in detail in a paper written jointly by Martin Fowler and Eric Evans: **http://martinfowler.com/apsupp/spec.pdf**

Therefore, the main idea is for the decision of "what" candidate data must be retrieved to be separated from the candidate objects searched for, and from the mechanism used to retrieve them.

We will explain this pattern logically below, as originally defined by MF and EE. **However, in the Domain layer implementation section we will see that the**

www.EBooksWorld.ir

**implementation we chose differs from the original logical pattern due to a more powerful language offered by .NET. This refers specifically to expression trees, which provide better implementation if we only work with specifications for objects in memory, as described by MF and EE.** However, we thought that it would be convenient to explain this here according to the original definition for a full understanding of the essence of this pattern.

**Cases in which the SPECIFICATION pattern is very useful**

SPECIFICATIONS are particularly useful in applications where users are allowed to make open and compound queries, and "save" such types of queries in order to have them available in the future (e.g., a client analyst saves a compound query that he made which shows only customers of a certain country that have placed orders above $200 plus other conditions that he has selected, etc.).

**Subsumption Pattern (Related Pattern)**

Once we use the **SPECIFICATION** pattern, another very useful pattern is the **SUBSUMPTION** pattern. Subsumption refers to the action and effect of subsuming. It comes from the prefix sub- and the Latin *'sumĕre'*, which means "to take"; to include something, such as a component, in a comprehensive summary or classification or to consider something as a part of a broader package or as a special case subject to a general principle or rule (*Dictionary of the Royal Academy of the Spanish Language*).

In other words, the normal use of specifications tests these against a candidate object to see if this object meets all the requirements specified in the specification. Subsumption allows us to compare specifications to see if meeting one specification implies meeting a second one. Sometimes it is also possible to use the Subsumption pattern to implement this compliance. If a candidate object can produce a specification that characterizes it, then testing a specification is like a comparison of similar specifications. The Subsumption works especially well in Composite applications (Composite-Apps).

Since this logical concept of SUBSUMPTION starts to make things quite complicated for us, it is better to see the clarifying table offered by *Martin Fowler and Eric Evans* in their public '*paper*' on which pattern to use and how to use it depending on our needs:

www.EBooksWorld.ir

**Table 10.- SPECIFICATION Pattern Table – By MF and EE**

| Problems | Solution | Pattern |
|---|---|---|
| • We need to select a subset of objects based on some criteria.<br>• We need to check that only certain objects are used for certain roles.<br>• We need to describe what an object can do without explaining the details of how the object does it and describe how a candidate could be built to meet the requirement. | Create a specification that is able to tell if a candidate object matches some criteria. The specification has a method IsSatisfiedBy(anObject) : Boolean that returns ”true” if all criteria are met by the anObject. | SPECIFICATION |
| How do we implement a SPECIFICATION? | • We code the selection criteria into the IsSatisfiedBy() method as a block of code. | Hard Coded SPECIFICATION |
| | • We create attributes in the specification for values that commonly vary. We code the IsSatisfiedBy() method to combine these parameters to make the test. | Parameterized SPECIFICATION |
| | • Creating “leaf” elements for the various kinds of tests. | |
| | • Creating composite nodes for the ‘and’, ‘or’ and ‘not’ operators (see Combining Specifications below) | COMPOSITE SPECIFICATIONS |
| How do we compare two specifications to see if one is a special case of the other, or is substitutable for another? | Creating an operation called IsGeneralizationOf(Specification) that will answer whether the receiver is in every way equal or more general than the argument | SUBSUMPTION |

www.EBooksWorld.ir

| | | |
|---|---|---|
| • We need to figure out what still must be done to satisfy the requirements.<br>• We need to explain to the user why the Specification was not satisfied. | Adding a method RemainderUnsatisfiedBy() that returns a Specification that expresses only the requirements not met by the target object. (Best used together with Composite Specification). | PARTIALLY SATISFIED SPECIFICATION |

**Table 11.- When to use the SPECIFICATION pattern**

**Rule # D16.**

**Use the SPECIFICATION pattern when designing and implementing dynamic or composite queries**

○ **Rule**

- Identify parts of the application where this pattern is useful and use it when designing and implementing Domain components (creating Specifications) and implement specification execution within Repositories.

✓ **When to use the SPECIFICATION pattern**

**PROBLEM**

- *Selection*: We need to select a set of objects based on certain criteria and "refresh" the results in the application at certain time intervals.
- *Validation*: We need to ensure that only the proper objects are used for a particular purpose.
- *Construction to be requested*: We need to describe what an object could do without explaining the details on how it does it, but in a way that a candidate can be constructed to meet the requirement.

www.EBooksWorld.ir

**SOLUTION**

- Create a specification capable of telling if a candidate object meets certain criteria. The specification will have a Boolean IsSatisfiedBy (anObject) method that returns True if all criteria are met by that object.

**Advantages of using Specifications**

- We decouple the design of requirements, compliance and validation.
- It allows definition of clear and declarative queries.

**When not to use the Specification Pattern**

- We can fall into the anti-pattern of overusing the SPECIFICATION pattern and end up using it too much and for all types of objects. If we find out that we are not using the common methods of the SPECIFICATION pattern or that our specification object is actually representing a domain entity instead of placing restrictions on others, then we should reconsider the use of this pattern.
- In any case, we should not use it for all types of queries, only for those that we identify to be suitable for this pattern. We should not overuse it.

**References**

*Specifications Paper* by **Martin Fowler** and **Eric Evans**:

**http://martinfowler.com/apsupp/spec.pdf**

The original definition of this pattern [M.F. and E. E.], shown in the following UML diagram, explains how objects and object sets must satisfy a specification.

www.EBooksWorld.ir

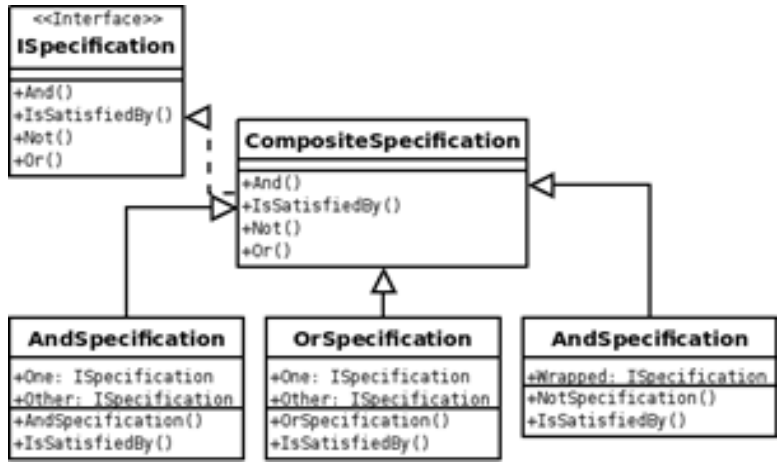



**Figure 6.- UML diagram of the Specification pattern [by Martin Fowler and Eric Evans]**

This is precisely what we stated as not making any sense in an advanced implementation when using .NET and EF (or other O/RMs) where we can work with queries that will be posed directly against the database instead of objects in memory, as the SPECIFICATION pattern originally suggested.

The main reason for the above statement comes from the pattern's own definition, which involves working with objects directly in memory; the **IsSatisfiedBy()** would take an instance of the object we want to check in order to determine if it meets certain criteria and return **true** or **false** depending on whether or not it is met. This, of course, is not desirable due to the overload entailed. Therefore, we could alter the definition of the SPECIFICATION slightly so that instead of returning a Boolean denying or confirming the compliance with a certain specification, it would return a statement with the criteria to be met.

This point will be covered and explained in more detail in our implementation of the SPECIFICATION pattern in the "Domain Layer Implementation" section, within this chapter.

## 2.3.- Domain Layer Design Considerations

When designing the Domain sub-layers, the main objective of the software architect should be to minimize the complexity, separating different tasks in different areas of concern and/or responsibility (for example, business processes, entities, etc., all of which represent different areas of responsibility.). The components we design for each area should target that specific area and should not include a code related to other areas of responsibility.

www.EBooksWorld.ir

The following guidelines should be considered when designing the business layers:

- **Define Different Types of Domain Components**: It is always a good idea to have different types of objects that implement different types of patterns, according to the type of responsibility. This will improve the maintainability and reusability of the application code. For example, we can define domain SERVICE classes and other differentiated components for query SPECIFICATION contracts and, of course, Domain ENTITY classes. Finally, we can even have executions of workflow type business processes (a workflow with dynamic business rules, etc.), although we would normally be interested in placing the coordination of workflows at a higher level; in the Application layer and not in the Domain layer.

- **Identify the Responsibilities of the Domain Layer**: The domain layer should be used to process business rules, transform data by domain logic requirements, apply policies, and implement validations related to the business requirements.

- **Design with High Cohesion.** The components should only contain specific functionality (concerns) associated with the component or entity.

- **Do not mix different types of components in the domain layers**: Domain layers should be used to decouple the business logic from the presentation and the data access code, and also to simplify unit testing of the business logic. Ultimately, this will dramatically increase the system's maintainability.

- **Reuse Common Business Logic**: It is good to use these business layers to centralize reusable business logic functions for different types of client applications (Web, RIA, Mobile, etc.).

- **Identify the Consumers of Domain Layers:** This will help determine how to expose the business layers. For example, if the presentation layer that will use the business layers is a traditional Web application, the best option is to access it directly. However, if the presentation layer is running on remote machines (RIA applications and/or *RichClient*), it will be necessary to render the Domain and Application layers through a Distributed Services layer (Web services).

- **Use abstractions to implement decoupled interfaces:** This can be achieved with interface type components, common definitions of interfaces or shared abstractions, where specific components depend on abstractions (interfaces) and not on other specific components. In other words, they do not directly depend on classes (this refers to the Dependency Injection principle for decoupling). This is especially important for the Domain SERVICES.

- **Avoid Circular Dependencies**: The business domain layers should only "know" the details related to the lower layers (Repository interfaces, etc.) and

www.EBooksWorld.ir

always (if possible), through abstractions (interfaces) and even through IoC containers. However, they should not directly "know" anything at all about the higher layers (e.g., Application layer, Service layer, Presentation layers, etc.).

- **Implement a decoupling between the domain layers and the lower (Repositories) or higher layers**: Abstractions should be used when an interface is created for the business layers. The abstraction may be implemented through public interfaces, common definitions of interfaces, abstract base classes or messaging (Web services or message queues). Additionally, the most powerful techniques to achieve decoupling between internal layers are IoC and DI.

## 2.4.- Designing and implementing Business Rules with EDA and Domain Events

> **Note**
> *It is important to point out that, in the current implementation (version 1.0) of this Architecture and its sample application v1.0, we do not use events and EDA. However, an interesting option would be to introduce the concept of "Event orientation" and Event-Sourcing as well as other possible designs and implementations.*

In connection to **EDA** (*Event Driven Architecture*), an application domain will include many business rules of the "condition" type. For example, if a customer has made purchases for over $100, he will receive offers or different treatment; in short, certain extra actions would take place. This is entirely a business rule related to the domain logic but the point is that we could implement it in different ways.

**Conditional Implementation (Traditional Code with Control** Statements**)**

We could simply implement that Rule through a conditional control statement ("if...then"); however, this type of implementation can become "spaghetti" code, as we add more and more domain rules. Moreover, we have neither a condition nor a rule reuse mechanism across the different methods of different domain classes.

**Event Oriented Domain Implementation**

Actually, for the given example, we would want something like this:

"*When a Customer has/is [something] the system must [something].*"

www.EBooksWorld.ir

This case can be very well coordinated by a model based on events. Thus, if we wanted to do more things/actions in the "do something" we could easily implement it as an additional events handler.

In short, the **Domain events** would become something similar to this:

*"When [something] has occurred, the system should [something]…"*

Of course, we could implement these events in the entities themselves, but it would be very advantageous to maintain these events entirely at the domain level.

At the time of implementation, we should implement an event at a global level and implement a subscription to this event in each business rule, while subsequently removing the subscription from the domain event.

The best way to implement this is to have each event managed by only one class that is not connected to any specific use case, but that can be activated generically by different use cases as needed.

## 2.4.1.-Explicit Domain Events

Global domain events can be implemented by specific classes so that any domain code can launch one of these events. This capacity can be implemented by static classes that use an IoC and DI container. This implementation is shown in the chapter corresponding to domain logic layer implementation.

## 2.4.2.-Unit Testing when Using Domain Events

Using domain events can complicate and undermine unit testing of these domain classes, since we need to use an IoC container to check what domain events have been launched.

However, by implementing certain functionalities in the domain event classes, we can solve this problem and run unit tests in a self-contained manner without the need to have a container. This is also shown in the chapter on domain logic layer implementation.

www.EBooksWorld.ir

# 3.- IMPLEMENTING THE DOMAIN LAYER WITH .NET 4.0 AND DECOUPLING OBJECTS WITH UNITY

The purpose of this chapter is to show the different technological options at our disposal when implementing the Domain layer and, of course, to explain the technical options chosen in our .NET 4.0 architecture reference.

We highlight the position of the Domain layer in the following *Layer* diagram of Visual Studio 2010:

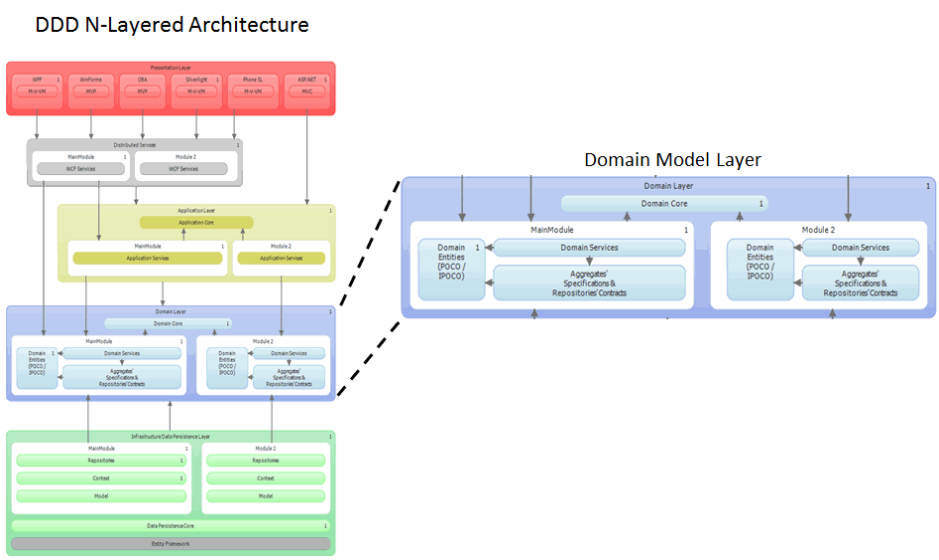



**Figure 7.- Location of the Domain Layer in the VS2010 Layer Diagram**

## 3.1.- Implementing Domain Entities

The first step we must take is to select a technology to implement the Domain entities. Entities are used to contain and manage our application main data and logic. In short, the domain entities are classes that contain values and render them through properties, but they also can and should render methods with business logic of the entity itself.

In the following sub-scheme we highlight where the entities are located within the Domain Model Layer:

www.EBooksWorld.ir

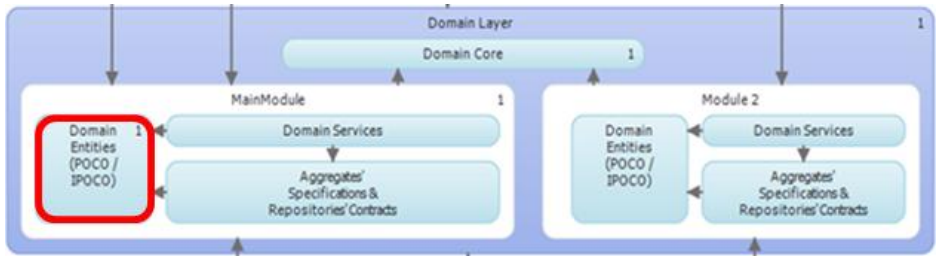



**Figure 8.- Domain Entities Sub-Scheme**

The decision of the data/technology type and format to be used for our domain entities is very important because it determines aspects affected by it, such as these questions:

- Is our Domain layer independent from the data access technology? Can we stop using our current data access technology and start using a different one while continuing to use our domain entity classes? The answer to this question would be totally different depending on the class type we are using for our domain entities (e.g., Datasets, custom classes, EF prescriptive classes, POCO/STE, etc.). For example, if we use DataSets as entities, then we certainly cannot simply change our data persistence framework, since that would require complete restructuring of the application, which will fully affect the heart of our application: the Domain Model Layer.

- In the event that we do not use DTOs and domain entities also travel to the presentation layer, the choice of entities type is even more critical as we are going to deal with interoperability issues and data serialization for Web Service remote communications, etc. Furthermore, the design and choice of technology to implement entities will greatly affect the performance and efficiency of the Domain layer.

Options of data/format/technology types:

- **POCO Classes**
  As discussed above, our entities will be implemented as simple .NET classes with fields and properties for the entity attributes. POCOs can be made manually or with the help of a code generator offered by an O/RM framework such as EF, which generates these classes automatically. This feature saves a lot of development time that otherwise would have to be spent to synchronize entities with the entity-relation model we are using. The most important rule of POCO classes is that they should not have dependency on other components and/or classes. For example, a typical EF 1.0 entity is not POCO, since it depends on the base classes of EF libraries. However, in EF 4.0 it is possible to generate POCO classes that are completely independent from the EF stack. These POCO classes are suitable for DDD N-layer architectures.

www.EBooksWorld.ir

Starting on EF 4.0 we can create POCOs using a T4 template, so the entity classes are generated and synchronized from the Entity Data Model. We might also choose to follow the new EF's Code First approach and manually create our POCOs from scratch (Please note that at the time of this writing this feature is not released yet and has been made available as a CTP). However, it will be released soon, so find out since it might be already released when you read this.

- **EF 4.0 *Self-Tracking Entities* (STE)**
  EF 4.0 *Self-Tracking Entities (STE)* are simple objects that implement some interfaces required by the EF 4.0 '*Self-Tracking'* system. To be more specific, those interfaces are **IObjectWithChangeTracker and INotifyPropertyChanged.** The important point is that these interfaces do not belong to any particular persistence framework. **IObjectWithChangeTracker** is generated by the T4 templates and **INotifyPropertyChanged** is part of the standard .NET Framework library. Therefore, these interfaces do not depend on any particular framework except .NET.
  The STE classes are very convvenient for N-Tier architectures.

- **DataSets and DataTables (basic ADO.NET classes)**
  The DataSets are similar to the disconnected databases in memory that are typically mapped fairly close to their own database schema. The use of DataSets is quite typical in .NET applications from the first version onwards, in a traditional and normal use of ADO.NET. The advantages of DataSets are that they are very easy to use, as well as being very productive in highly data oriented disconnected scenarios and CRUD applications (typically with an ADO.NET provider for a specific DBMS). "LINQ to DataSets" can also be used to work with them.
  However, DataSets have important disadvantages, which should be seriously considered:

  1. DataSets are not interoperable toward other non-Microsoft platforms, such as Java. Even though they can be serialized to XML, they may cause problems if they are used as data types in Web services.

  2. Even in cases where interoperability with other platforms is not required, using them in web services is still not recommended, since DataSets are quite heavy objects, especially when serialized into XML in order to be used in Web services. The performance of our Web Services would be much higher if we use lighter POCO classes, STE classes and of course, custom DTOs. Therefore, we do not recommend the use of DataSets in communications across boundaries defined by Web services or even in inter-process communications (i.e. between different .exe processes).

  3. O/RMs do not support/work with DataSets.

www.EBooksWorld.ir

4. DataSets are not designed to represent pure Domain entities with their domain logic included. The use of DataSets does not fit in a DDD Architecture because we will end up with an "Anemic Domain" having the domain entity logic separated (in parallel classes) from the domain entity data (in DataSets). Therefore, this option is not suitable for DDD.

- **XML**
  This is simply the use of fragments of XML text that contains structured data. Normally, we tend to make use of this option (representing domain entities with XML fragments) if the presentation layer requires XML or if the domain logic must work with XML content that should match specific XML schemes. Another advantage of XML is that, being simply formatted text, these entities will be fully interoperable.

For example, a system where this option would be commonly used is a message routing system where the logic routes the messages based on well-known nodes of the XML document. Keep in mind that the use and manipulation of XML may require great amounts of memory in scalable systems (many simultaneous users); if the XML volume is high, the access and process of the XML can also become a bottleneck when processing with standard APIs for XML documents.

The biggest problem with XML based entities is that it would not be "Domain Oriented" because we would have an "Anemic Domain" where the domain entity logic is separated from the domain entity data (XML). Therefore this option is not suitable in DDD either.

**Table 12.- Domain Entities Rule**

**Rule # I5.**

**By default, Domain entities will be implemented as POCO classes or Self-Tracking Entities (STE), generated by T4 templates or manually created by us. ('Code First approach')**

**Rule**

- According to the considerations above, since this Framework Architecture is Domain Oriented and we should achieve maximum independence in the Domain objects, the domain entities will be implemented as POCO classes or STE . In EF 4.0. These are typically generated by T4 templates to save a lot of time in the implementation of these classes.

- Creating them manually (e.g., using a *Code-First* approach in .NET 4.1) is another viable option which is even a more "pure DDD" way, but it will take more work in order to handle 'Optimistic Concurrency', etc.

www.EBooksWorld.ir

**Advantages of POCO entities.**

- They are independent of specific technology libraries.
- They are relatively lightweight classes that provide good performance.
- They are the most suitable option for DDD N-Layer Architectures.

**When to use EF Self-Tracking Entities**

- The use of **STEs** is recommended in most applications where we have complete control because they are more productive than POCO entities. STEs offer a very simplified optimistic concurrency management in N-Tier Applications.
- **STEs are suitable for N-Tier applications where we control their end-to-end development. They are not, however, suitable for applications where there is no intention of sharing real data types between the client and server; for example, pure SOA applications where we only control one end, either the service or the consumer. In these cases, where there is no possibility or permission to share data types, it is recommended to make use of DTOs in distributed services (Web Services, etc.).**

**When to use POCO Entities**

- On the contrary, if our application is an application or service with a strong SOA orientation, then only DTOs should be used and we would be managing the concurrency aspects (Optimistic Concurrency managed by us, etc.). The use of **POCO** domain entities is recommended in these cases. The POCO option will result in some very simplified entities, although we will have to make much greater efforts in the implementation of our system (e.g,. converting DTOs to domain entities, manual implementation of optimistic concurrency, etc.).

**References**

*POCO in the Entity Framework:*
**http://blogs.msdn.com/adonet/archive/2009/05/21/poco-in-the-entity-framework-part-1-the-experience.aspx**

*Self-Tracking Entities in the Entity Framework:*
**http://blogs.msdn.com/efdesign/archive/2009/03/24/self-tracking-entities-in-the-entity-framework.aspx**

www.EBooksWorld.ir

## 3.2.- Generation of POCO/STE Entities with EF T4 Templates (Model First and Database First)

In our sample application implementation, we have chosen to go with **STE** because of its very balanced approach. We can either do EF *database-first* approach or *model-first* approach when using STE. Regarding "*Code-Fist*" approach, it was not released at the time of this writing.

However, we also encourage evaluating the Code-First approach, as it fits very well with DDD concepts. (See short intro we make about it in this chapter, later on).

**Important**:
Although the concept and implementation of the entities correspond to the Domain Layer, when using STE and POCO templates, **the initial generation of these entities is usually accomplished using Visual Studio when we are implementing the data persistence infrastructure layer, while creating the EF Entity Data Model. Therefore, the process of "how to generate EF POCO/STE entities" is explained in the chapter on implementation of the Data Persistence Infrastructure Layer, but such entities are placed in an assembly/project belonging to the Domain Layer. Review the section Ways of generating entities with T4 templates in the Data Persistence chapter.**

Our custom entity classes (POCO/STE classes) generated by EF will be similar to the following STE classes:

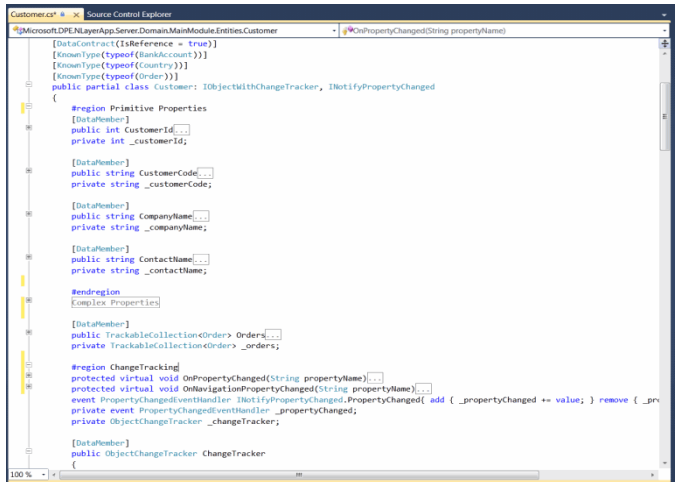



**Figure 9.- EF STE Custom classes**

www.EBooksWorld.ir

## 3.3.- 'Code First' approach for implementing POCO entity classes

This is another compelling option instead of using the T4 templates. Both options can be very useful and either one of them can be chosen, depending on requirements.

For an application with a very long life and many Domain logic changes made by a development team other than the GUI team (Presentation Layer) or other consuming applications, as well as a very high number of entities (many hundreds of entities), the recommended choice would be the '***Code-First + DTOs' approach***.

On the other hand, if the project/application is medium/high and we want to leverage EF productivity features (such as visual entity-model synchronization and automatic Optimistic Concurrency exceptions, for N-Tier applications, managed by STE), then the recommended option is 'Self Tracking Entities'.

***Code First*** **allows us to define our model using our own POCO C# or VB.Net classes**. Additional optional configuration can be performed using attributes on our classes and properties or by using the *Fluent API*. Our model can be used to generate a database schema or to map to an existing database.

### 3.3.1.- Mapping to an Existing Database

The information we are going to explain here is based on EF 4.1 CTP5, so please take into account that it could be slightly different when using the final released version.

In EF 4.1 CTP5 and 'Code-First' approach, if Code First detects that it is pointing to an existing database schema that it did not create, it will 'trust us' and attempt to use the schema. The easiest way to point Code First to an existing database is to add a App/Web.config connection string with the same name as your derived DbContext. Here is an example:

```
<connectionStrings>
  <add
    name=" ModuleContext"
    providerName="System.Data.SqlClient"

connectionString="Server=.\SQLEXPRESS;Database=MyDatabase;Trusted_Connec
tion=true;"/>
   </connectionStrings>
```

Same name as my derived DbContext

'*Code-First*' Connection string database

Similar steps could be taken in order to demonstrate Code First generating the database schema rather than mapping to an existing database, with the exception of 'Setting an Initialization Strategy' which does not apply to existing databases.

www.EBooksWorld.ir

## 3.3.2.- Creating the Model

For this implementation example, we are going to define a very simple object model. We would define it in a separate project (within the Domain Layer) specifically designed for holding Domain Entities.

Below we define the following two entity classes:

```
public class Order
{
    public string OrderId { get; set; }
    public string CustomerName { get; set; }

    public virtual ICollection<Product> OrderItems { get; set; }
}

public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public string Price { get; set;}
    public string CategoryId { get; set; }
   }
```

'*Code-First*' Entity

**At the moment they are only holding data attributes. We should add entity logic within the entity itself (methods), later on.**

## 3.3.3.- Create a Context

The simplest way to start using the classes for data access is to define a context that derives from *System.Data.Entity.DbContext* and expose a typed DbSet<TEntity> for each class in the model.

We could align this design with our UnitOfWork interface, but we will use a straightforward approach for the sake of simplicity and because we have not yet implemented it in our sample app.

We are going to use types defined in EF CTP5, so we need to add a reference to the CTP5 assembly:

*Project -> Add Reference…*
*Select the ".NET" tab*
*Select "EntityFramework" from the list*
*Click "OK"*

www.EBooksWorld.ir

We will also need a reference to the existing Entity Framework assembly:

*Project -> Add Reference…*
*Select the ".NET" tab*
*Select "System.Data.Entity" from the list*
*Click "OK"*

Add a using statement for System.Data.Entity at the top of the file.

```
using System.Data.Entity;
```

Then we need to add a derived context related to the existing entity classes that we have already defined.

Our Context / Unit of Work

```
public class ModuleContext : DbContext
{
    public DbSet<Order> Orders { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

This code will probably need to be located within the Data persistence Layer, since at this point we are dealing with EF context and, internally, dealing with database connections.

Also, in order for it to be a proper 'Unit of Work' we should have our own UoW interface/Contract. We will not do this now for the sake of simplicity.

That is all the code we need to write to start storing and retrieving data. Obviously there is quite a bit going on behind the scenes.

## 3.3.4.- Writing & Reading Data

Some of the code which could fit in the Repositories would be the following (remember that this code should be located within the data access and persistence Layer, not within the Domain or Application Layer):

```
using (var db = new ModuleContext())
{

   // Add an Order (Hardcoded, not for production application)

   var order = new Order { OrderId = "OR1001", CustomerName =
"Microsoft" };

   db.Orders.Add(order);

   int recordsAffected = db.SaveChanges();

}
```

Adding an Order to Orders

Persisting Data to

www.EBooksWorld.ir

After executing that code, check that the entity is updated into the database. We could also write some code in order to search for specific data, such as:

```
using (var db = new ModuleContext())
{
    // Use Find to locate the 'OR1001' Order
    var order = db.Orders.Find("OR1001");
}
```

Search based on Find() and ID field.

Or by using a LINQ query:

```
using (var db = new ModuleContext())
{
    // Query for all 'Software' products using LINQ
    var allSoftware = from p in db.Products
                      where p.CategoryId == "SOFTWARE"
                      orderby p.Name
                      select p;}
}
```

LINQ query on '*Code-First'* entities

## 3.3.5.- Where Is My Data persisted?

By convention, *DbContext* created a database for us on localhost\SQLEXPRESS. The database is named after the fully qualified name of the derived context. In our case, this will be "CodeFirstSample.ModuleContext". This name can be changed.

## 3.3.6.- Model Discovery

*DbContext* works out what classes to include in the model by looking at the DbSet properties that we have defined. It then uses the default 'Code First' conventions to find primary keys, foreign keys etc. There is a full set of conventions implemented for 'Code-First'.

## 3.3.7.- Changing the Database Name

One of the ways to change the name of the database that is created for us is to alter the constructor on the DbContext, specifying the desired database name.

If we want to change the name of the database to "***NLayerSampleDatabase***", we could add a default constructor to our derived context that passes this name down to DbContext:

www.EBooksWorld.ir

```
public class ModuleContext : DbContext
{
    public ModuleContext()
        : base("NLayerSampleDatabase")
    { }

    public DbSet<Orders> Orders { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

Specific Database name

There are a number of other ways to specify which database should be connected to, such as:

- App.config Connection String
    - Create a connection string in the App.Config file with the same name as the context.
- DbConnection
    - There is a constructor on DbContext that accepts a DbConnection.
- Replacing the Default Convention
    - The convention used to locate a database based on the context name is an AppDomain wide setting that can be changed via the static property *System.Data.Entity.Database.DbDatabase.DefaultConnectionFactory*

## 3.3.8.- Data Annotations

So far, we have simply let EF discover the model using its default conventions. However, at times our classes will not follow the conventions and we will need to be able to perform further configuration. There are two options for this:

- Data Annotations
- Code First Fluent API

We are going to show a glimpse of '*Data Annotations*'.
First of all, we could add a Customer entity class to our model:

```
public class Customer
{
    public string CustomerCode { get; set; }
    public string Name { get; set; }
    public string LastName { get; set; }
}
```

We also need to add a set to our current derived context.

www.EBooksWorld.ir

```
public class ModuleContext : DbContext
{
    public ModuleContext ()
        : base("NLayerSampleDatabase")
    { }

    public DbSet<Orders> Orders { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Customer> Customers { get; set; }
}
```

Now, if we run our application we will get an *InvalidOperationException* saying "*EntityType 'Customer' has no key defined. Define the key for this EntityType.*" because EF has no way of knowing that CustomerCode should be the primary key for Customer.

Since we are going to use Data Annotations, we need to add a reference:

- Project -> Add Reference…
- Select the ".NET" tab
- Select "System.ComponentModel.DataAnnotations" from the list
- Click "OK
- Add a using statement at the top of your C# Code file:

```
using System.ComponentModel.DataAnnotations;
```

Now we can annotate the *CustomerCode* property to identify that it is the primary key:

```
public class Customer
{
    [Key]
    public string CustomerCode { get; set; }
    public string Name { get; set; }
    public string LastName { get; set; }
    public string ZipCode { get; set; }
}
```

Entity Key annotation

The full list of annotations supported in EF CTP5 is the following:

www.EBooksWorld.ir

- KeyAttribute
- StringLengthAttribute
- MaxLengthAttribute
- ConcurrencyCheckAttribute
- RequiredAttribute
- TimestampAttribute
- ComplexTypeAttribute
- ColumnAttribute
    - Placed on a property to specify the column name, ordinal & data type
- TableAttribute
    - Placed on a class to specify the table name and schema
- InversePropertyAttribute
    - Placed on a navigation property to specify the property that represents the other end of a relationship
- ForeignKeyAttribute
    - Placed on a navigation property to specify the property that represents the foreign key of the relationship
- DatabaseGeneratedAttribute
    - Placed on a property to specify how the database generates a value for the property (Identity, Computed or None)
- NotMappedAttribute
    - Placed on a property or class to exclude it from the database

www.EBooksWorld.ir

## 3.3.9.- Validation

In EF 4.1 CTP5 there is a new feature that will validate if the instance data satisfies data annotations before attempting to save it into the database.

Let's add annotations to specify that **Customer.Name** must be mandatory and also that **Customer.ZipCode** must be exactly 5 characters long:

```
public class Customer
{
    [Key]
    public string CustomerCode { get; set; }
    [Required]
    public string Name { get; set; }
    public string LastName { get; set; }

    [StringLength(5, MinimumLength = 5)]
    public string ZipCode { get; set; }
}
```

Annotations attributes for data validations

Then, if we try to input invalid data, we will get exceptions like this:

*'CodeFirstSample.Customer failed validation ZipCode :*
*The field ZipCode must be a string or array type with a minimum length of '5'.*

**In conclusion, we strongly encourage an evaluation of this kind of EF approach because it strongly matches DDD principles and patterns**. In addition, at the time you are reading this, the Code First RTM will probably have been released.

## 3.4.- Domain Logic in Entity Classes

**In DDD it is essential to locate the logic related to internal operations of an entity within the class of that entity itself.** If the entity classes were used only as data structures and the entire domain logic was separated and placed into the Domain Services, this would constitute an anti-pattern called "Anemic Domain Model".

Therefore, we should add the domain/business logic related to the internal part of each entity's data within each entity class. If we use POCOs or STEs generated by T4 templates, we can add domain logic through partial classes, such as the one that can be seen in the following code called "*Custom-Partial Entity Classes*". For example, the following partial class *BankAccount* adds a kind of domain logic to the domain entity class itself. In particular, the operation is "charging an account" and the business performs necessary checks before making a charge to that account:

www.EBooksWorld.ir

```
namespace Microsoft.Samples.NLayerApp.Domain.MainModule.Entities
{
public partial class BankAccount
    {
        /// <summary>
        /// Deduct money to this account
        /// </summary>
        /// <param name="amount">Amount of money to deduct</param>
        public void ChargeMoney(decimal amount)
        {
            //Amount to Charge must be greater than 0. --> Domain logic.
            if (amount <= 0)
                throw new
ArgumentException(Messages.exception_InvalidArgument, "amount");


            //Account must not be locked, and balance must be greater
than cero.
            if (!this.CanBeCharged(amount))
                throw new
InvalidOperationException(Resources.Messages.exception_InvalidAccountToB
eCharged);

            //Charge means deducting money to this account. --> Domain
Logic
this.Balance -= amount;
        }
...
...
...
    }
}
```

Business checks / validations

Business/Domain logic for the Bank Account Charging process of the BankAccount

It's worth noting that if we had a Code First approach, then we would implement that logic within the entity class itself, since in Code First we usually would not create partial classes.

## 3.5.- Location of Repository Contracts/Interfaces in the Domain Layer

As explained during the theoretical chapters on DDD Architectural layers, the repository interfaces are the only things known about repositories by the Domain layer, and the instantiation of *Repository* classes will be made by the chosen IoC container (in this case *Unity*). Hence, we will have the Domain layer and the data persistence infrastructure layer's repository classes completely decoupled.

In the following sub-scheme we emphasized the location of the Repository contracts/interfaces within the Domain layer:

www.EBooksWorld.ir

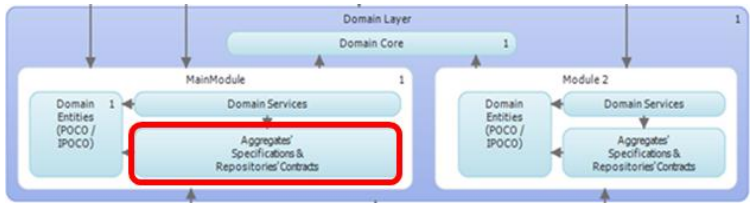



**Figure 10.- Domain Contract Scheme**

Thus, in our example, these interfaces will be defined in the following *namespace* within the Domain layer:

```
Microsoft.Samples.NLayerApp.Domain.MainModule.Repositories.Contracts
```

This allows us to completely replace the data persistence infrastructure layer, the repositories themselves or their implementations without affecting the Domain layer or having to change dependencies or making any re-compilations. Thanks to this decoupling, we can *mock* repositories and the domain business classes can dynamically instantiate "fake" classes (*stubs* or *mocks*) without having to change code or dependencies. This is done simply by specifying in the IoC container registration that, when it is asked to instantiate an object for a given interface, it must instantiate a fake (mock) class instead of the real one (both meeting the same interface, obviously).

**Important:**
Although the repository contract/interface should be located in the Domain layer for the reasons highlighted above, their implementation is explained with code examples in the chapter on "Implementation of Data Persistence Infrastructure Layer."

**Table 13.- Location of Repository Contract/Interface**

| Rule # I6 | Positioning Repository contracts/interfaces in the Domain Layer. |
|---|---|

**Rule**

- To maximize the decoupling between the Domain Layer and the Data Access and Persistence Infrastructure Layer, it is important to locate the repository contracts/interfaces in the Domain Layer, and not in the Data Persistence layer itself.

**References**

*Repository contracts in the Domain – (Book DDD by **Eric Evans**)*

www.EBooksWorld.ir

One example of Repository contract/interface within the Domain layer might be as follows:

Repository Contract *Namespace* is within the Domain layer

```
C#

namespace Microsoft.Samples.NLayerApp.Domain.MainModule.Orders
{
public interface IOrderRepository : IRepository<Order>
   {
    IEnumerable<Order> FindOrdersByDates(OrderDateSpecification
orderDateSpecification);

    IEnumerable<Order>
FindOrdersByShippingSpecification(OrderShippingSpecification
orderShippingSpecification);

    IEnumerable<Order> FindOrdersByCustomerCode(string customerCode);
   }
}
```

## 3.6.- Implementing Domain Services

In the following sub-scheme we emphasize where the "Domain SERVICES" classes are located within the Domain layer:

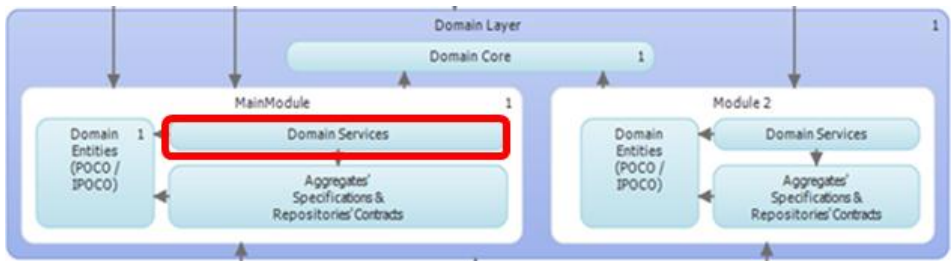



**Figure 11.- Domain Services**

**A SERVICE is an operation or set of operations offered as an interface that is only available in the model.**

The word "Service" comes from the SERVICE pattern and precisely emphasizes what is offered: "*What it can do and what actions are offered to the client by whom it is being used and emphasizes its relationship with other Domain objects (Incorporating several Entities, in some cases*)."

Normally, we will implement SERVICE classes as simple .NET classes with methods where the different possible actions related to one or several Domain entities are implemented; in short, implementation of actions as methods.

The SERVICE classes should encapsulate and isolate the data persistence infrastructure layer. It is in these business components, where all business rules and

www.EBooksWorld.ir

calculations –which are not internal of the ENTITIES themselves– such as complex/global operations involving the use of multiple entity objects, as well as business data validations required for a process must be implemented.

## 3.6.1.- Domain SERVICES as Business Process Coordinators

As explained in detail in the chapter on designing the Domain Layer Architecture, **the SERVICE classes are primarily business process coordinators which normally cover different concepts and ENTITIES associated with scenarios and complete use cases.**

For example, a Domain service would be a class that coordinates an operation that incorporates different entities, and even operations of other related Services.

This code is an example of a domain's SERVICE class aimed at coordinating a business operation:

```
C#
…
…
namespace Microsoft.Samples.NLayerApp.Domain.MainModule.Services
{
    public class TransferService : ITransferService
    {
public void PerformTransfer(BankAccount originAccount, BankAccount
destinationAccount, decimal amount)
        {
//Domain Logic
            //Process: Perform transfer operations to in-memory Domain-
Model Objects
            // 1.- Charge money to origin acc
            // 2.- Credit money to destination acc
            // 3.- Annotate transfer to origin account

            //Number Accounts must be different
            if (originAccount.BankAccountNumber !=
destinationAccount.BankAccountNumber)
            {
                //1. Charge to origin account (Domain Logic)
originAccount.ChargeMoney(amount);

                //2. Credit to destination account (Domain Logic)
destinationAccount.CreditMoney(amount);

                //3. Anotate transfer to related origin account
originAccount.BankTransfersFromThis.Add(new BankTransfer()
                {
                    Amount = amount,
                    TransferDate = DateTime.UtcNow,
```

Domain Service *Namespace* in a specific module

Domain Service

Contract/Interface to be achieved

Charge to Account

Pay to Account

Register/Log Operation

www.EBooksWorld.ir

```
                        ToBankAccountId = destinationAccount.BankAccountId
                    });
                }
                else
                    throw new
InvalidOperationException(Resources.Messages.exception_InvalidAccountsFo
rTransfer);
            }
        }
    }
```

As can be seen, the Domain Service code above is very clean and is only related to the business logic and business data. There are no "application plumbing" operations such as the use of Repositories, Unit of work, transaction creation, etc. (Sometimes, however, we will need to use/call Repositories from within Domain Services when we need to query data depending on specific domain conditions).

In the Domain Service methods we simply interact with the logic offered by the entities that are involved. In the example above we call methods (*ChargeMoney*(), *CreditMoney*(), etc.) that belong to the entities themselves.

**Recall that**, normally, **in the implementation of Domain Service methods, all the operations are only performed against objects/entities in memory, and when the execution of our Domain Service method is completed, we have simply modified the data Entities and/or Value Objects of our domain model. Nonetheless, all these changes are still only in the server's memory (EF's entities context).** The persistence of these objects and changes in data performed by our logic will not be made until we coordinate/execute it from the Higher Layer of our Application which will invoke the Repositories within complex application logic (UoW and transactions).

The Application Layer, which is a higher layer level, will normally call Domain services, providing the necessary entities after having made the corresponding queries through the Repositories. And finally, this Application Layer will also be what coordinates the persistency in storages and databases.

### Important:

Knowing how to answer the following question is important: 'What code should I implement within Domain Layer Services?' The answer is:

'*Only business operations we would discuss with a Domain Expert or an end user*'.

We would not talk about "application plumbing", how to create transactions, UoW, use of Repositories, persistence, etc. with a domain expert, which is why everything is related to coordination. On the other hand, the pure logic of the Domain should not be located in the Application Layer, so we do not get the Domain Logic "dirty".

As mentioned above, the only exception to this is when a Domain Service needs to obtain and process data depending on specific and variable domain states. In that case, we will need to consume/call Repositories from Domain Services. This is usually done just to query data.

www.EBooksWorld.ir

## 3.7.- SPECIFICATION Pattern

As explained in the Domain logic layer section, the **SPECIFICATION** pattern approach concerns **separating the decision on the object type to be selected in a query from the object that performs the selection**. The "Specification" object will have a clear and limited responsibility that should be separated and decoupled from the Domain object that uses it.

Therefore, the main idea is that the decision of "what" candidate data is to be obtained should be separated from the candidate objects searched for and the mechanism used to find them.

### 3.7.1.- Use of the SPECIFICATION Pattern

The *specification* pattern will normally be used from the application layer where we define the logical queries we want to make, but this will be decoupled with regard to the actual implementation of these logical queries in the data access and persistence infrastructure layers.

Below we show the code where we use a specification.

```
… //Application Layer
…
                          Method with simple use of SPECIFICATION

public Customer FindCustomerByCode(string customerCode)
{
    //Create specification
CustomerCodeSpecification spec = new
CustomerCodeSpecification(customerCode);

    return _customerRepository.FindCustomer(spec);
}
…                         Method with complex use of SPECIFICATION
…

public List<Customer> FindPagedCustomers(int pageIndex, int pageCount)
{
   //Create "enabled variable" transform adhoc execution plan in
prepared plan
   bool enabled = true;
Specification<Customer> onlyEnabledSpec = new
DirectSpecification<Customer>(c => c.IsEnabled == enabled);

   return _customerRepository.GetPagedElements(pageIndex, pageCount, c
=> c.CustomerCode, onlyEnabledSpec, true)
                                        .ToList();
}
```

www.EBooksWorld.ir

## 3.7.2.- Implementation of the SPECIFICATION Pattern

However, the implementation we choose differs in part from the original logical pattern defined by MF and EE. This is due to the increased power language offered in .NET, such as expression trees, where we can achieve much greater benefit than if we work only with specifications for objects in memory, as originally defined by MF and EE.

This is precisely what was mentioned as not making any sense in advanced implementation with .NET and EF (or other O/RMs), where we can work with queries that will be posed directly against the database instead of objects in memory, as originally assumed by the SPECIFICATION pattern.

The main reason for the statement above derives from the definition of the pattern. It involves working with objects directly in memory because the **IsSatisfiedBy()** method will take an instance of the object where we want to confirm if it meets certain criteria and return **true** or **false** depending on whether there is compliance or not. This, of course, is undesirable due to the overload it would entail. Given all of the above, we might modify our SPECIFICATION definition a bit so that, instead of returning a Boolean denying or confirming the compliance with a certain specification, it returns an expression with the criteria to be met.

In the following code fragment we would have a skeleton of our base contract with this slight modification:

C#

*Skeleton/Interface of our base contract*

```
public interface ISpecification<TEntity>
        where TEntity : class,new()
{
/// <summary>
/// Check if this specification is satisfied by a
/// specific lambda expression
/// </summary>
/// <returns></returns>
Expression<Func<TEntity, bool>> SatisfiedBy();
}
```

***We use* SatisfiedBy() instead of the original IsSatisfiedBy()**

At this point we could say that we already have the base and the idea of what we want to build, so now all we need to do is follow the rules and guidelines of this pattern to begin creating our direct specifications or "*hard coded specifications*", as well as our composite specifications, in the "And", "Or" style, etc.

**Table 14.- Objective of SPECIFICATION pattern implementation**

| Objective of SPECIFICATION pattern implementation |
| --- |
| In short, while maintaining the principle of separation of responsibility, and considering that a SPECIFICATION is a business concept (a special search type, that is perfectly explicit) we are searching for an elegant manner to perform different queries in terms of parameters using expressions of conjunctions or disjunctions. |

www.EBooksWorld.ir

We could state specifications as follows:

```
C#

/// <summary>
/// AdHoc specification for finding orders
/// by shipping values
/// </summary>
public class OrderShippingSpecification
    : Specification<Order>
{
    string _ShippingName = default(String);
    string _ShippingAddress = default(String);
    string _ShippingCity = default(String);
    string _ShippingZip = default(String);

    public OrderShippingSpecification(string shippingName, string
shippingAddress, string shippingCity, string shippingZip)
    {
        _ShippingName = shippingName;
        _ShippingAddress = shippingAddress;
        _ShippingCity = shippingCity;
        _ShippingZip = shippingZip;
    }

    public override System.Linq.Expressions.Expression<Func<Order,
bool>> SatisfiedBy()
    {
        Specification<Order> beginSpec = new
TrueSpecification<Order>();

if (_ShippingName != null)
            beginSpec &= new DirectSpecification<Order>(o =>
o.ShippingName != null && o.ShippingName.Contains(_ShippingName));

if (_ShippingAddress != null)
            beginSpec &= new DirectSpecification<Order>(o =>
o.ShippingAddress != null &&
o.ShippingAddress.Contains(_ShippingAddress));

if (_ShippingCity != null)
            beginSpec &= new DirectSpecification<Order>(o =>
o.ShippingCity != null && o.ShippingCity.Contains(_ShippingCity));

if (_ShippingZip != null)
            beginSpec &= new DirectSpecification<Order>(o =>
o.ShippingZip != null && o.ShippingZip.Contains(_ShippingZip));

        return beginSpec.SatisfiedBy();

    }

}
```

**Specification to get Orders depending on the Shipping Address**

**Constructor with values required by the Specification. Consider that it is pointless to use DI/IoC to instantiate a Specification object**

**The SatisfiedBy() method returns a Lambda expression of Linq**

Please note how the above specification, *OrderShippingSpecification*, provides us with a mechanism to know the criteria of the elements we want to search for; but it does not know anything about who will perform the search operation thereof. In addition to this clear separation of responsibilities, the creation of these elements also

www.EBooksWorld.ir

helps us to make important domain operations, such as types of search criteria, perfectly clear. Otherwise, these would be scattered around different parts of the code, making them more difficult and expensive to modify. Finally, another advantage of specifications, as proposed here, is the possibility of performing logical operations on them, which provides us with a simple mechanism to perform dynamic queries in LINQ.

## 3.7.3.- Composing Specifications with AND/OR Operators

Although there is evidently more than one approach to implement these operators, we chose to implement them with the VISITOR pattern to evaluate the expressions (ExpressionVisitor):
**http://msdn.microsoft.com/en-us/library/system.linq.expressions.expressionvisitor(VS.100).aspx)**.

What we need is the following class to come up with a re-composition of the expression instead of an **InvocationExpression** (not compatible with EF 4.0).

This class of support is displayed below:

**C#**

```
/// <summary>
/// Extension method to add AND and OR with rebinder parameters
/// </summary>
public static class ExpressionConstructor
{
    public static Expression<T> Compose<T>(this Expression<T> first,
Expression<T> second, Func<Expression, Expression, Expression> merge)
    {
        // build parameter map (from parameters of second to
parameters of first)
        var map = first.Parameters.Select((f, i) => new { f, s =
second.Parameters[i] }).ToDictionary(p => p.s, p => p.f);

        // replace parameters in the second lambda expression with
parameters from the first
        var secondBody = ParameterRebinder.ReplaceParameters(map,
second.Body);
        // apply composition of lambda expression bodies to
parameters from the first expression
        return Expression.Lambda<T>(merge(first.Body, secondBody),
first.Parameters);
    }
    public static Expression<Func<T, bool>> And<T>(this
Expression<Func<T, bool>> first, Expression<Func<T, bool>> second)
    {
        return first.Compose(second, Expression.And);
    }
    public static Expression<Func<T, bool>> Or<T>(this
Expression<Func<T, bool>> first, Expression<Func<T, bool>> second)
    {
        return first.Compose(second, Expression.Or);
   }
}
```

**EXPRESSIONS constructor**

www.EBooksWorld.ir

The complete definition of an AND specification, therefore, is shown in the following code:

**C#**

```
/// <summary>
/// A logic AND Specification
/// </summary>
/// <typeparam name="T">Type of entity that checks this
specification</typeparam>
public class AndSpecification<T> : CompositeSpecification<T>
    where T : class,new()
{

    private ISpecification<T> _RightSideSpecification = null;
    private ISpecification<T> _LeftSideSpecification = null;

    /// <summary>
    /// Default constructor for AndSpecification
    /// </summary>
    /// <param name="leftSide">Left side specification</param>
    /// <param name="rightSide">Right side specification</param>
    public AndSpecification(ISpecification<T> leftSide,
ISpecification<T> rightSide)
    {
        if (leftSide == (ISpecification<T>)null)
            throw new ArgumentNullException("leftSide");

        if (rightSide == (ISpecification<T>)null)
            throw new ArgumentNullException("rightSide");

        this._LeftSideSpecification = leftSide;
        this._RightSideSpecification = rightSide;
    }

    /// <summary>
    /// Left side specification
    /// </summary>
    public override ISpecification<T> LeftSideSpecification
    {
        get { return _LeftSideSpecification; }
    }

    /// <summary>
    /// Right side specification
    /// </summary>
    public override ISpecification<T> RightSideSpecification
{
        get { return _RightSideSpecification; }
    }
    public override Expression<Func<T, bool>> SatisfiedBy()
    {
        Expression<Func<T, bool>> left =
_LeftSideSpecification.SatisfiedBy();
        Expression<Func<T, bool>> right =
_RightSideSpecification.SatisfiedBy();

return (left.And(right));
    }
}
```

**AND specification**

**The SatisfiedBy() method required by our SPECIFICATON pattern**

www.EBooksWorld.ir

The hierarchy of specifications proposed in the paper by Evans and Fowler includes specifications from the NOT specification to a basis for LeafSpecifications that we would have to build.

## 3.8.- Implementing Unit Testing for the Domain Layer

Like most elements of the solution, our Domain Layer is one of the areas that should be covered by unit testing. It should, of course, meet the same requirements demanded in the rest of the layers or parts of a project. The main items within this layer that must have good code coverage are the entities and the domain services sub-layer.

Regarding entities, we should create tests for internal business methods therein, since the rest of the code is generated automatically by EF T4 templates as discussed in the previous sections. The case of domain services is different because the entire code is *adhoc, so* we should have tests for each one of the developed elements. Basically, a Domain layer testing project should be added for each solution module. If we have a *Main Module,* we should have a test project such as *Domain.MainModule.Tests* where we will have a set of tests both in entities and in services.

In the following code example we can see some tests of the domain entity *BankAccount.*

**C#**

```
[TestClass()]
public class BankAccountTest
{

    [TestMethod()]
    public void CanTransferMoney_Invoke_Test()
    {
        //Arrange
        BankAccount bankAccount = new BankAccount()
        {
            BankAccountId = 1,
            Balance = 1000M,
            BankAccountNumber = "A001",
            CustomerId = 1,
            Locked = false
        };

        //Act
        bool canTransferMoney =
bankAccount.CanTransferMoney(100);

        //Assert
        Assert.IsTrue(canTransferMoney);
    }

    [TestMethod()]
    public void CanTransferMoney_ExcesibeAmountReturnFalse_Test()
    {
```

www.EBooksWorld.ir

```
                //Arrange
                BankAccount bankAccount = new BankAccount()
                {
                    BankAccountId = 1,
                    Balance = 100M,
                    BankAccountNumber = "A001",
                    CustomerId = 1,
                    Locked = false
                };

                //Act
                bool canTransferMoney =
bankAccount.CanTransferMoney(1000);

                //Assert
                Assert.IsFalse(canTransferMoney);
            }
            [TestMethod()]
            public void CanTransferMoney_LockedTruetReturnFalse_Test()
            {
                //Arrange
                BankAccount bankAccount = new BankAccount()
                {
                    BankAccountId = 1,
                    Balance = 1000M,
                    BankAccountNumber = "A001",
                    CustomerId = 1,
                    Locked = true
                };

                //Act
                bool canTransferMoney =
bankAccount.CanTransferMoney(100);

   //Assert
   Assert.IsFalse(canTransferMoney);
            }
        }
```

www.EBooksWorld.ir

**Table 15.- Domain layer tests**

**Rule # I7.**

**Always implement Unit Testing for the Domain Layer.**

**Recommendation**

- Add the option for domain layer testing to be run in isolation from any dependency, such as a database. This allows tests to run faster so the developer will not have any problems in running a set of them in each code change.
- Check that all the tests are repeatable, that is, two sequential executions of the same test return the same result, without requiring a prior step.
- Avoid excessive test cleanup and preparation code since it might affect readability.

**References**

*Unit Test Patterns:* **http://xunitpatterns.com**

www.EBooksWorld.ir

CHAPTER 6

# Application Layer

## 1.- APPLICATION LAYER

Following the trend of DDD architecture, the Application layer should be a Layer that coordinates the activities of the Application as such, but it is essential that it not include any domain logic or business/domain state. However, it can contain progress states of the application tasks.

The SERVICES that typically reside within this layer (remember that the SERVICE pattern is applicable to different Architecture layers), are services that usually coordinate the SERVICES of other lower level layers.

The most common case of an Application Service is a Service that coordinates all the "plumbing" of the application, that is, orchestration of calls to the Domain Services and later, calls to Repositories to perform persistence, using UoW, transactions, etc.

Another more collateral case would be a SERVICE of the APPLICATION layer responsible for receiving E-Commerce purchase orders in a specific XML format. In this scenario, the APPLICATION layer will be responsible for reformatting/rebuilding such Purchase Orders from the original XML received and converting them into Domain Model ENTITY objects. This example is a typical case of APPLICATION logic. We need to perform a format conversion, which is a requirement of the application and not something that forms part of the Domain logic, so it should not be located in the Domain layer but rather in the Application layer.

**In short, we will locate all the coordination necessary to perform complete operations/scenarios in the Application layer SERVICES, but we would never**



www.EBooksWorld.ir

**discuss the application tasks, informally known as application "plumbing" coordination, with a domain expert (end user). The Application layer is generally useful as a *Façade Layer* exposing the server components to consumer layers or external applications (Presentation layers or other remote services.).**

## 2.- APPLICATION LAYER LOGICAL DESIGN AND ARCHITECTURE

In the following diagram we show how the Application layer typically fits into our *N-Layered Domain Oriented* architecture:

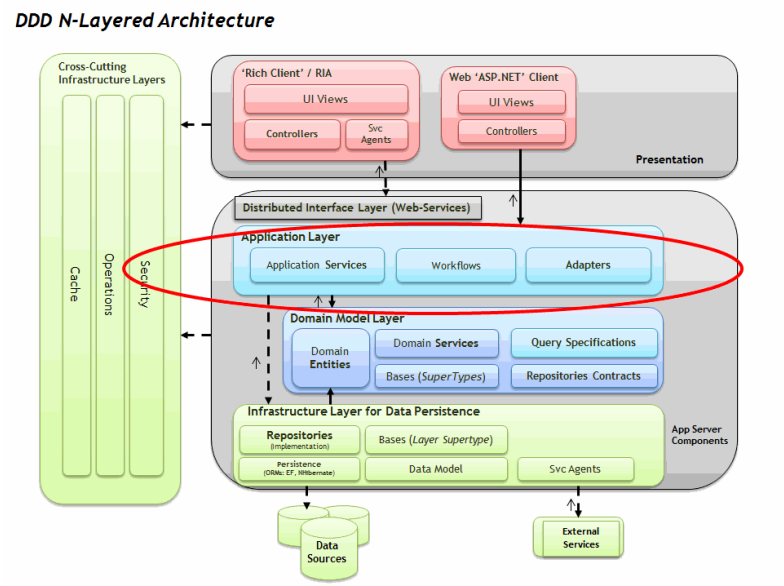



**Figure 1.- Location of the Application layer in DDD N-Layered Architecture**

The Application Layer defines the tasks that the software is supposed to perform. These are usually related to making calls to the Domain and Infrastructure Layers. Again, these are the application tasks, and not the Domain tasks. For example, coordination of calls to Repositories to persist data in databases, data conversion, offering highly granulated interfaces in order to improve performance in communications, implementation of DTO Adapters to perform data conversions, etc. are the tasks that should be coordinated in this layer.

www.EBooksWorld.ir

The elements to be included in the Application Layer might be:

- **Application Services** (the most common element in this layer.)
- **Workflows** (for long executions processes.)
- **Adapters/Converters** (e.g., Converters from DTO to Domain entities and vice versa)

**Table 1.- Framework Architecture Guide**

**Rule # D17.**

**An Application Layer will be designed and implemented for coordination of tasks related to coordinating technical Application requirements.**

**Rules**

- The Application logic should not include any Domain logic, but only coordination of the tasks related to technical requirements of the application, such as coordination of calls to Repositories in order to persist data, Domain Entities data format conversion, and ultimately, calls to the Infrastructure components so that they perform complimentary tasks to the application.
- It should not have states reflecting the business process status; however, it may have states that reflect the progress of the application task.

**Advantages of using the Application Layer**

- We satisfy the "*Separation of Concerns*" principle, that is, we isolate the Domain layer from tasks/requirements of the application itself also known as "plumbing tasks". These are actually not business logic, but rather technological integration aspects, data formats, performance optimization, data persistence coordination, etc.

**References**

*'Application' Layer. By* ***Eric Evans*** *in his book DDD.*

www.EBooksWorld.ir

## 2.1.- Application Layer Design Process

When designing the server component layers, we should consider the requirements of the application design. In this section we briefly explain the main activities associated with the design of components usually located in the Application Layer. When designing the business layers, the following key activities should be performed on each area:

- **Creating a general application layer design:**
  - Identify consumers of the application layer.
  - Determine the security requirements of the application layer.
  - Determine requirements and data validation strategies in the application layer.
  - Determine the Cache strategy.
  - Determine the exception management system of the application layer.
- **Design of application logic components (Coordination):**
  - Identify domain components that will be coordinated from the application layer.
  - Make decisions on localization, coupling and interactions of the business components.
  - Choose an adequate transactional support.
  - Identify how to implement the coordination of business rules.
    - Directly in the code, using Application services.
    - Workflows (scenarios with long executions).
  - Identify Application layer design patterns that meet the requirements.

www.EBooksWorld.ir

## 2.2.- The importance of decoupling the Application Layer from the Infrastructure

In the Application layer design, and generally in the design of the rest of the internal layers, we should ensure that we implement a decoupling mechanism between the objects of these layers. This will make it easier to support a high level of business rules in business application scenarios with a large amount of domain logic components (business rules) and a high level of data source access. This is usually accomplished by **decoupling component layers through contracts or interfaces, and even beyond that, by using DI (*Dependency Injection*) and IoC (*Inversion of Control*). These techniques provide excellent maintainability.**

For more information on core concepts of Inversion of Control and Dependency Injection, see the initial chapter on global N-Layer Architecture.

# 3.- APPLICATION LAYER COMPONENTS

The Application layer may include different types of components. However, the main component is the Application SERVICE, as discussed below.

## 3.1.- Application Services

The Application SERVICE is another type of Service, complying with the guidelines of its pattern, the "*SERVICE pattern".* Basically, they should be stateless objects that coordinate certain operations. In this case they would be operations and tasks related to the Application layer (Tasks required by the application, not by the Domain logic.).

Another function of the Application Services is to encapsulate and isolate the data persistence infrastructure layer. Therefore, in the application layer, we perform **coordination** of transactions and data persistence (only coordination or calls to Repositories), data validation and security issues such as authentication requirements and authorization for the execution of specific components, etc.

In addition, the SERVICES should be the only entry point in the architecture through which higher layers access the data persistence infrastructure classes (*Repositories*) when working with data. For example, it should not be possible to directly invoke Repository objects from the Presentation layer.

www.EBooksWorld.ir

As we can see in the following diagram, the interaction between the different objects of the layers will usually start in an Application Service, which will be the concentrator or hub of the different types of application actions.

*Application Services interacting with Domain Services and Infrastructure Repositories*

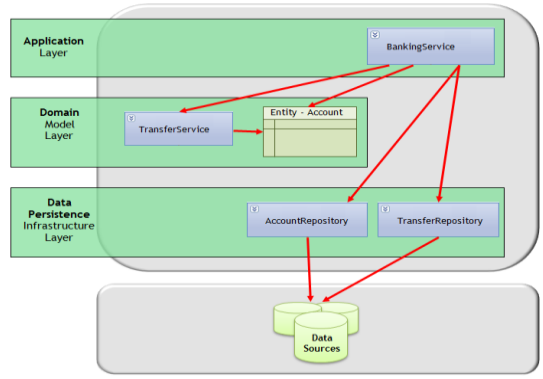



**Figure2.- Interaction Sample between the objects of different layers**

The fundamental aspect of this layer is not to mix Software requirements (persistence coordination, conversions to different data formats, optimizations, Service Quality, etc.) with the Domain layer, which should only contain pure Business logic.

The following UML sequence diagram shows an Application layer object (the Service that originates the bank transfer in the application), the Domain objects and how the Repository and UoW objects are invoked from the Application Service at the end.

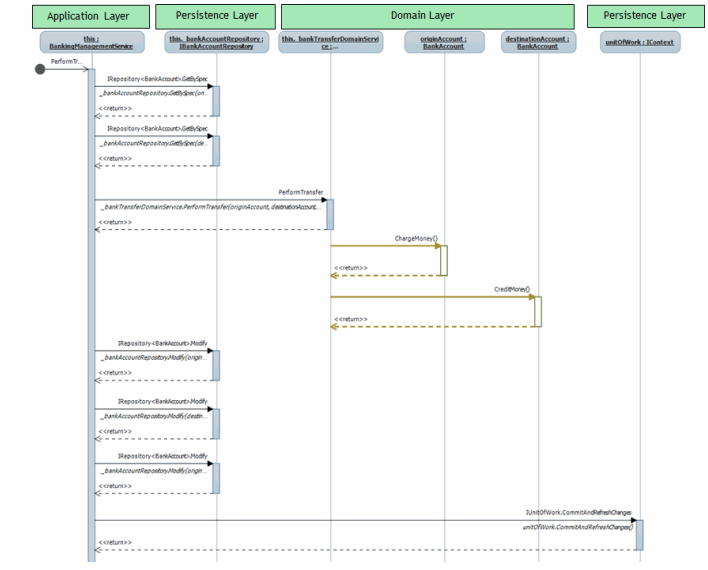



**Figure 3.- Sequence Diagram**

www.EBooksWorld.ir

In this interaction between objects, only the calls to *ChargeMoney()* and *CreditMoney()* methods are purely business/Domain calls. The rest of the interactions are necessary for the application (e.g., query the data of every account and data persistence through Repositories, use of UoW, etc.) and are therefore coordinated from the Application Layer.

**Table 2.- Recommendation Service Classes**

**Rule # D18.**

**SERVICE classes are mainly responsible for the intercommunication with the Data Persistence Infrastructure Layer Classes (Repository classes).**

**Recommendation**

- It is recommended for the application SERVICE classes to have the coordination and the communication (interlocutors or access paths) with the Repository classes (lower layer of the Infrastructure) as their main responsibility. For example, there should be no access to a Repository class directly from the Web Services or the Presentation layer. Usually, we do not instantiate Repositories from a domain class either, although there may be exceptions, like when we need to query some data for the domain variable states.

**Note1:**
Coordination of Repositories, UoW, transactions, etc. can be implemented from the Domain objects/services themselves. In fact, there are many implementations of N-Layer architectures, including those which are DDD, that do so. Placing the Repository coordination in one layer or another is simply for reasons of design preferences. However, by leaving these aspects in the Application layer (as we prefer to do), the Domain layer becomes much cleaner and more simplified, containing only the domain logic.

**Note2:**
Additionally, and although as a rule Repositories are only used from the Application layer, it is also possible to make exceptions, and make queries invoking Repositories from the Domain services as necessary. But this should be avoided as much as possible in order to achieve homogeneity in our developments.

www.EBooksWorld.ir

**Table 3.- Rule Application Layer Class**

**Rule # D19.**

**Do not implement persistence/data access code in the Application Services**

**Rule**

- Never implement any data access or persistence code (such as code directly using 'LINQ to Entities', 'LINQ to SQL', ADO.NET classes, etc.) or SQL statements code or stored procedures names, directly in a method of an Application layer class. For data access, you should only invoke classes and methods from the Infrastructure layer (i.e., Repository classes).

**References**

*"Separation of Concerns" principle*
**http://en.wikipedia.org/wiki/Separation_of_concerns**

**Table 4.- Implement The Layer Supertype Pattern**

**Rule # D20.**

**Implement the *Layer Supertype* pattern**

**Recommendation**

- It is common and very useful to have "base classes" for each layer to group and reuse common functionalities which otherwise would be duplicated in different parts of the system. This simple pattern is called *Layer SuperType*.
- However, it should only be implemented if necessary (YAGNI).

**References**

'*Layer Supertype' pattern*. By ***Martin Fowler***.
**http://martinfowler.com/eaaCatalog/layerSupertype.html**

www.EBooksWorld.ir

## 3.2.- Decoupling between APPLICATION SERVICES and REPOSITORIES

When decoupling all the objects with dependencies through **IoC and DI, the application layer should also be decoupled from the lower layers, such as Repositories (belonging to the Data persistence infrastructure layer). This would allow us, for example, to dynamically configure (or when compiling and testing) whether or not we actually want to access the real data repositories (usually relying on databases), a secondary system of repositories/storages or even "fake repositories" (stub or fake). Thus**, if we simply want to run a large number of unit tests right after making changes to the business logic, this can be done easily and quickly (without slowing down development) because we will not be accessing databases (tests are run only against the mock or stub repositories).

Additionally, we should be able to run integration tests. In this case we are testing Repositories execution against real data sources.

An application SERVICE class method usually invokes other objects (domain Services or data persistence Repositories), setting up rules or complete transactions. **As we saw in the example, an Application Service Method implementing a Bank transfer**, called *BankingManagementService.PerformTransfer()*, made a call to the Domain to carry out the business operation related to the transfer (internally within the Domain entities logic by *Credit()* and *Debit() methods*), and later on, following the Application Service execution, it called the Repository persistence methods so that the transfer was recorded/reflected in the persistent storage (most likely a database). **All of these calls between different objects of different layers (especially those regarding the infrastructure) should be decoupled calls made through interfaces or dependency injections.** The only case in which it is pointless to decouple through DI is when we use Domain entities, as there is no point in replacing entities to another version that complies with the same interface.

### 3.2.1.- Unit of work pattern

The UNIT OF WORK pattern (UoW) concept is closely linked to the use of REPOSITORIES. In short, a Repository does not directly access the storages (databases in most cases) when we tell it to perform an update (*update/insert/delete*). Instead, it only registers the operations it "wants" to perform 'in memory' (Context). Therefore, in order for them to be effectively performed on the storage or database, a higher level element needs to commit these changes against the storage. This element or higher level concept is the UNIT OF WORK.

The UNIT OF WORK pattern fits perfectly with transactions, because we can place a UNIT OF WORK in a transaction, so that, just before "committing" the transaction,

www.EBooksWorld.ir

the UoW is applied with the different operations grouped together all at once, optimizing performance and minimizing blockage in the database. On the other hand, if we place complete data access operations (traditional DAL) within a transaction, the transaction will take longer to complete, and the objects applying transaction operations will be mixed in with domain logic. Hence, it will take more time to perform the transaction and there will be a subsequent increase in blockage time.

The UNIT OF WORK pattern was defined by Martin Fowler (Fowler, Patterns of Enterprise Application Architecture, page 184). According to Fowler, "*A UNIT OF WORK maintains a list of objects affected by a business transaction and coordinates the updating of changes and the resolution of concurrency problems.*"

The operational design of a UNIT OF WORK may be performed in different ways. However, the most suitable (as stated above) is probably for the Repositories to delegate the job of accessing the data storage to the UNIT OF WORK (UoW). That is, the UoW effectively makes the calls to the storage (in the case of databases, communicating to the database server to execute the SQL statements). The main advantage of this approach is that messages sent by UoW are clear for the repository consumer; the repositories only tell the UoW operations what should be done when the repository consumer decides to apply the unit of work.

The following scheme shows the operation of traditional data access classes (DAL), **without using any UoW**:

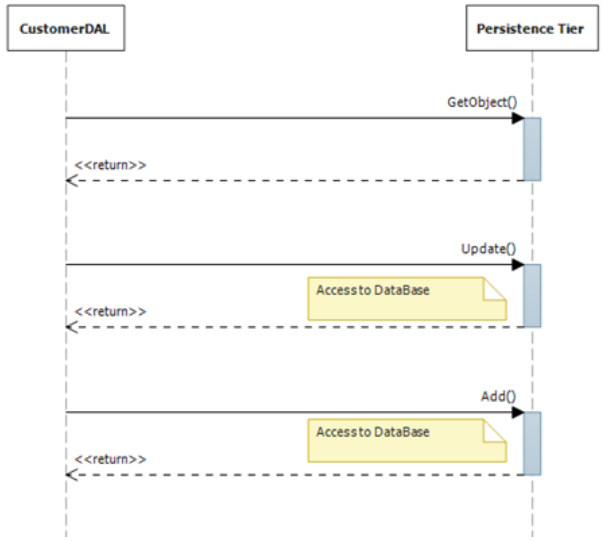



**Figure 4.- Scheme of data access classes (DAL)**

www.EBooksWorld.ir

The following scheme shows the operation of a REPOSITORY class along with a UoW, which is what we recommend in this Architecture guide:

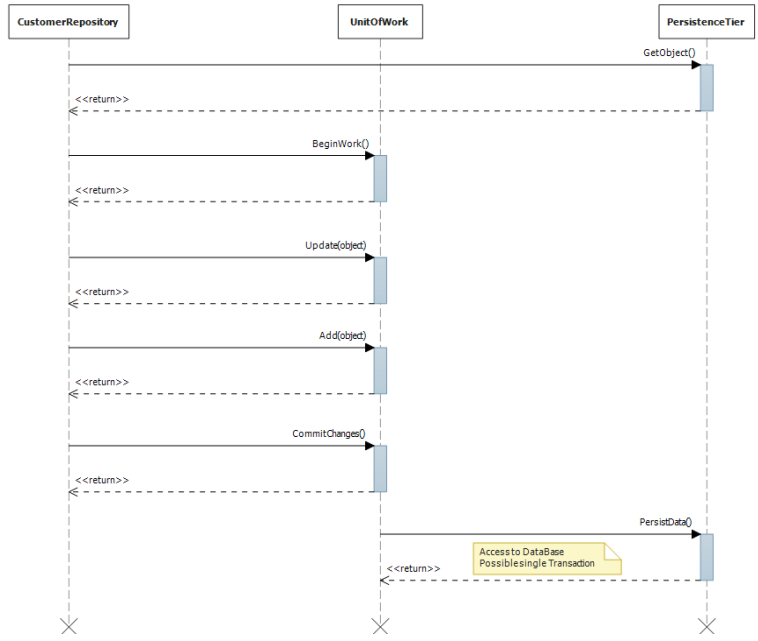



**Figure 5.- Operation of UoW and REPOSITORY classes**

## 3.2.2.- Application Layer Workflow Services (optional)

Actually, this sub-layer is a special case of Application SERVICES that provides a solution to certain casuistry in different software solutions. Long running processes or processes where there is interaction both with humans and other software systems are clear examples for the use of workflows. Modeling a process with human interaction directly in the code will often obscure its true purpose and in many cases hinder the possibility of understanding it, thereby reducing its readability. On the other hand, the workflow layer allows for modeling different interactions through activities and a control designer gives a clear visual idea of the purpose of the process to be carried out.

www.EBooksWorld.ir

**Table 5.- Designing and implementing a workflow**

**Rule # D21.** **Designing and implementing a Workflow Service Sub-layer in the Application Layer**

○ Recommendations

- This layer is optional. In fact, it is not commonly found in applications that focus to a great degree on data without business processes with human interactions.
  Try to encapsulate the processes in "Activities" in a workflow so they are reusable in other flows.
  Although the workflows may implement "business logic", they should always rely on domain services and repositories in order to perform the different tasks assigned to their activities.

**References**

*Workflow Patterns –*
**http://www.workflowpatterns.com/**

When we refer to workflows, we generally talk about the steps of these workflows, typically referred to as activities. When there is an implementation of this sub-layer, it is very important to make the best of its reusability and to pay attention to how they are implemented. The following shows the most important highlights:

- If the activities use persistence mechanisms, they should use the already defined repositories as much as possible.

- The activities may orchestrate different application sub-layers and domain service methods.

Most of the workflow engines that exist today have mechanisms to ensure their durability in long-running processes and/or in the case of a system outage. These systems are usually based on relational databases; therefore, they have different operations that could be incorporated in this sub-layer. Some examples of operations are as follows:

- Rehydration of workflows from the persistence system to the memory.

www.EBooksWorld.ir

- Download of workflows from memory to the persistence system.
- Checking the existence of certain workflows in the persistence system.
- Workflow instance correlation storage in the persistence system.

## 3.3.- Application Layer Errors and Anti-patterns

There are certain problematic items and common errors in many applications, which should be analyzed when designing the Application layer. This table lists these items grouped by categories.

**Table 6.- Application Layer Anti-patterns**

| Category | Common Mistakes |
|---|---|
| Authenti_ cation | • Applying the authentication of the application itself in the application layers when it is not required and when a global authentication outside the application could be used.<br>• Designing an authentication mechanism of its own.<br>• Failure to have a 'Single-Sign-on' when appropriate. |
| Authori_ zation | • Incorrect use of role granularity.<br>• Use of impersonation and delegation when not required.<br>• Mixing up authorization code with business process code. |
| Application components | • Mixing up data access logic (TSQL, LINQ, etc.) in the Application Services.<br>• Overload of business components when mixing non-related functionalities.<br>• Failure to consider the use of interfaces based on messages (Web-Services) when exposing the business components. |
| Cache | • Making a volatile data cache.<br>• Caching too much data in the application layers.<br>• Failure to get to cache data in a ready to use format.<br>• Caching sensitive/confidential information in a non-encrypted format. |

www.EBooksWorld.ir

| | |
|---|---|
| Coupling and cohesion | • Designing layers that are tightly-coupled with each other.<br>• There is no clear separation of responsibilities (*concerns*) between the different layers. |
| Concurrency and transactions | • A correct model of data concurrency was not chosen.<br>• Use of ACID transactions that are too long and causing too many blockages in the database. |
| Data access | • Access to the database directly from the business/application layers.<br>• Mixing-up data access logic with business logic in the business components. |
| Exception management | • Showing confidential information to the end user (such as connection strings when there are errors).<br>• Use of exceptions to control application flow.<br>• Failure to show the user error messages with useful information. |
| Instrumentalization and *Logging* | • Failure to adapt the instrumentation in business components.<br>• Failure to log critical business events or critical system events. |
| Validation | • Exclusively relying on validation made in the presentation layer.<br>• Failure to correctly validate length, range, format and type.<br>• Failure to reuse validation logic. |
| *Workflow* | • Failure to consider the application management requirements.<br>• Choosing an incorrect workflow pattern.<br>• Failure to consider how to manage all the state exceptions.<br>• Choosing incorrect workflow technology. |

www.EBooksWorld.ir

## 3.4.- Design aspects related to the Application Layer

The following items are specifically cross-cutting aspects of an Architecture and are explained in detail in the chapter on ***'Cross-Cutting/Horizontal Infrastructure Layers."*** However, it is important to show which of these aspects are related to the Application Layer.

### 3.4.1.- Authentication

Designing an effective authentication strategy for the Application Layer is essential when dealing with application security and reliability concerns. If this is not correctly designed and implemented, the application may be vulnerable to attacks. The following guidelines should be followed when defining the type of application authentication to be used:

- Do not perform authentication in the Application layer if you are only going to use it for a presentation layer or at a Distributed Services level (Web Services, etc.) within the same trusted boundary. In these cases (commonly in business applications), the best solution is to propagate the client's identity to the Application and Domain layers when it should be authorized based on the initial client's identity.

- If the Application and Domain layers are used in multiple applications with different user's storages, then the implementation of a *"single sign-on"* system should be considered. Avoid designing your own authentication mechanisms; it is preferable to use a generic platform.

- Consider using a "Claims based" security, especially for applications based on Web Services. This way, the benefits of federated identity can be used, and different types and technologies of authentication may be integrated.

This cross-cutting aspect (Authentication) is further explained in the chapter ***''Cross-Cutting/Horizontal Infrastructure Layers'.***

www.EBooksWorld.ir

## 3.4.2.- Authorization

Designing an effective authorization strategy for the Application Layer is essential when dealing with security and reliability of the application. If this is not correctly designed and implemented, the application can be vulnerable to attacks. The following guidelines should be considered when defining the type of application authorization to be used:

- Protect the resources of the Application and Domain Layer (services classes, etc.) by applying the authorization to consumers (clients) based on their identity, roles, claims of role type, or other contextual information. If roles are used, try to minimize the number of roles in order to reduce the number of combinations of required permissions.

- Consider use of authorization based on roles for business decisions, authorization based on resources for system audits, and authorization based on claims when the support of federated authorization is necessary and is based on a mix of information such as identity, role, permissions, rights and other factors.

- Avoid using impersonation and delegation insofar as possible because it may significantly affect performance and scalability. Generally, as regards performance, it is more expensive to impersonate a client in a call than to make the call itself.

- Do not mix the authorization code.

This cross-cutting aspect (Authorization) is further explained in the chapter ***'Cross-Cutting/Horizontal Infrastructure Layers'.***

www.EBooksWorld.ir

### 3.4.3.- Cache

Designing an effective cache strategy for the application is essential when dealing with the performance and scalability concerns of the application. Cache should be used to optimize master data queries, to avoid unnecessary remote calls through the network and to eliminate duplicated processes and queries. As a part of the strategy there should be a decision on when and how to upload data to the cache. This depends entirely on the nature of the Application and Domain, since it depends on each entity.

To avoid unnecessary client wait time, load the data in an asynchronous manner or use batch processes.

The following guidelines should be considered when defining the cache strategy of the application:

- Make a cache of static data that will be regularly reused in the different layers (in the end, they will be used/handled in the Domain and Presentation Layers), but avoid making a cache of very volatile data. Consider making a cache of data that cannot be quickly and efficiently retrieved from the database (sometimes normalized data in databases can be hard to obtain, we could therefore cache de-normalized data). At the same time, however, avoid making a cache containing very large volumes of data that may slow down the caching process. Make a cache of minimally required volume.

- Avoid cache of confidential data or design a protection mechanism of such data in the cache (such as encryption of such confidential data).

- Consider deployments in "Web Server Farms", which may affect standard caches in the memory space of the Services. If any server in the Web-Farm can handle queries of the same client (Balancing without affinity), the cache to be implemented should support data synchronization between different servers of the Web-*Farm*. Microsoft has appropriate technologies for this purpose (Distributed cache) as explained further on in this guide.

This cross-cutting aspect (Cache) is further explained in the chapter ***'Cross-cutting/Horizontal Infrastructure Layers'.***

### 3.4.4.- Exception Management

Designing an effective strategy of Exception Management for the application layer can be essential when dealing with stability and even security concerns of the application. If exceptions are not properly managed, the application may be vulnerable to attacks; it

www.EBooksWorld.ir

may disclose confidential information about the application, etc. Also, originating business exceptions and exception management itself are operations with a relatively expensive process cost, so we should consider the impact on performance in our design.

When designing an exception management system, the following guidelines should be followed:

- Catch only the exceptions that can actually be managed or if it is necessary to add information.

- Under no circumstance should the exception management system be used to control the flow of the application or business logic. The implementation of exceptions catching (Catch, etc.) has very low performance and in these cases (normal execution flow of the application) it would cause a negative impact on application performance.

- Design a proper strategy for exception management; for example, allow exceptions to flow up to the "boundary" layers (e.g., the last level of the component server) where they can (should) be persisted in a logging system and/or transformed as necessary before passing to the presentation layer. It is also best to include a context identifier so that the related exceptions may be associated throughout the different layers, making it easier to identify the origin/cause of the errors.

This cross-cutting aspect (Exception management) is further explained in the chapter ***''Cross-Cutting/Horizontal Infrastructure Layers'.***

## 3.4.5.- Logging, Audit and Instrumentalization

Designing an effective strategy of *Logging, Audit and Instrumentalization* for the Domain and Application layer is important for application security, stability and maintainability. If it is not properly designed and implemented, the application may be vulnerable to rejection actions when certain users deny their actions. The log/record files may be requested to test incorrect actions in legal procedures. The Audit is considered more accurate if the information log is generated at the exact moment of access to the resource and through the routine that accesses the resource.

The instrumentation may be implemented with events and performance counters as well as the subsequent use of monitoring tools to provide administrators with information on the state, performance and health of the application.

www.EBooksWorld.ir

Consider the following guidelines:

- Centralize the *logging*, audits and instrumentation in the Application and Domain layers.
- Make use of simple and reusable classes/libraries. For more advanced aspects (clear publication in different repositories and even in SNMP traps), we recommend using libraries such as '***Microsoft Patterns & Practices Enterprise Library***' or those of third parties, such as *Apache Logging Services* "*log4Net*" or Jarosław Kowalski's "*NLog*".
- Include instrumentation in the system and/or critical business events within the components of the Application Layer and Domain Layer.
- Do not store confidential information in the log files.
- Make sure the failures in the logging system do not affect the normal operation of the Application and Domain layers.

## 3.4.6.- Validations

Designing an effective strategy for validations in the Application and Domain layer is not only important for the stability of the application, but also for the use of the application by the end user. If it is not properly designed, it may lead to data inconsistencies and violations of the business rules, and finally, to a mediocre user experience due to the errors subsequently originated, which should have been detected earlier. In addition to this, if it is not correctly built, the application may also be vulnerable to security aspects like '*Cross-Site-Scripting*' attacks in web applications, SQL injections attacks, '*buffer overflow*', etc.

Consider the following guidelines:

- Validate all the input data and method parameters in the Application layer, even when data validation has been previously performed in the presentation layer. The data validation in the presentation layer is more closely related to user experience and the validation performed in the Application layer is more associated with aspects of application security.
- Centralize the validation strategy to enable tests and reuse.
- Assume that all the input data of the users may be "malicious". Validate data length, ranges, formats and types as well as other more advanced concepts of the business/domain rules.

www.EBooksWorld.ir

## 3.4.7.- Deployment Aspects of the Application Layer

When deploying the Application and Domain layer, consider performance aspects and security of the production environment. Consider the following guidelines:

- If you want to maximize performance, consider deploying the application and domain layer in the same physical level as the Web presentation level. It should only be taken to another physical level for security reasons and in some special cases of scalability.

## 3.4.8.- Concurrency and Transactions

When we design Concurrency and Transactions aspects, it is important to identify the proper model of concurrency and determine how to handle transactions. For concurrency you may choose between the optimistic and the pessimistic model.

**Optimistic Concurrency Model**

In this model, the blocks are not kept in the database (only the minimum required while updating, but not while the user is working or simply with the update window open), and therefore updates are required to check that the data have not been changed in the database since the original retrieval of the data to be modified. Generally, it is articulated based on *timestamps*.

**Pessimistic Concurrency Model**

Data to be updated are blocked in the database, and they cannot be updated by other operations until unblocked.

Consider the following guidelines related to concurrency and transactions:

- The transaction boundaries should be considered so that retries and compositions can be made.

- When a commit or rollback cannot be applied, or if long execution transactions are used, choose the option to implement compensatory methods to undo operations performed on data and leave it in the previous state in case an operation is about to fail. The reason for this is that you cannot keep the database blocked due to a long running transaction.

- Avoid maintaining blockage for long periods of time; for example, do not perform long running transactions that are '*Two Phase Commit*'.

www.EBooksWorld.ir

- Choose a suitable isolation level for the transaction. This level defines how and when changes will be available for other operations.

## 3.5.- Map of possible patterns to be implemented in the Application layer

In this table you can see the key patterns for the application layers, organized by categories. The use of these patterns should be considered when decisions are made for each category.

**Table 7.- Key patterns**

| Categories | Patterns |
|---|---|
| Application layer components | • Application Façade<br>• Chain of Responsibility<br>• Command |
| Concurrency and transactions | • Capture Transaction Details<br>• Coarse-Grained Lock<br>• Implicit Lock<br>• Optimistic Offline Lock<br>• Pessimistic Offline Lock<br>• Transaction Script |
| Workflows | • Data-driven workflow<br>• Human workflow<br>• Sequential workflow<br>• State-driven workflow |

www.EBooksWorld.ir

**Patterns References**

*Information on 'Command', 'Chain of Responsibility' and 'Façade' patternsor "data & object factory"* at **http://www.dofactory.com/Patterns/Patterns.aspx**

*Information on "Entity Translator" pattern*:
**http://msdn.microsoft.com/en-us/library/cc304800.aspx**

*"Capture Transaction Details pattern", see "Data Patterns" in:*
**http://msdn.microsoft.com/en-us/library/ms998446.aspx**

# 4.- IMPLEMENTING THE APPLICATION LAYER USING .NET

The explanation and logical definition of this layer is provided in the previous section, so in this section we will focus on showing the possibilities of the Application Layer implementation in .NET 4.0.

In the following diagram we emphasize the location of the Application Layer with a *Layer Diagram* implemented with Visual Studio 2010, with a real map of each layer to the different *namespaces* that implement them:

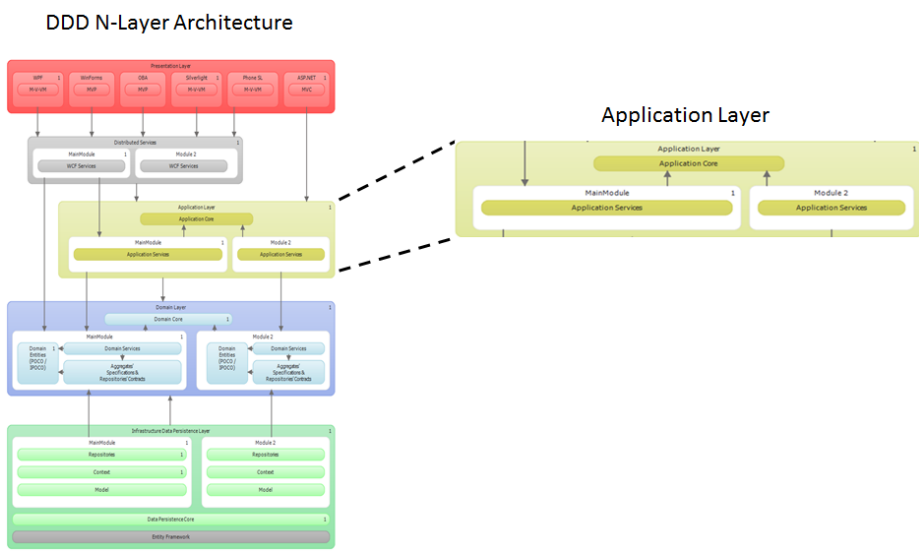



**Figure 6.- Application Layer Situation diagram**

www.EBooksWorld.ir

Steps to be followed:

1.- After identifying the application characteristics and the software requirements (not the Domain requirements), we should create a structure of this layer, that is, the project or projects in Visual Studio that will host the .NET classes implementing the Application SERVICES.

2.- We will add and implement .NET classes of Application SERVICES as necessary. It is important to remember that we should also continue working with abstractions (interfaces) in this layer. Therefore, for each class of SERVICE implementation we should also have an interface declaring all its operations (operations contract). This interface will be used from the higher layer (Web services or Presentation in ASP.NET) through the Unity container. When resolving dependencies, the UNITY container will resolve every object dependency related to the Service interface that we order. The process is similar to the one followed in Domain SERVICE implementation.

3.- SERVICES of the application layer may be implemented with WORKFLOW technologies, and not only through .NET classes.

## 4.1.- Implementation of Application Layer Services

In general and with a few exceptions, the APPLICATION SERVICES should be the only item or type of component of the architecture through which there is access to the data persistence infrastructure classes (*Repositories*). There should not be direct access to the Repositories through the presentation layers or Web services. Otherwise, we would be bypassing the application logic, as well as the business/domain logic.

The following figure shows an application Service and related Repository classes (Repositories form part of the Data Persistence Infrastructure layer) of a sample application module:

www.EBooksWorld.ir

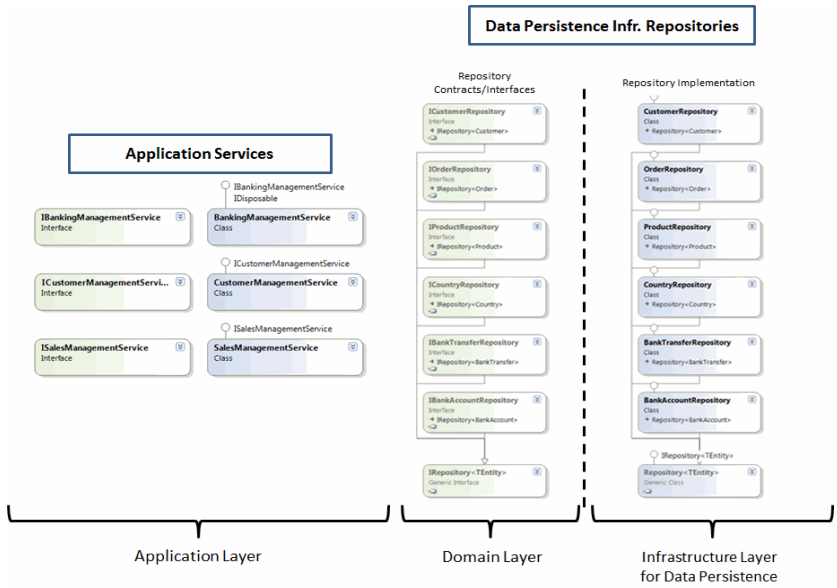



**Figure 7.- Repositories and Application Service Classes Diagram**

In addition to the Service Application and Repository classes, we also have the Domain Services. However, these are not shown in the diagram above because the relationship with the Repositories (creation and use of Repositories) will generally be done from the Application service Layer.

Below is an example of the implementation of an Application SERVICE class to control everything associated with the **Customer** entity:

```
C#
```

Interface for abstraction and instantiation through IoC container (Unity), from higher layers (e.g. Web-Services)

```
public class CustomerManagementService : ICustomerManagementService
{
  ICustomerRepository _CustomerRepository;
```

Constructor of required Dependency (Repository) to be inferred and instantiated by the IoC container (Unity).

```
public CustomerManagementService(ICustomerRepository
customerRepository)
{
  _CustomerRepository = customerRepository;
}
```

Application logic for the 'Customer' entity.

```
public List<Customer>FindPagedCustomers(int pageIndex, int pageCount)
{
```

www.EBooksWorld.ir

Validations and raising Business Exceptions

```
    if (pageIndex < 0)
                    throw new   ArgumentException(
                          Resources.Messages.exception_InvalidPageIndex,
                          "pageIndex");

    if (pageCount <= 0)
                    throw new   ArgumentException(
                          Resources.Messages.exception_InvalidPageCount,
                          "pageCount");


    Specification<Customer> onlyEnabledSpec = new
    DirectSpecification<Customer>();
```

Access to Data Sources through Repositories.

```
    return _customerRepository.GetPagedElements(
                                      pageIndex,
                                      pageCount, c => c.CustomerCode,
                                      onlyEnabledSpec,
                                      true)
                                    .ToList();
}

// Other CustomerManagementService methods to be implemented afterwards
(With UoW and Specifications patterns)
// ...
}
```

The code above is quite straightforward, except for one item: **Where is the Repository object of type 'ICustomerRepository' contract being instantiated and created?**

This is precisely related to the Dependency Injection and decoupling between objects through the Unity IoC container which is discussed below.

## 4.1.1.- Decoupling and Dependency Injection between Application Services and Repositories through UNITY IoC

The following scheme shows where Dependency Injection is being implemented with UNITY, between the "Application Service" classes and the "Data Access and Persistence Infrastructure" layer Repositories:

www.EBooksWorld.ir

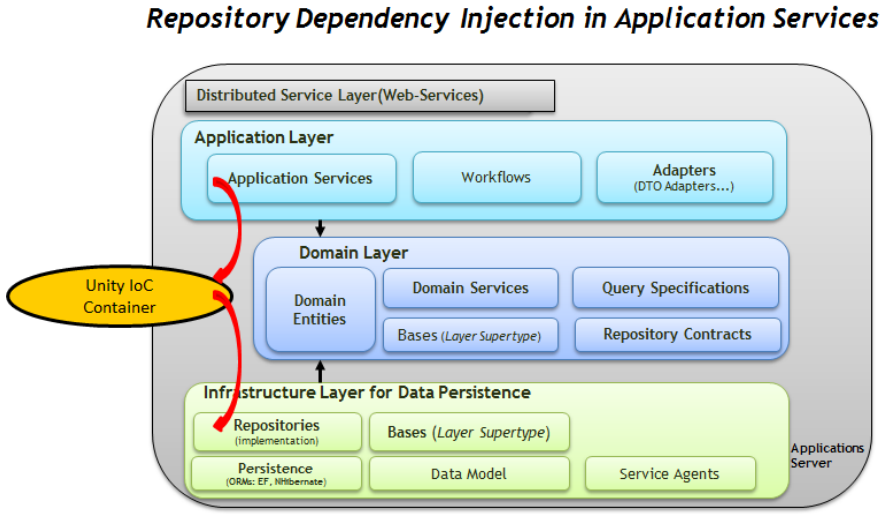



**Figure 8.- Domain Service Scheme**

Below, we can see how this decoupled integration can be performed between both layers (domain components and Repositories). If you are not familiar with Unity, you should first read the chapter on "Implementation of Dependency Injection and IoC with UNITY", which is part of this Architecture and implementation guide.

### Interface and Class Registration in the Unity Container

Before instantiating any class through the Unity container, we need to logically "register" the types in the Unity IoC container, both the interfaces and the classes**. This registration process may be done through code (e.g. C#) or as a statement through the Unity configuration XML.**

In the case of registering class types and mappings using XML, you may choose to mix the Unity configuration XML with the XML of web.config or App.config of the process that hosts your application/service. However, a better (and cleaner) approach would be to have a specific XML file for Unity linked to the app.config/web.config configuration file. In the sample implementation we used a specific configuration file for Unity, called Unity.config.

This would be the XML link from the web/app.config to the Unity configuration file:

```
Web.config (From a WCF Service or an ASP.NET app, etc.)
…
<!-- Unity configuration for solving dependencies-->
<unity configSource="Unity.config"/>
…
```

www.EBooksWorld.ir

This is the XML configuration to register the interface and Repository class:

```
    Web.config (WCF Service config, etc.)
    …
    <!-- Unity configuration for solving dependencies-->
    <unity configSource="Unity.config"/>
    …
    XML - Unity.config
    <?xml version="1.0" encoding="utf-8" ?>
    <unity>
    <typeAliases>
       …
       …

    <typeAlias alias="ICustomerRepository"

type="Microsoft.Samples.NLayerApp.Domain.MainModule.Contracts.ICustomerR
epository,
            Microsoft.Samples.NLayerApp.Domain.MainModule" />


    <typeAlias alias="CustomerRepository"

type="Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule.Reposit
ories.CustomerRepository,
            Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule"
/>

    …
    …
```

Repository Contract/Interface registration

Repository class registration

Below is where the interesting part comes in, that is, the mapping that we can specify to the container between the contracts/interfaces and the class to be instantiated by the Unity container. In other words, a mapping that states: "*When I order an object for ICustomerRepository, instantiate and give me an object of the CustomerRepository class*". The interesting part is that at another moment, if we want to run unit tests against a fake implementation (a stub/mock), it could specify something similar to the following statement: "*When I order an object for ICustomerRepository, instantiate an object of the CustomerFakeRepository class*".

www.EBooksWorld.ir

So the XML statement in the Unity.config file where we specified this mapping for our sample Repository is the following:

**XML – Unity.config**

```
<?xml version="1.0" encoding="utf-8" ?>
<unity>
<typeAliases>
   …
   …
</typeAliases>
  …
  …

<!-- UNITY CONTAINERS -->
<containers>
<container name="RootContainer">
<types>
        …
<type type="ICustomerRepository" mapTo="CustomerRepository">
</type>
        …
</types>
</container>

…
```

**Container**: We can have a containers hierarchy, created by program. Here we simply define the mapping of each container.

Interface mapping to the class that will be instantiated by the Unity

This registry of types and mappings from interfaces to classes may also be done through .NET code (C#, VB, etc.), which is probably the most suitable way when we are in the middle of the project development. In the sample application it has been done with C# code in the IoC factory class, with a code similar to the following snippet:

```
//Register Repositories mappings
            container.RegisterType<IProductRepository,
ProductRepository>(new TransientLifetimeManager());
            container.RegisterType<IOrderRepository,
OrderRepository>(new TransientLifetimeManager());
            container.RegisterType<IBankAccountRepository,
BankAccountRepository>(new TransientLifetimeManager());
            container.RegisterType<ICustomerRepository,
CustomerRepository>(new TransientLifetimeManager());
            container.RegisterType<ICountryRepository,
CountryRepository>(new TransientLifetimeManager());

            //Register application services mappings

            container.RegisterType<ISalesManagementService,
SalesManagementService>(new TransientLifetimeManager());
            container.RegisterType<ICustomerManagementService,
CustomerManagementService>(new TransientLifetimeManager());
            container.RegisterType<IBankingManagementService,
BankingManagementService>(new TransientLifetimeManager());

            //Register domain services mappings
            container.RegisterType<IBankTransferDomainService,
BankTransferDomainService>(new TransientLifetimeManager());
```

www.EBooksWorld.ir

```
            //Register crosscuting mappings
            container.RegisterType<ITraceManager, TraceManager>(new
TransientLifetimeManager());
   …
   …
```

Once we have defined the mappings, we may implement the code where we really order the Unity container to instantiate an object for a given interface. We could do something similar to the following code (Please note that we usually do not explicitly invoke Resolve method to instantiate Repositories.).

```
   C#

IUnityContainer container = new UnityContainer();
ICustomerRepository customerRep = container.Resolve<ICustomerRepository
>();
```

**Keep in mind that if we want to apply the DI (Dependency Injection) correctly, we will usually invoke the Resolve() method only from the highest level classes of our application server, that is, from the incoming or initial items that are usually Web services (WCF) and/or ASP.NET presentation layer. We should not do an explicit Resolve() against Repositories, because we would be using the container almost exclusively as a type selector. It would not be correct from a DI point of view**.

In short, as we should have a chain of built-in layers decoupled from each other through Unity, the best option is to let Unity detect our dependencies through each class constructor. That is, **if our Application Service class has a dependency to a Repository class (it needs to use a Repository object), we simply specify it in our constructor and the Unity container will create an object of this dependency (a Repository object), and will provide it as a parameter of our constructor**.

For example, a SERVICE class called 'CustomerManagementService', will be like this:

```
C#

public class CustomerManagementService : ICustomerManagementService
{
ICustomerRepository _CustomerRepository;
```

Constructor with required Dependency (Repository) to be inferred and instantiated by the IoC (Unity) container.

```
public CustomerManagementService(ICustomerRepository customerRepository)
{

 CustomerRepository = customerRepository;
}
   …  }
```

www.EBooksWorld.ir

It is important to point out that, as shown, we have not made any explicit "new" CustomerRepository object. The Unity container is the one that automatically creates the CustomerRepository object and provides it as a parameter to our constructor. **This is precisely the dependency injection in the constructor.**

Then, within the constructor, we store the dependency (Repository in this case) in a field inside the object so that we can use it from different methods of our Application Layer Service class.

So our Application Service class called **CustomerManagementService** would be something like this:

**C#**

Interface for abstraction and instantiation through IoC container (Unity)

```
public class CustomerManagementService: ICustomerManagementService
{

   ICustomerRepository _CustomerRepository;
```

Constructor with required Dependency (Repository) to be inferred and instantiated by the IoC container (Unity).

```
   public CustomerManagementService(ICustomerRepository
   customerRepository)
   {
       _CustomerRepository = customerRepository;
   }
```

Business/Domain logic for the Customer entity

```
   public List<Customer> FindPagedCustomers(int pageIndex, int
   pageCount)
   {
      if (pageIndex < 0)
                  throw new
ArgumentException(Resources.Messages.exception_InvalidPageIndex,
"pageIndex");
```

Validations and generations of Business exceptions

```
      if (pageCount <= 0)
                  throw new
ArgumentException(Resources.Messages.exception_InvalidPageCount,
"pageCount");


      return _CustomerRepository.GetPagedElements(pageIndex, pageCount,
c => c.ContactTitle, true).ToList();
   }
```

Data Sources Access through Repositories.

```
   public Customer FindCustomerByCode(string customerCode)
   {
      //Create specification
      CustomerCodeSpecification spec = new
   CustomerCodeSpecification(customerCode);
```

Use of the SPECIFICATION

```
      return _CustomerRepository.FindCustomer(spec);
   }

   public void ChangeCustomer(Customer customer)
```

www.EBooksWorld.ir

```
    {
       //Begin unit of work
       IUnitOfWork unitOfWork = _CustomerRepository.StoreContext as
    IUnitOfWork;
       _CustomerRepository.Modify(customer);
       //Complete changes in this unit of work
       unitOfWork.Commit(CommitOption.ClientWins);
    }
}
```

Use of UoW (UNIT OF WORK) pattern

Finally, the following code snippet shows how a graph of objects would be instantiated using dependency injection based on constructors.

This consumer code is not part of the Application Layer. Typically, this kind of code would be implemented in a WCF Services Layer or in an ASP.NET web presentation layer, which would be executed inside the same application server.

```
C# (In WCF service layer or in ASP.NET application)
…
{
    IUnityContainer container = new UnityContainer;
    ICustomerService custService =
container.Resolve<ICustomerManagementService>();
custService.AddCustomer(customer);
}
```

However, in the sample application we use a static utility class for Unity (IoCFactory), which is a cleaner and more extensible approach:

```
C# (In WCF service layer or in ASP.NET application)
   …
   {
   ICustomerManagementService custService =
ServiceFactory.Current.Resolve<ICustomerManagementService>();
       custService.AddCustomer(customer);
   }
```

The following shows the diagram for Repository and Application Service classes, related to the "Customer" Domain Entity:

www.EBooksWorld.ir

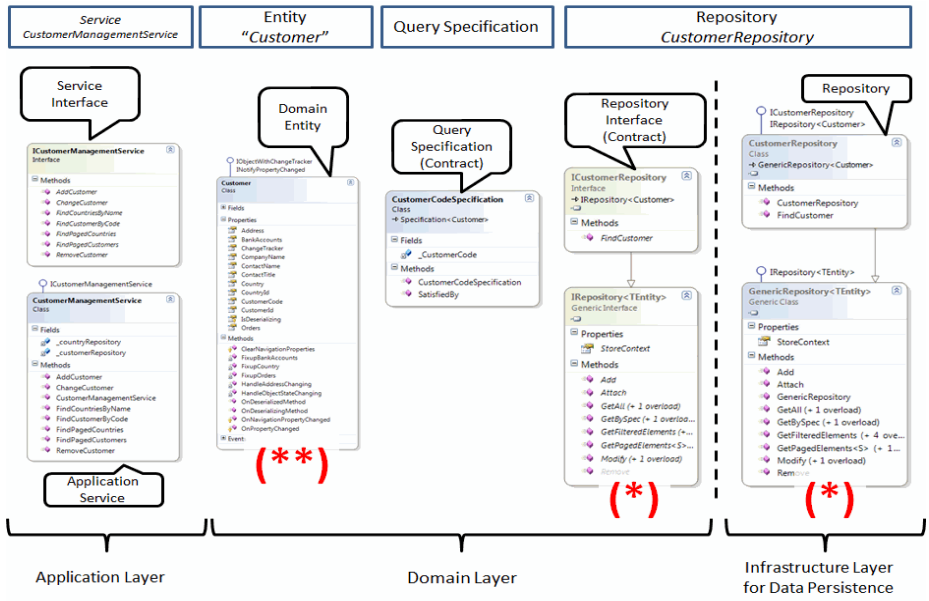



**Figure 9.- Repository and App Service Classes Diagram**

Although it may seem that there are many classes and interfaces associated with a single Domain entity, they are necessary if there is a need for decoupling; in fact, it does not require much work to be implemented, because:

- Of all these classes, the ones marked with one asterisk (*) in the lower part are base classes, so they are only implemented once for the entire project.
- The "Customer" Domain entity class marked with two asterisks (**) is generated by the T4 of Entity Framework, so it does not require any work.
- Interfaces are only method declarations or signatures, just like a contract, very quick to create and modify.

Therefore, we only need to implement the "*CustomerManagementService*" service class itself, together with the Application layer logic that we require, and also the "CustomerRepository" repository with persistence logic and data access if we cannot reuse the one already contained in the repository base class.

www.EBooksWorld.ir

## 4.2.- Implementing Transactions and Using UoW in Application Layer Services

Before showing the internal implementation of the sample Service, we will first show the different options for transaction implementation in .NET. Then we will implement it in the code of the sample Service "BankTransferService" because this sample implementation is closely associated with the implementation of transactions in .NET.

### 4.2.1.- Transactions in .NET

A transaction is an exchange of information and associated sequential actions treated as an atomic unit in order to satisfy a request, and simultaneously ensuring, specific data integrity. A transaction is only deemed complete if all the transaction information and actions have been completed and all the associated changes to the database are permanently applied. The transactions support the "undo" action (*rollback*) when there is a mistake, which helps to preserve the data integrity in the databases.

Historically, there have been many possible ways of implementing transactions in.NET. Basically, the following options are available:

- Transactions in TSQL (In their own SQL statement).
- ADO.NET transactions (Based on the Connection and Transaction objects)
- Enterprise Services transactions (Distributed transactions based on COM+)
- **Transactions System.Transactions (local and those upgradable to distribute).**

The first type (transactions in SQL statements and/or stored procedures) is feasible for any programming language and platform (.NET, VB, Java, etc.) and is the one that may achieve the best performance. Therefore, for special and specific cases, it may be the most suitable approach. However, using it in an N-layer architecture business application is not recommended because it has the huge disadvantage of having the transaction concept completely coupled (a business concept such as a transfer) with the data access code (SQL statements). Remember that one of the basic rules of an N-layer application is that the application code and domain/business should be completely separated and decoupled from the persistence and data access code. **Transactions should be exclusively declared/implemented in the Application layer (or Domain layer depending on the preference)**.

On the other hand, in .NET 1.x we basically had two main options, ADO.NET and COM+ transactions with *Enterprise Services*. If the ADO.NET transactions were used in an application, we should keep in mind that these transactions are closely linked to

www.EBooksWorld.ir

the *Transaction and Database Connection* objects, which are associated with the data access level. It is therefore very difficult to define transactions exclusively in the business component level (only through a Framework itself based on aspects, etc.) In short, we have a problem similar to using transactions in SQL statements. Now, however, instead of defining transactions in the SQL itself, we would be tightly coupled to the ADO.NET objects implementation. It is not the ideal context for transactions that should be defined exclusively at the business level.

Another option enabled by .NET Framework 1.x was to use *Enterprise Services* transactions (based on *COM+*), which can be exclusively specified at the business class level (through .NET attributes).However, in this case we have the problem wherein its use seriously impacts performance (*Enterprise Services* are based on *COM+* and therefore *COMInterop* is used from .Net, as well as an inter-process communication with *DTC*). The development also becomes more tedious because the components must be signed with a safe name (*strong-name*) and recorded as *COM* components in *COM+*.

However, as of **.NET 2.0** (also continued in .NET 3.0, 3.5 and 4.0) we have the '***System.Transactions***' *namespace*. This is generally the most recommended way of implementing transactions because of its flexibility and higher performance when dealing with Enterprise Services. This is especially true as of SQL Server 2005 where there is a **possibility of "automatic promotion from a local transaction to a distributed transaction".**

The below Table summarizes the different technological options to coordinate transactions in .NET:

**Table 8.- Technological options to coordinate transactions in .NET**

| Type of transaction | V. Framework .NET | Description |
|---|---|---|
| Internal transactions with T-SQL (in DB) | From .NET Framework 1.0, 1.1 | Transactions internally implemented in its own SQL statements (it can also be defined in stored procedures) |
| Enterprise Service transactions (COM+) | From .NET Framework 1.0, 1.1 | Enterprise Services (COM+)<br><br>Web ASP.NET transactions<br><br>XML Web Service (WebMethod) transactions |
| ADO.NET transactions | From .NET Framework 1.0, 1.1 | Implemented with ADO.NET Transaction and Connection objects. |
| System.Transactions transactions | .NET Framework 2.0, 3.0, 3.5 and 4.0 | Powerful system of local transactions that are upgradable to distributed transactions. |

www.EBooksWorld.ir

The next table shows the resources and objectives as well as assumptions and the transaction technology to be used:

**Table 9.- Resources and goal premises**

| What do I have? + Objectives | What to use |
|---|---|
| • A SQL Server 2005/2008/2008R2 for most transactions and there could also be distributed transactions with other DBMS and/or transactional environments 'Two Phase Commit'<br>• Objective: Maximum performance in local transactions | System.Transactions (From .NET 2.0) |
| • Only one older DBMS server (e.g. SQL Server 2000), for the same transactions<br>• Objective: Maximum flexibility in the business components design. | System.Transactions (From .NET 2.0) |
| • Only one older DBMS server (e.g. SQL Server 2000), for the same transactions<br>• Objective: Maximum performance in local transactions | ADO.NET transactions |
| • 'n' DBMS servers and Transactional Data Sources for Distributed Transactions.<br>• Objective: Maximum integration with other Transactional environments (HOST, MSMQ transactions, etc.) | System.Transactions (From .NET 2.0)<br>Enterprise Services (COM+) could be used too, but it is an older technology associated with COM+ and COM components. |
| • Any DBMS and execution of very critical specific transaction regarding its maximum performance.<br>• Objective: Maximum full performance, even when rules of design in N-layers are broken. | Internal transactions with Transact-SQL |

www.EBooksWorld.ir

Therefore, as a general rule and with a few exceptions, **the best option is System.Transactions**.

**Table 10.- Framework Architecture Guide**

**Rule # I8.**

**The transaction management system to be used by default in .NET will be 'System.Transactions'**

**Rule**

- The most powerful and flexible system for transaction implementation in .NET is **System.Transactions**. It offers aspects such as upgradable transactions and maximum flexibility by supporting local and distributed transactions.

- For most transactions of an N-layer application, the recommendation is to use the **implicit model** of **System.Transactions**, that is to say, using '**TransactionScope'.** Although this model is not at the same performance level as manual or explicit transactions, they are the easiest and clearest to develop, so they adapt very well to the Domain layers. If we do not want to use the Implicit Model (TransactionScope), we can implement the Manual Model by using the **Transaction** class of the **System.Transactions** *namespace*. Consider it for certain cases or those with heavier transactions.

**References**

*ACID Properties*
**http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconacidproperties.asp**

*System.Transactions*
**http://msdn.microsoft.com/en-us/library/system.transactions.aspx**

www.EBooksWorld.ir

## 4.2.2.- Transaction Implementation in the Domain Services Layer

The initiation and coordination of transactions following a correct design should generally be done in the SERVICE layer of the APPLICATION components. This is also feasible in the Domain layer, as preferred. However, in this guide, as we've explained, we propose to perform all the plumbing coordination, such as use of Repositories and UoW from the application layer in order to leave the Domain layer much cleaner with only business logic.

Any application design with business transactions should include transaction management in its implementation, so that a **sequence of operations can be performed as a single unit of work and be completely or unitarily applied or revoked if there is a mistake being made.**

**Any N-layer application should have the ability to establish transactions at the business or application component levels and not embed them within the data layer,** as shown in this scheme:

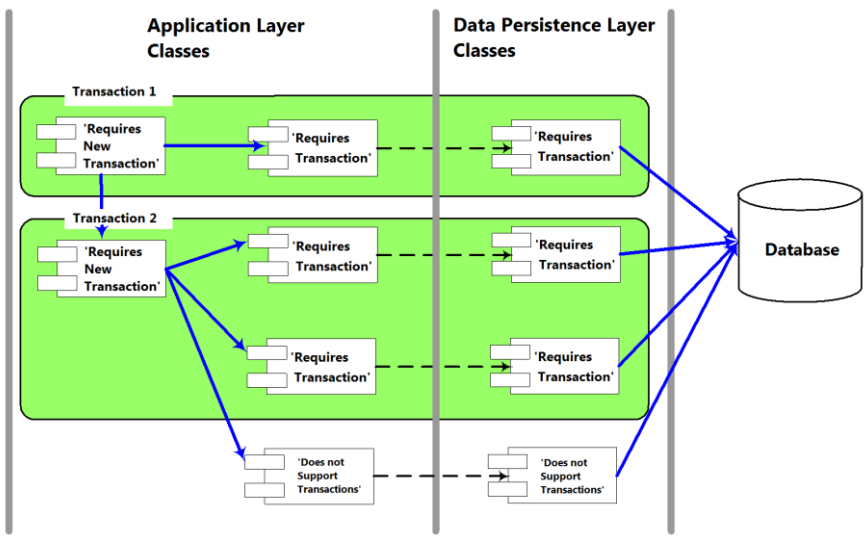



**Figure 10.- Transactions diagram - Classes level**

All native standard transactions (not compensatory transactions) must satisfy the **ACID principles**:

- **Atomicity:** A transaction should be an atomic unit of work, that is, everything is done or nothing is done.

www.EBooksWorld.ir

- **Consistency:** It should leave data in a consistent and coherent state once the transaction is over.
- **Isolation:** Modifications made by transactions are independently treated, as if there were only one user of the database.
- **Durability:** After the transaction is completed, its effects will be permanent and they will not be undone.

## 4.2.3.- Concurrency Model During Updates

It is important to identify the proper concurrency model and to determine how to manage the transactions. For concurrency, we can choose between an optimistic or pessimistic model. There are no blockages maintained in the data sources with the optimistic concurrency model; however, the updates require a certain checking code, generally against a '*timestamp*' in order to verify that the data to be modified have not changed in the source (DB) since the last time they were obtained. In the pessimistic concurrency model, data are blocked and they cannot be updated by any other operation until they are unblocked. The pessimistic model is quite typical for Client/Server applications where support for a great scalability of concurrent users is not required (e.g., thousands of concurrent users). On the other hand, **the optimistic concurrency model is much more scalable because it does not maintain such a high level of blockage in the database and is therefore the model to be chosen in general by most Web applications, N-Tier, and SOA**.

**Table 11.- Architecture Framework Guide**

**Rule # I9.**

**The concurrency model by default will be "Optimistic Concurrency".**

**Rule**

- The concurrency model in DDD N-layer applications with SOA, N-Tier or Web deployment will be the 'Optimistic Concurrency' model.
  At the implementation level, it is much easier to implement an Optimistic Concurrency exception management through Entity Framework Self Tracking.
  Of course, if there are important reasons for using the pessimistic concurrency model in specific cases, then it should be used but as an exception.

www.EBooksWorld.ir

**Advantages**

- Higher scalability and independence of data sources.
- Fewer blockages in database than the pessimistic model.
- For applications that require high scalability, such as Internet applications, it is mandatory to use this type of concurrency model.

**Disadvantages**

- Higher effort in managing exceptions while developing, if there is not additional help such as *Entity Framework* '*Self-Tracking Entities'*.
- In certain on-off operations where the concurrency control and the operation order are critical and we do not intend to depend on the end user's decisions when exceptions occur, the pessimistic concurrency model always offers a stronger and tighter concurrency control.
- If there is a high possibility of data conflicts due to concurrent users working, then consider using the pessimistic concurrency to avoid a high number of exceptions to be decided by the end users.

## 4.2.4.- Types of Transaction Isolation

Use a suitable isolation level for the transaction. There should be a balance between consistency and containment. That is, a high level of transaction isolation will offer a high level of data consistency, but it will have a higher level of blockages. On the other hand, a lower transaction isolation level will improve overall performance by lowering containment, but the level of consistency may be lower.

Therefore, when executing a transaction, it is important to know the different types of isolation options available in order to apply the most suitable for the operation to be performed. These are the most common ones:

- **Serialized**: data read by the current transaction will not be modified by other transactions until the current transaction is completed. No new data will be inserted during the execution of this transaction.
- **Repeatable Read**: data read by the current transaction will not be modified by other transactions until the current transaction is completed. New data could be inserted during the execution of this transaction.

www.EBooksWorld.ir

- **Read Committed**: a transaction will not read data being modified by another transaction if it is not reliable. This is the Microsoft SQL Server and Oracle's default isolation level.
- **Read Uncommitted**: a transaction will read any data, even though it is being modified by another transaction. This is the lowest possible level of isolation, although it allows higher data concurrency.

**Table 12.- Architecture Framework Guide**

**Rule # I10.**

**The level of isolation should be considered in each application and application area. The most common are 'Read-Committed' or 'Serialized'.**

**Recommendation**

In cases where the transaction has a critical level of importance, use of the '**Serialized'** level is recommended, although we should be aware that this level will decrease performance and increase the surface blockage in the database.
In any case, the transaction isolation level should be analyzed depending on the particular case of each application.

Consider the following guidelines when designing and implementing transactions:

- Consider what the boundaries of transactions are, and activate them only if necessary. In general, the queries will not require explicit transactions. It is also convenient to know the database transaction isolation level. By default, SQL Server runs each individual SQL statement as an individual transaction (*auto-commit transactional mode)*.

- Transactions should be as short in duration as possible to minimize the blockages time maintained in the database tables. Also, avoid blockages in shared data as much as possible because they may block access to another code. Avoid using exclusive blockage because it may cause inter-blocking.

- Avoid blockages in long running transactions. In cases where we have long-running processes but we would like them to behave as one transaction, compensatory methods should be implemented in order to return data to the initial state in case an operation fails.

www.EBooksWorld.ir

Below, there is a sample of a Domain SERVICE class method (**BankTransferService**) that initiates a transaction involving operations of associated Repositories, Domain Services and Domain Entities to persist changes in the operation:

**C#**

*Namespace* **of the Application Layer Services in a sample module**

```
…
namespace
Microsoft.Samples.NLayerApp.Application.MainModule.BankingManagement
{
```

**Domain Service**

**Contract/Interface to be met**

```
    public class BankingManagementService:IBankingManagementService
    {
        IBankTransferDomainService _bankTransferDomainService;
        IBankAccountRepository _bankAccountRepository;
```

**Constructor with Dependency Injection**

```
public BankingManagementService(IBankTransferDomainService
bankTransferDomainService, IBankAccountRepository bankAccountRepository)
        {
            _bankTransferDomainService = bankTransferDomainService;
            _bankAccountRepository = bankAccountRepository;
        }
```

**Method of App Service to perform a Transaction**

```
public void PerformTransfer(string fromAccountNumber, string
toAccountNumber, decimal amount)
        {
            //Process: 1° Start Transaction
            //         2° Get Accounts objects from Repositories
            //         3° Call PerformTransfer method in Domain Service
            //         4° If no exceptions, save changes using
repositories and Commit Transaction

            //Create a transaction context for this operation
            TransactionOptions txSettings = new TransactionOptions()
            {
                Timeout = TransactionManager.DefaultTimeout,
                IsolationLevel = IsolationLevel.Serializable
            };
```

**Type of transaction isolation**

**It requires a transaction**

```
using (TransactionScope scope = new
TransactionScope(TransactionScopeOption.Required, txSettings))
            {
```

**UoW (Unit of work) pattern for operations with Repositories**

**Query specification creation**

```
                //Get Unit of Work
                IUnitOfWork unitOfWork =
_bankAccountRepository.StoreContext as IUnitOfWork;


                //Create Queries' Specifications
```

www.EBooksWorld.ir

```
                BankAccountNumberSpecification originalAccountQuerySpec
= new BankAccountNumberSpecification(fromAccountNumber);
                BankAccountNumberSpecification
destinationAccountQuerySpec = new
BankAccountNumberSpecification(toAccountNumber);

                //Query Repositories to get accounts
                BankAccount originAccount =
_bankAccountRepository.GetBySpec(originalAccountQuerySpec as
ISpecification<BankAccount>).SingleOrDefault();

                BankAccount destinationAccount =
_bankAccountRepository.GetBySpec(destinationAccountQuerySpec as
ISpecification<BankAccount>).SingleOrDefault();

                ////Start tracking STE entities (Self Tracking Entities)
                originAccount.StartTrackingAll();
                destinationAccount.StartTrackingAll();

                //Excute Domain Logic for the Transfer (In Domain
Service)
_bankTransferDomainService.PerformTransfer(originAccount,
destinationAccount, amount);

                //Save changes and commit operations.
                //This opeation is problematic with concurrency.
                //"balance" propety in bankAccount is configured
                //to FIXED in "WHERE concurrency checked predicates"

                _bankAccountRepository.Modify(originAccount);
                 bankAccountRepository.Modify(destinationAccount);

                //Complete changes in this Unit of Work
                unitOfWork.CommitAndRefreshChanges();

                //Commit the transaction
                scope.Complete();
}
        }
}
}
```

**Retrieval of entities and data required for transfer**

**Call Domain Operations for transfer**

**Use of Repositories: 'Marked' for update**

**Commit of Unit of Work. The DB is being updated at this point**

**Commit of Transaction**

www.EBooksWorld.ir

Some considerations concerning the example above are found below:

- As shown, this Application Layer Service is where we implement all the "plumbing" coordination. In other words, the creation of a transaction and configuration of its type, use of 'Unit of Work', calls to Repositories to obtain entities and to finally persist them, etc. Ultimately, this includes all the necessary coordination of the application which is basically the aspects that we would not discuss with a business/domain expert. Instead, the entire Domain logic (BankTransfer operations) is encapsulated in the Domain Service and business logic of the entities themselves (in this case, the BankAccount entity and the BankTransfer Domain Service).

- Since **using** is being employed, it is not necessary to manually manage the transaction *rollback*. Any exception being thrown during insertion of any of the regions will cause the transaction to be aborted.

- The **UoW** (*Unit of work*) enables a context where the Repositories point out/record the persistence operations to be performed, but they are not actually made until we explicitly call 'unitOfWork.CommitAndRefreshChanges()'.

**Transactions Nesting**

*System.Transactions* clearly allows for nesting transactions. A common example is having another "TransactionScope" within an internal method (for example, in one of the methods of the "*BankAccount*" class, etc.). The original transaction will be extended with the new TransactionScope in one way or another, depending on the specified 'TransactionScopeOption' in the internal TransactionScope.

As shown, the advantage of this model resides in its flexibility and ease of development.

**Table 13.- Framework Architecture Guide**

**Rule # I11.** **The type of TransactionScope by default will be 'Required'.**

**Recommendation**

- If a transaction scope is not specified in the services at the lowest level, that is, the ones using REPOSITORIES, then operations will be listed to the highest transaction level that can be created. But if in these SERVICES we also implement *TransactionScope*, it should be configured as '*Required.*'

www.EBooksWorld.ir

This is because, in the case of calling the Service with our transaction from a code that has not created any transaction yet, then a new transaction will be created with the corresponding operations. However, if it is called from another class/service that has already created a transaction, this call will simply extend the current transaction. Then, as '*Required'* (TransactionScopeOption.Required) it will be correctly aligned to the existing transaction. On the other hand, if it appears as *"RequiredNew",* although there is an initial transaction in existence, a new transaction would be created by calling this transaction. Of course, all this depends on the specific business rules involved. In some cases, we might be interested in this other behavior.

This transaction configuration is implemented through the System.Transactions ***'TransactionScope()***' syntax.

**References**

*Introducing System.Transactions in the .NET Framework 2.0:*
**http://msdn2.microsoft.com/en-us/library/ms973865.aspx**
*Concurrency Control*
**http://msdn.microsoft.com/enus/library/ms978457.aspx**.
*Integration Patterns*
**http://msdn.microsoft.com/enus/library/ms978729.aspx**.

## 4.3.- Testing Implementation in the Application Layer

The application layer tests should normally perform *testing,* especially on the Application Services.

The application service tests are relatively complex because they involve dependencies to other elements such as the *IContext* used or other services (application or domain services) while, of course, invoking domain entity logic.

```
C#

[TestClass()]

[DeploymentItem("Microsoft.Samples.NLayerApp.Infrastructure.Data.MainMod
ule.Mock.dll")]

[DeploymentItem("Microsoft.Samples.NLayerApp.Infrastructure.Data.MainMod
ule.dll")]
   public class BankingManagementServiceTests
       {
```

www.EBooksWorld.ir

```
            [TestMethod()]
            public void PerformTransfer_Invoke_Test()
            {
                //Arrange

                IBankingManagementService bankTransfersService =
ServiceFactory.Current.Resolve<IBankingManagementService>();
                IBankingManagementService bankAccountService =
ServiceFactory.Current.Resolve<IBankingManagementService>();

                string bankAccountFrom = "BAC0000001";
                string bankAcccountTo = "BAC0000002";
                decimal amount = 10M;
                decimal actualBanlance = 0M;

                //Act

                //find actual balance in to account
                actualBanlance =
bankAccountService.FindBankAccountByNumber(bankAcccountTo).Balance;

                bankTransfersService.PerformTransfer(bankAccountFrom,
bankAcccountTo, amount);

                //Assert
                //check balance
                decimal balance =
bankAccountService.FindBankAccounts(bankAcccountTo,
null).SingleOrDefault().Balance;
                Assert.AreEqual(actualBanlance + amount, balance);
    }
```

It is evident that the configuration file of the dependency container may include the possibility, as in the persistence infrastructure layer, of incorporating a simulation of the *IContext* interface, that is, making the tests finally run against a real database or not, which highly affects the testing speed. Here we should remember **SlowTest,** a well-known anti-pattern in unit testing which is vitally important if we do not want the developers to omit tests due to their slowness.

In the specific case of our sample application NLayerApp, this change is configurable from the Web.config of the WCF service projects, so that tests are run against structures in memory instead of the database:

```
Web.config of WCF hosting project in sample application NLayerApp

<appSettings>
<!--RealAppContext - Real Container-->
<!--FakeAppContext - Fake Container-->
<!--<add key="defaultIoCContainer" value="FakeAppContext" />-->
<add key="defaultIoCContainer" value="RealAppContext" />
</appSettings>
```

Internally, a *mocking* of Entity Framework context is being made against a simulated environment of structures in memory. By not accessing the database, the unit

www.EBooksWorld.ir

testing will run faster; this is particularly noticeable when having to perform hundreds or even thousands of unit tests.

www.EBooksWorld.ir

CHAPTER 7

# The Distributed Services Layer

## 1.- LOCATION IN THE N-LAYERED ARCHITECTURE

This section describes the architecture area related to this layer, which is logically 'Service Oriented'. In many ways, SOA (*Service Oriented Architecture*) overlaps with 'Service Orientation', but they are not exactly the same concept.

**IMPORTANT**:
During this chapter, when we use the term 'Service' we are referring to Distributed-Services or Web-Services, by default. We are not referring to internal Domain/Application/Infrastructure Services (DDD patterns).

The following diagram shows how this layer (Distributed Services) typically fits into the '*Domain Oriented N-Layered Architecture'*:



www.EBooksWorld.ir

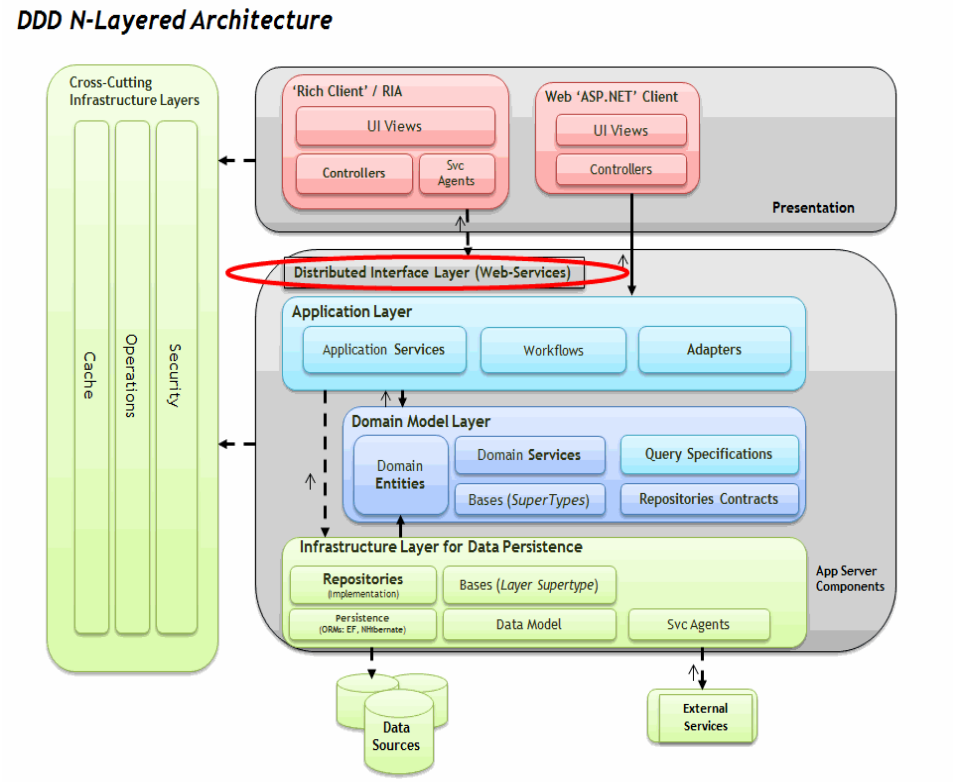



**Figure 1.- Location of the Distributed Services Layer**

The Service Layer typically includes the following topics:

- **Service Interfaces/Contracts**: Services expose interfaces that receive incoming messages. In short, services are like a façade layer for remote clients. Services expose the application and domain logic to the potential consumers such as Presentation Layer or other remote Services/Applications.

- **Message Types**: In order to exchange data via the Service Layer, data structures are serialized to messages. The service layer will also include data types and contracts that define the data types used in the messages.

**SOA**, however, covers a lot more than the design and implementation of an internal distributed Service Layer for only one N-layer application. The advantage of SOA is that it can share certain Services/Applications and provide access to them in a standard way. It is able to perform integrations in an interoperable manner which, in the past, has always been very expensive.

Before focusing on the design of a Service Layer within an N-layer application, we will provide an introduction to SOA.

www.EBooksWorld.ir

## 2.- SERVICE ORIENTED ARCHITECTURES AND N-LAYER ARCHITECTURES

It is worth noting that **SOA** trends do not contradict ***N-Layered architectures.*** On the contrary, they are complementary architectures. **SOA** is a high level architecture that defines "how" some applications intercommunicate (Services) with others. Simultaneously, each one of the **SOA services/applications** can be internally structured following the design patterns of the ***N-Layer architecture***.

SOA tries to define standard corporate communication buses between the different applications/services of a company, and even between services on the Internet owned by different companies.

The following diagram shows a *standard communication bus* example (following SOA trends) with several corporate applications integrated:

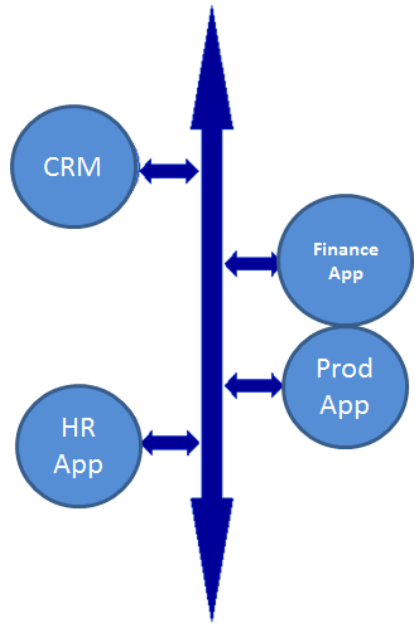



**Figure 2.- SOA and Enterprise Service Bus**

Each SOA Service/Application has to have an internal implementation where the application business logic, data access and entities (states) are implemented. Additionally, the Service input/output communication is based on messages (SOAP or REST messages).

www.EBooksWorld.ir

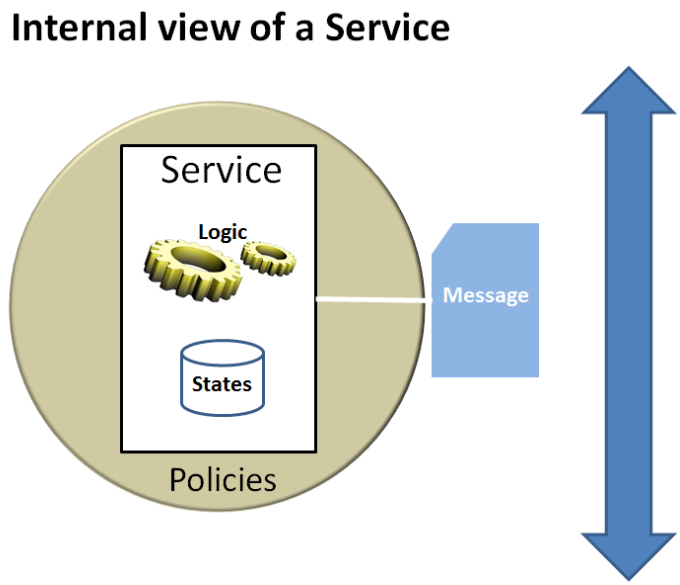



**Figure 3.- Internal View of a distributed service**

This internal implementation is normally carried out (structurally) following the design patterns of the logical N-layer architectures and physical distribution (deployment in servers) according to N-Tier architecture.

At a deeper level, the specific N-Layered Architecture for that SOA Service could be aligned with the layered architecture we propose in this guide, that is, a DDD NLayered Architecture, following DDD trends. This point will be explained later in more detail.

## 3.- N-LAYERED ARCHITECTURE RELATIONSHIP WITH ISOLATED APPLICATIONS AND SOA SERVICES

The internal architecture of an SOA service can therefore be similar to that of an isolated application, that is, implementing the internal architecture of both (SOA service and isolated Application) as an ***N-Layer architecture*** (component logical N-layer architecture design).

**The main difference between them** is that an SOA service is seen from "the outside" (from another external application) as something "not visual." By contrast, an isolated application will also have a **Presentation layer** (that is, the "client" part of the application to be used visually by the end-user).

www.EBooksWorld.ir

Keep in mind that **an "independent and visual" application may also be simultaneously a SOA service which could be publishing its components and business logic to other external applications**.

The order we will follow in this guide is: first, an explanation of the basis of SOA Architecture and second, an explanation of the implementation of Distributed Services with **WCF** (*Windows Communication Foundation)*.

## 4.- WHAT IS SOA?

SOA (*Service Oriented Architecture*) is an evolution of object oriented programming (OOP) and applies aspects learned over time in the development of distributed software.

The reasons for the appearance of SOA are basically the following:

- Integration between applications and platforms is difficult
- Certain systems are heterogeneous (different technologies)
- There are multiple integration solutions, which are independent and unrelated to each other.

A standard approach is necessary, which can provide the following:

- Service oriented architecture
- Based on a "common messaging bus"
- Standard for all platforms

'*Service orientation*' is different from '*Object orientation*', primarily in how it defines the term '*application*'. The "Object oriented development" is focused on applications whose construction is based on libraries of interdependent classes. SOA, however, emphasizes systems that are constructed on the basis of a set of autonomous services. This difference has a profound impact on the assumptions that can be made about development.

A "service" is simply a program used to interact via messages. A set of services installed/deployed would be a "system". Individual services should be constructed consistently (availability and stability are crucial to a service). An aggregated/composite system composed by various services should be constructed to allow change and evolution of these services and the system should be adapted to the presence of new services that appear over time after the services and original clients have been deployed/ installed. Furthermore, these changes should not break the functionality of the current system.

www.EBooksWorld.ir

Another aspect to note is that an SOA-Service should be, as a general rule, interoperable. Therefore, it should be based on standard specifications at the **protocol levels, serialized data format in communication levels, etc**.

Currently, there are two trends of Architecture regarding Web Services:

- SOAP (WS-I, WS-* specifications)
- REST (RESTful services)

**SOAP** is based on **SOAP messages,** logically. These messages are composed by **XML**, following a specific schema (format). SOAP uses **HTPP** as the communications protocol. ASMX Web-Services and WCF Services using WS Basic Profile or WS-* specifications are current Microsoft implementations for SOAP.

**REST** is highly oriented to the URI. The addressing of resources is based on the HTTP URL and therefore exchanging messages is simpler and lighter than with SOAP XML messages.

At the technological level, as we will explain in detail in the chapter on implementation, ***WCF* (*Windows Communication Foundation*)** also allows us to have other types of data formats and transport protocols that are not interoperable, compatible only with the .NET ends (such as ***NetTcpBinding, NetNamedPipeBinding*** or ***NetPeerTcpBinding***). These can be very useful as remote communication protocols **within the same application/service,** but they are not the most suitable for interoperable SOA-Services.

## 5.- INTERNAL ARCHITECTURE OF THE SOA SERVICES

SOA aims to solve problems of distributed application development. A "Service" can be described as an application that exposes an interface based on messages, encapsulates data and also manages ACID transactions (Atomic, Consistent, Isolated and Durable), with their respective data sources. Typically, SOA is defined as a set of service providers that expose their functionality through public interfaces (which can also be protected/ secured). The interfaces exposed by the service providers can be used individually or by adding several services and forming composite service providers.

The SOA services may also provide RPC style interfaces, if required. However, the "synchronized request-response" scenarios should be avoided whenever possible. On the contrary, the asynchronous consumption of Services should be favored.

Services are typically constructed internally by the following layers:

- **Service interface (Contract)**
- **Application and Domain layers**
- **Data access (Infrastructure and Data Access)**

www.EBooksWorld.ir

In the following diagram we show how the above sample service would be internally structured:

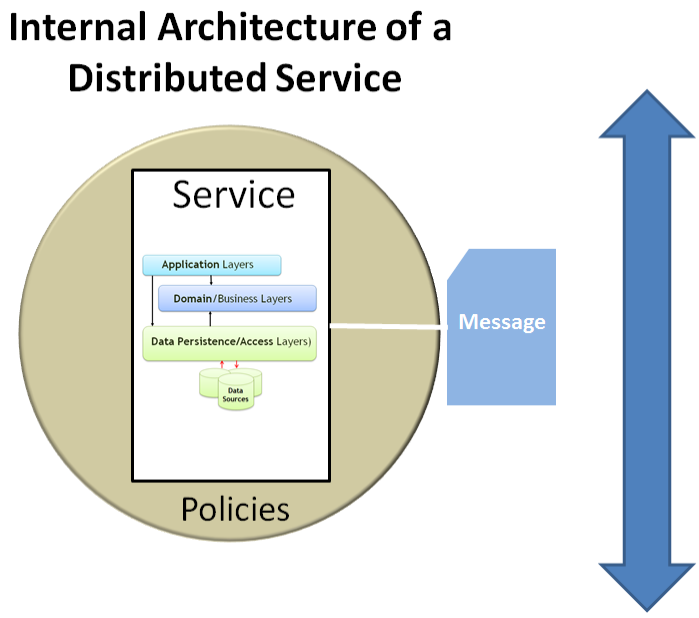



**Figure 4.- Logical layers of a Service**

It is very similar when compared to the internal architecture of an N-layer application. The difference is that, logically, a service does not have a presentation layer.

The 'Interface' is logically placed between the service clients and the facade of service processes. One single service can have several interfaces, such as a *Web-Service* based on HTTP, a message queue system (like MSMQ), a WCF service with binding based on TCP (a TCP port chosen by us), etc.

**Normally, a distributed service should provide a "thick" or not very granulated interface. That is, the intention is to perform the highest number of actions within a method to minimize the number of remote calls from the client.**

In addition, the services are frequently stateless (without state or an internal object life relative to each external call) although they do not always have to be so. A basic *Web Service* (WS-I specifications) is *stateless*, but WCF advanced services (WS-* specifications or Net proprietary) may also have shared objects and states such as the *Singleton*, *Session* types, etc.).

www.EBooksWorld.ir

## 6.- DESIGN STEPS FOR THE SERVICES LAYER

The best approach when designing a service starts by defining the service contract, which consists of the interfaces that we plan to expose from our service. This is commonly referred to as '*Contract First Design*'. Once the service interface has been defined, the next step is to design the service implementation, which is used to translate data contracts into domain entities and to interact with the domain and application layer. The following basic steps can be used when designing a service layer:

1.- Define data contracts and messages that represent the scheme to be used for the messages (These data contracts could be DTOs or even Entities).

2.- Define the service contract that represents the operations supported by our service.

3.- Design transformation objects that translate between domain entities and service data contracts, as in the case of DTO-Adapters (transformation to Domain Entities). These operations and components may be situated at the Application layer, instead of within the Distributed Service Layer.

4.- Defining fault contracts that return information on errors to the consumers of the distributed service.

5.- Designing the integration with the internal Layers (Domain, Application, etc.). A good approach is to start DI (Dependency Injection) in this Web-Services level by using the resolution of IoC containers in this layer (Distributed services) only, since this is the first point of access to the application server, and letting the IoC system create all the internal dependency object graph for the rest of the layers.

## 7.- DATA OBJECT TYPES TO BE TRANSFERED

We should determine how we will transfer the entity data through the physical boundaries of our Architecture (Tiers). In most cases, when we want to transfer data from one process to another and especially from one server to another, we must serialize data.

We could also use this serialization when going from one logical layer to another, However, this is generally not a good idea, since we will have penalizations in performance.

www.EBooksWorld.ir

In general and from a logical point of view, the data objects to be transferred from one tier to another can be classified as follows:

- **Scalar Values**
- **DTOs (Data Transfer Objects)**
- **Serialized Domain Entities**
- **Sets of records (disconnected artifacts or data sets)**

All of these types of objects must have the capacity to be serialized (to XML, JSON or binary format) and transferred over the network.

**Scalar Values**

If we are going to transfer a very small amount of data (like a few input arguments) it is quite common to use scalar values (such as int, string, etc.). On the other hand, even when we have a few parameters it is a better practice to create a complex type (a class) merging all those parameters.

**Serialized Domain Entities**

When we are dealing with volumes of data related to domain entities, a first option (and the most immediate one) is to serialize and transmit their own domain entities to the presentation layer. This may be good or bad, depending on the context. In other words, if the implementation of the entities is strongly linked to a specific technology, it is contrary to the DDD Architecture recommendations because we are contaminating the entire architecture with a specific technology. However, we have the option of sending domain entities that are POCO (Plain Old CLR Objects), that is, serialized classes that are 100% custom code and do not depend on any data access technology. In this case, the approach can be good and very productive, because we could have tools that generate code for these entity classes for is. In addition, the work can be streamlined because even these entities can perform concurrency handling tasks for us. This concept (Serializing and transferring Domain Entities to other Tiers / physical levels) will be discussed in the chapter about Web Services implementation.

Thus, this approach (Serialization of Domain entities themselves) has the disadvantage of leaving the service consumer directly linked to the domain entities, which could have a different life cycle than the presentation layer data model and even different changing rates. Therefore, this approach is suitable only when we maintain direct control over the whole application (including the client that consumes the web-service), like a typical N-Tier application. On the other hand, when implementing SOA services for unknown consumers it is usually a better option to use DTOs, as explained below.

www.EBooksWorld.ir

**DTOs (*Data Transfer Objects*)**

To decouple clients/consumers of Web Services from the Domain Entities, the most common option is to implement DTOs (*Data Transfer Objects*). This is a design pattern that consists in packaging multiple data structures in a single data structure to be transferred between "physical boundaries" (remote communication between servers and/or machines). DTOs are especially useful when the application that uses our services has a data representation or even a model that does not exactly match the Domain Entity Model. This pattern allows us to change the Domain entities (internal implementation in our application Server) as long as the interfaces of Web Services and the DTOs structure do not change. So, in many cases, changes in the server will not affect the consumer application. It also supports a more comfortable version management towards external consumers. This design approach is, therefore, the most suitable when there are external clients/consumers using data from our web-services and when the development team does not have control over the development of these client applications (Consumer client applications could be developed by others).

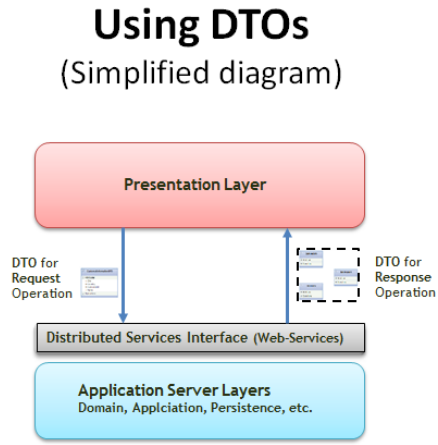



**Figure 5.- DTOs diagram (Data Transfer Objects)**

The typical design of DTOs tries to adapt to the hypothetical needs of the consumer (either presentation layer, or another type of external application). It is also important to design them so that they minimize the number of calls to the web service (minimize round-trips), therefore improving the performance of the distributed application.

Working with DTOs requires having certain adaptation/conversion logic from DTOs to Domain entities and vice versa. In DDD N-layered architecture, these Adapters would be typically placed by us in the Application layer, since this is a requirement of the Application Architecture and not of the Domain. Placing them within the Web Services would not be the best option either, since this layer should be as thin as possible.

www.EBooksWorld.ir

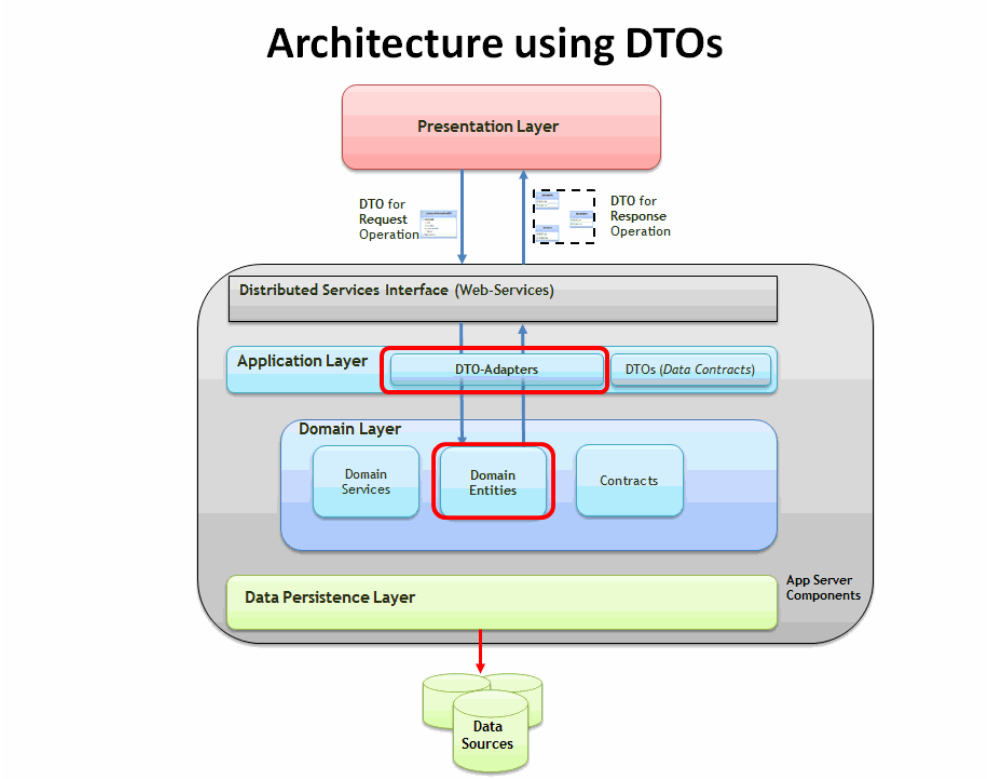



**Figure 6.- Architecture Diagram using DTOs (Data Transfer Objects)**

In short, the option of using **DTOs** (*Data Transfer Objects*) should be considered to consolidate data in unified structures that minimize the number of remote calls (round-trips) to Web Services. The DTOs promote a thick granulation of operations by accepting DTOs that are designed to carry data between different physical levels (Tiers).

This is the right approach from a pure Software Architecture point of view, since we decouple Domain data entities from "outside the domain world". In the long term, decoupling Domain entities from consumer applications (using DTOs) has great benefits when dealing with changes on either side (Domain vs. Presentation layer or external consumer). However, the use of DTOs requires significantly more initial work than when using directly serialized domain entities (which in some cases can even perform concurrency handling tasks for us), as we will see in the section about Distributed Services implementation in .NET.

There can also be mixed approaches; for example, using serialized domain entities for controlled presentation layers, and the use of DTOs for an SOA layer/facade outwards (other initially unknown consumers).

www.EBooksWorld.ir

**Sets of Records/Changes (*disconnected devices*)**

The sets of records/changes are usually implementations of disconnected complex data, such as DataSets in .NET. They are mechanisms that are very easy to use. However, they are closely linked to the underlying technology and tightly coupled components regarding the data access technology As such, they are completely contrary to the DDD approach (Domain Layer isolated from the infrastructure layers) and would not be recommended in this type of domain oriented architecture. They are more likely to be recommended in architectures for less complex applications and to be developed in a more RAD manner (*Rapid Application Development*).

Any of these logical concepts (Entity, DTO, etc.) may be serialized to different types of data (XML, binary, different XML formats/schemes, etc.), depending on the specific implementation chosen. However, this implementation is already associated with technology, so we will analyze this later in the section about Distributed Services Implementation in .NET.

**References about DTOs**

*"Pros and Cons of Data Transfer Objects"*- ***Dino Esposito:***
***http://msdn.microsoft.com/en-us/magazine/ee236638.aspx***

*"Building N-Tier Apps with EF4"*- ***Danny Simons****:*
***http://msdn.microsoft.com/en-us/magazine/ee335715.aspx***

# 8.- CONSUMPTION OF DISTRIBUTED SERVICES BASED ON AGENTS

The Service Agents basically establish a sub-layer within the client application (Presentation Layer) which centralizes and locates the "consumption" of Web Services in a methodical and homogeneous manner, instead of directly consuming the Services from any area of the client application (form, page, etc.). Ultimately, the use of agents is a way to design and program (pattern) the consumption of Web services.

**Definition of Service Agent**

*"A Service Agent is a component located in the presentation layer, acting as the front-end of communications towards Web Services. It should be solely responsible for actions of direct consumption of Web Services".*

www.EBooksWorld.ir

An agent could also be defined as a "*smart-proxy*" class that is an intermediary between a service and its consumers. Consider that the Agent is physically placed on the client side.

From the point of view of the client application (WPF, Silverlight, OBA, etc.), an agent acts "in favor" of a Web-Service. That is, as if it was a local "mirror" offering the same functionality of the service in the server.

Below we show a diagram with agents consuming a Distributed Service:

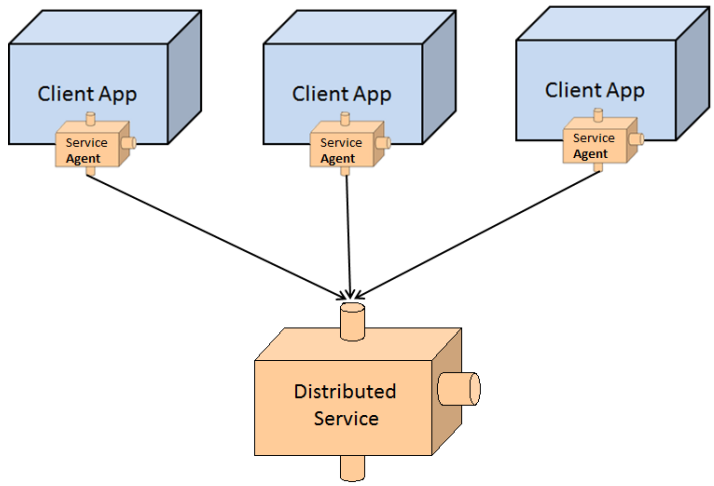



**Figure 7.- Diagram of the agents in a Service "Consumption" architecture**

The **Agent** should help prepare service requests and interpret the responses from the service.

It is important to consider that an agent is not a part of the service (it must be loosely coupled to the distributed service) and therefore the service must not trust the agent. All the interaction between an agent and a service should be authenticated, authorized and validated by the service in the same way in which a service is accessed directly without an agent.

Some of the advantages of using Agents are:

- **Easy integration**: If a Service has its corresponding Agent, providing this developed agent to whomever is going to consume the service may simplify the development process.

- **Mocking**: Mocking a Web Service is very useful, especially when the Service exposes a system which is not always available during development time, such as a Host, corporate ERP, etc. We should be able to test our client application in an isolated way. The Service Agent would be the place to implement the fake web service.

www.EBooksWorld.ir

- **Error Management**: Reacting correctly to the error conditions is essential and one of the most complex tasks for the developer that uses a Web service from an application. The Agents should be designed to understand mistakes a service can make, greatly simplifying the development of subsequent integrations.

- **Offline data management and cache**: An agent may be designed to make a "cache" of data for the service correctly and so it can be understood. This can sometimes dramatically improve response times (and therefore performance and scalability) of the requests and even enable applications to work offline.

- **Request validations**: Agents can verify the input data sent to the server components and ensure they are correct before making any remote call (cost in latency to the server). This in no way exempts the server from having to validate data, since the safest way is in the server (the client may have been hacked) but it can normally save time.

- **Intelligent routing**: Some services may use agents to send requests to a specific service server, based on the contents of the request.

In short, the concept is very simple; the agents are classes located in an *assembly* on the client side and they are the only classes on this side that should interact with the proxy classes of the Services. On a practical level, the usual way is to create a class library project in order to implement these Agent classes. After that, we simply have to add a reference to this *assembly* in the client application.

Before reviewing other aspects, it is important to emphasize that the use of Agents does not depend on technology. This pattern can be used for consuming any type of Distributed Service.

## 9.- INTEROPERABILITY

The main factors affecting interoperability of the applications are the availability of proper communication channels (standard) and formats and protocols that can be understood by the parties involved in different technologies. Consider this guideline:

- In order to communicate with most platforms and devices from different manufacturers, the use of standard protocols and standard data formats are recommended, such as HTTP and XML, respectively. Bear in mind that decisions on protocol may affect availability of consumer clients. For example, target systems can be protected by Firewalls that block some protocols.

- The chosen data format may affect interoperability. For example, the target systems may not understand specific data types related to a technology (for example, ADO.NET Datasets are hardly consumed from JAVA applications) or they may have different ways of managing and serializing the data types.

www.EBooksWorld.ir

- The selected communications security can also affect interoperability. For example, some encryption/decryption techniques (like 'message based security') may not be available in many consumer systems.

## 10.- PERFORMANCE

The design of communication interfaces and data formats to be used will have a considerable impact on the application performance, especially when we cross "boundaries" in communication between different processes and/or different machines. There are techniques we can use to improve performance related to communications between different application tiers.

Consider the following guidelines and best practices:

- Minimize the volume of data transmitted through the network, this reduces overload during objects serialization (For instance, server paging 'is a must').

- **Coarse-grained** Web Services: It is important to bear in mind that we should always **avoid** working with **fine-grained** Web Service interfaces (this is how the internal components are usually designed within the Domain). This is problematic because it forces us to implement the consumption of Web Services in a "chatty" way. This type of design strongly impacts performance because it forces the client application to make many remote calls (many round-trips) for a single global operation and since remote calls have a performance cost (latency because of Web Service activation, data serialization/de-serialization of data, etc.), it is critical to minimize the round-trips. In this regard, the use of DTOs is best when deemed convenient (it allows grouping of different Domain entities into a single data structure to be transferred) although ORM technologies (such as 'Entity Framework') also allow serialization of graphs containing several entities.

- Consider using a Web Services Facade which provides a **coarse-grained interface**, encapsulating the Domain components that usually are fine-grained designed.

- If web-service data serialization (XML serialization) impacts on the application performance, consider using binary serialization (although binary serialization is usually not interoperable with other technical platforms).

- Consider using other protocols (such as TCP, Named-Pipes, MSMQ, etc.). In most cases, they substantially improve communication performance. However, we may lose HTTP interoperability.

www.EBooksWorld.ir

# 11.- ASYNCHRONOUS *vs.* SYNCHRONOUS COMMUNICATION

We should consider the advantages and disadvantages of communicating with Web services in a synchronous vs. asynchronous manner.

**Synchronous** communication is appropriate for scenarios where we must guarantee certain operations sequence or when the user must wait to see the requested information (although this last point can also be obtained through asynchronous communication).

**Asynchronous** communication is suitable for scenarios where the response from the application must be immediate or in scenarios where there is no guarantee that the target is available.

Consider these guidelines when deciding on synchronous or asynchronous communications:

- We should consider an **asynchronous communication model** to obtain the highest performance and scalability, a nice loosely-coupled architecture regarding the back-end, and to minimize the system load. . If some clients can only make synchronous calls, a component can be implemented (Service Agent in the client) that is synchronous towards the client but can use web services in an asynchronous manner. This provides the possibility of making different calls at the same time, increasing the overall performance in that area.

- In cases where we must ensure the sequence in the execution of operations or when operations that depend on the outcome of previous operations are used, the most suitable scheme is probably **synchronous communication**. In most cases, a synchronous operation with a certain request can be simulated with coordinated asynchronous operations. However, depending on the particular scenario, the effort put forth in implementing it may or may not be worth the trouble.

- If asynchronous communication is chosen but network connectivity and/or availability of destination cannot always be guaranteed, consider using a system of "saving/sending" messaging **that ensures communication** (**Message queue** system, such as MSMQ), to avoid missing messages. These message queue advanced systems can even extend transactions by sending asynchronous messages to the message queues. If, in addition, you need to interoperate and integrate with other business platforms, consider the use of integration platforms (such as Microsoft BizTalk Server.)

www.EBooksWorld.ir

## 12.- REST *vs.* SOAP

REST (*Representational State Transfer*) and SOAP (*Simple Object Access Protocol*) **represent two very different styles** to implement a Distributed Service. Technically, REST is a pattern of architecture constructed with simple verbs that fit perfectly with HTTP. Although REST architecture principles could apply to protocols other than HTTP, in practice, REST implementations are fully based on HTTP.

SOAP is a messaging protocol based on XML (SOAP messages with a specific XML format). It can be used with any communications protocol (Transport) including HTTP.

The main difference between these two approaches is the way the service state is maintained. We are referring to a very different state from that of session or application. We are referring to the different states that an application goes through during its lifetime. With SOAP, changing through different states is made by interacting with a single service endpoint which encapsulates many operations and message types.

On the other hand, with REST, we have a limited number of operations and these operations are applied to resources represented and addressed by URIs (HTTP addresses). The messages are composed by current resources states or the required resources state. REST works very well with Web applications where HTTP can be used as protocol for data types other than XML (like JSON). The service consumers interact with the resources through URIs in the same way people can navigate and interact with Web pages through URLs (web addresses).

The following diagram shows what scenarios REST or SOAP fit better:

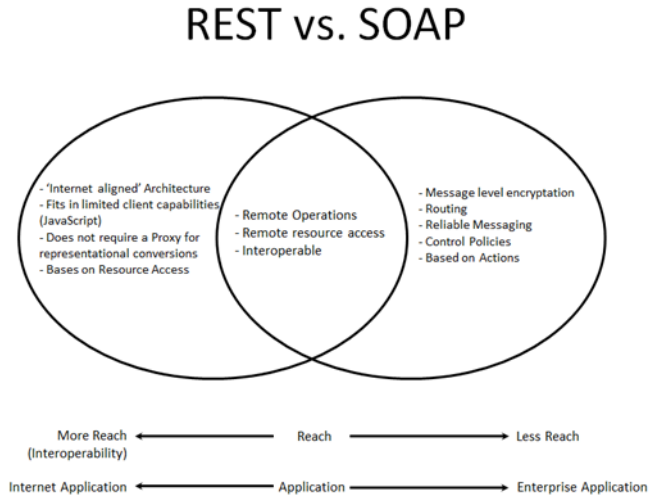



**Figure 8.- REST vs. SOAP scenarios**

www.EBooksWorld.ir

From a technical point of view, these are some of the advantages and disadvantages of both of them:

**SOAP advantages:**

- Good for data (communications are strict and structured)
- Strongly typed proxies thanks to WSDL
- Works over different communication protocols. The use of protocols other than HTTP (such as TCP, NamedPipes, MSMQ, etc.), can improve performance in certain scenarios.

**SOAP disadvantages:**

- SOAP messages are not 'cacheable' by CDNs.
- SOAP messages are not JavaScript friendly (For AJAX, REST is the best choice).

**REST advantages:**

- Governed by HTTP specifications, so the services act as Resources, such as images or HTML documents.
- Data can be maintained strictly or decoupled (not as strict as SOAP)
- REST resources can be easily used from the JavaScript code (AJAX, etc.)
- Messages are light, so the performance and scalability offered are high. This is important for many Internet applications.
- REST can use XML or JSON as data format.

**REST disadvantages:**

- Working with strongly typed objects in the server code is hard, although this depends on technological implementations and it is improving in the latest versions.
- Only works over HTTP
- REST calls are restricted to HTTP verbs (GET, POST, PUT, DELETE, etc.)

Even though both approaches (REST and SOAP) may be used for similar types of services, the approach based on REST is normally more suitable for Distributed Services that are publicly accessible (Internet) or in cases where a Service can be accessed by unknown consumers. SOAP, on the contrary, is much better for

www.EBooksWorld.ir

implementing procedural implementation ranges, such as an interface between the different layers of Application architecture or, ultimately, private Business Applications.

SOAP does not limit us to HTTP. The standard specification WS-*, which can be used on SOAP, provides a standard and therefore interoperable path to work with advanced aspects such as SECURITY, TRANSACTIONS, ADDRESSING AND RELIABLE-MESSAGING.

REST also offers a great level of interoperability (due to the simplicity of its protocol); however, for advanced aspects, such as those previously mentioned, it would be necessary to implement their own mechanisms, which would be non-standard.

In short, both protocols allow us to interchange data by using verbs. The difference lies in the fact that, with REST, this set of verbs is restricted to coincidence with HTTP verbs (GET, PUT, etc.) and in the case of SOAP, the set of verbs is open and defined in the Service endpoint.

Consider the following guidelines when choosing one approach or the other:

- SOAP is a protocol that provides a messaging framework that a layer abstraction can be built on, and it is mostly used as an RPC calls system (synchronous or asynchronous) where data is transferred as XML messages.

- SOAP manages aspects, such as security and addressing, through its internal implementation of SOAP protocol.

- REST is a technique that uses other protocols, such as JSON (JavaScript Object Notation) Atom as a publication protocol, and simple and light formats of the POX type (Plain Old XML).

- REST enables the use of standard HTTP calls such as GET and PUT to make queries and modify the state of the system. REST is stateless by nature, which means each individual request sent from the client to the server must contain all the necessary information in order to understand the request, since the server will not store data about the state of the session.

## 12.1.- Design Considerations for SOAP

SOAP is a protocol based on messages that is used to implement the messages layer of a Web Service. The message consists of an "envelope" with a header and a body. The header can be used to provide information external to the operation to be performed by the service (e.g., security aspects, transactional aspects or message routing, included in the header).

The body of the message has contracts, in the form of XML schemes, which are used to implement the Web service. Consider this design guideline which is specific to SOAP Web Services:

www.EBooksWorld.ir

- Determine how to manage errors and faults (Normally exceptions generated in internal layers of the server) and how to return the proper information on errors to the Web Service consumer. (See "Exception Handling in Service Oriented Applications" in **http://msdn.microsoft.com/en-us/library/cc304819.aspx**.)

- Define the schemes of operations that can be performed by a service (Service Contract), the structures of data passed when requests are made (Data Contract) and errors and faults that can be returned from a request to the Web service.

- Choose a proper security model. For more information, see "*Improving Web Services Security*: Scenarios and Implementation Guidance for WCF" in **http://msdn.microsoft.com/en-us/library/cc949034.aspx**

- Avoid using complex types with dynamic schemes (such as Datasets). Try to use simple types, DTO classes or entity classes to maximize interoperability with any platform.

## 12.2.- Design Considerations for REST

REST represents an architecture style for distributed systems and is designed to reduce complexity by dividing the system into resources. The resources and operations supported by a resource are represented and exposed through a set of URIs (HTTP addresses) logically on the HTTP protocol. Consider this guideline specifically for REST:

- Identify and categorize the resources that will be available for Service consumers

- Choose an approach for representation of resources. A good practice would be using names with meaning (Ubiquitous language in DDD?) for REST input points and unique identifiers for specific resource instances. For example, **http://www.mycompany.employee/** represents the input point to access an employee and **http://www.mycompany.employee/smith01** uses an employee ID to indicate a specific employee.

- Decide if multiple representations for different resources should be supported. For example, we can decide if the resource should support an XML format, Atom or JSON and make it part of the resource request. A resource may be exposed by multiple representations. For example:

    - **http://www.mycompany.employee/smith01.atom** and

    - **http://www.mycompany.employee/ smith 01.json**

www.EBooksWorld.ir

- Decide if multiple views for different resources will be supported. For example, decide if the resource should support GET and POST operations or simply GET operations. Avoid excessive use of POST operations, if possible, and also avoid exposing actions in URI.

- Do not implement user session state maintenance within a service and do not try to use HYPERTEXT (like hidden controls in Web pages) to manage states. For example, when a user makes a request to add an item to the shopping cart of an e-commerce company, the cart data must be stored in a persistent state storage or a cache system prepared for that purpose, but not in memory as states of the own services (which would also invalidate scalable scenarios of the "Web Farm" type).

## 13.- INTRODUCTION TO SOAP AND WS-*

SOAP, originally defined as 'Simple Object Access Protocol', is a specification for exchanging information structured in the Web Service implementation. It is specially based on XML as message formats and HTTP as communication protocols (But it can use other communication protocols, as well).

SOAP is the stack base of Web Service protocols, providing a basic frame of messaging on which the Web Services can be built.

This protocol is defined in three parts:

- A message **envelope**, that defines the contents of the body or contents of the message and how to process it

- A set of **serialization** rules to express instances of application data types

- A **conversation** to represent calls and answers to remote methods

In short, it is a system of remote calls based on XML messages at a low level. A SOAP message will be used both for requesting the execution of a method of remote Web Service and for using another SOAP message as an answer (containing the requested information). Due to the fact that the data format is XML (text, with scheme, but text finally), it can be used from any platform or technology. SOAP is interoperable.

The basic standard of SOAP is 'SOAP WS-I *Basic Profile'*.

www.EBooksWorld.ir

## 14.- WS-* SPECIFICATIONS

Basic web services (such as SOAP WS-I, *Basic Profile*) offer more than communications between the Web service and the client applications that use it. However, the standards of basic web services (*WS-Basic Profile*) were just the beginning of SOAP.

Transactional and complex business applications require many more functionalities and service quality requirements (QoS) than simple communications between client and web services. The following needs are usually required by business applications:

- Message base security or mixed security in communications, including authentication, authorization, encryption, non-tampering, signature, etc.
- Reliable messaging
- Distributed transactions support between different services.
- Routing and addressing mechanisms.
- Metadata to define requirements as policies.
- Support to attach large volumes of binary data when invoking web services (images and/or attachments of any kind).

To define all these "advanced needs", the industry (different companies such as Microsoft, IBM, HP, Fujitsu, BEA, VeriSign, SUN, Oracle, CA, Nokia, CommerceOne, Documentum, TIBCO, etc.) has been and continues to be defining some theoretical specifications that set forth how the extended aspects of the Web services should operate.

All these "theoretical specifications" are known as **WS-* specifications.** (The '*' is given because there are many advanced web services specifications, such as *WS-Security*, *WS-SecureConversation*, *WS-AtomicTransactions*, etc.) To learn more about these specifications, you can review the standards in: **http://www.oasis-open.org**.

In short, these WS-* specifications theoretically define the advanced requirements of the business applications.

The following scheme shows the different functionalities the WS.* tries to solve at a high level.

www.EBooksWorld.ir

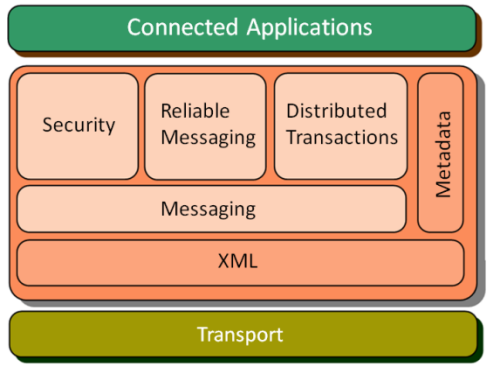



**Figure 9.- WS-* functionalities Diagram.**

All the central modules (Security, Reliable Messaging, Transactions and Metadata) are precisely the functionalities the basic XML Web services do not have, and what defines WS.*.

The **WS.* specifications** are therefore formed by subsets of specifications:

- WS-Security
- WS-Messaging
- WS-Transaction
- WS-Reliability
- WS-Metadata

They, in turn, are subdivided into other subsets of specifications, which are deeply defined, as shown below:

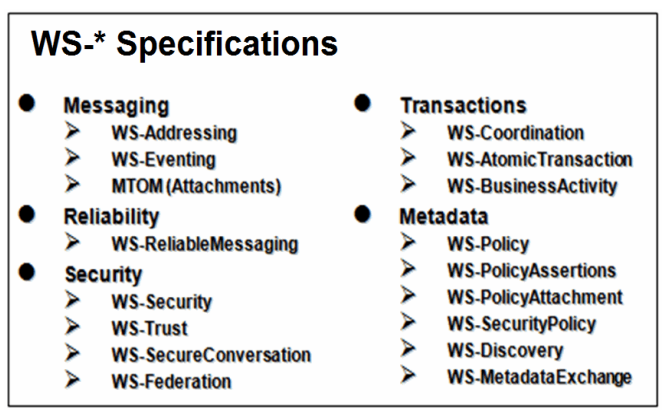



**Figure 10.- WS-* specifications**

www.EBooksWorld.ir

Therefore, WS-* specifications are virtually a whole world, they are not just a small extension to the basic Web Services.

Below we show a table with the needs in SOA business distributed applications and the WS-* standards that define them:

**Table 1.- WS-* specifications**

| Advanced needs in services | WS-* Specifications that define them |
|---|---|
| Advanced, including different types of authentication, authorization, encryption, non-tampering, signature, etc. | → WS-Security<br>→ WS-SecureConversation<br>→ WS-Trust |
| Stable and reliable messaging | → WS-ReliableMessaging |
| Distributed Transactions support between different services | → WS-AtomicTransactions |
| Addressing and routing mechanisms | → WS-Addressing |
| Metadata to define requirements as policies | → WS-Policy |
| Support to attach great volumes of binary data when invoking services (images and/or attachments of any type) | → MTOM |

www.EBooksWorld.ir

# 15.- INTRODUCTION TO REST

**What is REST?** REST was introduced by *Roy Fielding* in a speech where he described an "architecture style" of interconnected systems. Also, REST is the acronym of "*Representational State Transfer*".

Why is it called "*Representational State Transfer*"? The Web is a set of resources. A resource is an element of interest. For example, Microsoft can define a type of resource on a product of its own, which could be Microsoft Office SharePoint. In this case, clients can access this resource with an URL such as:

**http://www.microsoft.com/products/sharepoint**

To access this resource, a **representation** of the resource will be returned (e.g., SharePoint.htm). This **representation** places the client application in a **state**. The result of the client accessing a link within said HTML page will be another accessed resource. The new representation will place the client application in another state. So the client application changes (**transfers**) the state with each resource representation. In short, there is a "*Representational State Transfer*".

So REST's goal is to show the natural features of the Web that made the Internet a success. These are precisely the features used to define REST.

REST is not a standard, it is an architecture style. We probably won't see W3C publishing a REST specification, because REST is only an architecture style. A style cannot be packaged but only understood, and web services can be designed accordingly. It is comparable to an N-tier architecture style, or SOA architecture. We don't have an N-Tier standard or SOA standard.

However, although REST is not itself a standard, it is based on the Internet standards:

- HTTP
- URL
- XML/HTML/PNG/GIF/JPEG/etc (Resources representations)
- Text/xml, text/html, image/gif, etc. (MIME type)

## 15.1.- The URI in REST

In conclusion and as an essential concept in REST, the most important thing in REST is the URI (URI is a cool and more technical way of saying URL, so it is better to name it like this...) Seriously, the URI is very important in REST because it bases all the definitions of access to Web Services on syntax of a URI. In other words, we will

www.EBooksWorld.ir

explain it with several examples of URIs of Web Services based on REST. As seen, the definition is self-explanatory, which is one of the objectives of REST; simplicity and self-explanation, so we will not explain such sample URIs as:

**http://www.mydomain.com/Providers/GetAll/**
**http://www.mydomain.com/Providers/GetProvider/2050**
**http://www.mydomain.com/Providers/GetProviderByName/Smith/Joe**

As stated, we will not explain this, considering how easy it is.

## 15.2.- Simplicity

Simplicity is one of the fundamental aspects in REST. Simplicity is pursued in any aspect, from the URI to the XML messages sent or received from the Web service. This simplicity is a big difference as compared with SOAP, which is quite complex in its headers, etc.

The benefit of this simplicity is the ability to achieve good performance and efficiency because it is light (even when we are working with less than efficient standards, such as HTTP and XML). In the end, however, the data (bits) transmitted are always those of minimum necessity. We have something light, so performance will be quite optimal. On the other hand, if we are based on something quite simple, then complex capabilities (which can be done using SOAP) are almost impossible to achieve using REST. For example, advanced security standards, signature and message level encryption, Distributed transactions between several web services and many other advanced functionalities that are defined in WS-* specifications based on SOAP.

But the goal of REST is not to achieve large or complex functionality, but to achieve a minimum functionality needed by a great percentage of web services on the Internet that are interoperable; that simply transmit the information and are very efficient.

Below we show an example of a REST message returned by a Web Service. The simplicity is in contrast to the SOAP WS-* message that may be quite complex and therefore heavier. REST messages are very light:

```
<?xmlversion="1.0"?>
<p:Clientxmlns:p="http://www.mycompany.com"
   xmlns:xlink="http://www.w3.org/1999/xlink">
<Client-ID>00345</Client-ID>
   <Name>Smith & Brothers</Name>
   <Description>Great company</ Description>
   <Details xlink:href="http://www.mycompany.com
/clients/00345/details"/>
</p:Client>
```

www.EBooksWorld.ir

Evidently, it is difficult to design a more simplified XML message than the one above. It is interesting to note the "Details" item of the example, which is that of a link or hyperlink type. The importance of these “link” type elements is explained below.

## 15.3.- Logical URLs versus Physical URLs

A resource is a conceptual entity. A representation is a concrete manifestation of this resource. For example:

**http://www.mycompany.com/customers/00345**

The above URL is a logical URL, not a physical URL. For example, there is no need for an HTML page for each client in this example.

A correct design aspect of URIs in REST is that the technology used in URI/URL should not be disclosed. There should be freedom to change implementation without affecting the client applications that are using it. In fact, this involves a problem for WCF services hosted in IIS, since these services usually work based on a .svc page. However, in the latest WCF version, we can work using REST with no .svc extension.

## 15.4.- Core characteristics of REST Web Services

- Client-Server: “pull” interaction style. Complex methods of communications, of the Full-Duplex or Peer-to-Peer type, cannot be implemented with REST. REST is for simple Web services.

- *Stateless*: each request that is made by the client to the server must have all the information necessary to understand and execute the request. No type of server context should be used. This is what the basic Web services are also like in .NET (single-call, stateless); however, in WCF there are more types of instantiation, such as Singleton and shared instantiation (with sessions). This also cannot be implemented with a REST Web service.

- *Cache*: to improve the network efficiency, the answers should be classified as “*cacheable*” and “non-*cacheable*”

- Uniform interface: all resources are accessed with a generic interface (for example: HTTP GET, POST, PUT, DELETE); however, the most important or predominant interface in REST is GET (like the URLs shown in the above example). GET is considered as “special” for REST.

- The content type is the object model

www.EBooksWorld.ir

- Image, XML, JSON, etc.
- Named resources. The system is limited to resources that can be named through an URI/URL.
- Representations of interconnected resources: representations of resources are interconnected through URLs; this enables the client to go from one state to the next.

## 15.5.- Design Principles of REST Web Services

- The key to creating web services in a REST network (e.g., the Web on the Internet) is to identify all the conceptual entities to be exposed as services. We saw some examples earlier, such as clients, products, invoices, etc.
- Create an URI/URL for each resource. Resources should be nouns, not verbs. For example, the following URI would be wrong:

  **http://www.mycompany.com/customers/getcustomer?id=00452**

- The verb "GetCustomer" would be wrong. Instead, only the name would appear, like this:

  **http://www.mycompany.com/customers/customer/00452**

- Categorize resources according to whether the client applications can receive a representation of the resource, or whether client applications can modify (add) to the resource. For the former item, the resource should be made accessible with a HTTP GET, for the latter item, the resources should be made accessible with HTTP POST, PUT and/or DELETE.
- The representations should not be isolated islands of information. That is why links should be implemented within the resources to allow client applications to search for more detailed or related information.
- Design to gradually reveal data. Do not reveal everything in a single document response. Provide links to obtain more details.
- Specify the format of the response using an XML scheme (W3C Schema, etc.)

www.EBooksWorld.ir

**Additional Resources**

*"Enterprise Solution Patterns Using Microsoft .NET" in:*
***http://msdn.microsoft.com/en-us/library/ms998469.aspx***

*"Web Service Security Guidance" in:*
***http://msdn.microsoft.com/en-us/library/aa480545.aspx***

*"Improving Web Services Security: Scenarios and Implementation Guidance for WCF" in* ***http://www.codeplex.com/WCFSecurityGuide***

*"WS-* Specifications" in:*
***http://www.ws-standards.com/ws-atomictransaction.asp***

## 16.- ODATA: OPEN DATA PROTOCOL

**OData** is a higher-level concept than SOAP and REST. It is also the most recent, as it is a proposed standard for high level protocols based on **REST** and **AtomPub**.

Let's start from the beginning. **What exactly is OData?**

**Table 2.- OData definition**

| Definition | |
|---|---|
|  | **OData** (*Open Data Protocol*) is a web protocol to perform queries and remote updates to access services and data stores. OData emerged based on the **AtomPub** experiences of server and client implementations. OData is used to expose and access information from different resources, including, but not limited to, relational databases. Actually, it can publish any type of resource. |

**OData** is based on certain conventions, especially on **AtomPub** using data oriented **REST** services. These services share resources identified through the use of URIs (*UniformResourceIdentifiers*) and defined as an abstract model of data to be read/queried and edited by clients of such Web services HTTP-REST.

**OData** consists of a set of specifications such as [OData:URI], [OData:Terms], [OData:Operations], [OData:Atom], [OData:JSON] and [OData:Batch].

www.EBooksWorld.ir

So, OData is a **high level protocol designed to share data in the network**, especially in public and interoperable Internet environments. In short, it is a higher level than REST, but following the same trend, using URIs to identify each piece of information in a service, HTTP to transport requests and answers and AtomPub and JSON to manage sets and representation of data.

**The main goal of OData is to offer a standard manner of using data via the network and getting consumers of data services to use a series of high level conventions that would be of much interest if widely adopted**. Ultimately, using schemes and predefined conventions instead of "reinventing the wheel" during development and birth of each distributed or web service.

Finally, keep in mind that OData is a standard proposed by Microsoft that is born initially from protocols used originally in **ADO.NET Data Services** (currently called ***WCF Data Services***), but the interesting part is that Microsoft has made it evolve and released it through the OSP (*Open Specification Promise*) so that any manufacturer can create implementations of OData.

The benefits of OData as proposed open standard protocol are interoperability and collaboration with other platforms, as well as how it can be implemented by any platform that supports HTTP, XML, AtomPub and JSON. For example, **IBM Web Sphere** is one of the products and manufacturers that support OData (the service called *IBM WebSphereeXtremeScale REST* supports OData), along with many Microsoft products and technologies, primarily the following:

- Base technology/implementation of Microsoft OData
    - ***WCF Data Services***
- Higher level products and technologies:
    - *Windows Azure Storage Tables*
    - *SQL Azure*
    - *SharePoint 2010*
    - *SQL Server Reporting Services*
    - *Excel 2010 (with SQL Server PowerPivot for Excel)*

For the complete list, see **http://www.odata.org/producers**

Due to its nature (REST, Web, AtomPub and interoperability) it is highly oriented to publication and use of data in heterogeneous environments and the Internet and therefore, 'Data Oriented' services instead of 'Domain Oriented' (DDD). In a complex and private business application, implementation of its internal distributed services is probably more powerful using SOAP and WS-* specifications (Security, transactions, etc.). However, a 'Domain Oriented' application may want to publish information to the outside (other applications and/or initially unknown services). That is where OData fits perfectly as an additional access interface to our 'Domain Oriented' application/service from the outside world, other services and ultimately "the network".

Currently, in our implementation of the sample application related to the present Architecture (*Domain-oriented N-Layered*) we do not use OData because DDD does

www.EBooksWorld.ir

not offer a '*Data Oriented*' architecture/application, but one that is '*Domain Oriented*'. Moreover, the distributed services are being essentially used from another layer of our application (Presentation layer within our application), so it is more flexible to use SOAP or even REST at a lower level. OData is more oriented to publishing data directly as CRUD services (*Create-Read-Update-Delete*), with pre-set specifications, which is based on *WCF Data Services*.

Finally, because OData is really strategic for Microsoft, in the future OData could evolve towards many more scenarios further than Data-Driven Services/Apps. Keep an eye on **http://odata.org** for more details.

## 17.- GLOBAL DESIGN RULES FOR SOA SYSTEMS AND SERVICES

**Table 3.- Global Design Rules**

**Rule # D22**

**Identify what server components should be SOA services**

**Rule**

- Not all the components of an application server should be accessed exclusively by Distributed Services.
- **Bear "the end"** in mind, **not "the means"**.
- The components that have **business value** and are **reusable in different applications** and those that should necessarily be accessed remotely (because the Presentation layer is remote, Windows Client Type) should be identified as **SOA Services**.
- If the presentation layer is remote (e.g., WPF, Silverlight, OBA, etc.), a "Distributed Service Interface" should be published through Services.
- The goal of "transparency" and "interoperability" is achieved.

www.EBooksWorld.ir

**Table 4.- Global Design Rules**

**Rule # D23**

**The internal Architecture of a service should follow the guidelines of N-layer architecture**

**Rule**

- Each independent service must be internally designed in accordance with the N-layer architecture, similar to the one described in this guide.

**Table 5.- Global Design Rules**

**Rule # D24**

**Identify the need to use DTOs vs. serialized Domain Entities, as data structures to communicate between different tiers or physical levels**

**Rule**

- This rule means that we have to identify when it is worth the excessive effort of implementing DTOs and DTO adapters versus the direct use of serialized Domain entities.

- In general, if the party that uses our services (client/consumer) is controlled by the same development team as the server components, it will be much more productive to use serialized Domain Entities. However, if the consumers are external, initially unknown and not under our control, the decoupling offered by DTOs will be crucial and the excessive effort of implementing them will really be worthwhile.

**References**:

*Pros and Cons of Data Transfer Objects*,***Dino Esposito***
***http://msdn.microsoft.com/en-us/magazine/ee236638.aspx***

*Building N-Tier Apps with EF4*, ***Danny Simons****:*
***http://msdn.microsoft.com/en-us/magazine/ee335715.aspx***

www.EBooksWorld.ir

**Table 6.- Global Design Rules**

**Rule # D25**

**The boundaries of Services must be explicit**

**Rule**

- Whoever develops the **client application** that uses a service should be aware of when and how a service is remotely used in order to consider scenarios of errors, exceptions, low band width on the network, etc. All of this should be implemented in the **Service Agents.**
- Web Services interfaces should be **coarse-grained**, minimizing the number of round-trips from the client application to the Service.
- Maintain **maximum simplicity in the service interfaces.**

**Table 7.- Global Design Rules**

**Rule # D26**

**Services must be independent in pure SOA architectures**

**Rule**

- In a pure SOA architecture, services should be designed, developed and versioned independently. The services must not depend heavily on their life cycles with respect to applications that use them. In general this requires the use of DTOs (Data contracts).
- Services should offer ***ubiquity***, that is, they must be ***locatable*** (through ***UDDI***) and above all, ***self-descriptive*** through standards such as **WSDL and MEX** (***Metadata-Exchange***). This is easily achieved simply by developing services with technologies that provide it directly, such as ***ASMX and/or WCF***.
- **Security**, especially authorization, should be **managed by each service within its boundaries.** It is recommended, however, that authentication be based on propagated authentication systems, based on standards such as ***WS-Federation***.

www.EBooksWorld.ir

**Table 8.- Global Design Rules**

**Rule # D27**

**Service compatibility must be based on Policies**

**Rule**

- Implement horizontal requirements and compatibility restrictions at the security level (such as **required security, monitoring**, types of communication and protocols, etc.) in the form of **policies** (as defined in configuration files of the type .**config**) whenever possible, instead of implementing restrictions based on code (*hard-coded*).

**Table 9.- Global Design Rules**

**Rule # D28**

**Context, composition and state of global SOA services**

**Rule**

- If the **SOA services** we are treating are GLOBAL SERVICES (to be used by "n" applications), then they should be designed so that **they ignore the context from which they are being "consumed".** This does not mean the Services cannot have a state (stateless), but rather that they should be independent from the context of the consumer, because each consumer context will, in all likelihood, be different.

- '**Loosely coupled**': the SOA Services that are GLOBAL can be reused in "client contexts" which may not be known at the time of design.

- **Value can be created when combining Services** (e.g., booking a holiday with a flight-booking service, with another service to book a car and another to book a hotel).

www.EBooksWorld.ir

For more general information on SOA concepts and patterns to be followed, see the following references:

**SOA and Service references**:

*Service pattern*
***http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesServiceInterface.asp***

*Service-Oriented Integration*
***http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/archserviceorientedintegration.asp***

# 18.- IMPLEMENTING THE DISTRIBUTED SERVICES LAYER WITH WCF 4.0

The purpose of this chapter is to show different options we have at the technology level to implement the Distributed Services layer and of course, to explain the technical options chosen in our .NET 4.0 reference architecture.

We highlight the Location of the Distributed Services layer in the Architecture diagram shown below:

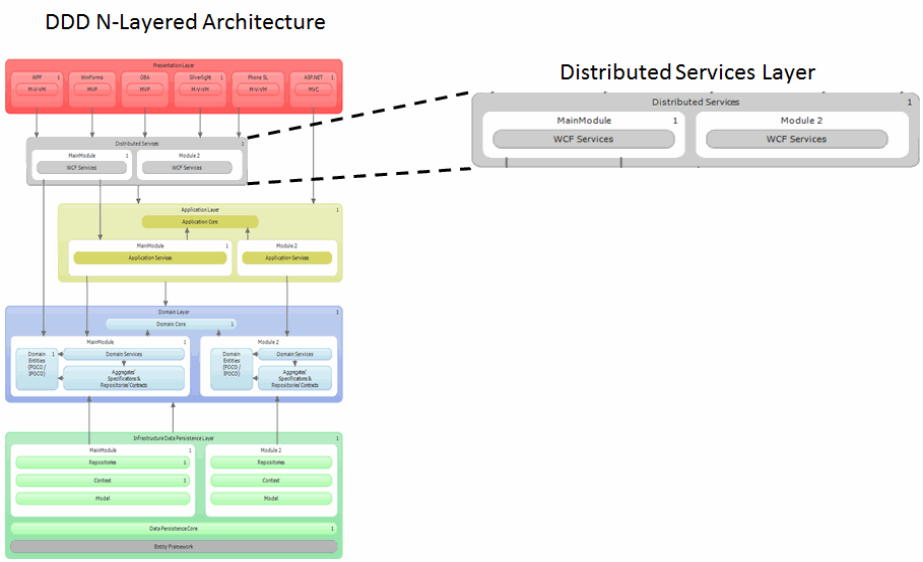



**Figure 11.- Distributed Services layer in Layer diagram - Visual Studio 2010**

www.EBooksWorld.ir

There are several possibilities for the implementation of Distributed Services with Microsoft technology, as we will analyze below. However, the most powerful technology is WCF (*Windows Communication Foundation*), so this is how we recommend implementing this layer within our proposed architecture.

# 19.- TECHNOLOGICAL OPTIONS

In a Microsoft platform, we can currently choose between two message oriented base technologies and Web services:

- *ASP.NET Web Services (ASMX)*
- *Windows Communication Foundation (WCF)*

As well as other derivative technologies of a higher level:

- *Workflow-Services (''WCF+WF'')*
- *RAD (Rapid Application Development)*:
    - *WCF Data.Services (aka. ADO.NET DS)*
        - *Implementation of OData of Microsoft.*
    - *WCF RIA Services*

However, it is not feasible to use higher level technologies (RAD) for this architecture where we need decoupling between components of the different layers, as they are usually tightly-coupled and data oriented technologies. That is why the only two options to be initially considered are the core technologies with which we can implement Web services: WCF or ASP.NET ASMX and in some cases *Workflow-Services*.

## 19.1.- WCF Technology

WCF provides decoupled technology in many ways (protocols, data formats, hosting process, etc.), providing very good configuration control. Consider WCF in the following cases:

- Web services to be created require interoperability with other platforms that also support SOAP and/or REST, such as JEE application servers
- You require either SOAP Web-Services or *RESTful* Services.

www.EBooksWorld.ir

- A higher performance is required in communications and support of both SOAP messages and binary format are required.
- WS-Security implementation is required to implement authentication, data integrity, data privacy and message-based encryption.
- The implementation of WS-MetadataExchange is required in SOAP requests to obtain descriptive information on services, such as its WSDL definitions and policies.
- The implementation of 'WS-ReliableMessaging' is required to implement end to end reliable communications, even performing a route between the different intermediates of Web services (not just a point to point origin and destination).
- Consider WS-Coordination and WS-AT (AtomicTransaction) to coordinate *'two-phasecommit'* transactions in the context of Web Services conversations. See: **http://msdn.microsoft.com/en-us/library/aa751906.aspx**
- WCF supports several communication protocols:
    - For public services, those of the Internet and those that are interoperable, consider HTTP
    - For services with higher performance and end to end .NET, consider TCP
    - For services used within the same machine, consider named-pipes
    - For services that must ensure communication, consider MSMQ, which ensures communication through messages queues

## 19.2.- ASMX technology (Web ASP.NET services)

ASMX provides a simpler technology for developing Web services, although it is also an older technology and more coupled/linked to certain technologies, protocols and formats.

- ASP.NET web services are exposed through IIS Web server
- It can only be based on HTTP as a communication protocol
- It does not support transactions distributed between different web services

www.EBooksWorld.ir

- It does not support advanced standards of SOAP (WS-*), it only supports the SOAP WS-I Basic Profile
- It provides interoperability with other platforms that are not .NET through
- SOAP WS-I, which is interoperable.

## 19.3.- Technology Selection

To implement simple web services, ASMX is very easy to use. However, for the context we are addressing (Domain oriented complex business applications), **we strongly recommend the use of WCF** for its greater flexibility regarding technical options (standards, protocols, formats, etc.).Ultimately it is much more powerful than ASP.NET .ASMX web services.

## 19.4.- Types of WCF Service deployment

The Distributed Services (which is ultimately the whole server application) can be deployed at the same physical tier (same servers) being used for other layers such as a web presentation layer or it can be deployed in a separate tier (other servers) specifically for application/business logic. This last option is often required by security policies or even for scalability reasons (under certain special circumstances).

In most cases, the Service layer will reside in the same level (Servers) as the Domain layers, Application layers, Infrastructure layers, etc. to maximize performance. If we separate the layers into different physical tiers we are adding some latency caused by remote calls. Consider the following guidelines:

- **Deploy the Service Layer in the same physical tier as the Domain, Application, Infrastructure layers, etc**., to improve performance of the application, unless there are security requirements and policies that prevent it. This is the most common case for N-Tier architectures with RIA and Rich Clients.

www.EBooksWorld.ir

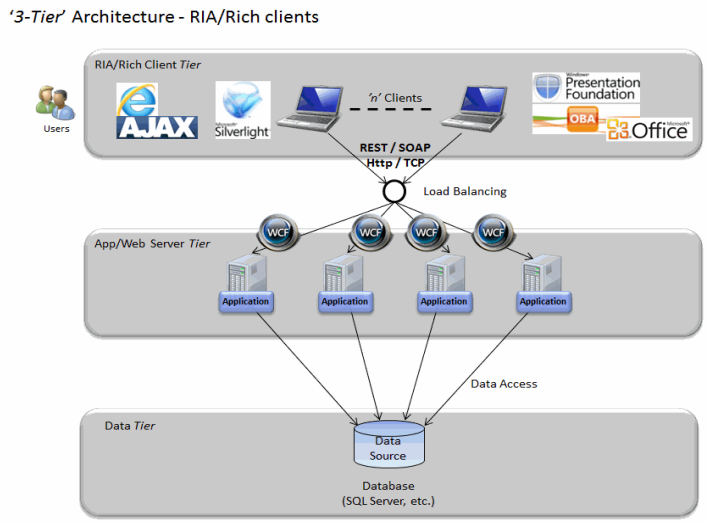



**Figure 12.- RIA/Rich Clients remotely accessing to WCF services**

- **Deploy the Service layer in the same physical tier as the Presentation layer if this layer is a Web presentation layer (like ASP.NET),** to improve application performance. Separating it from the ASP.NET web tier should only be done for security reasons or because of certain special scalability reasons that are not so common and must be demonstrated. If the Web services are located in the same physical level as the consumer, consider using *named-pipes* as communications protocol. However, another option in this case could be not to use Web Services and using objects directly through the CLR. This is probably the option that offers the best performance. It is pointless to use Distributed Services if we consume them from within the same machine. According to **Martin Fowler**: '*The first law of distributed programming, is "Do not distribute" (unless absolutely necessary).*' However, this approach could be preferable at times for the sake of homogeneity, if we do not want to maintain several versions of the same software, and we prefer to maintain fully SOA software.

www.EBooksWorld.ir

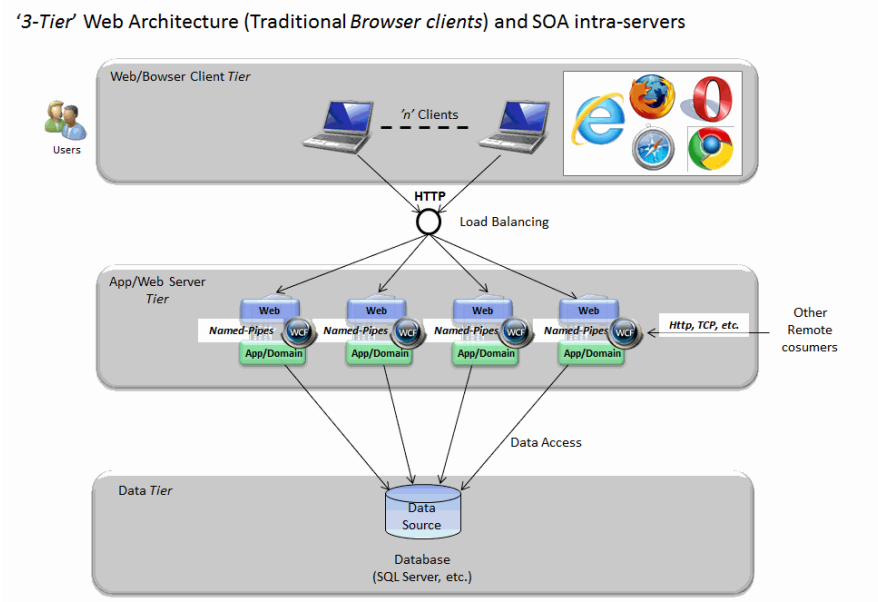



**Figure 13.- App-Web with intra-server WCF services, reusable for external calls from other remote consumers**

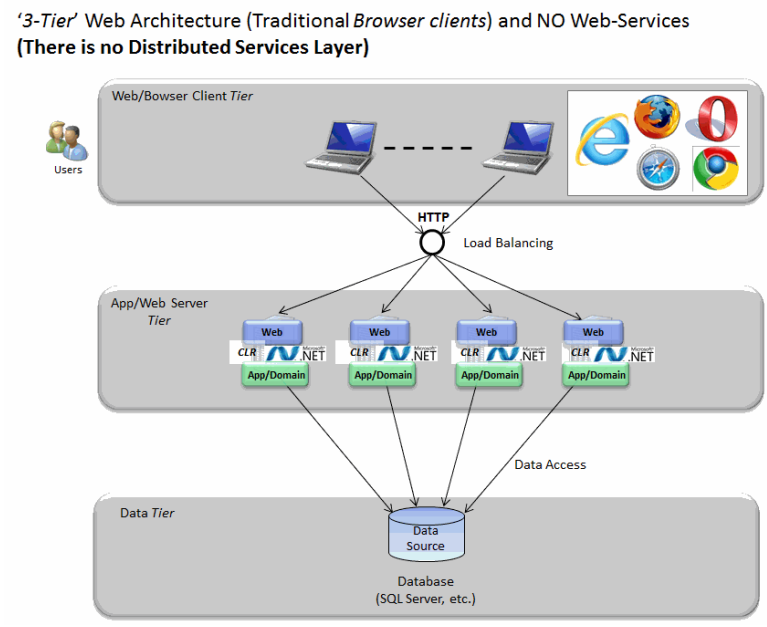
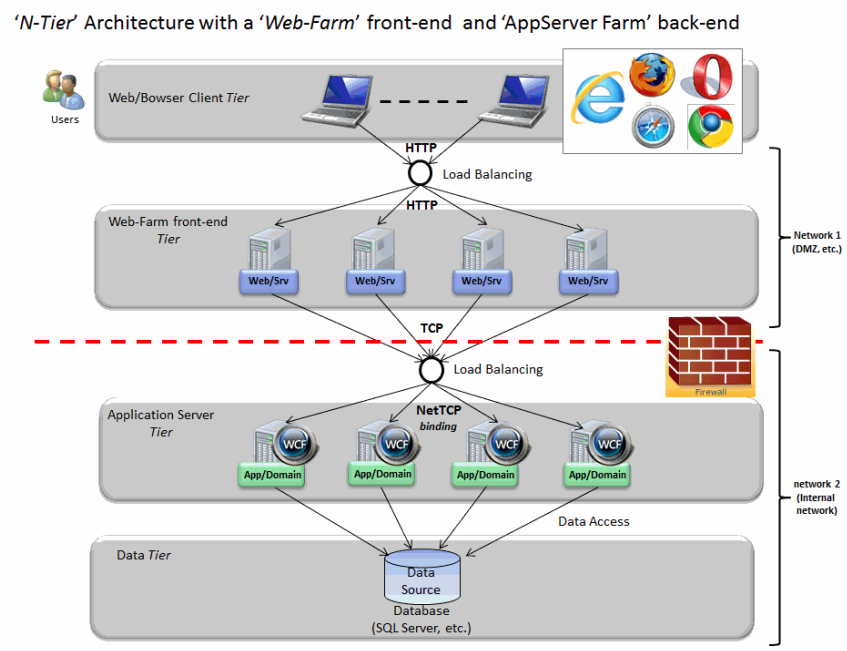



**Figure 14.- App-Web without Distributed Service layer**

www.EBooksWorld.ir

- **Deploy the Service layer in a different physical layer than the Web presentation layer**. In certain situations separating visual Web front-ends from distributed services back-ends may increase scalability, although this must be proven with a load test. Any introduction of remote communication is, by default, a reason for loss of performance due to latency introduced in communications, so the contrary must be proven if you want to separate layers in different physical servers. Another reason for separating the visual Web front-end (ASP.NET) from the Applications Server (Web services) may be for security reasons and corporate policies, such as having separate public and private networks for internal components. In this case, there is no problem in separating these tiers.




- If consumers are .NET applications within the same internal network and highest performance is desired, consider TCP binding in WCF for communications.

- If the service is public and interoperability is required, use HTTP

www.EBooksWorld.ir

# 20.- INTRODUCTION TO WCF (WINDOWS COMMUNICATION FOUNDATION)

***'Windows Communication Foundation'*** (called '***Indigo***' earlier in its BETA phase in 2004-2005) is the strategic platform in .NET technologies to develop '**Connected Systems**' (Distributed applications, etc.). It is a communication infrastructure platform constructed from the evolution of Web service architectures. The support of advanced Services in WCF provides programmatic messaging that is **secure, reliable, transactional and interoperable** with other platforms (Java, etc.). Mostly, WCF is designed following the guidelines of the ***'Service Oriented' (SOA)*** model. Finally, WCF unifies all the different technologies of distributed systems that Microsoft provides on a single **componentizable, decoupled and extensible architecture,** and is capable of **changing** in a declarative manner **transport protocols, security, messaging** patterns, type of **serialization** and **hosting models**.

It is important to note that WCF is re-designed from scratch and not based on ASMX 2.0 (Basic Web Services of ASP.NET). It is really much more advanced than ASMX 2.0.

**WCF** in turn is part of ***.NET Framework, starting on .NET 3.0 (2006)***.

Below, we show a scheme of the evolution and unification of Microsoft protocol *stacks*, as mentioned earlier:

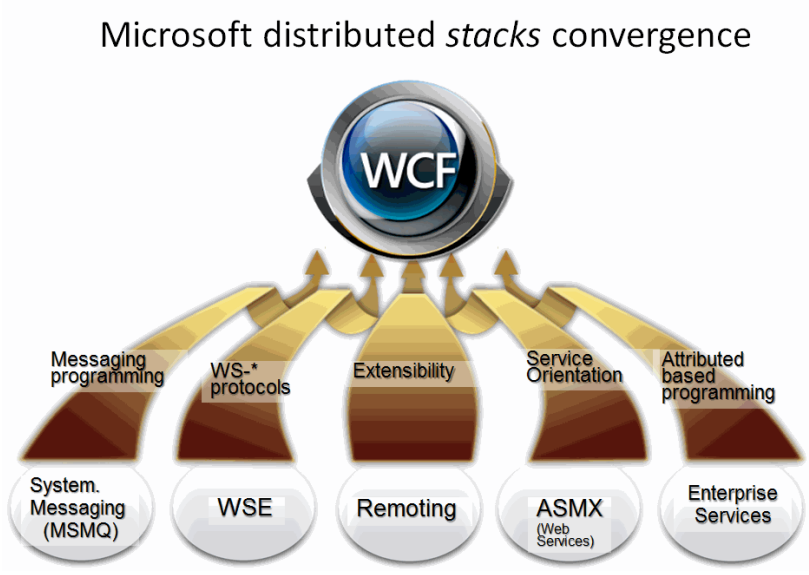



**Figure 15.- Evolution and distributed stacks convergence**

www.EBooksWorld.ir

The practical goals of WCF is that there are no design and architecture decisions being made on distributed technologies (**ASMX** vs. **Remoting** vs. **WSE**, etc.), depending on the type of application. This is something we had to do before the advent of WCF, but sometimes due to changing application requirements there could be problems in areas not supported by the technology initially chosen.

The main **goal** of WCF is **to be able to perform an implementation of any combination of requirements with a single communication technological platform**.

In the following Table, we show the different characteristics of the different previous technologies and how, with WCF, they are unified into a single technology:

**Table 10.- Communication Technology features**

| | ASMX | .NET Remoting | Enterprise Services | WSE | MSMQ | WCF |
|---|---|---|---|---|---|---|
| Interoperable Basic Web Services | X | | | | | X |
| .NET communications | | X | | | | X |
| Distributed transactions, etc. | | | X | | | X |
| WS-* Specifications | | | | X | | X |
| Message queues | | | | | X | X |

For example, if the intention is to develop a Web Service with reliable communications that supports sessions and propagation of transactions between different services and even to extend it depending on the types of messages that come into the system, this can be done with WCF. Although it is not entirely impossible to do this with previous technologies, it would require a lot of development time of and a strong knowledge of all the different communication technologies (and the different programming schemes, etc.) Therefore, another goal of WCF is to be more productive not only in the initial development stage but also in developments that subsequently will have requirement changes (functional and technical ones). It will only have to learn a single programming model that unifies all the positive aspects of ASMX, WSE, Enterprise Services (COM+), MSMQ and .Net Remoting. Moreover, we should not forget that WCF is Microsoft's "flagship implementation" for the WS-* specifications, which have been elaborated and standardized by different manufacturers (including Microsoft) during the last eight years in order to achieve true interoperability across different platforms and programming languages (NET, Java, etc.). It can however, perform advanced aspects (encryption, signature, propagation of transactions between different services, etc.).

Finally, it is important to emphasize that **WCF is interoperable,** based on the SOAP standards of ***WS-****, or on REST.

www.EBooksWorld.ir

## 20.1.- The 'ABC' of Windows Communication Foundation

The acronym 'ABC' is essential to WCF, as it matches the basic concepts about how WCF Services '*End-Points*' are composed.

The '*End-Points*' are basically the communications ends when using WCF and are therefore also the entry points to the services. An '*End-Point*' is internally quite complex since it offers different possibilities regarding communication, addressing, etc.

To be precise and returning to the acronym ABC as something to remember, an '*End-Point*' is composed by 'ABC', that is:

- "**A**" for ***'Address'***: **Where** is the service located?
- "**B**" for ***'Binding'***: **How** do I talk to the service?
- "**C**" for ***'Contract'***: **What** does the service offer?

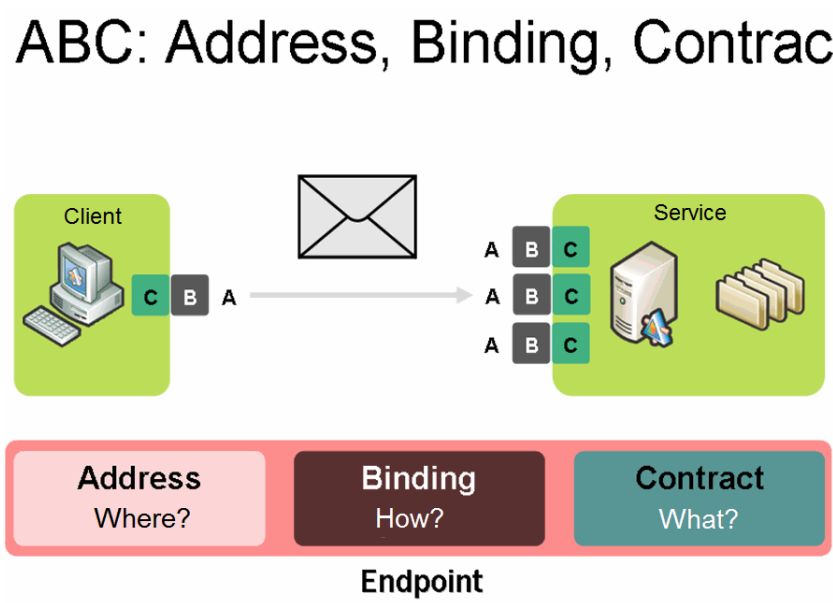



**Figure 16.- Address, Binding, Contract**

Keep in mind that these three elements are independent and there is a great amount of decoupling between them. A contract may support several bindings, and a binding may support several contracts. A Service may have many '*endpoints'* (contract bound to address) coexisting and available at the same time. For example, a service can be exposed through HTTP and SOAP 1.1 to offer the highest interoperability and, at the same time, expose it through TCP and binary format to offer the highest performance.

www.EBooksWorld.ir

The result would be two end-points that can reside at the same time on the same Service.

**Address**

Like a web page or web service, all WCF services must have an address. The thing is that, unlike the previous ones, a WCF service can provide addresses for the following protocols:

- HTTP
- TCP
- *NamedPipes*
- PeerToPeer (P2P)
- MSMQ

**Binding**

A **binding** specifies how to access the service. It defines, among others, these concepts:

- Transport protocol used: HTTP, TCP, NamedPipes, P2P, MSMQ, etc.
- Message codification: plain text, binary, etc.
- WS-* Protocols to be applied: transactional support, messages security, etc.

**Contract**

The service contract represents the interface offered by this service to the outside world. Therefore, the methods, types and operations intended to be exposed to the service consumers are defined at this point. Usually, the service contract is defined as an interface type class to which the attribute *ServiceContractAttribute* is applied. The business logic of the service is codified implementing the interface designed earlier.

The simplified architecture of the WCF components is shown in the following diagram:

www.EBooksWorld.ir

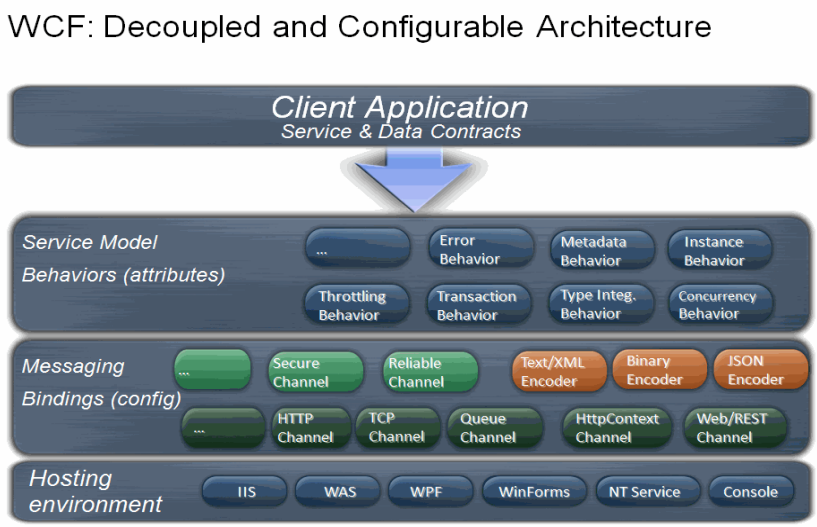



**Figure 17.- WCF is decoupled and configurable**

And in this other diagram we can see **the WCF decoupling and combinations**:

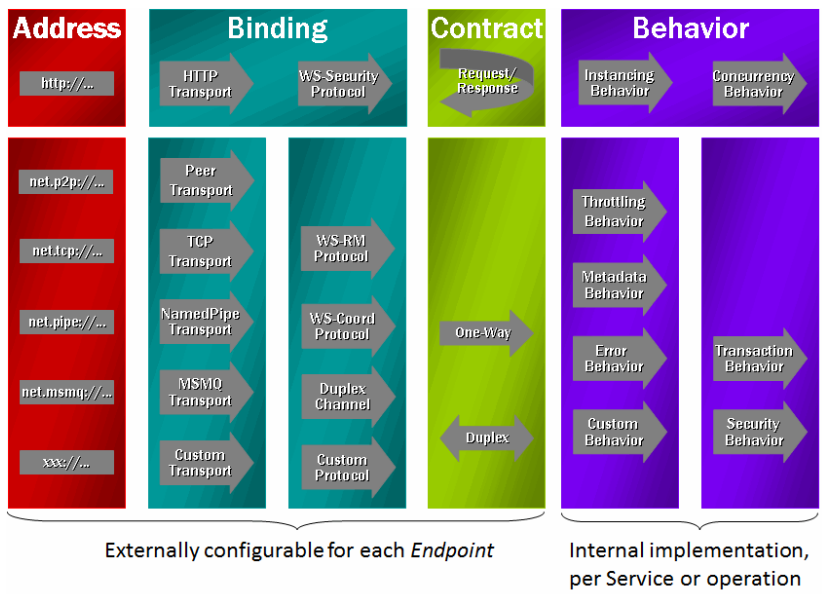



**Figure 18.- WCF Decoupling and combinations**

www.EBooksWorld.ir

**"ABC"** also means that the development and configuration of a WCF service are performed in three steps:

1.- The contract and its implementation are defined.

2.- A binding is configured that selects a transport along with the features of quality of service, security and other options. It is equal to the "Messaging" and "Bindings" in the previous diagram.

3.- The service and the endpoint are deployed in a hosting environment (Execution process. Equal to the Hosting Environments in the previous diagram).

We will see these steps in detail in the following sections.

## 20.2.- Implementing a WCF service

The development of a basic WCF service, that is, one in which we simply implement communication between the client and the service, is relatively simple. It is not as simple as developing a basic web service in ASP.NET (ASMX), because WCF decoupling has certain implementation costs. On the other hand, this decoupling is a great benefit for the future, since we are able to change key features of our development (communication protocol, format, etc.) by only changing declarations and without changing our programming model or technology.

Let's analyze these steps.

**Definition of Contract of a WCF service**

Both the contract and the implementation of a service are performed in a .NET class library (.DLL). This is equivalent to the '***Service Model***' in the above diagram.

Service contracts are modeled in .NET using regular .NET interfaces (using C#, VB.NET, etc.). We can use any .NET interface as a starting point, such as the one shown below:

```
Namespace MyCompany.MyApplication.MyWcfService
{
   public interface IGreeting
   {
       string SayHello(string name);
   }
}
```

To convert this normal .NET interface into a service contract, we simply have to "decorate" it with certain attributes, specifically with the attribute [ServiceContract] for the interface and then the attribute [OperationContract] for each method we want to expose, such as a service operation, as shown below:

www.EBooksWorld.ir

```
Using System.ServiceModel;

namespace MyCompany.MyApplication.MyWcfService
{
   [ServiceContract]
   public interface IGreeting
   {
      [OperationContract]
      string SayHello(string name);
   }
}
```

These attributes affect the mapping between the .NET and the SOAP world. WCF uses the information found in the service contract for dispatching and serialization. When *dispatching*, there is a process to decide which method to call for each incoming SOAP messages. *Serialization* is the process for mapping between the data found in the SOAP message and the .NET objects used for holding the data. This mapping is controlled by an operation *data contract*. WCF sends messages based on the action of the message. Each method in a service contract is automatically assigned an action value based on the *namespace* of the service and the name of the method.

A WCF service may be implemented with a single class (without interface) but we highly recommend separating the contract in an interface and simply placing the service internal implementation in a class. This offers several advantages, such as:

- It allows us to modify the service implementation without breaking the contract.
- It enables service versioning by establishing new interfaces.
- An interface can extend/inherit from other interface.
- A single class may implement several interfaces.

**Implementing the WCF service class**

We can now develop the service (the code we want to execute) by simply implementing the .NET interface in a .NET class:

```
Using System.ServiceModel;

namespace MyCompany.MyApplication.MyWcfService
{
   Public class Greeting : IGreeting
   {
      Public string SayHello(string name)
      {
         Return "Welcome to this book " + name;
      }
   }
}
```

www.EBooksWorld.ir

By doing this, we ensure the **Greeting** class supports the service contract defined by the **IGreeting** interface. When an interface is used to define a service contract, we do not need to apply any attribute associated with the contract to the class. However, we can use attributes of the **ServiceBehavior]** type to affect its behavior/local implementation:

```
usingSystem.ServiceModel;

namespace MyCompany.MyApplication.MyWcfService
{
   ... // We omit the interface definition

   [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
                    ConcurrencyMode = ConcurrencyMode.Multiple)]
   Public class Greeting : IGreeting
   {
          ...
```

This example specifically tells WCF to manage a singleton instance (the same instantiated object of the service is shared between all the clients) and also to allow a *multi-thread* access to the instance (therefore we should control the concurrent accesses to the shared memory zones of such object, through critical sections, traffic lights, etc.). We can also use the **[OperationBehavior]** attribute to control behaviors at the operation/method level.

Behaviors affect the processing within the host (execution process of the service, we will see this later in more detail), but they do not impact on the service contract. The behaviors are one of the main WCF extensibility points. Any class implementing **IServiceBehavior** can be applied to a service through the use of its own attribute ('*custom*') or through a configuration element.

### Defining Data Contracts

When calling a service, WCF automatically serializes the standard input and output parameters of .NET (basic data types such as string, int, double, and even more complex types of data such as EF entities or DataTable and DataSet.)

However, on many occasions, our WCF methods have input parameters or return value defined by custom classes in our code (DTOs, *custom entity* classes or any other custom data class). To be able to use these custom classes in WCF operations/methods, they need to be serializable. Therefore, the recommended mechanism is to label the class with the **DataContract** attribute and their properties with the **DataMember** attribute. If you want to hide this property from the consumers, you will only have to abstain from labeling it with the **DataMember** attribute.

To illustrate this point, we have modified the example above by changing the *SayHellow()* method so that it takes the *Person Entity* class as the input parameter.

www.EBooksWorld.ir

```
Using System.ServiceModel;
Namespace MyCompany.MyApplication.MyWcfService
{
    //SERVICE CONTRACT
    [ServiceContract]
    public interface IGreeting
    {
        [OperationContract]
        string SayHello(Person person);
    }

    //SERVICE
    Public class Greeting : IGreeting
    {
        Public string SayHellow(Person  person)
        {
            return "Welcome to this book" +
                                          person.Name + " " +
                                          person.LastName;
        }
    }

    // DATA CONTRACT
    [DataContract]
    publicclassPersonEntity
    {
        string  name;
        string _lastName;

        [DataMember]
        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        [DataMember]
        public string LastName
        {
            get { return _lastName; }
            set { _lastName = value; }
        }
     }

   }
```

## 20.3.- Service Hosting and configuration (Bindings)

After we have defined our service, we should select a host (process) where our service can be executed (do not forget that, so far, our service is simply a .NET class library, a .DLL that cannot be executed on its own). This hosting process can be almost any type of process; it may be IIS, a console application, a Windows service, etc.

www.EBooksWorld.ir

To decouple the host from the service itself, it is convenient to create a new project in the Visual Studio solution that defines our host. Different projects could be created, depending on the hosting type:

- Hosting in IIS/WAS/AppFabric: **Web Application project**
- Hosting in an executable app-console: **ConsoleApplication project**
- Hosting in a 'Windows Service': **Windows Service project**

WCF flexibility allows us to have several host projects hosting the same service. This is useful because we can host it in a console application during the development stage of the service. This facilitates debugging and, subsequently, a second project of the web type can be added to prepare the deployment of the service in an IIS or Windows service.

As an example, we will create a console project as a hosting process and we will add a reference to the **System.ServiceModel** library.

Below, in the Main class, the **ServiceHost** object should be instantiated by passing the *Greeting* service type as a parameter.

```
using System.ServiceModel;
static void Main(string[] args)
{
  Type serviceType = typeof(Greeting);
  using (ServiceHost host = new ServiceHost(serviceType))
  {
    host.Open();

    Console.WriteLine("The Service is available-WCF of Application.");
    Console.WriteLine("Press a key to close the service");
    Console.ReadKey();

    host.Close();
  }
}
```

In the example above we can see how the service host is defined. If we start it, the process will be running until the user presses a key. At the same time, the WCF Service will be 'listening' and waiting for any request:

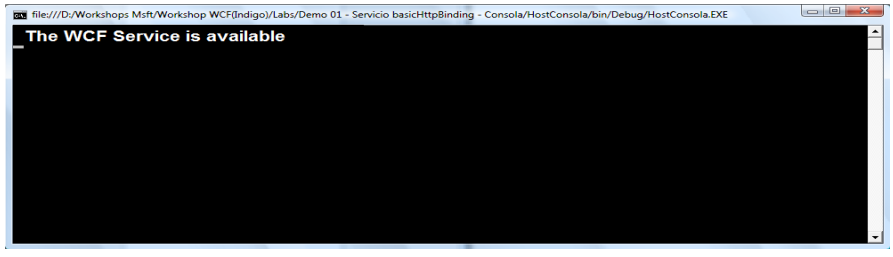



**Figure 19.- WCF Service hosted in a Command-Console App (Just for demos/tests)**

www.EBooksWorld.ir

This execution of the console application by hosting the WCF service actually cannot be performed until we also have the configured service through the XML sections of the .config, as we shall see in a moment. Once we have configured the host with the endpoint information, WCF can then build the necessary runtime to support our configuration. This happens at the moment Open() is called on a particular ServiceHost, as we can see in the code C# above (host.Open(); )

After ending the method Open(), the WCF runtime is already constructed and ready to receive messages at the specified address based on each *endpoint*. During this process, WCF generates an *endpoint listener* for each configured endpoint. (An *endpoint*-listener is a piece of code that actually listens and waits for incoming messages by using the specified transport and address).

Information can be obtained on the service at run time through the object model exposed by the **ServiceHost** class. This class allows us to obtain anything we want to know about the initialized service, such as what *endpoints* are exposed and what *endpoints listeners* are currently active.

It is important to emphasize that the use of "console applications" as hosting process for WCF services should be used only for proof of concepts, demos, and testing services and never, of course, for WCF services in production. A deployment of WCF service in a production environment would normally be performed in a hosting process of any of the following types:

**Table 11.- Possibilities of WCF Service Hosting**

| Context | Hosting process | Protocols | Req. Operative System |
|---|---|---|---|
| Client-Service | IIS 6.0 | http/https | • Windows Server 2003<br>• Windows XP (or later Windows version) |
| Client-Service | Windows Service | TCP<br>Named-Pipes<br>MSMQ<br>http/https | • Windows Server 2003<br>• Windows XP (or later Windows version) |
| **Client-Service** | **IIS7.x-WAS AppFabric** | **Tcp<br>Named-Pipes<br>MSMQ<br>http/https** | • **Windows Server 2008 or later version**<br>• Windows Vista |
| Peer-To-Peer | WinForms or WFP client | Peer-to-Peer | • Windows Server 2003<br>• Windows XP (or later Windows version) |

www.EBooksWorld.ir

**IIS 7.x, in Windows Server** is usually the preferred option for production environments.

## 20.4.- WCF Service Configuration

However, in order to run a service we first need to configure it. This configuration may be performed in two ways:

- *Hard-coded*: Configuration specified by C#/VB code.
- Service Configuration based on configuration files (**.config*). This is the most highly recommended option, since it offers flexibility to change service parameters such as addresses, protocols, etc. without recompilation. Therefore, it facilitates deployment of the service and ultimately provides greater simplicity because it allows us to use the **Service Configuration Editor** utility provided by Visual Studio.

So far, we had already defined the contract (interface), implementation (class) and even the hosting process within the '**ABC' model.** But now it is necessary to associate this contract with a specific binding, address and protocol. This will normally be done within the .config file (app.config if this is our own process or web.config if IIS is the selected *hosting* environment).

So, a very simple example of an app.config XML file is the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
...
<system.serviceModel>
<services>
<service name="MyCompany.MyApplication.MyWcfService.Greeting">
<endpoint

address="http://localhost:8000/ServiceGreeting/http/"
                  binding="basicHttpBinding"

contract="MyCompany.MyApplication.MyWcfService.IGreeting"
                  bindingConfiguration="" />
</service>
</services>
</system.serviceModel>
...
</configuration>
```

Of course, app.config or web.config may have additional XML sections. In the app.config above, we can clearly see the '**ABC of WCF**': **Address, Binding and Contract.**

www.EBooksWorld.ir

Keep in mind that in this example our service is "listening" and offering service via HTTP (specifically through **http://localhost:8000**). However we are not using a web server such as IIS; this is simply a console application or it could also be a Windows service that also offers service through HTTP. This is because WCF provides internal integration with **HTTPSYS**, which allows any application to become an ***http-listener***.

The ".config" file configuration can be done manually by writing the XML directly into the .config (we recommend doing it like this, because this is how you really learn to use bindings and their configurations).Or, if we are starting with WCF or even if there are things we do not remember, such as XML configuration, we can do it through the Visual Studio wizard. This wizard is available in Visual Studio.

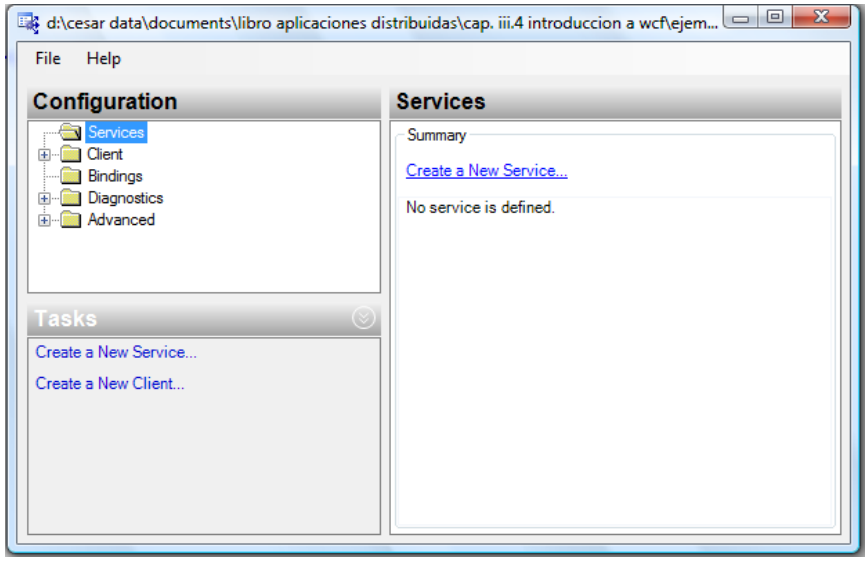



**Figure 20.- WCF Configuration Wizard of Visual Studio**

To do this in our example console project once we have added an *app.config* file, right-click on this .config file and select *"Edit WCF Configuration…"* The **Service Configuration Editor** window will be displayed.

www.EBooksWorld.ir

# 21.- IMPLEMENTATION OF WCF SERVICE LAYER IN N-LAYER ARCHITECTURE

In our sample solution there is a projects solution tree similar to the following, where we highlight the location of projects implementing the WCF service:

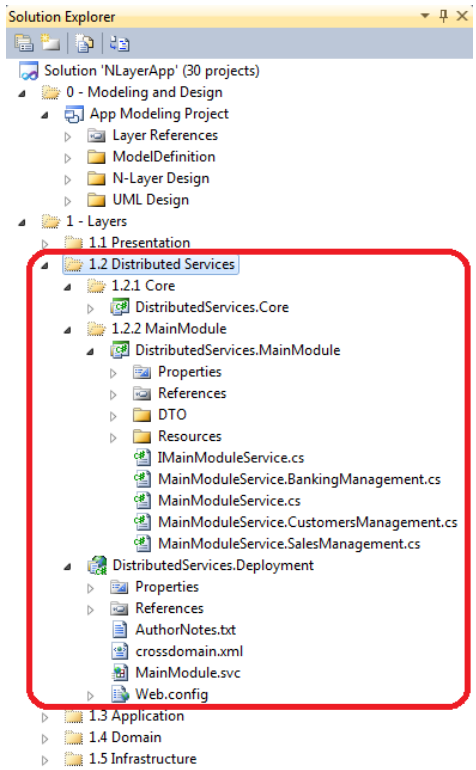



**Figure 21.- WCF service projects location tree**

As shown above, for each application the Distributed Service Layer will be comprised by a single hosting project called '*DistributedServices.Deployment*' (one by default, but there could be several types of hosting, depending on the needs). In this case, we chose a Web project that will be hosted in IIS server (or *Cassini*, the VS Web Server, during development environment).

For each vertical module of the application there is an implementation assembly of the WCF service. In this case, there is a single vertical application module, so we have a single WCF service implementation assembly called "*DistributedServices.MainModule*".

www.EBooksWorld.ir

In addition, there may be a class library with reusable code for the different services of different corresponding modules. In this example, the library is called "*DistributedServices.Core*" and contains certain reusable code for WCF *Faults* and Error management.

## 22.- TYPES OF DATA OBJECTS TO COMMUNICATE WHEN USING WCF SERVICES

As we explained in the chapter on Distributed Services (at the logical level), and in order to unify options, the usual types of data objects to communicate when passing from one tier to another remote tier within an N-Tier architecture are:

- Scalar values
- DTOs (*Data Transfer Objects*)
- Serialized Domain Entities
- Sets of records (disconnected artifacts)

With these options, at the technology level we would see the following mapping:

**Table 12.- Mapping Options**

| Logical object type | .NET technology |
|---|---|
| Scalar values | • String, int, etc. |
| DTOs (Data Transfer Objects) | • Custom .NET data classes gathering several entities |
| Serialized Domain Entities depending on the data access infrastructure technology | • Simple/native entities of Entity Framework (this was the only direct possibility in EF 1.0) |
| Serialized Domain Entities NOT depending on the data access infrastructure technology | • *POCO* entities mapped to Entity Framework entity model (available from EF 4.0)<br>• *SELF-TRACKING* STE entities mapped to Entity Framework entity model (available from EF 4.0)<br>• Other *POCO* entities mapped to other ORMs such as NHibernate, etc. |
| Sets of records depending on the data access infrastructure technology (disconnected devices) Concept 1 | • **DataSets, DataTables** of ADO.NET |

Although simple/native EF Entities are normally not the best pattern for N-Tier applications (they have direct dependence on EF technology), it was the most viable

www.EBooksWorld.ir

option in the first version of EF (EF 1.0). However, EF4 significantly changes the options for N-Tier programming. Some of the new features are:

- New methods that support offline operations, such as *ChangeObjectState* and *ChangeRelationshipState*, which change an entity or relation to a new state (for example, added or modified); *ApplyOriginalValues*, which allows us to establish original values of an entity and the new event *ObjectMaterialized* that is activated each time the *framework* creates an entity.
- Compatibility with POCO entities and foreign key values in the entities. These features allow us to create entity classes that can be shared between implementation of Server components (Domain Model) and other remote levels that may not even have the same version of *Entity Framework* (.NET 2.0 or Silverlight, for example). The POCO entities with foreign keys have a simple serialization format that simplifies the interoperability with other platforms, such as JAVA. The use of external keys also allows a much simpler concurrency model for relationships.
- T4 templates to customize generation of code of the POCO or STE classes (*Self-Tracking Entities*).
  The Entity Framework team has also used these features to implement the STE entity pattern (Self-Tracking Entities) in a T4 template. Therefore, this pattern is much easier to use because we will have a code generated without implementing it from scratch.

In general, when making decisions on design and technology about data types to be handled by the Web Services with these new capabilities in EF 4.0, we should evaluate the pros and cons of the aspects from **Purist Architecture approaches** (decoupling between Domain entities of data managed in Presentation Layer, Separation of Concerns, Service Data Contracts different from Domain Entities) **versus Productivity and *Time-To-Market*** (optimistic concurrency management already implemented without having to develop conversions between DTOs and Entities, etc.)

www.EBooksWorld.ir

**If we place the different types of patterns we can use for implementing data objects to communicate in N-Tier applications (data traveling via the network thanks to the Web Services), we would have something like this**:

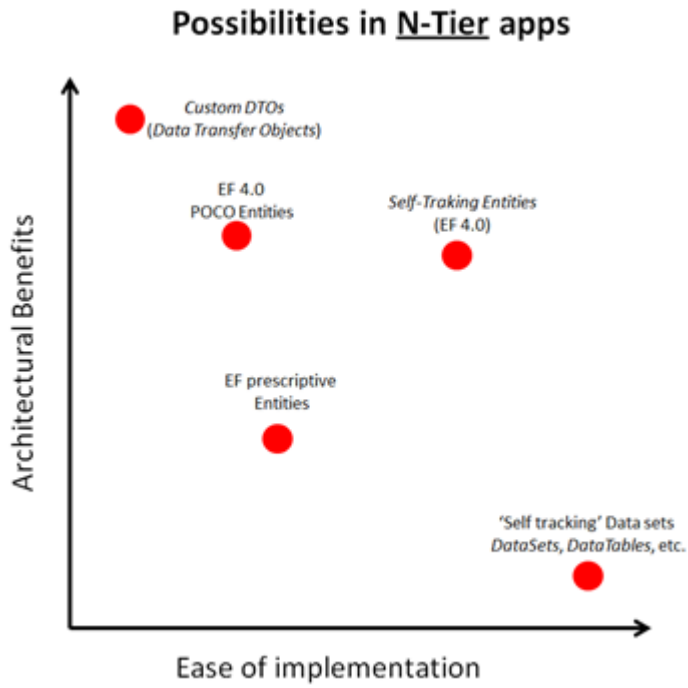



**Figure 22.- Balance between Ease of Implementation and Architectural Benefits**

The right pattern for each particular situation depends on many factors. In general, DTOs (as introduced in a logical way in the initial section of current chapter), provide many architectural advantages, but have a high implementation cost. The "Record change sets" (DataSets, DataTables, etc.) do not have many advantages at the Architecture level, because they do not meet the PI principle (*Persistence Ignorance*) or provide alignment with logical Entities of a model, etc. However they are very easy to implement (e.g., the ADO.NET DataSet is ideal for applications that are not as complex at a business logic level and very data oriented, that is, Architectures that are not Domain Oriented or "non DDD").

In short, we recommend a pragmatic and streamlined balance between these concerns (Productivity *vs*. Purist Architecture). **One convenient choice (using EF 4.0) could be the STE (Self-Tracking Entities)** if there is end to end control of the application (Client Consumer and Server). The other option would be **steering towards the DTO** pattern as the situation requires it, like if we need to decouple the domain entity model from the data model used by the presentation layer.

The STE will provide high productivity while obtaining very significant architecture benefits. In addition, they represent a much better balance than the

www.EBooksWorld.ir

DataSets or prescriptive/native entities of EF. On the other hand, **DTOs are definitely the best choice as an application gets bigger and more complex**. This is also true if we have requirements that cannot be met by the STE, such as different development changes speed in the Application/Domain Layers regarding the Presentation layer, so that makes it desirable to decouple the domain entities from the presentation layer data model.

These two types of serialized objects traveling between physical levels (**STE** or DTO) are currently probably the most important ones to consider in **N-Tier** architectures that are at the same time **DDD** Architectures (for complex *Domain Driven Design* applications with a lot of domain and application logic coordinated by Services).

**OData (WCF Data Services) and DDD**

On the other hand, if we are building a long term *Data Driven SOA* Service, where we might not know who is going to consume this Service (maybe published in the Internet so it needs the best interoperability available), then the best approach would be **OData (WCF Data Services)**, but in this case, it should be a *Data Driven Service* (Not DDD).

In a DDD scenario, regarding Reads/Queries, OData would be ok in many situations (We could even create a special OData endpoint for external publishing). But regarding updates and business logic, it is a different matter.

OData is oriented to update directly against data entities, from the client/consumer (using http verbs like UPDATE, INSERT, DELETE directly against data entities).

But, when dealing with a DDD app/service, we are dealing with Application/Domain Services (commands, tasks, transactions, etc.). In DDD we do not directly expose entities for data updates, we work against business/application/domain services. For doing so, OData is not the best choice; actually, WCF Data Services is oriented to publish data entities, directly.

Also, OData entites are anemic-entities (no own domain logic within entities themselves). This is also contrary to DDD where we do not want to have anemic domain models (See chapter 5).

In short, OData is great for N-Tier Data-Driven Apps (and a very strategic technology for Microsoft), but it is not ideal for N-Tier DDD Apps, at least for the transactional front-end (Business/Domain Services Façade). Of course, internally we can use OData for accessing/updating 'only data' stuff, engaging to any database/storage. But that will be a 'data only matter' related to the Data Persistence Layer.

Finally, and as we already said, because OData is really strategic for Microsoft, in the future OData could evolve towards many more scenarios further than Data-Driven Services/Apps. Keep an eye on **http://odata.org** for more details.

www.EBooksWorld.ir

**CQRS Architecture and OData**

For a short intro about CQRS, please read the last pages of chapter number 10.

Regarding CQRS, OData (WCF Data Services) fits perfectly with the CQRS Read-Model (where we only want to query a data model), but it does not fit well with the WCF Services attacking the Write-Model, because in this case we are using Commands and executing Business/Domain Services. Commands and Services make the domain entities transparent when executing Business/Domain tasks.

# 23.- PUBLISHING APPLICATION AND DOMAIN LOGIC

The publication of Application and Domain logic in general should not be direct. In other words, we will generally not provide direct visibility of all the classes and methods from the Domain or Application Layer. On the contrary, we should implement a Web Service interface that shows what we want to be used by remote Clients (Presentation Layer or other external applications). Therefore, it is common to create a coarse-grained interface, trying to minimize the number of remote calls from the Presentation Layer.

## 23.1.-Decoupling the Architecture internal layer objects using UNITY

Using UNITY from the Web Services Layer is crucial because it is here, in the WCF Web Services, where we have the entry point to the Application Server components. Therefore, this is where we have to start with the initial explicit use of the UNITY container (explicitly determining the objects to be instantiated, using Resolve<>). UNITY is used as well in the rest of the Layers (Application, Domain, Persistence), but automatically through the dependency injection that we use in constructors. However, the recommendation for a correct and consistent "Dependency injection" is: '**we should only use "Resolve()" at the entry point of our Server (Web Services in an N-Tier application, or Controllers code in ASP.NET MVC for a Web application).**'

**Use the UNITY container explicitly only at the entry point to the Server**: We should only use "**Resolve<>**" at the entry point to our Server (Web Services in an N-Tier application or Controllers code in ASP.NET MVC for a Web application). Other than that, we would be using the IoC container only as a 'ServiceLocator pattern' which in many cases is an anti-pattern.

www.EBooksWorld.ir

Below, we show an example of the WCF Service class that publishes certain Application Layer logic:

```
C#
namespace Microsoft.Samples.NLayerApp.DistributedServices.MainModule
{
    public partial class MainModuleService
    {
```

If we control the Client applications (Consumers), we return a POCO/STE Entity of EF 4.0. If we do not know who will consume the Web Service, a DTO will be more suitable

```
        public Client GetClientByCode(string clientCode)
        {
            try
            {
```

**Root resolution with UNITY**: Use of Resolve<Interface>() of UNITY only at the entry points to the Server, like WCF services methods

```
IClientManagementService clientService =
        ServiceFactory.Current.Resolve<IClientManagementService>();
```

Call to the Application Layer service method.

```
                return clientService.FindClientByCode(clientCode);
            }
            catch (ArgumentNullException ex)
            {
                //Trace data for internal health system and return
specific FaultException here!
                //Log and throw is a known anti-pattern but at this
point (entry point for clients this is admitted!)

            }
        }
    }
   }
```

## 23.2.- Handling Exceptions in WCF Services

By default, the internal exceptions that occur are transformed by WCF into FAULTS (to be serialized so that they reach the consumer client application). However, unless we change it, the information included about the Exception within the FAULT is generic (A message such as "*The server was unable to process the request due to an internal error.*"). That is, for security reasons, no information about the Exception is included, since we could be sending sensitive information relating to a problem that has occurred.

However, when we are *Debugging* and we see the errors that have occurred, we need to be able to send the client specific information on the internal error/exceptions of a server (for example, specific error of a Database access problem, etc.). To do so,

www.EBooksWorld.ir

we should include this configuration in the Web.config file, indicating that we want to include the information on all exceptions in the WCF FAULTS when the type of compilation is 'DEBUG':

```
CONFIG XML

<behavior name="CommonServiceBehavior">
                    ...
<serviceDebug includeExceptionDetailInFaults="true" />

                    ...
</behavior>
```

Include in the FAULTS info on exceptions, only in DEBUGGING mode

In this case, as long as the compilation is in "*debug*" mode, the exception details will be included in the FAULTS.

```
CONFIG XML

<system.web>
<compilation debug="true" targetFramework="4.0" />
</system.web>
```

We set the "debug" compilation → Exceptions Details will be sent

## 23.3.- Hosting WCF Services

We can host a WCF service in any process (.exe), although there are two main types:

- ***Self-Hosting***
    - The Service will be hosted in a custom process (.exe). The hosting process is performed by our own code, either a console application, as we have seen a Windows service or a form application, etc. In this scenario, we are responsible for programming this WCF hosting process. This, in WCF terms, is known as 'Self-Hosting.'
- ***Hosting (IIS/WAS and Windows Server AppFabric)***
    - The process (.exe) where our service will run is based on IIS/WAS. IIS 6.0/5.5 if we are in Windows Server 2003 or Windows XP, or IIS 7.x and WAS only for IIS versions as from version 7.0. Therefore, that process (.exe) will be any IIS pool process.

www.EBooksWorld.ir

The following table shows the different characteristics of each major type:

**Table 13.-Self-Hosting vs. IIS and AppFabric**

| | *Self-Hosting* | IIS/WAS/AppFabric |
|---|---|---|
| **Hosting process** | Our own process, our source code | Process of IIS/WAS. This is configured with a .svc file/page |
| **Configuration file** | App.config | Web.config |
| **Addressing** | The one specified in the EndPoints | This depends on the configuration of the virtual directories |
| **Recycling and automatic monitoring** | NO | YES |

We will review the most common specific types of *hosting* below.

### Hosting in Console Application

This type of hosting is very similar to the one shown above, so we will not explain it again. Keep in mind that it is useful to perform demos, ***debugging***, etc., but it should not be used to deploy WCF services in a production environment.

### Hosting a Windows Service

This is the type of hosting we would use in a production environment if we do not want to/cannot rely on IIS. For example, if the server operating system is Windows Server 2003 (we do not have WAS within IIS), and we also want to use a protocol other than HTTP (for example, TCP, Named-Pipes or MSMQ), then the option we should use for hosting our Service has to be a Windows service developed by us (Service managed by the **Windows Service Control Manager** to start/stop the service, etc.).

In short, the configuration is similar to console application hosting (like the one we have seen before, app.config, etc.), but it varies in the place where we should program the start/creation code of our service, which will be similar to the following (code in a project of the "Service-Windows" type):

```
namespace HostServicioWindows
{
  public partial class HostServicioWin : ServiceBase
  {
     //(CDLTLL) Host WCF
     ServiceHost _Host;

     public HostServiceWin()
     {
       InitializeComponent();
     }
```

www.EBooksWorld.ir

```
        protected override void OnStart(string[] args)
        {
          Type serviceType = typeof(Greeting);

          _Host = new ServiceHost(serviceType);

          _Host.Open();

          EventLog.WriteEntry("Host-WCF  Service",  "The  WCF  Service  is
    available.");
        }

        protected override void OnStop()
        {
            Host.Close();
           EventLog.WriteEntry("Host-WCF Service", "WCF service stopped.");
        }
      }
    }
```

We have to instantiate the WCF Service when the Windows-service starts (OnStart() method), and close the WCF service in the OnStop() method. Other than that, the process is a typical Windows Service developed in .NET, and we will not review this topic any further.

**Hosting in IIS (Internet Information Server)**

WCF services can be activated using IIS with hosting techniques similar to the previous traditional Web-Services (ASMX). This is implemented by using the .svc extension files (comparable to .asmx files), within which the service to be hosted is specified in just one line:

**.svc page content**

```
<%@Service class="MyCompany.MyApplication.MyWcfService.Greeting" %>
```

This file is placed in a virtual directory and the service assembly is deployed (.DLL) within its \bin directory or in the GAC (Global Assembly Cache). When this technique is used, we should specify the *endpoint* of the service in the web.config. However, there is no need to specify the address, since it is implicit in the location of the .svc file:

```
    <configuration>
    <system.serviceModel>
    <services>
    <service type=" MyCompany.MyApplication.MyWcfService.Greeting ">
    <endpoint address=" binding="basicHttpBinding"
    contract=" MyCompany.MyApplication.MyWcfService.IGreeting"/>
    </service>
          ...
```

If you navigate with a *browser* to the .svc file, you can see the help page for this service, showing that a ServiceHost has been automatically created within the

www.EBooksWorld.ir

application domain of the ASP.NET. This is performed by a 'HTTP-Module' that filters incoming requests of the .svc type, and automatically builds and opens the appropriate ServiceHost when necessary. When a website is created based on this project template, the .svc file is generated automatically, along with the corresponding implementation class (located within the *App_Code* folder). It also brings us a configuration of an endpoint by default, in the *web.config* file.

**NOTE about REST Services**:
When developing a *RESTfull* Service using WCF, we usually only rely on URIs and don't need (and it is better not to) the .svc files.

### Hosting in IIS 7.x – WAS

In 5.1 and 6.0 versions of IIS, the WCF activation model is linked to the '*pipeline of ASP.NET*', and therefore, to the HTTP protocol. However, starting with **IIS 7.0** or higher, (starting on **Windows Vista**, **Windows Server 2008 or higher**), an activation mechanism is introduced that is independent from the transport protocol. This mechanism is known as '*Windows Activation Service*' (WAS). Using WAS, WCF services can be published on any transport protocol, such as TCP, Named-Pipes, MSMQ, etc. (as we do when using a '*Self-Hosting*' process).

Use of a different protocol (like TCP) based on WAS, required prior configuration of IIS, both at the Web-Site level and at the virtual directory level.

### Adding a binding to a site of IIS 7.x/WAS

Adding a new *binding* to an IIS/WAS site can be done with the IIS 7.x GUI or from the command line by invoking IIS commands.

The example below shows the configured bindings in a Web Site of IIS 7.x:

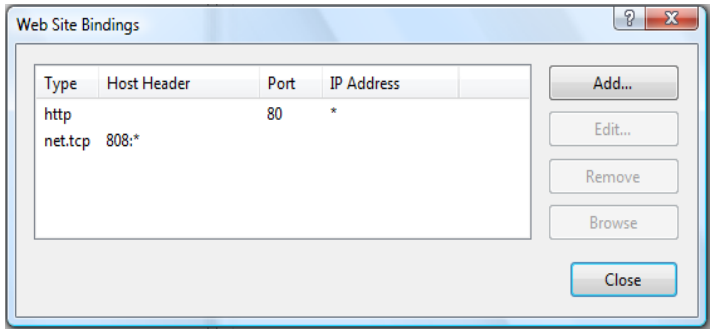



**Figure 23.- Bindings in an IIS 7.x Web Site**

www.EBooksWorld.ir

It can also be added with a command *script*:

```
%windir%\system32\inetsrv\appcmd.exe set site "Default Web Site" -
+bindings.[protocol='net.tcp',bindingInformation='808:*']
```

We should also enable *bindings* later, which we want to be able to use in each virtual directory where our WCF service resides:

```
%windir%\system32\inetsrv\appcmd.exe set app "Default Web
Site/MyVirtualDirectory" /enabledProtocols:http,net.tcp
```

**Hosting in 'Windows Server AppFabric'**

This is similar to the *hosting* in IIS 7.x (WAS), but AppFabric brings extended capabilities for Monitoring WCF and Workflow Services. Nonetheless, the deployment over AppFabric is exactly the same, since AppFabric is based on IIS/WAS infrastructure.

**Hosting in WPF or Windows Form Application**

It may seem strange to want to host a service (which apparently is a server aspect), in a Windows form application or even in a WPF form application (Windows Presentation Foundation). However, this is not so rare since in WCF there is a type of binding that is '*Peer-to-Peer*', where client applications are WCF services at the same time. In other words, all the applications talk to each other because they are WCF services. There are many examples of 'Peer-to-peer' communications on the Internet, such as file sharing software, etc. This is 'yet another type' of *Self-Hosting*.

# 24.- WCF SERVICE DEPLOYMENT AND MONITORING IN WINDOWS SERVER APPFABRIC (AKA DUBLIN)

***Windows Server AppFabric*** has two very different main areas:

- **Service Hosting and Monitoring** (Web services and Workflow Services), called '*Dublin*' during beta version.
- **Distributed Cache** (called '*Velocity*' during its beta version)

The AppFabric Cache is explained in the chapter on '*Cross-cutting Layers*'.

The '**Service Hosting and monitoring**' capabilities are related to the WCF 4.0 Services and Workflow-Services deployment.

This part brings us the added value of a series of functionalities for easy **deployment** and mainly for **monitoring**. Now, with AppFabric, an IT professional (no

www.EBooksWorld.ir

need to be a developer) may localize issues in a WCF service application (or validate correct executions) simply by using the IIS 7.x Management Console. Before the appearance of *Windows Server AppFabric*, monitoring WCF services had to be done exclusively by developers and at a much lower level (tracing, *debugging*, etc.). For an ITPro, the WCF Services of any application were "black boxes" with unknown behaviors.

Actually, *Hosting* a Service on AppFabric does not create a completely new hosting environment. Instead, it is based on IIS 7.x and WAS (*Windows Process Activation Service*) but it adds certain execution and WCF service management capabilities including *Workflow-Services*. The following diagram shows *Windows Server AppFabric* architecture.

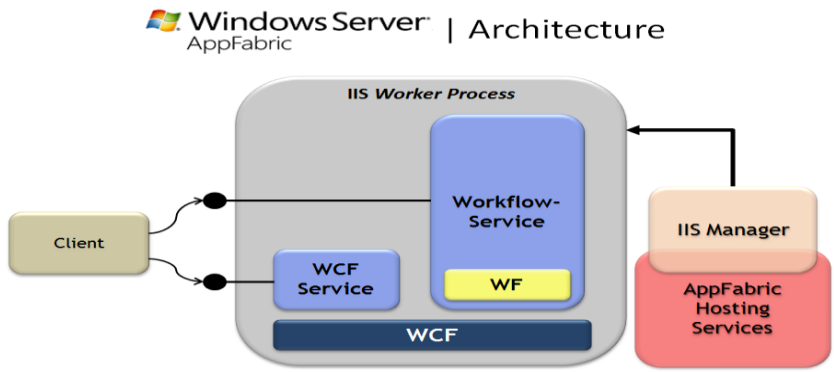



**Figure 24.- 'Windows Server AppFabric Architecture'**

## 24.1.- Windows Server AppFabric Installation and Configuration

Logically, the first thing we should do is install *Windows Server AppFabric*, whether from the public *download* or using the '*Web Platform Installer*'. After installing 'the bits', we should configure it with the '*Windows Server AppFabric Configuration Wizard*', where the aspects related to Cache and WCF monitoring are configured. In this case we only need to review the required points in order to configure WCF Services management and monitoring. We now ignore the cache configuration.

First, we should configure the monitoring database to be used, as well as the credentials for such access.

www.EBooksWorld.ir

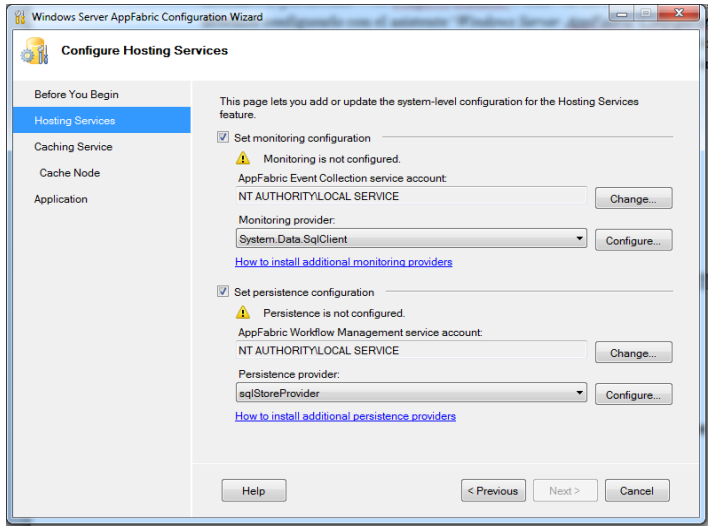



**Figure 25.- Hosting Services configuration**

At the 'hosting services' level we need two SQL Server databases; one to monitor any WCF service and Workflow-Services (called '*Monitoring Store*') and a second database for the persistence of workflow services execution.

In the example, we used *SQL Server Management Studio* to create an empty database (called *AppFabricMonitoringDb*) for the monitoring database ('*Monitoring Store*') in a local SQL Server Express. However, a remote Standard SQL Server could also have been used instead. Once the empty database is available, we specify it in the AppFabric configuration, as shown in the diagram below.

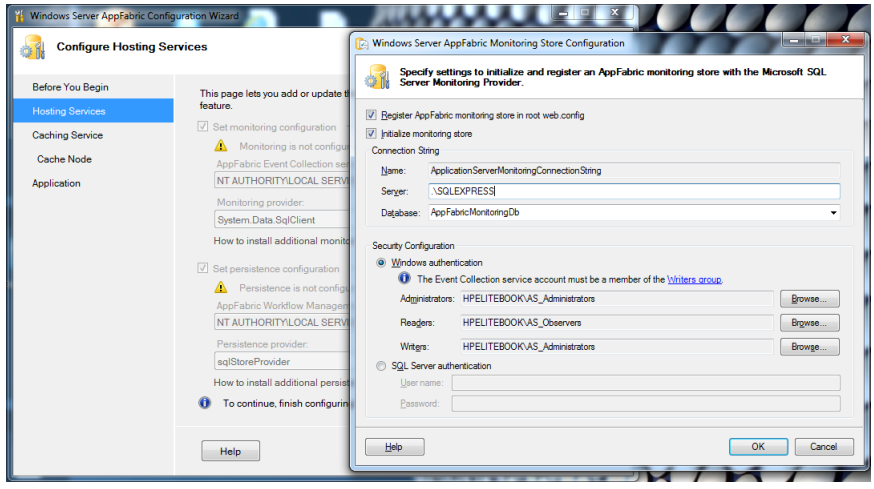



**Figure 26.- Hosting Services - Monitoring Database config**

www.EBooksWorld.ir

We can also opt to use Standard SQL authentication with SQL Server own users.

For the long workflows executions we also need a persistence database where WF automatically dehydrates the workflow execution. That requires creating a second database called, for example, '*AppFabricPersistenceDB*' and also specified in the AppFabric configuration.

**Note**:

In NLayerApp Sample we are not using WF or Workflow Services, so all AppFabric Workflow Services configuration could be omitted.

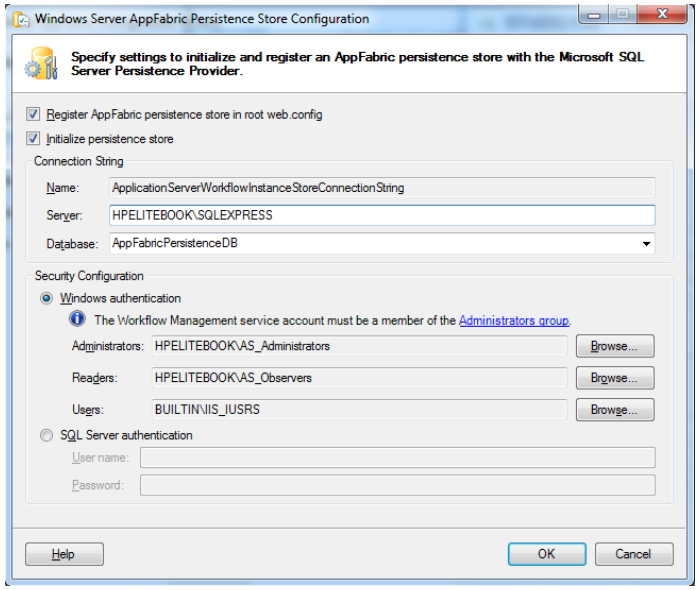



**Figure 27.- Hosting Services – WF Persistence Database config**

This is how the AppFabric monitoring section is finally configured.

www.EBooksWorld.ir

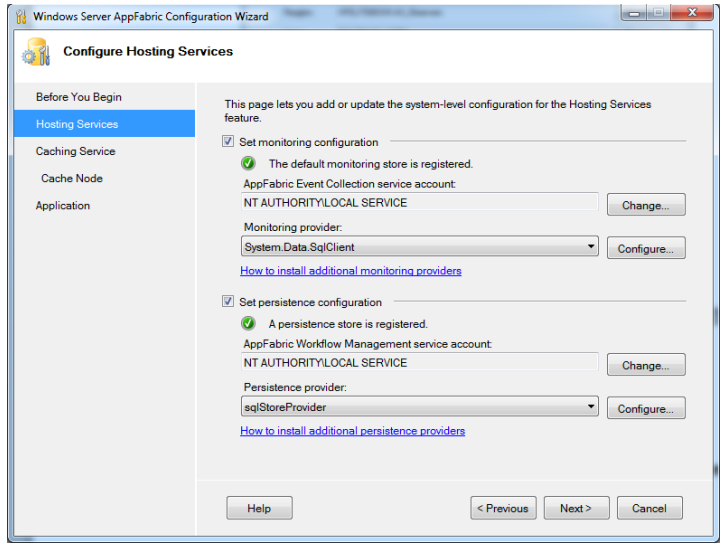



**Figure 28.- Final Hosting Services configuration**

The rest of the wizard configuration will not be addressed in this guide because it is related to Cache services.

If we start IIS Manager, we can see some new icons for AppFabric management:

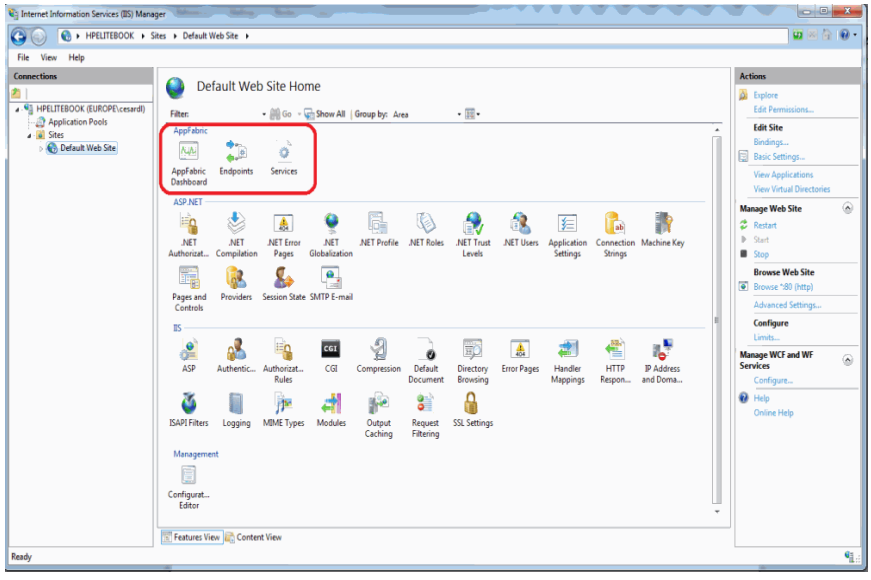



**Figure 29.- AppFabric snap-in – IIS Manager integrated**

www.EBooksWorld.ir

## 24.2.- WCF Service Deployment in Windows Server AppFabric.

After installing and configuring *AppFabric*, we can deploy our WCF services and application/domain components on *AppFabric*, so WCF services will be now monitored.

In the development environment (our PC), we will directly map our WCF service project to an IIS website. To do so, we should first create an IIS website with the TCP port chosen for our WCF services, such as the 8888. We will then specify our Distributed Services project directory as the root path for this Website:

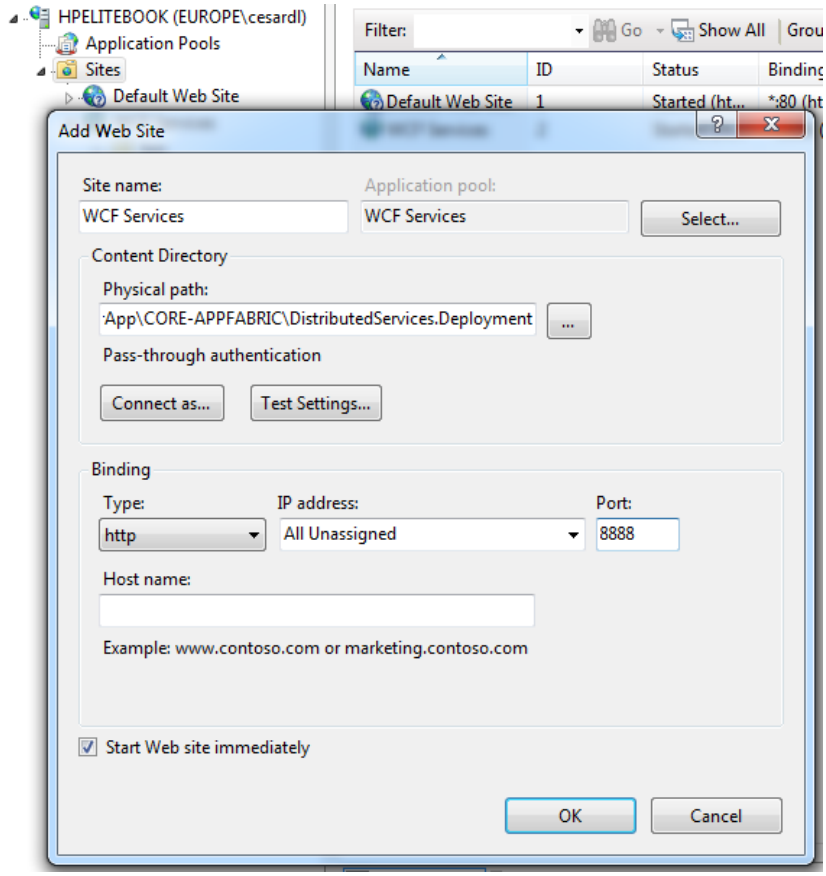



**Figure 30.- WCF WebSite configuration**

We need to change the properties of our WCF project so that, when we *debug*, it will be done over IIS and not over VS.2010 'Cassini'. In order to do that, we have to

www.EBooksWorld.ir

visualize the properties of the hosting project in VS.2010, and specify in the "Web" tab that we want to use the local IIS Web server. We will specify 'http://localhost:8888/' as the project URL. Finally, we must specify in which physical directory we want to map the Virtual Directory of IIS.

When we are done, we can press the '*Create Virtual Directory*' button to verify if it is properly connected to IIS (although in this case no virtual directory is created because there is already a root Website).

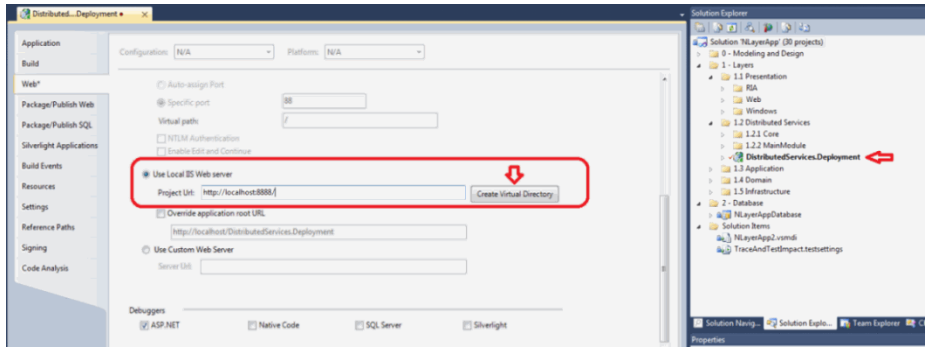



**Figure 31.- WCF project deployment from VS.2010**

Now, we can test our WCF service with the testing page, running on the IIS website. This can be done by launching it from the VS.2010 debugging environment or directly from the browser or IIS Manager.

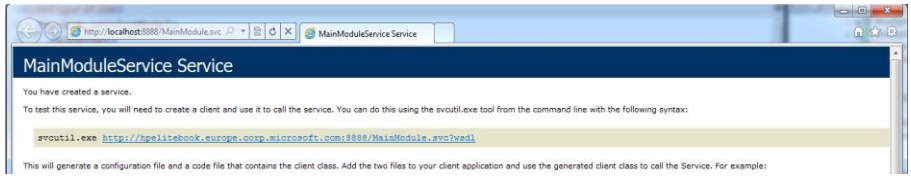



**Figure 32.- Testing our WCF Service already deployed on IIS**

Finally, in order for the application clients to work correctly, we should change the WCF endpoint configuration URL in all the Client Applications who are consuming our Service (Silverlight, WPF, etc.), specifying that the TCP port is now 8888 or any other change we have made for the URL.

www.EBooksWorld.ir

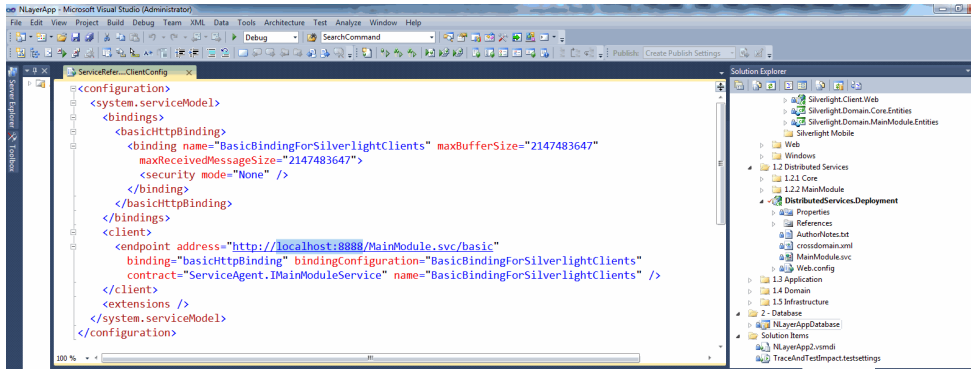



**Figure 33.- Changing Client WCF endpoint configs**

Finally, we also need to deploy the *Silverlight* web/hosting project in IIS. We simply need to specify it in the project properties and create it as a virtual ISS directory with the project name. The virtual directory in IIS will be created by pressing the '*Create Virtual Directory*' button.

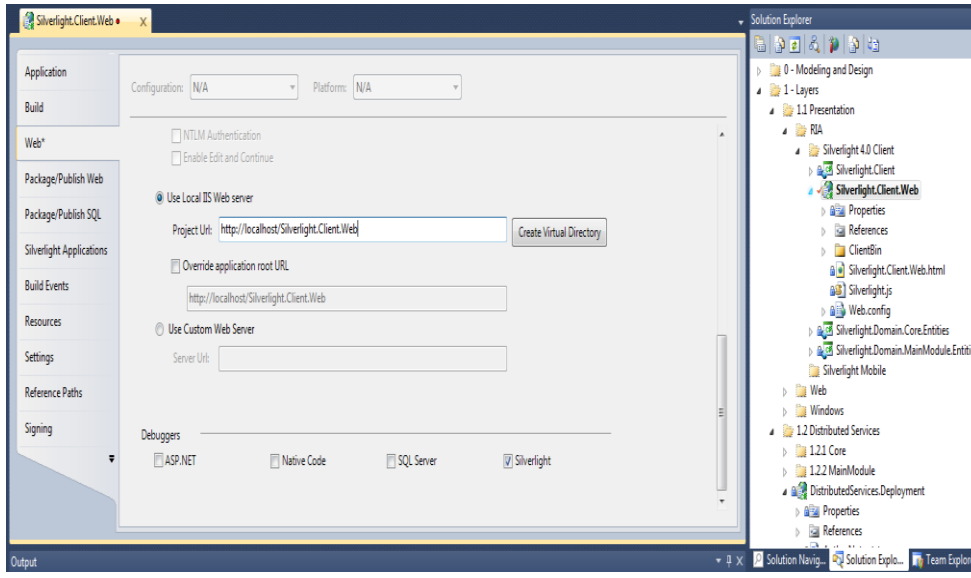



**Figure 34.- Silverlight project deployment in IIS**

The same operation may be performed with any *ASP.NET MVC* project so that it also runs on *IIS*.

Finally, the application should be run in a similar way, but underneath it will be running on *IIS* and *Windows Server AppFabric*.

www.EBooksWorld.ir

## 24.2.1.- SQL Server DB Access Identity and WCF Impersonation

The following should be considered: When we are running our application in *'debugging'* mode from Visual Studio and with WCF services based on the '*Dev Web Server*' of Visual Studio (*Cassini*), and the connection string is using "Integrated Security", the identity of the user who accesses SQL Server through ADO.NET EF connections is the identity of the user we logged onto the machine with, who will probably not have any problem accessing our local SQL Server.

On the other hand, if our WCF service is running on IIS/AppFabric, the identity it will try to access the SQL Server with will be the identity of the user-account that our Website process pool is being run with; this is normally 'ApplicationPoolIdentity' or 'NetworkService', which probably do not have access to our SQL Server, by default.

For our WCF services to have access to SQL Server, we recommend following the "Trusted Subsystem":

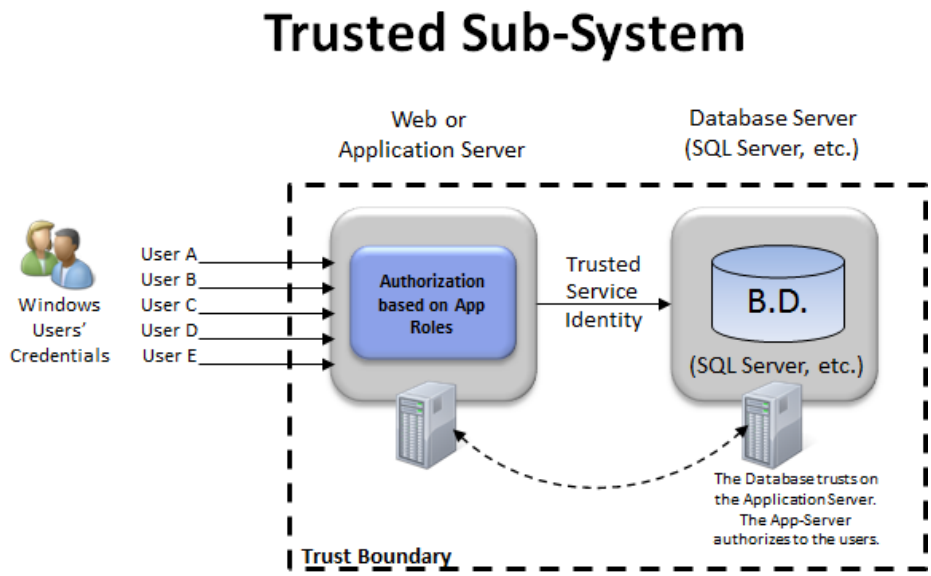



**Figure 35.- Trusted Subsystem**

Having an EF/ADO.NET connection string with integrated security model means we need the IIS pool we are using to be able to access our database in SQL Server. This is either because we provide DB access to the user the IIS pool is run with by default or because we change the identity the IIS pool is run with. The first option is probably more recommendable. This provides the necessary DB access to a default

www.EBooksWorld.ir

user of the IIS pool, so we do not have to create specific new Active Directory or Windows users. This, of course, will depend on our application, if we need to make other remote accesses using that same identity, etc.

**Granting Access to the IIS pool user-account to the SQL Server database** **Changing the IIS pool identity (Optional)**

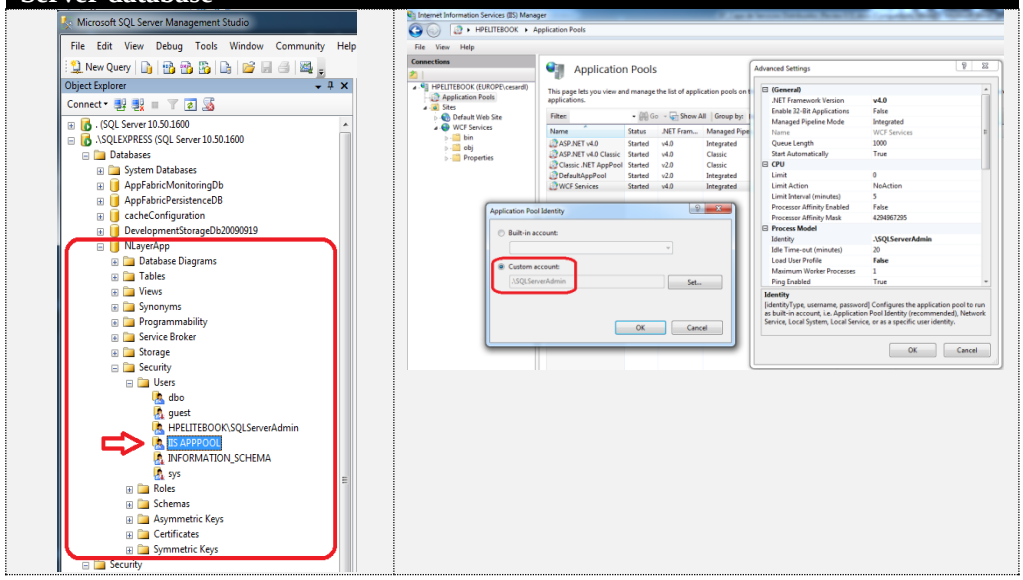



**Figure 36.- DB access identity options**

**Another viable option is to use a connection string with SQL Server standard security**, where the access credentials to SQL Server will be within the connection string. This has the disadvantage of being less secure & recommended because we have to specify (within this connection string) the user name and the password and these data remain visible in our web.config file. Normally, the web.config file may only be accessed by administrative staff and developers. In any case, it is less secure than specifying some administrative level credentials in the IIS console and being encrypted and protected by the IIS system (within the *applicationHost.config* file, the credentials to be used by the IIS pool remain encrypted).

## 24.3.- Monitoring WCF services from the Windows Server AppFabric Console in the IIS Manager.

After consuming/accessing any WCF Service hosted in AppFabric (from any client app, such as WPF or Silverlight in the *NLayerApp* sample), several times, we will be able to check those accesses and their performance from the AppFabric console.

The following is a global view from the Dashboard.

www.EBooksWorld.ir

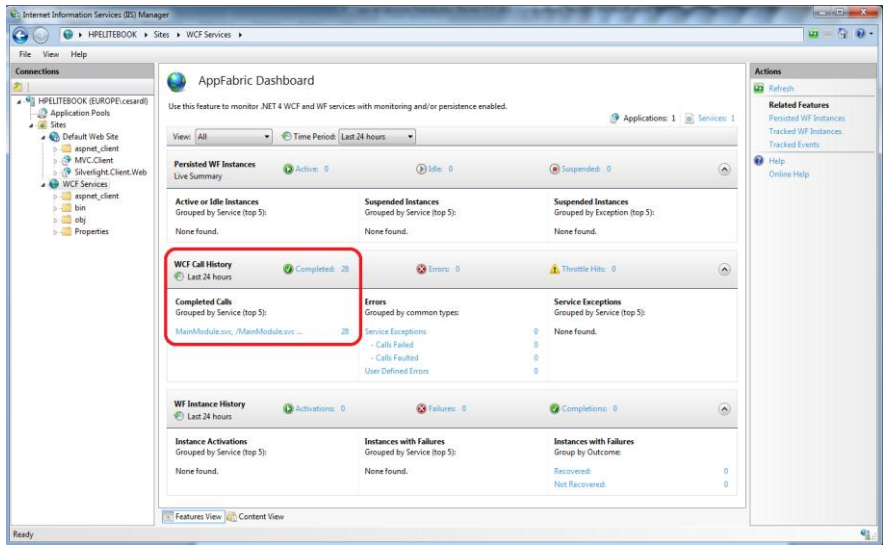



**Figure 37.- Monitoring WCF Services through AppFabric**

We can then review the entire list of calls to WCF services in detail. **Therefore, an IT Pro (no need to be a developer), can examine the behavior of an WCF Web service application, and even analyze the performance of every call to Web Services**, in order to detect performance issues, without prior knowledge of the details of the development.

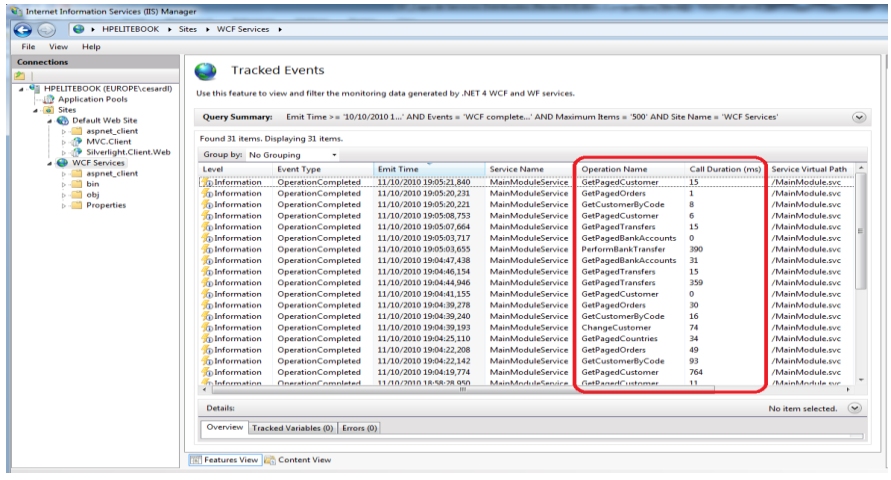



**Figure 38.- AppFabric Monitoring details**

We can also analyze the global statistics for a specific Web service.

www.EBooksWorld.ir

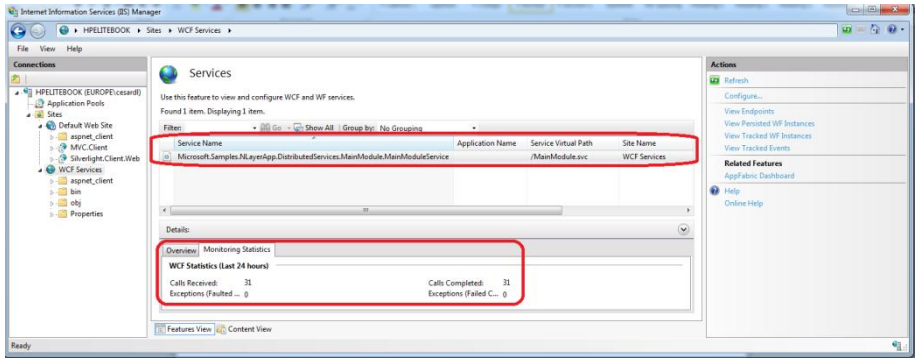



**Figure 39.- AppFabric Monitoring – Global statistics**

# 25.- SERVICES AND WCF GLOBAL REFERENCES

## References

*"Exception Handling in Service Oriented Applications"*
***http://msdn.microsoft.com/en-us/library/cc304819.aspx***

*"Data Transfer and Serialization":*
***http://msdn.microsoft.com/en-us/library/ms730035.aspx***

*"Endpoints: Addresses, Bindings, and Contracts":*
***http://msdn.microsoft.com/en-us/library/ms733107.aspx***

*"Messaging Patterns in Service-Oriented Architecture":*
***http://msdn.microsoft.com/en-us/library/aa480027.aspx***

*"Principles of Service Design: Service Versioning":*
***http://msdn.microsoft.com/en-us/library/ms954726.aspx***

*"Web Service Messaging with Web Services Enhancements 2.0":*
***http://msdn.microsoft.com/en-us/library/ms996948.aspx***

*"Web Services Protocols Interoperability Guide":*
***http://msdn.microsoft.com/en-us/library/ms734776.aspx***

*"Windows Communication Foundation Security":*
***http://msdn.microsoft.com/en-us/library/ms732362.aspx***

www.EBooksWorld.ir

*"XML Web Services Using ASP.NET":*
***http://msdn.microsoft.com/en-us/library/ba0z6a33.aspx***

*"Enterprise Solution Patterns Using Microsoft .NET":*
***http://msdn.microsoft.com/en-us/library/ms998469.aspx***

*"Web Service Security Guidance":*
***http://msdn.microsoft.com/en-us/library/aa480545.aspx***

*"Improving Web Services Security: Scenarios and Implementation Guidance for WCF":*
***http://www.codeplex.com/WCFSecurityGuide***

*"WS-* Specifications":*
***http://www.ws-standards.com/ws-atomictransaction.asp***

www.EBooksWorld.ir

CHAPTER 8

# Presentation Layer

## 1.- SITUATION IN N-LAYER ARCHITECTURE

This section describes the area of architecture as related to this layer (related to the User Interface). We will first introduce the logical aspects of the design (design patterns for the Presentation Layer) and then, in a differential manner, the implementation of the different patterns through the use of technologies.

The following diagram shows how the Presentation layer typically fits within our '*Domain Oriented N-layer Architecture'*:



www.EBooksWorld.ir

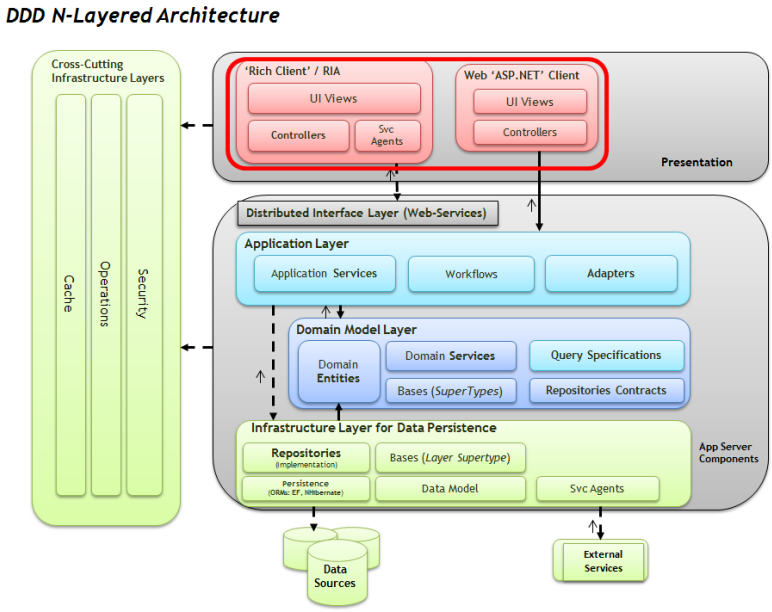



**Figure 1.- Domain-Oriented N-layered Architecture**

The Presentation Layer includes different elements, such as Views, Controllers, Models, etc. However, before reviewing aspects of the Architecture, we will introduce important aspects regarding the nature of this layer.

This layer´s function is to introduce the user to business concepts through a User Interface (UI), facilitate the operation of these processes, report on their condition and implement the validation rules of that interface. In the end, from the end user's point of view, this layer is the "application" itself. It is of no value to have created the best architecture in the world if we cannot leverage its capabilities in the most satisfactory manner for the user.

One of the peculiarities of the user interface is that it requires skills beyond the developer's scope, such as artistic design skills, knowledge of accessibility, usability, and control of the location of applications. Therefore, it is highly recommended that a professional in this area, such as a graphic designer, work together with the developer to achieve a high-quality user interface. It is the responsibility of this layer to enable cooperation between both roles. The developer will program in the chosen object-oriented language, creating the presentation layer logic, and the designer will use other tools and technologies, such as HTML or XAML, to create the GUI and make it interactive, among other things.

Another consideration to bear in mind is that this layer must be tested in the same manner as the lower layers. Therefore, there should be a mechanism to automate such testing and to include it in the process of automated integration, whether it is continuous or not.

www.EBooksWorld.ir

In this section of the guide, it is of utmost importance to approach this at two levels. A first logical level (Logical Architecture), which can be implemented with any technology and language (any version of .NET or even other platforms) and, subsequently, a second level of technology implementation that we will deal with specifically with specific versions of the technology.

## 2.- REQUIREMENT TO INVEST IN USER INTERFACE

The investment of companies in applications with intuitive user interfaces (simple and/or tactile) has been possible thanks to the increase in consumers of these types of proposals through cell phones, multi-tactile tables, or embedded systems. All of these new devices have many business advantages. Additionally, many studies have been done and many presentations have been given about this subject and all of them lead to the same conclusion: When using this software, user productivity is increased, expenses are reduced and sales grow.

Large companies such as Microsoft, Apple, Google, Yahoo or Amazon invest a lot on user experience. An efficient design for a user interface enables users to solve tasks more quickly and in the best possible way, thereby providing a great impact on user productivity. Please note that a well-designed and optimized user interface involves reducing the chances of users making mistakes and leads to an improvement in productivity. It is not only easy to use and intuitive, but fast. Users will be able to do more in less time; again increasing productivity. Psychology also plays a role: if a good user interface does not create problems and failures, its users will feel more comfortable when they are working and, therefore, become more productive.

Nowadays, expense reduction is one of the most important things for companies to accomplish. If we think we have intuitive, user-friendly tools, the investment in training or documentation of the tool may be reduced. Another reduction we will see is that when users feel comfortable using the tools they require less support. Another very important factor is using the proper UI technology to reduce complexity and therefore the cost in the deployment of the application.

The most important issue is market differentiation and the ability to gain competitive advantage by providing users with better user experience than what is offered in the current market. It is said that first impressions are what sell-- a bad image can turn a very powerful product into a terrible failure.

Some external studies provide very revealing data about the importance of user interfaces. The **Gartner** study *"The Increasing Importance of the Presentation Layer to the Enterprise"* published in 2004 already stated that *"the presentation layer is an integral part of all the aspects of the development process. Because of this, we see everywhere that, from 25% to nearly 40% of the total time and effort is somehow related to the presentation layer in all parts of the development process".*

*Infragistics* also has some interesting figures. They say that *"every dollar invested in UX brings in between 2 and 100 dollars"* and the evidence for this is based on the following:

www.EBooksWorld.ir

- ***Gilb, T***. *(1988). "Software Engineering Direction Principles". Addison Wesley*
- ***Pressman, R.S.*** *(1992). "Software Engineering". McGraw-Hill: New York, New York.*

An SDS Consulting (Strategic Data Consulting) report also shows interesting figures. In its report called *"Special Report: Business Impacts of UX and ROI"* the authors conclude that investment in user experience is essential, and results in the decrease in development expenses, an increase in income and reduction in time before it is launched onto the market. In this report, which is based on the review of 735 Internet companies, it was concluded that, as an average, companies invest 11.5% of their development budgets for user interface products. It also concludes that the user interface accounts for between 47% and 66% of the total code written for a project, which then accounts for 40% of the development effort. It is interesting to note that they quote the real case of McAffee, which reduced 90% of its support expenses when it updated the user interface.

# 3.- THE NEED FOR ARCHITECTURE IN THE PRESENTATION LAYER

The treatment of the presentation layer during all parts of the development cycle (even during the early stages of product planning) considerably reduces development time and economic expenses associated with the subsequent design changes.

## 3.1.- Decoupling Between Layers

Investing time in the separation between the presentation logic code and the interface reduces the expenses incurred by future changes, but it also favors cooperation between designers, developers and project managers, which helps to avoid downtime due to poor communication. If the architecture is done in such a way that the designer's work does not interfere with the developer's work, it will reduce the project's completion time.

The main problem we encounter is coupling between components and layers of the user interface. When each layer knows how the other layer does its job, the application is considered to be coupled. For example, in an application typical coupling occurs when the search query is defined and the logic of this search is implemented in the controller of the "Search" button that is in the code behind. Since the requirements of the application change and we have to modify it as we go along, we will have to update the code behind the search screen. If, for example, there is a change in the data model, we will have to make these changes in the presentation layer and confirm that

www.EBooksWorld.ir

everything is working correctly again. When everything is highly coupled, any change in one part of the application can cause changes in the rest of the code and this is an issue of complexity and quality of the code.

When we create a simple application, such as a movie player, this type of problem is not common because the size of the application is small and the coupling is acceptable and manageable. However, in the case of Business Applications, or LOBs (Line of Business), with hundreds of screens or pages, this becomes a critical issue. As the size of a project is increased, it becomes more complex and we have more user interfaces to maintain and understand. Therefore, the Code behind approach is considered an anti-pattern in business line applications.

## 3.2.- Performance Trade-Off

On the other hand, we have performance requirements. Optimizations frequently consist of complicated tricks that make it difficult to understand the code. These tricks are not easily understood, which makes application maintenance more difficult. This might not always be the case, but it is true in most cases. There are many performance tricks that are not at all complex, and there are many applications with high-performance and maintainability, but our experience has proven that these types of situations are not common. Therefore, with regard to business applications, it is better to have good maintainability than to have the greatest graphic performance, so we must concentrate on optimizing only the parts which are critical to the system.

## 3.3.- Unit testing

Another part of the problem is testing the code. When an application is coupled, the functional part is the part that can be tested, which is the user interface. Again, this is not a problem with a small project, but as a project grows in size and complexity, being able to check the application layers separately is crucial. Imagine that something changes in the data layer, how do you confirm it has affected the presentation layer? You would have to perform functional tests on the entire application. The performance of functional tests is very expensive because they are not usually automated, and they require people to test all the functionalities one by one. It is true that there are tools that are suitable for recording the users' movements and automating those tests, but labor and cost incurred by this well suggests that we should avoid them.

If we are also capable of providing the product with a set of automated tests, we will improve quality of the after sale support, since we will be able to reduce time in locating and solving the failures (Bugs). This will increase our product quality, customer satisfaction and we will reinforce his loyalty.

www.EBooksWorld.ir

For these reasons, the presentation layer is required to have a sub architecture of its own that fits into the rest of the proposed architecture.

# 4.- ARCHITECTURE PATTERNS IN THE PRESENTATION LAYER

There are some well-known architecture patterns that can be used for the design of the presentation layer architecture, such as MVC, MVP, MVVM and others. Right now, these patterns are used to separate the concepts between the user interface and the presentation logic. At the beginning, however, they were created to separate the presentation layer from the business layer, since presentation technologies at that time did not separate themselves from the business logic.

## 4.1.- MVC pattern (Model-View-Controller)

The *model-view-controller* pattern first appeared in the eighties, with its first implementation in the popular *SmallTalk* programming language. This pattern has been one of the most important and influential in the history of programming and is a fundamental pattern even today. This pattern belongs to a set of patterns grouped in separated presentation architecture styles. The purpose when using this pattern is to separate the code responsible for representing data on screen from the code responsible for executing logic. To do so, the pattern divides the presentation layer into three types of basic objects: models, views and controllers. The use of the pattern is to describe communication flows between these three types of objects, as shown in the diagram below:

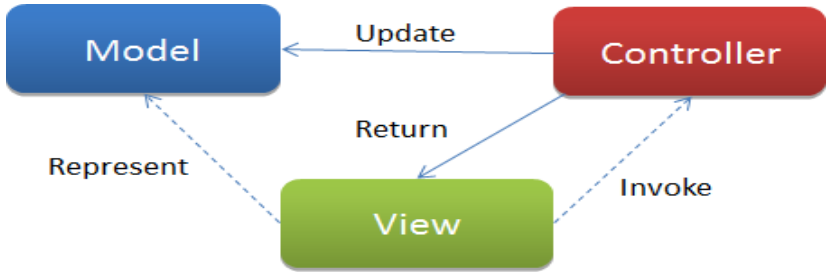



**Figure 2.- MVC Architecture - Presentation Layer**

In a classic scenario, this view represents what the user sees about the data, and invokes actions of a controller in response to the user's actions. The controller monitors actions of the user, interacts with the model and returns a specific view as an answer.

www.EBooksWorld.ir

In most MVC implementations, the three components can directly relate from one to another, but in some implementations, the controller is responsible for determining which view to show. This is the evolution of the Front Controller pattern (**http://msdn.microsoft.com/en-us/library/ms978723.aspx**).

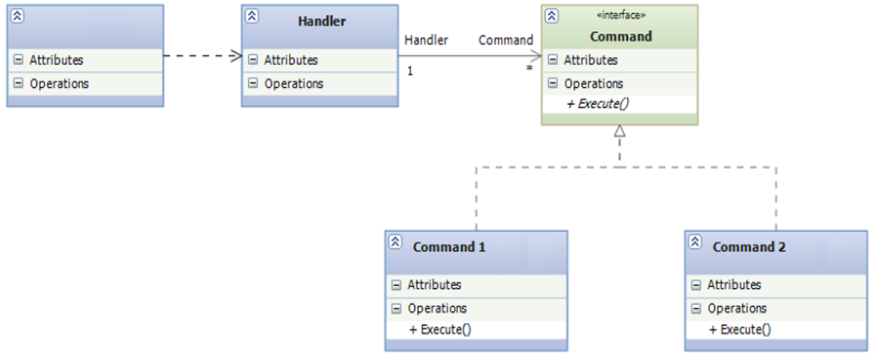



**Figure 3.- 'Front Controller' Pattern Evolution**

In '*Front-Controller*,' the controller is divided into two parts, one is the handler that collects relevant information from the requests that come from the presentation and directs them to a particular command, which executes the action performed in the presentation.

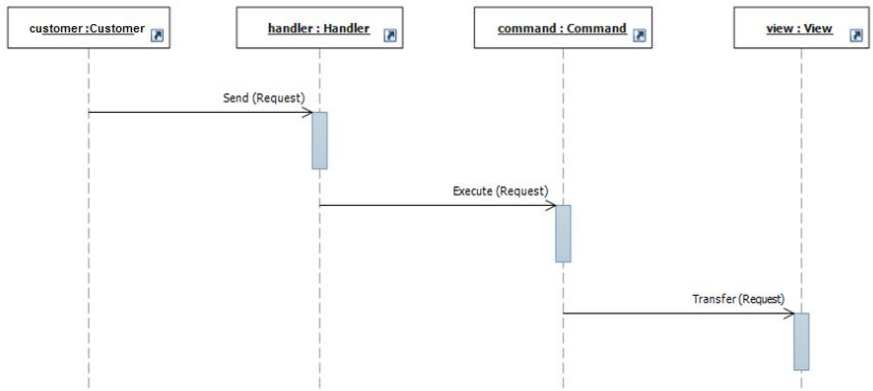



**Figure 4.- Front-Controller Sequence Diagram**

We will describe the responsibilities of each component in more detail below.

www.EBooksWorld.ir

## 4.2.- The Model

The model is the set of classes responsible for representing the information the user works with. It is important to understand that these classes can be domain classes, DTOs or *ViewModels* (Remember that a *ViewModel* is a class that stores data that represents a certain view, not a *ViewModel* as in Silverlight). The option we choose will depend on the situation:

- Domain classes: the view model corresponds to a domain class to a large degree and we are not using DTOs in the application. For example, in an entity editing action.

- DTOs: the view model corresponds to a large degree to the data of the DTO we are using in our application.

- *ViewModels*: the data needed by the view does not correspond directly to the DTOs or to the domain classes of our application. This ViewModel stores these pieces of additional information and probably includes small query methods (Properties) that simplify the description of the view. For example, a paginated list of a set of data, where the ViewModel is composed of the page's data and properties such as page number, size, the total number of pages, etc.

## 4.3.- The Views

The views are responsible for graphically representing the model and offering controller actions for the user to be able to interact with the model. It is clear that a view should not invoke any method that causes a change of state in the model or call upon any method requiring parameters. In other words, it should only access simple properties and object query methods that do not have any parameters.

## 4.4.- The Controller

The controller implements interactions between views and the model. It receives the requests from the user, interacts with the model making queries and modifications to it, decides which view to show in response and provides the model with the required data for its rendering, or delegates the answer to another action of another controller.

www.EBooksWorld.ir

## 4.5.- MVP Pattern (Model View Presenter)

Before explaining the MVP pattern itself, let's look at the history.

In the 90´s the Forms and Controllers model was the trend, driven by development environments and programming language such as Visual Basic or Borland Delphi. These development environments enabled the developer to define the screen layout with a graphic editor that allowed dragging and dropping controls within a form. These environments increased the number of applications because it enabled rapid application development and applications with better user interfaces to be created.

The form (design surface in development enviroments) has two main responsibilities:

- Screen layout: definition of positions of the screen controls, with the hierarchical structure of some controls with regard to others.

- Formula logic: behavior of controls that usually respond to events launched by controls set by the screen.

The proposed architecture for Forms & Controller became the best approach in the world of presentation architectures. This model provided a very understandable design and had reasonably good separation between reusable components and the specific code of form logic. But Forms & Controller does not have something provided by MVP, which is the 'Separated Presentation pattern', and the facility to integrat to a Domain Model, both of which are essential.

In 1996, Mike Potel, who was behind IBM's Mainstream MVP, published the Model View Presenter, MVP, which took a step towards merging these streams (MVC and others) by trying to take the best characteristics from each of them.

The MVP pattern separates the domain model, presentation and actions based on interaction with the user into three separate classes. The view delegates the responsibility for handling user events to its presenter. The Presenter is responsible for updating the model when an event is raised in the view, but it is also responsible for updating the view when the model indicates that a change has occurred. The model does not know about the existence of the Presenter. Therefore, if the model changes due to the action of any component other than the Presenter, it must trigger an event to make the Presenter aware of it.

When implementing this pattern, the following components are identified:

- IView: The interface used by the Presenter to communicate with the view.

- View: The view concrete class implementing the IView interface, responsible for handling the visual aspects. It always refers to its Presenter and delegates the responsibility of handling the events to it.

www.EBooksWorld.ir

- Presenter: Contains the logic to provide an answer to the events and handles the view state through a reference to the IView interface. The Presenter uses the model to know how to provide an answer to the events. The presenter is responsible for setting and managing the view state.

- Model: Composed of the objects that know and manage data within the application. For example, they can be the classes that compose the business model (business entities).

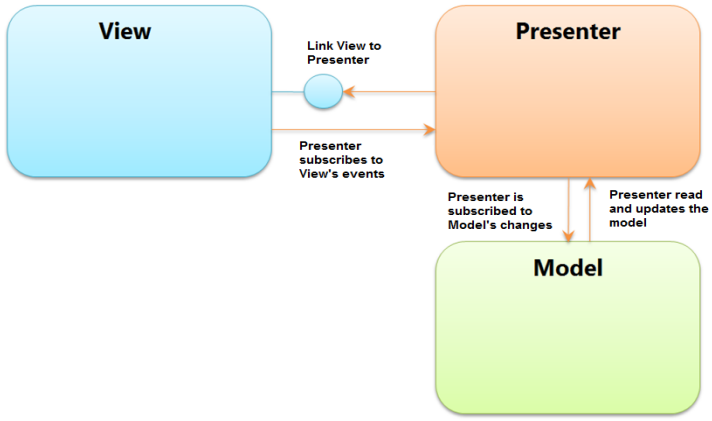



**Figure 5.- MVP Pattern (Model View Presenter)**

In July 2006, the renowned author *Martin Fowler* published a note suggesting the removal of the Presentation Model pattern (as Fowler called the MVP) from the patterns catalogue. He said that after a long study and analysis he decided to divide the MVP pattern into two patterns, depending on the responsibility of the view level:

- **Supervising Controller:** Decomposes presentation functionality into two parts: a presenter and view.

- **Passive View:** where the view is completely "controlled" by the Presenter and has hardly any functionality.

www.EBooksWorld.ir

## 4.6.- MVVM Pattern (Model-View-ViewModel)

The Model View ViewModel (MVVM) pattern is also derived from MVP and this, in turn, from MVC. It emerged as a specific implementation of these patterns when using certain very powerful capabilities of newly available user interface technologies. Specifically, this is due to certain Windows Presentation Foundation (WPF) capabilities.

This pattern was adopted and used by the Microsoft Expression product team since the first version of Expression Blend was developed in WPF. Actually, without the specific aspects brought by WPF and Silverlight, the MVVM (Model-View-ViewModel) pattern is very similar to the 'Passive-View' defined by Martin Fowler, but because of the capabilities of current user interface technologies, we can see it as a generic pattern of presentation layer architecture.

The fundamental concept of the MVVM is to separate the Model and the View by introducing an abstract layer between them, a "ViewModel". The view and the view model are usually instantiated by the container component. The view maintains a reference to the ViewModel. The ViewModel exposes commands and entities that are "observable" or linkable to those where the View can be linked. The user's interactions with the View will trigger commands against the ViewModel and in the same way, updates in the ViewModel will be propagated to the View automatically through a data link.

The following diagram shows this at a high level, without specifying details of technological implementation:

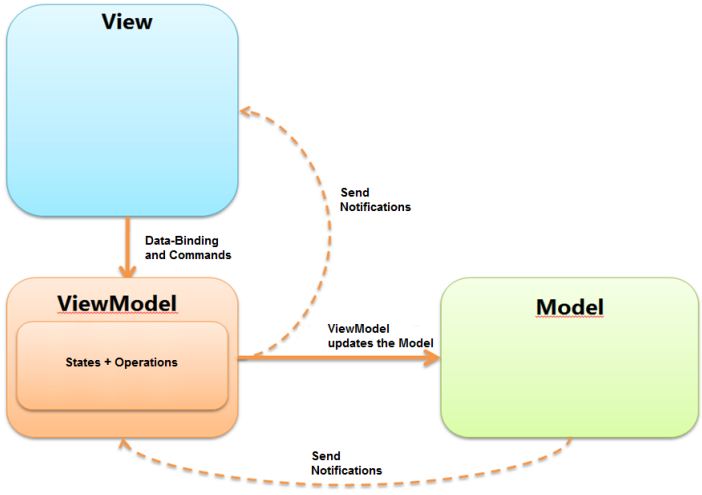



**Figure 6.- MVVM pattern (Model-View-ViewModel)**

www.EBooksWorld.ir

The objective of the MVVM is to present and manage data transferred to the view in the easiest possible way. The ViewModel exposes the data from the model so, in this sense, the ViewModel is a model rather than a view. However, it also manages the viewing logic of the view so, from this perspective, the view is more similar to a view than to a model. Currently, the mix of responsibilities is still a matter of discussion and continuous exploration in this sector.

## 4.7.- Global Vision of MVVM in Domain-Oriented Architecture

Within the domain oriented N-layered architecture, the MVVM presentation layer sub-architecture is not positioned in a straight line as you might think. The diagram below shows a view on how the MVVM layers can communicate with the architecture layers.

Please note how the Model may or may not be communicated with the rest of the architecture. The Model defines the entities to be used in the presentation layer when this model is not the same as the entity model. If we have entities that are the same in the domain layer and in the presentation layer, we will not have to repeat our work. This diagram shows that the View Model does not have to communicate uniquely and exclusively with the Model, but it can call directly to the rest of the domain oriented architecture and it can even return domain entities instead of the Model.

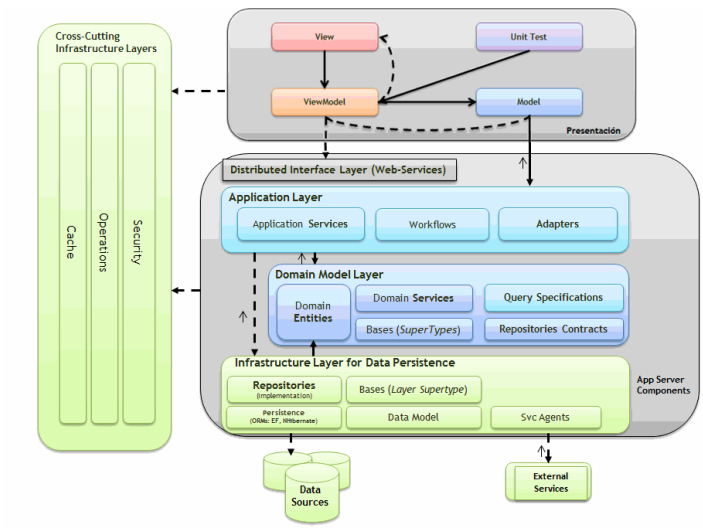



**Figure 7.- Global vision of MVVM within the DDD architecture**

www.EBooksWorld.ir

## 4.8.- Design Patterns used in MVVM

We describe some design patterns used in the MVVM architectural pattern below.

### 4.8.1.- Command Pattern

The goal of the Command design pattern, also known as Action or Transaction, is to provide an operation management abstract interface on a certain receptor, allowing a client to execute the operations without having to know exactly what type of operation is involved or who is implementing it.

One of the reasons for this is that sometimes we want to be able to send requests to objects without having to know exactly the type of operation involved or who is implementing it. Generally, a button or menu object follows the request but the request is not implemented within itself.

Command is used to:

- Parameterize objects according to the actions they perform.
- Specify, manage and execute requests at different times. The Command object has a life time that does not depend on the request of the command that instantiates it.
- Support the capacity to undo the request. The Command object can maintain a state that allows the execution of the command to be undone.
- Support the capacity to generate a history that allows the recovery of the state in case of system failure.
- Allow system structuring around high level operations constructed based on primitive or low level operations.

www.EBooksWorld.ir

The structure is as follows:

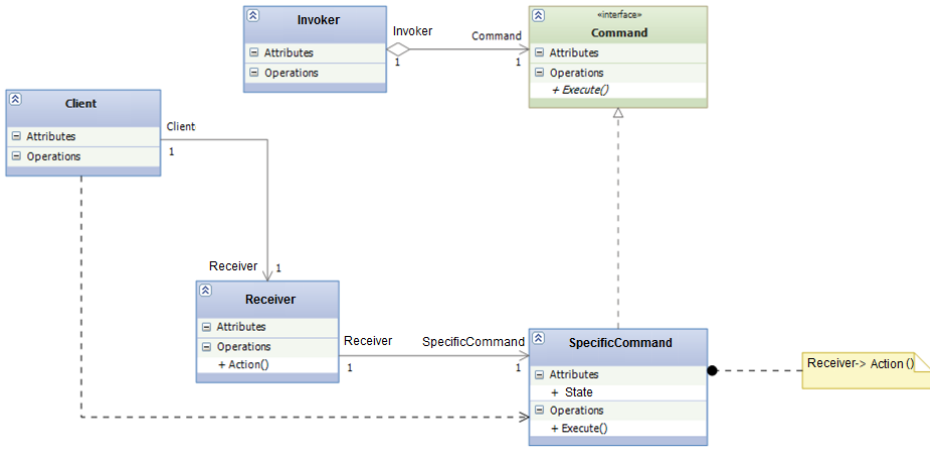



**Figure 8.- Command structure**

Where the responsibilities are:

- **Command**: States the interface for execution of the operation.
- **SpecificCommand:** Defines the relationship between the Receiver object and an action, and implements Execute() when invoking the corresponding operations in Receiver.
- **Client**: Creates a SpecificCommand object and associates it to its Receiver.
- **Invoker**: Makes requests to the *Command* object.
- **Receiver**: Knows how to execute operations associated with the requests. Any class can be a receiver.

The collaboration between objects is as follows:

- A client creates a SpecificCommand object.
- The Invoker object saves the SpecificCommand object.
- The Invoker object requests a call to *Command* Execute().
- The SpecificCommand object invokes the necessary operations to resolve the request.

www.EBooksWorld.ir

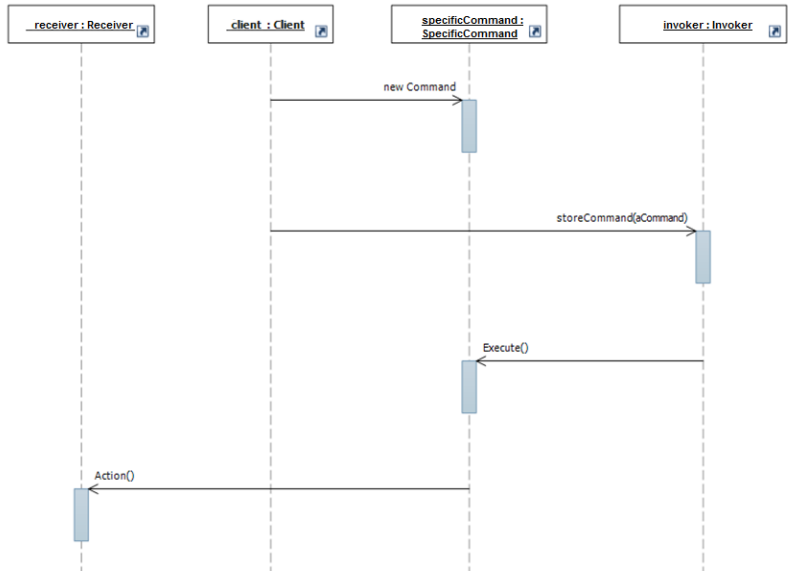



**Figure 9.- Sequence between objects**

## 4.8.2.- Observer Pattern

The observer pattern, also known as publication-subscription pattern, defines the relationship of an object with many objects, so that, when an observable object changes state, it notifies all the observers of this change.

The basic ideas of the pattern are simple: the data object contains methods through which any observer object can subscribe by passing along a reference to itself. Thus, the data object provides a list of the references to its observers.

The observers are obligated to implement certain methods that enable the data object to notify its subscribed observers about the changes it has undergone, so that all of them have the chance to refresh the contents represented therein.

This pattern is usually seen in object-oriented graphical user interface (GUI) frameworks, where events are collected by subscribing to the objects that can trigger events.

www.EBooksWorld.ir

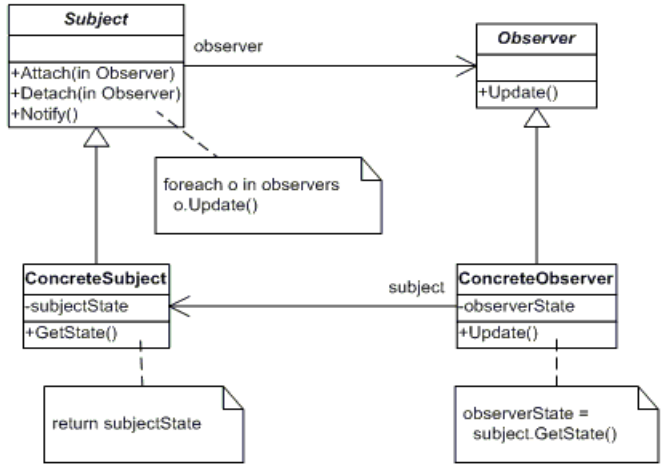



**Figure 10.- The Observer pattern**

## REFERENCES:


- *"Presentation Model"* - ***Martin Fowler***, *July 2004.* ***http://martinfowler.com/eaaDev/PresentationModel.html***

- *"Design Patterns: Elements of Reusable Object-Oriented Software" (ISBN 0-201-63361-2)* ***http://en.wikipedia.org/wiki/Special:BookSources/0201633612***

- *"Head First Design Patterns****" O. Freeman, E. Sierra, K. Bates****, B (2004) Ed. O'Reilly.*

- *"Introduction to Model /View /ViewModel Pattern for building WPF apps"* - ***John Gossman***, *October VS. r 2005.* **http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx**

- *"Separated Presentation"* - ***Martin Fowler***, *June 2006.* ***http://www.martinfowler.com/eaaDev/SeparatedPresentation.html***

- *"WPF Pattern"s* - ***Bryan Likes***, *September 2006* ***http://blogs.sqlxml.org/bryantlikes/archive/2006/09/27/WPF-Patterns.aspx.***

- *"WPF patterns: MVC, MVP or MVVM or…?"* - ***The Orbifold***, *December 2006.* ***http://www.orbifold.net/default/?p=550***


www.EBooksWorld.ir

- *"Model-see, Model-do, and the Poo is Optional"* - ***Mike Hillberg***, *May 2008.*
  ***http://blogs.msdn.com/mikehillberg/archive/2008/05/21/Model-see_2C00_-model-do.aspx***

- *"PRISM: Patterns for Building Composite Applications with WPF"* - ***Glenn Block***, *September 2008.*
  ***http://msdn.microsoft.com/en-us/magazine/cc785479.aspx***

- *"The ViewModel Pattern"* - ***David Hill***, *January 2009.*
  ***http://blogs.msdn.com/dphill/archive/2009/01/31/the-viewmodel-pattern.aspx***

- *"WPF Apps with the Model-View-ViewModel Design Pattern"* - ***Josh Smith***, *February 2009.*
  ***http://msdn.microsoft.com/en-us/magazine/dd419663.aspx***

- *"Model View Presenter"* – ***Jean-Paul Boodhoo***, *August 2006.*
  ***http://msdn.microsoft.com/en-us/magazine/cc188690.aspx***


www.EBooksWorld.ir

# 5.- IMPLEMENTING THE PRESENTATION LAYER

The objective of this chapter is to show the different options we have, at the technical level, to implement the Presentation Layer, depending on the nature of each application and the chosen design patterns.

In this Architecture diagram we highlight the location of the Presentation Layer:

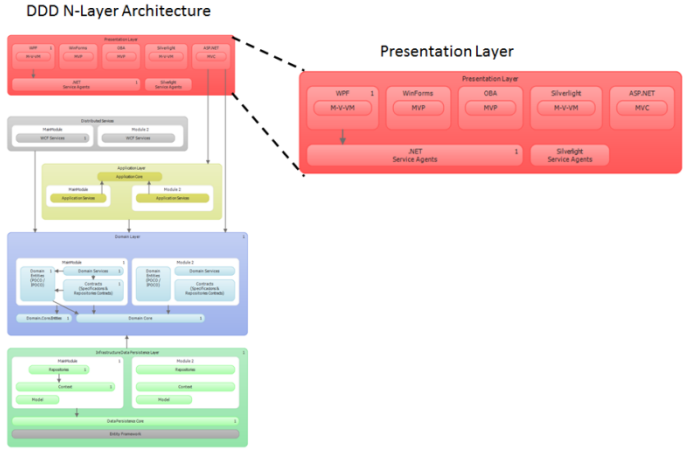



**Figure 11.- Presentation Layer Location in Layer Diagram using VS.2010**

As can be seen in detail in the Presentation Layer, current applications have various technologies and user interfaces that enable the use of different functionalities in different ways. For example, in **Microsoft Office** we can change the text format from the menu bar or from the secondary menu associated with the right button of the mouse.

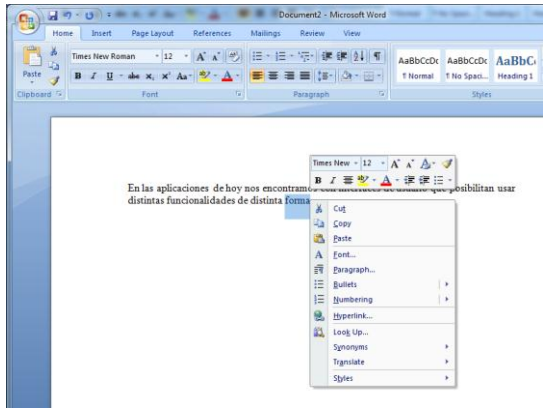



**Figure 12.- Microsoft Office Word GUI as a Presentation Layer**

www.EBooksWorld.ir

If we simplify the problem and we want to implement an interface with repeated functionalities from multiple sites, we see that in XAML we can define an event handler such as the following:

```
    <Button Name="UnoBtn" Click="UnoBtn_Click" Width="100"
Height="25">Button</Button>
```

And also

```
    Function test()
    <my1:Label KeyDown="Label_KeyDown">One</my1:Label>
```

We can then implement this in the ***code behind***:

```
    private void OneHl_Click(object sender, RoutedEventArgs e)
    {
        Functionality();
    }
    private void Label_KeyDown(object sender, KeyEventArgs e)
    {
        Functionality();
    }
```

We should note one detail. On one hand, we are defining the event handler from the XAML itself, and we are mixing two roles. The designer that generates XAML visually with Blend does not have to know anything about the handler. He understands the concepts or orders. For example, he knows that the button he is designing is for saving and will therefore be associated to the order "Save."

On the other hand, we are creating a strong dependency on XAML with the functionality programming. This is to say, if we want to test to see if something is being saved, we will have to test the code behind. If we were to reuse this functionality in other XAMLs, we would have to bring it to a different class. Then again, we still require the programmer to see the XAML and link the event handlers to each XAML. In this case, we do not have total independence of responsibility, either the designer has a good understanding of programming because he has to link events or he has to link events with what the designer has done.

In the example above, since the action is performed by two different XAML components, we must use two event handlers if we are unlucky and these handlers have different delegations. If we put in 10 repeated functionalities from three different components, we put in 30 event handlers. It seems unmanageable, doesn't it? Therefore, we recommend using architecture that helps solve these problems.

www.EBooksWorld.ir

## 5.1.- Associated Archetypes, UX technologies and Design Patterns

Currently, there are a series of Archetype applications distinguished by the nature of their visual technology, their User Experience and the technology with which they are implemented. These Archetypes are also explained one by one in this Architecture guide (including other Archetypes not marked by UX technologies). However, after analyzing the detailed study of the Presentation Layer patterns and technologies, we will only review some of them.

The list of current Archetypes defined by UX aspect (User Experience) would be:

- **Rich Client** applications (Desktop applications / Windows)
- **Web** applications (HTML dynamic applications)
- **RIA** applications (*Rich Internet Applications*)
- **Mobile** applications
- **OBA** applications (*Office Business Applications*)

Depending on each Archetype, there are one or more technologies to implement the applications. In turn, the design and implementation of one or another architecture pattern of the Presentation Layer is recommended depending on each technology. These possibilities are exposed in the matrix shown below:

**Table 1.- Presentation Layer Architecture Archetypes and Patterns**

| Archetype | Technology | Architecture Pattern – Presentation layer |
|---|---|---|
| Rich applications (*Desktop applications / Windows*) | **WPF (*)** | → **MVVM** |
| | WinForms | → MVP |
| Web applications | **ASP.NET MVC (*)** | → **MVC** |
| | ASP.NET Forms | → MVP |

www.EBooksWorld.ir

| | | | |
|---|---|---|---|
| RIA applications | **Silverlight (*)** | → | **MVVM** |
| Mobile- Rich applications | .NET Compact Framework | → | MVP |
| Mobile applications – RIA | Silverlight Mobile | → | MVVM |
| OBA Applications (Office) | .NET VSTO | → | MVP |

In this edition of this Architecture guide, we will specifically emphasize the Archetypes and patterns related to technologies in bold and with an (*), that is:

- WPF
- Silverlight
- ASP.NET MVC

## 5.2.- Implementing MVVM Pattern with WPF 4.0

The community initially associated with WPF has created the MVVM pattern. As mentioned, this model is an adaptation of the MVC and MVP patterns where the ViewModel provides the data model and the view's behavior, but allows the view to link them through declaration in the view model. The view becomes a combination of XAML and C#. The model represents the available data for the application and the ViewModel prepares the model to be linked through the view.

MVVM was designed to use specific functions of WPF (also available now from Silverlight 4.0), which better enables the separation of development/design between the View's sub-layer from the rest of the pattern, virtually eliminating the entire codebehind of the Views sub-layer. Instead of requiring the graphic designers to write the .NET code, they can use the XAML brand language (a specific format based on XML) of WPF to create links to the ViewModel (which is .NET code maintained by developers). This separation of roles allows the Graphic Designers to focus on the UX (User Experience) instead of having to know something about programming or business logic, and finally allowing different Presentation sub-layers to be created by different teams with different technical profiles.

www.EBooksWorld.ir

The following diagram represents this operation:

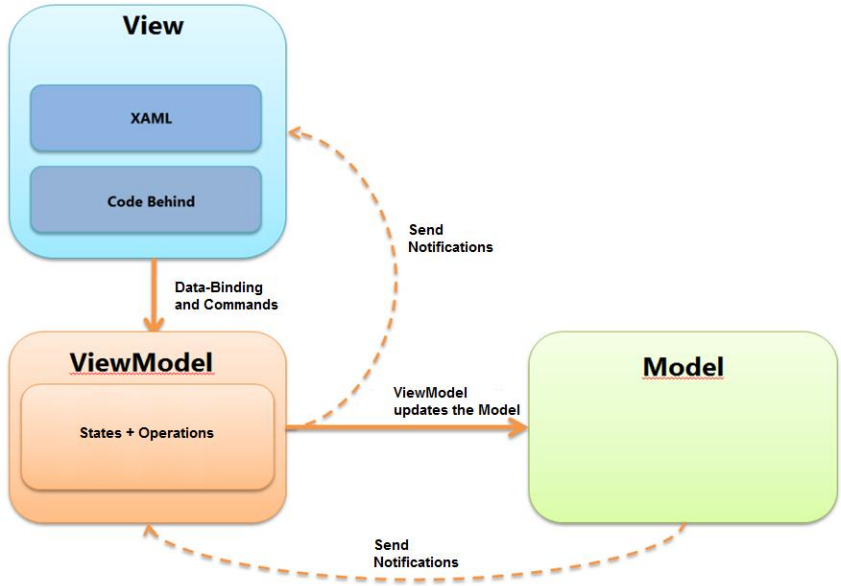



**Figure 13.- MVVM using WPF**

## 5.2.1.- Justification of MVVM

**Simplistic design of Application**

To prove the benefits of using an architecture that separates the presentation from the model, we will start with a simple scenario of an application where we use the *Separated Presentation* pattern.

Let's assume a scenario where we have two windows (Views): One View with a table/list where we can see all the clients of the application and another view showing the details of simply one client (one "record" if we simplify it). We would have a User interface similar to the following:

www.EBooksWorld.ir


**Figure 14.- Customer list view**


**Figure 15.- Customer detail view**

To distinguish between Model and View for this particular scenario, the Model will be the Customer's data and the Views will be the two Windows shown.

www.EBooksWorld.ir

Following our simple model, the design would look like this:

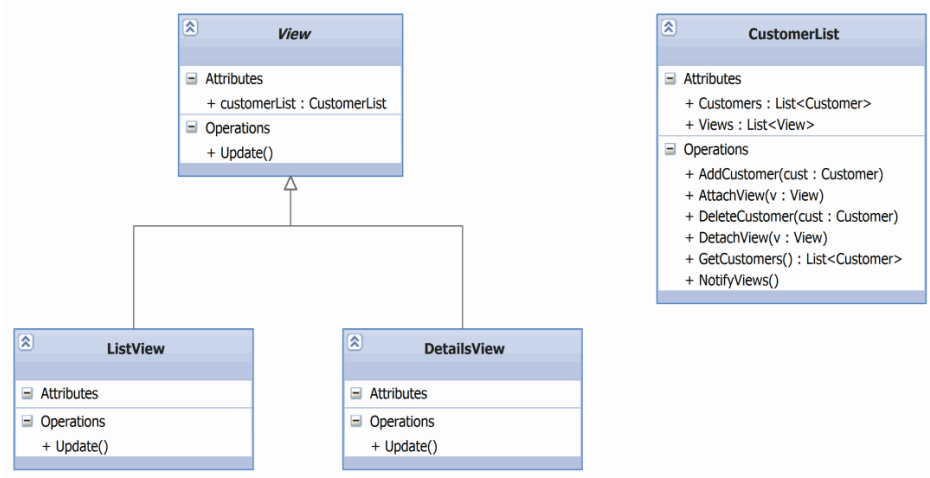



**Figure 16.- Simplistic Design – The views are directly associated with the Model**

As can be seen, **CustomerList** defines a View list (that can be added or removed with **AttachView**() and **DetachView**()). When a contact changes, **CustomerList** will notify all the Views by calling the **Update**() method and all the views would be updated by calling the **GetCustomers()** method of the model. Once instantiated, the ListView and DetailsView will have a reference to the CustomerList, defined as a field member, (it is actually defined in the abstract base class 'View'). As you may have noticed, the 'observer' pattern has been applied.

**Note**:
Note that the two Views in this case are STRONGLY COUPLED with the **CustomerList** class. If we add more client business logic, it will only increase the coupling level.

### WPF Version of the Application simplistic design

Before continuing the analysis of the logical design, we will turn it into a WPF-friendly design. Keep in mind that WPF provides two very powerful aspects that we can use directly:

- **Databinding**: The capacity to link GUI elements to any type of data.
- **Commands**: Provide the capability to notify the underlying data, that there were changes made in the GUI.

www.EBooksWorld.ir

According to these capabilities, the WPF-friendly design would look like this:

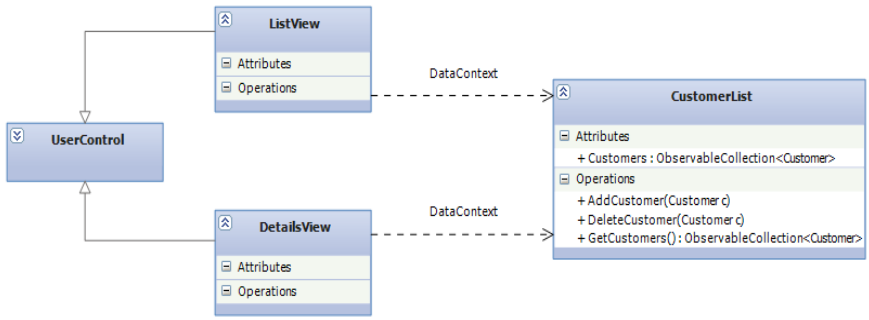



**Figure 17.- Friendly and simple design of the WPF application**

Views in this case are derived from the **UserControl** (we do not need a **View** abstract class) and **CustomerList** does not have to maintain a Views list any longer, since it does not have to "know" anything about the views. Instead, views aim at **CustomerList** as their ***DataContext*** and they use ***WPF Data-Binding*** in order to link to the Customer list.

Also note that we have replaced `List<Customer>` by `ObservableCollection<Customer>` to allow views to be linked to **CustomerList** by using **data-binding** mechanisms**.**

> **Note:**
> Note that the WPF data-*binding* mechanism allows us to create a much more de-coupled design (at least until we have to add a certain client business logic).

**Problems of the WPF simple design**

The problem is that things get complicated when we introduce the following functionality:

1.- Propagation of the selected element, so that we update the **DetailsView** as long as the selection of an element in **ListView** changes, and vice versa.

2.- Enabling or disabling parts of the User Interface of **DetailsView** or **ListView** based on any rule (e.g., highlighting an entry with a ZIP code that does not belong to Spain).

Point "1" can be implemented by adding a `CurrentEntry` property directly in `CustomerList`. However, this solution has some problems if we have more than one instance of UI connected to the same **CustomerList**.

www.EBooksWorld.ir

Point "2" can be implemented in the views (ListView and DetailsView), but there is a problem if we do that: if we want to change the rule, then we will need to change both views. Changes will start to affect multiple sites.

In short, it seems convenient to gradually add a third sub-layer in our application. We need a sub-layer between the views and the CustomerList model that saves the states shared by the views. In fact, we need a **ViewModel** concept.

## 5.2.2.- Design with the Model-View-ViewModel (MVVM) pattern

A ViewModel provides an abstraction that acts as a meta-view (a view's model), saving states and policies shared by one or a group of Views.

When introducing the **ViewModel** concept, our design will look like this:

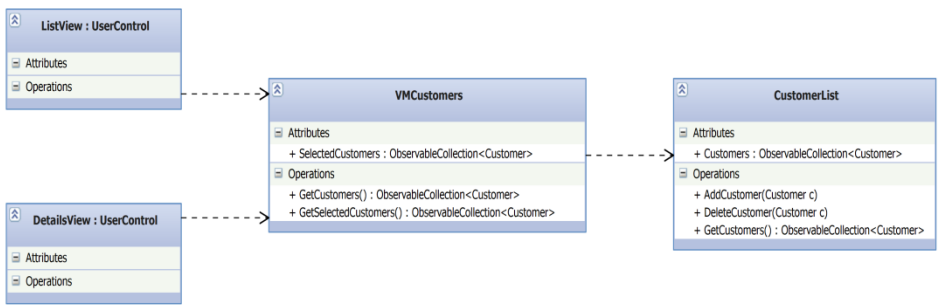



**Figure 18.- Application design using VIEW-MODEL**

In this design, the views know the **ViewModel** and are linked to its data, so they can reflect any change that occurs. The **ViewModel** does not have any reference to the **Views,** only a reference to the **Model,** in this case **CustomerList**.

For Views, the ViewModel acts as a facade of the model but it also acts as a way to share states among views (`selectedCustomers` in the example). In addition, the ViewModel usually exposes 'Commands'to which the Views can be linked.

### Using 'Commands' in MVVM WPF applications

WPF implements the Commands design pattern as an event entry programming mechanism. These commands allow decoupling between the origin and the action managed with the window, and has the advantage that multiple sources (view controls) can invoke the same command. In addition, the same command can be managed in different ways, depending on the target.

www.EBooksWorld.ir

From the user's point of view, a command is simply a user interface element property launched by the logical command after invoking an event handler directly (remember the problem we addressed at the beginning of this section). We can have multiple components of that user interface linked to the same command. In addition, we can do it directly from the XAML, so a designer only sees an order and a programmer will have the command's functionality in some part of its code (it is not recommended to be in the codebehind).

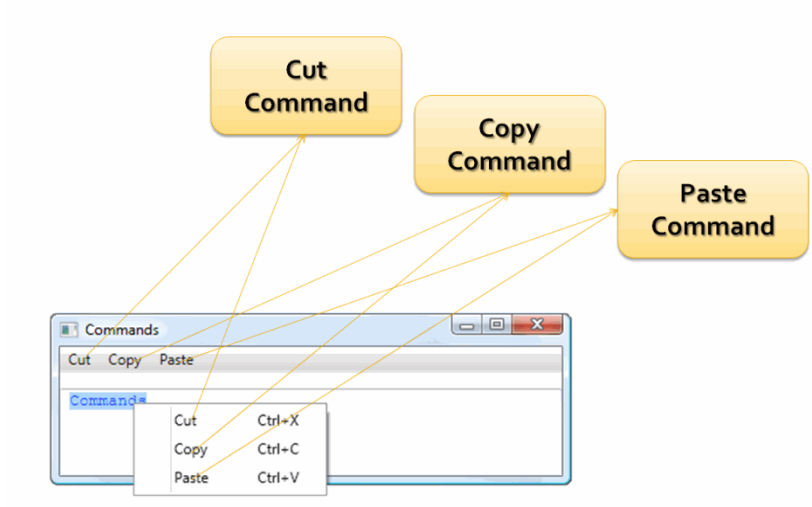



**Figure 19.- Using a Command**

There are also more functionalities that could be requested to the Commands such as requesting that if an action is not available, it cannot be launched. For example, if we have not modified anything in a document, why would we activate the Undo action? The XAML components of our application that launch the Undo action should be disabled.

All the 'commands' are implemented in the **ICommand** interface.

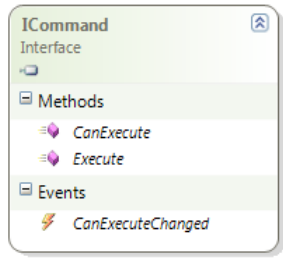



**Figure 20.- ICommand Interface**

www.EBooksWorld.ir

This interface consists of two methods and one event.

- **Execute**: Contains the logic of the action that must be implemented by the command in the application.
- **CanExecute**: Returns to the command state and tells us whether it is enabled or disabled.
- **CanExecuteChanged**: Each time the CanExecute value changes, an event will be launched informing us of this.

This would be an example of a command implementation:

```
public class SaveCommand : ICommand
{
   private CustomerViewModel _view;

   public SaveCommand(CustomerViewModel view)
   {
      _view = view;
   }

   public bool CanExecute(object parameter)
   {
     return true;
   }

   public event EventHandler CanExecuteChanged;

   public void Execute(object parameter)
   {
     //to do something, like saving a customer
   }
}
```

However, WPF provides a special class for Graphical Interface actions, called **RoutedCommand**. This is a command object that does not know how to perform the task it represents.

When asked if it can be executed (CanExecute) or when asked to be executed (Execute), the RoutedCommand delegates the responsibility to another command. Routed commands travel through the WPF element visual tree, giving each UI element the chance to execute the command that performs the task. Also, all the controls using RoutedCommand can be automatically "disabled" when they cannot be executed.

www.EBooksWorld.ir

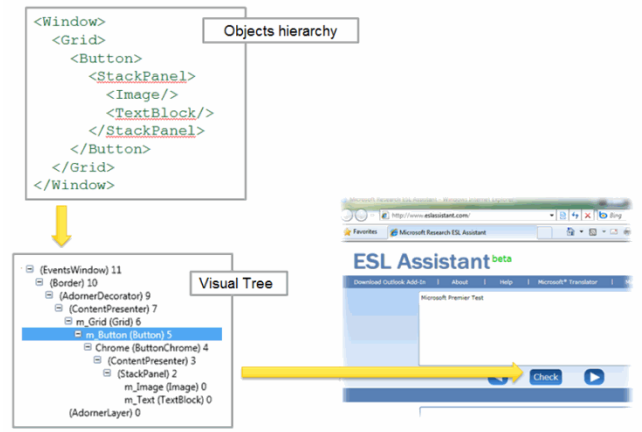



**Figure 21.- RoutedCommand in WPF**

In MVVM designs, however, the command's objectives are often implemented in ViewModels, though theseare not typically a part of the visual element tree. This requires the introduction of a different class of command: the Linkable Command (**DelegateCommand)** that implements **ICommand** and can have non-visual elements as objectives.

Considering linkable commands, the design applying MVVM would look like this:

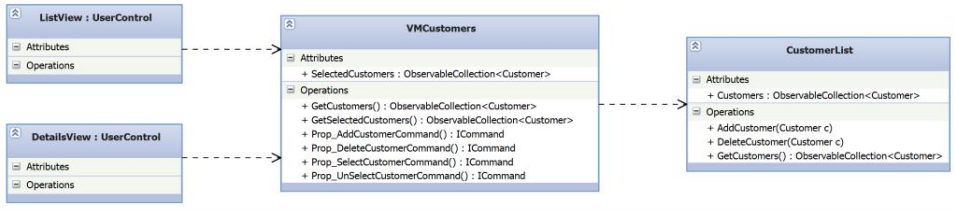



**Figure 22.- Application design using VIEW-MODEL and exposing linkable Command.**

A **DelegateCommand** allows views to be linked to the **ViewModel**. Although the diagram above does not show this explicitly, all the **ICommand** exposed by the **ViewModel** are **DelegateCommands**. For example, the code for **AddCustomerCommand** in the **ViewModel** would be something like this:

**C#**

```
public class VMCustomerList : ObservableObject
{
```

**ICommand** property for Edition of the selected Customer

```
   private ICommand  editCommand;

   public ICommand EditCommand
```

www.EBooksWorld.ir

```
    {
       if (_editCommand == null)
       {
          _editCommand = new DelegateCommand<Customer>(EditExecute,
                                                       CanEditExecute);
       }
         return _editCommand;
    }

    private void EditExecute(...) { ... }
    private bool CanEditExecute() { ... }
}
```

Creation of **DelegateCommand**

And the XAML code that uses this Command, to make it simple, would be something like this:

```
XAML de View

<UserControl>
...
<StackPanel>
...
<Button Content="Button" Command="{Binding EditCommand, Mode=OneWay}"
CommandParameter="{Binding SelectedItem, ElementName=listBox}"/>
...
</StackPanel>
...
</UserControl>
```

Binding with **Command** EditCommand

### Use of INotifyPropertyChanged in MVVM WPF applications

Finally, remember that in WPF there is an interface called INotifyPropertyChanged, which can be implemented to notify the user interface that an object's property has been modified and, therefore, the interface must update its data. This entire subscription mechanism is done by the WPF data links automatically. As explained above, when we want to return a set of objects we use the observable collection (ObservableCollection). However, when we have to go from the model to the view, passing only one object through the ViewModel, we will have to use this Interface.

This interface defines a single event called PropertyChanged that must be launched to inform us of the property change. Each model's class is responsible for launching the event whenever applicable:

```
public class A : INotifyPropertyChanged
{
    private string _name;

    // Event defined by the interface
    public event PropertyChangedEventHandler PropertyChanged;

    // Launching the "PropertyChanged" event
```

www.EBooksWorld.ir

```
        private void NotifyPropertyChanged(string info)
        {
            var handler = this.PropertyChanged;
            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(info));
            }
        }

        // Property that informs changes
        public string Name
        {
            get { return _name; }
            set
            {
                if (_name != value)
                {
                    _name = value;
                    NotifyPropertyChanged("Name");
                }
            }
        }
    }
```

This code is too long to be performed in classes with many properties, and it is easy to make mistakes. Thus, it is important to create a small class that prevents us from having to repeat the same code over and over again. Therefore, we recommend doing something like this:

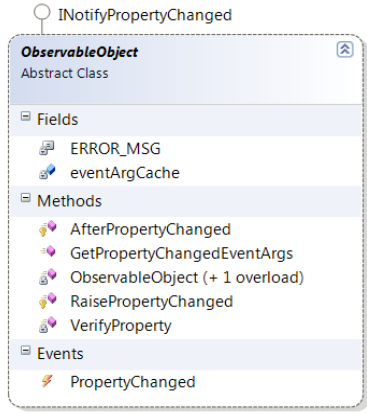



**Figure 23.- INotifyPropertyChanged Interface**

### Using INotifyPropertyChanged in MVVM WPF Applications

One of the most common tasks to be performed in MVVM applications is managing the loading of different views from the ViewModel, depending on the type of command that has been executed. If, for instance, we press a button in a view whose

www.EBooksWorld.ir

command is (ShowDetails) we almost certainly will need to go from that ViewModel that operates the command to another view - ShowDetailsView), which, in turn, will have another ViewModel (VMMostrarDetails – VMShowDetails).

We will always need to perform the same operations of Navigating to another view and assign the new ViewModel, so we recommend creating a class that implements this functionality.

```
public static void NavigateToView(UserControl view)
{
    NavigateToView(view, null);
}

public static void NavigateToView(UserControl view,
                                  ObservableObject viewModel)
{
   if (view != null)
   {
      ((MainWindow)App.Current.MainWindow).
                           ContentHolder.
                           Children.
                           Remove(NavigationController.currentView);

      if ( viewModel != null )
         view.DataContext = viewModel;

      ((MainWindow)App.Current.MainWindow).
                                        ContentHolder.
                                        Children.
                                        Add(view);
      NavigationController.currentView = view;
   }
}
```

## 5.3.- Implementing MVVM Pattern with Silverlight 4.0

There are two clear differences when implementing the MVVM pattern in Silverlight, with regard to the guidelines we have seen in WPF applications.

These two differences can be divided into the following categories:

1. Asynchronous programming model
2. Validation model

www.EBooksWorld.ir

## 5.3.1.- Asynchronous Programming Model

One of the main advantages of using the user interface architecture created from the Model-View-ViewModel Pattern is that we can leverage its asynchronous model to create interfaces that respond even to heavy tasks. Avoiding delay times between requests and response due to the network overload or tasks requiring very complicated calculations is one of the great benefits of using a **ViewModel**.

- A **ViewModel** object is executed in a different thread from the one responsible for constructing the interface (**DispatcherThread**), so no action started through a **ViewModel** object will affect the screen drawing process or the visual transitions carried out by that screen.

- Silverlight, in turn, forces us to use an asynchronous proxy model to use services referred to by our application, by using the **IAsyncResult** interface. Therefore, it is different from the service utilization model used by the WPF applications.

These two characteristics provide an application model that leads us, unequivocally, to obtaining an application with a high degree of response to end user interactions.

Let's look at an example of how our Silverlight application's client proxy object utilization methods are created (automatic generation):

```
private void OnGetPagedBankAccountsCompleted(object state)
{
  if ((this.GetPagedBankAccountsCompleted != null))
  {
    InvokeAsyncCompletedEventArgs e =
      ((InvokeAsyncCompletedEventArgs)(state));
    this.GetPagedBankAccountsCompleted(this,
    new GetPagedBankAccountsCompletedEventArgs(
      e.Results,
      e.Error,
      e.Cancelled,
      e.UserState)
    );
  }
}

Public void GetPagedBankAccountsAsync(
ServiceAgent.PagedCriteria pagedCriteria)
{
    this.GetPagedBankAccountsAsync(pagedCriteria, null);
}
```

Just as there are differences in the proxy object we generated to use the service that exposes our model's data, there are differences in the way the **ViewModel** classes are implemented.

www.EBooksWorld.ir

While the base implementation of our **ViewModel** remains unaltered with respect to *Windows Presentation Foundation*, the way the service is used really does contemplate a change in the body of the **Execute**() action of the **RoutedCommands**.

In *Windows Presentation Foundation* we have seen the use of the **BackgroundWorker** type to prevent the main thread of our application from being blocked, hindering interaction with our interface. The Silverlight service use model already shows us this asynchrony naturally.

We see this difference by comparing the implementation of the **Execute**() action in the two technologies:

Execution of the `GetPagedCustomer` action in WPF:

```
    using (BackgroundWorker worker = new BackgroundWorker())
    {
      worker.DoWork += delegate(object sender, DoWorkEventArgs e)
      {
     MainModuleServiceClient client = new MainModuleServiceClient();

     e.Result = client.GetPagedCustomer(new PagedCriteria()
                      { PageIndex = this._pageIndex, PageCount = 10 });
    };

    worker.RunWorkerCompleted += delegate(object sender,
                                          RunWorkerCompletedEventArgs e)
    {
    if (!e.Cancelled && e.Error == null)
    {
       List<Customer> customers = e.Result as List<Customer>;

       if (customers != null)
       {
         this.Customers = new ObservableCollection<Customer>(customers);
         this._viewData =
                    CollectionViewSource.GetDefaultView(this.Customers);
         this._viewData.Filter = null;
       }
    }
    Else
      MessageBox.Show(e.Error.Message, "Customer List",
MessageBoxButton.OK, MessageBoxImage.Error);
                    };

    worker.WorkerSupportsCancellation = true;
    worker.RunWorkerAsync();
    }
```

Execution of the `GetPagedCustomer` action in Silverlight:

```
try
{
   MainModuleServiceClient client = new MainModuleServiceClient();

   client.GetPagedCustomerAsync(new PagedCriteria() { PageIndex = 0,
                                                      PageCount = 10 });
```

www.EBooksWorld.ir

```
client.GetPagedCustomerCompleted += delegate(object sender,
                                GetPagedCustomerCompletedEventArgs e)
{
   Customer[] listCustomers = e.Result;
   if (listCustomers != null && listCustomers.Length > 0)
   {
       Customer = listCustomers[0];
       GetCustomerOrders();
   }
               };
}
catch (Exception excep)
{
  Debug.WriteLine("GetPagedCustomer: Error at Service:" +
                                              excep.ToString());
   }
```

## 5.3.2.- Validation Model

The second big difference when dealing with an application implementation using Silverlight is derived from the internal implementation of the validation model.

WPF uses types derived from the **ValidationRule** abstract class, where we define the customized logic of client validation and in which we connect a certain data binding (**Binding**). This is how WPF uses this intermediate object to validate data that travels from the View to our **ViewModel** linked to the view.

The main disadvantage of using the **ValidationRule** type comes from the behavior of **Binding** class. To clarify this, the validation rules are not checked until the value of the property related to the **Binding** object is modified and this change has been sent to the user interface. Therefore, if we want to use them explicitly, we must go through the object tree to invoke these validations in a procedural manner.

In Silverlight, this validation model has been modified to provide a better control of these tasks through the implementation of our **ViewModel** of the **INotifyDataErrorInfo** interface.

In the section about validations we will provide more details on the use of this interface.

## 5.4.- Benefits and Consequences of using MVVM

The use of the MVVM pattern provides several fundamental **benefits**:

- A ViewModel provides a unique state storage and presentation policy, which improves the reuse of the Model (decoupling it from the Views) and enables replacement of the Views (by removing specific presentation policies of the Views).

- A MVVM design makes it easier to run *Tests* (Unit Testing, specifically) on the application. When separating the logic from the views and visual controls,

www.EBooksWorld.ir

we can easily create unit tests exclusively responsible for the Model and *ViewModel* (since the Views will be normally just XAML, without *code-behind*). In addition, MVVM enables implementation of MOCKs in the Presentation Layer, because decoupling the Views from the Model and placing the client logic in the ViewModels makes this logic easily replaceable by MOCKs (execution simulation), which is essential for *testing* of complex applications.

- The MVVM pattern provides a decoupled design. The Views only reference the ViewModel and the ViewModel references only the Model. The rest is performed by the *databinding* and the *Commands* of the WPF infrastructure.

The **consequences** of using the MVVM pattern are:

- The typical relationship between a ***ViewModel*** and the corresponding ***Views*** is often 'one to many,' but in certain situations this is not true. In general, any client business logic and data entry validation business logic (element selection tracking, etc.) should be implemented in the *ViewModel*.

- There are situations where a ***ViewModel*** is "aware" of another ViewModel within the same application. These situations appear when there is a master-detail relationship between two *ViewModels* or when a *ViewModel* represents a loose element (for example, the visual representation of a single Client). When this happens, a *ViewModel* can represent a collection of *ViewModels*, as in the case shown previously.

## 6.- DATA VALIDATION IN THE INTERFACE (WPF)

What are Validation rules?

When we want to check that the value entered by the user in the interface is the right one or the expected one, we have the Validation Rules function in WPF.

Through the use of Validation Rules we can validate the value entered by the user before the linked property is updated. By default, we have the `ExceptionValidationRule` which allows us to catch exceptions when the binding data source is updated. The WPF binding engine will check every binding with the corresponding validation rule every time a value changes. In this case this means any view model property in this case, which will depend on the configuration of the binding itself, that is to say, it will depend on the value of the *Mode* and `UpdateSourceTrigger` properties.

We can also create our own validation rules by implementing the `ValidationRule` abstract class, whose Validation method will be the one executing the binding engine to check if the value is the right one or not. Moreover, in the response result we can include an error message that we can show the user interface to provide more information about why the entered data was not valid.

www.EBooksWorld.ir

```
    public class RequiredValidationRule: ValidationRule
    {
        public override ValidationResult Validate(object value,
System.Globalization.CultureInfo cultureInfo)
            {
                if (value == null )
                    return new ValidationResult(false,
    "This field is required");

                if (string.IsNullOrEmpty(value.ToString().Trim()))
                    return new ValidationResult(false,
    "This field is required");

                return new ValidationResult(true, null);
            }
        }
```

It is very easy to use a validation rule:

- First, we should add the xml namespace of the customized validation rules in the view where they will be used. In the example, this corresponds to the *AddCustomerView*.

```
xmlns:ValidationRules="clr-
namespace:Microsoft.Samples.NLayerApp.Presentation.Windows.WPF.Client.Va
lidationRules"
```

- Once we add this, we look for the binding where we want to include the validation rule and we modify it as follows:

```
<TextBox Margin="0" TextWrapping="Wrap" Grid.Column="1" Style="{DynamicR
esource TextBoxStyle1}" x:Name="txtCompanyName" GotFocus="txtCompanyName
_GotFocus" LostFocus="txtCompanyName_LostFocus"><Binding Path="CompanyNa
me" Mode="TwoWay" NotifyOnValidationError="True" UpdateSourceTrigger="Pr
opertyChanged" ValidatesOnDataErrors="True" ValidatesOnExceptions="True"
>
<Binding.ValidationRules>
<ValidationRules:RequiredValidationRule />
</Binding.ValidationRules>
</Binding>
</TextBox>
```

- Next, we should edit the TextBox control style and establish the *Validation.ErrorTemplate* property so that, in this case, the text box with an error appears with a red border. We should also add a trigger so that, when the *Validation.HasError* property is true, the tooltip of the text box is set with the error message.

```
<Style BasedOn="{x:Null}" TargetType="{x:Type TextBox}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type TextBox}">
                (...)
```

www.EBooksWorld.ir

```
</ControlTemplate>
</Setter.Value>
</Setter>
<Setter Property="Validation.ErrorTemplate">
<Setter.Value>
<ControlTemplate>
<Border BorderBrush="Red" BorderThickness="1" CornerRadius="5">
<AdornedElementPlaceholder></AdornedElementPlaceholder>
</Border>
</ControlTemplate>
</Setter.Value>
</Setter>
<Style.Triggers>
<Trigger Property="Validation.HasError" Value="true">
<Setter Property="ToolTip"
            Value="{Binding RelativeSource={RelativeSource Self},
            Path=.,
            Converter={StaticResource errorInfo}}"/>
</Trigger>
</Style.Triggers>
</Style>
```

Now that we have the visual part ready, we should modify the view code, so that its ViewModel is "aware" of the existence or non-existence of errors, and acts on them accordingly. To do so, there is an event that we can subscribe to. The event is **Validation.ErrorEvent** and it will be launched when there is a validation error, but only in those bindings where the **NotifyOnValidationError** property is true.

We are still in the *AddCustomerView* view and we subscribe to the aforementioned event.

```
private Dictionary<string, bool> _errors = new Dictionary<string,
bool>();

private void AddCustomerView_Loaded(object sender, RoutedEventArgs e)
           {
               //add handler for validation errors
               this.AddHandler(Validation.ErrorEvent, new
RoutedEventHandler(OnErrorEvent));
           }

   private void OnErrorEvent(object o, RoutedEventArgs args)
           {
               if (args.OriginalSource != null)
               {
                   TextBox txtBox = (TextBox)args.OriginalSource;

                   if (!_errors.Keys.Contains(txtBox.Name))
                       _errors.Add(txtBox.Name, false);

                   _errors[txtBox.Name] =
Validation.GetHasError(txtBox);
               }

               this.IsValidData = (this._errors.Where(k => k.Value ==
true).Count() == 0);
   }
```

www.EBooksWorld.ir

When the event is launched, we will be checking to see if the element has an error. We will then record this in an element dictionary to control those that have modified their error state and those have not. Finally, we should check to see if there are elements with errors and assign the value to the IsValidData property.

**Note:**
We can force validation of a text box through the use of the UpdateSource() method:

```
GetBindingExpression(TextBox.TextProperty).UpdateSource()
```

**Note**:
If we implement the same system to visually warn the user that there is an error in a text box, showing it with a red border (using *AdornedElementPlaceholder* when the displayed elements are superimposed on the text box with the error), the red border will probably still be seen. This is because the *AdornedElementPlaceholder* will search in the visual tree for the first *AdornerDecorator* that is to be painted. When this is not specified, the entire tree shares *AdornerDecorator*, so the border is superimposed on anything that is above the text box. The easiest way to avoid this is to place an *AdornerDecorator* in the user control that has text boxes, where validations are made.

## 6.1.- Data Validation in the User Interface (Silverlight)

The user interface data validation in Silverlight 4 using MVVM is different from what we have just seen in WPF.

In Silverlight 4 we have an **INotifyDataErrorInfo** interface that we will use in our Model Views. The advantage, as compared to WPF, is that we can stop worrying about the visual part since Silverlight 4 has a system to show the user if there is an error.

To use **INotifyDataErrorInfo,** we simply have to implement this interface. We have implemented it in the **ObservableObject** class, which is the base of all the view models. Implementation is very easy. We have created a series of help methods to add the properties with errors to a dictionary and to keep track of them as we did in the case of WPF.

www.EBooksWorld.ir

```
public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
private readonly Dictionary<string, List<string>> _currentErrors;

public IEnumerable GetErrors(string propertyName)
{
   if (string.IsNullOrEmpty(propertyName))
          return (_currentErrors.Values);

   MakeOrCreatePropertyErrorList(propertyName);
          return _currentErrors[propertyName];
}

public bool HasErrors
{
    get
    {
      return (_currentErrors.Where(c => c.Value.Count > 0).Count() > 0);
    }
}

void FireErrorsChanged(string property)
{
    if (ErrorsChanged != null)
           ErrorsChanged(this, new
DataErrorsChangedEventArgs(property));
   }

public void ClearErrorFromProperty(string property)
{
    MakeOrCreatePropertyErrorList(property);
    _currentErrors[property].Clear();
    FireErrorsChanged(property);
}

public void AddErrorForProperty(string property, string error)
{
    MakeOrCreatePropertyErrorList(property);
     currentErrors[property].Add(error);
    FireErrorsChanged(property);
}

void MakeOrCreatePropertyErrorList(string propertyName)
{
    if (!_currentErrors.ContainsKey(propertyName))
             _currentErrors[propertyName] = new List<string>();
}
```

Validation rule handlers will be created in the next example. In this version, however, we will only have one required rule implemented in the same **ObservableObject** class.

www.EBooksWorld.ir

The rule is simple, check that the property has a value and if not, add this property to the error dictionary and display the error.

```
public void CheckRequiredValidationRule(string propertyName, string
value)
{
   ClearErrorFromProperty(propertyName);
   if (string.IsNullOrEmpty(value) ||
                              string.IsNullOrWhiteSpace(propertyName))
   {
      AddErrorForProperty(propertyName, "This field is required");
   }

   RaisePropertyChanged(propertyName);
}
```

Finally, the only thing left is to display the validation rule in the property we want to validate, as follows:

```
    public string CompanyName
    {
        get { return _currentCustomer.CompanyName; }
        set
        {
            _currentCustomer.CompanyName = value;
            _currentCustomer.CustomerCode = (value.ToString().Length > 3)
                                ? value.ToString().Substring(0, 3) : null;
            CheckRequiredValidationRule("CompanyName", value);
        }
    }
```

Then, perform the binding in the **Text** property of the text box, but without forgetting to set the **ValidatesOnNotifyDataErrors** binding property as true.

```
{Binding CompanyName, Mode=TwoWay, ValidatesOnNotifyDataErrors=True}
```


**Figure 24.- Validation in Silverlight**

www.EBooksWorld.ir

# 7.- IMPLEMENTING MVC WITH ASP.NET MVC

ASP.NET MVC is the implementation of the MVC pattern for the web. This implementation is based on the ASP.NET platform and therefore allows us to use many existing components such as authentication, authorization or cache.

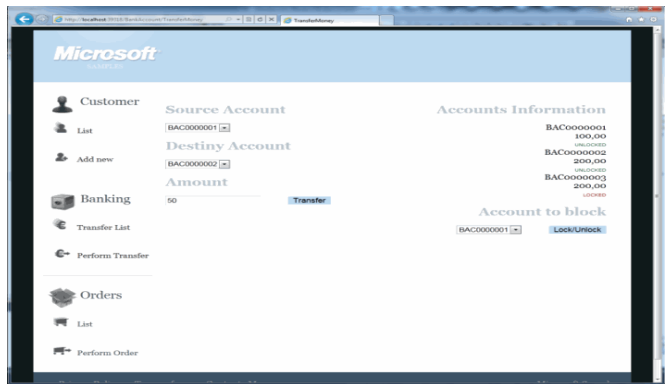



**Figure 25.- MVC – 'Bank Transfers'**


**Figure 26.- MVC – View of "Customer list"**

www.EBooksWorld.ir

## 7.1.- Basics of ASP.NET MVC

ASP.NET MVC organizes the user's interaction with the system of controllers that expose actions responsible for modifying or querying the system model. These actions return a result that, after being executed, returns an answer to the user. Let's look at the pipeline in more detail in order to understand it more easily.

## 7.2.- The ASP.NET MVC Pipeline

When an HTTP request appears, the Routing system tries to find an associated controller and action to answer the request based on the content and the same URL. After finding the proper controller, it requests an instance of it from the controller factory. Once it gets an instance from the controller, it searches within itself for the action to be invoked, maps the action parameters based on the content of the request (ModelBinding) and invokes the action with the parameters obtained. The controller action interacts with the system and returns a result. This result can be a file, a redirection to another controller, a view, etc. The most common case is the view (ViewResult). The results of an action are derived from ActionResult, and it is important to understand that they simply describe the result of the action and nothing happens until they are executed. In the case of a ViewResult, when executed, a ViewEngine is invoked, which is responsible for finding the requested view and rendering it to the HTML sent as a response.

www.EBooksWorld.ir

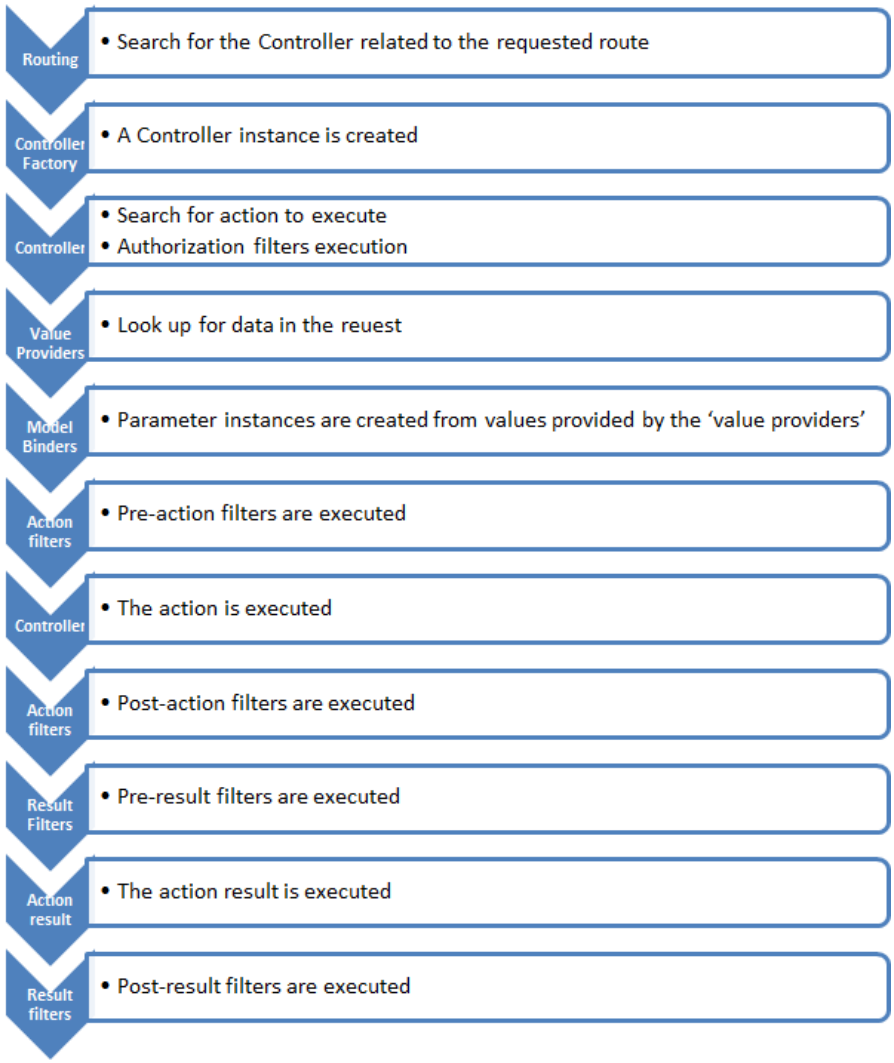



**Figure 27.- ASP.NET MVC Pipeline**

A controller communicates with a view through an object called ViewData, which is a dictionary in charge of storing the information necessary to render the view. Views can be strongly typed. In the case of being strongly typed, the view is a closed generic class (all the parameters of this type have been specified) and the ViewData exposes a Model property with the type of generic parameter of the view.

As a result, the intention of the view is better expressed, its contract with any controller is more explicit and clearer, and we have Intellisense to access the members of the model while coding the view.

www.EBooksWorld.ir

## 7.3.- A Complete Example: Customer's Update

The first thing to do is delineate the scenario that we are implementing. Typically, in any MVC applications, updating an entity requires two actions: one to view a form with the entity we want to modify and another to send changes to the server. It is important to note that in this type of application we try to follow the REST protocol (Representational State Transfer) and this will be reflected in the controllers. Let's first look at what the two actions are like:

```
public ActionResult Edit(string customerCode)
{

Customer customer = _customerService.FindCustomerByCode(customerCode);
    return View(customer);
}

[HttpPost]
public ActionResult Edit(Customer customer)
{
    try
    {
         customerService.ChangeCustomer(customer);
        return RedirectToAction("Details", new
                          { customerCode = customer.CustomerCode });
    }
    catch
    {
            return View();
    }
}
```

As we can see, we have a primary action to request the customer we want to edit and a secondary action to send the changes back. All the controller's work is divided into two responsibilities: to modify or query the domain, and to decide the screen to be displayed.

Let's look at the update view. If we are perceptive, we will realize that the first action is responsible for entering the current data in the form and the second action is responsible for processing that data.

If we take a look at the customer´s edition view, we see that it is divided into two parts; on one hand, the edition view corresponding to the editing action that is in charge of fitting the view within the page's global design, and linking the form to the corresponding action:

www.EBooksWorld.ir

```
<% using (Html.BeginForm("Edit","Customer",FormMethod.Post,
                                new { enctype = "multipart/form-
data"}))
{ %>
   <fieldset>
   <legend><%: ViewData.ModelMetadata.ModelType.Name %></legend>
    <%: Html.EditorForModel() %>
    <%: Html.SerializedHidden(Model) %>
    <input type="submit" value="Editar" />
    </fieldset>
<%} %>
```

On the other hand, the customer form view is responsible for showing the editable fields of a customer and performing as well as displaying their validation result:

```
   <div class="display-label">
   <%: Html.DisplayNameFor(x => x.CustomerCode) %></div>
   <div class="display-field">
   <%: Html.EditorFor(x => x.CustomerCode) %></div>
   <div class="validation-field">
   <%: Html.ValidationMessageFor(x => x.CustomerCode) %></div>
   <div class="display-label">
   <%: Html.DisplayNameFor(x => x.ContactName) %></div>
   <div class="display-field">
   <%: Html.EditorFor(x => x.ContactName) %></div>
   <div class="validation-field">
   <%: Html.ValidationMessageFor(x => x.ContactName) %></div>
   <div class="display-label">
   <%: Html.DisplayNameFor(x => x.ContactTitle) %></div>
   <div class="display-field">
   <%: Html.EditorFor(x => x.ContactTitle) %></div>
   <div class="validation-field">
   <%: Html.ValidationMessageFor(x => x.ContactTitle) %></div>
   <div class="display-label">
   <%: Html.DisplayNameFor(x => x.CompanyName) %></div>
   <div class="display-field">
   <%: Html.EditorFor(x => x.CompanyName) %></div>
   <div class="validation-field">
   <%: Html.ValidationMessageFor(x => x.CompanyName) %></div>
   <div class="display-label">
   <%: Html.DisplayNameFor(x => x.IsEnabled) %></div>
   <div class="display-field">
   <%: Html.EditorFor(x => x.IsEnabled,true) %></div>
   <div class="validation-field">
   <%: Html.ValidationMessageFor(x => x.IsEnabled) %></div>
   <div class="display-field">
   <%: Html.EditorFor(x => x.Country) %></div>
   <div class="display-field">
   <%: Html.EditorFor(x => x.Address) %></div>
   <div class="display-field">
   <%: Html.EditorFor(x => x.CustomerPicture) %></div>
```

The most important aspect we have to consider in this type of application is how we manage this state. When we update an entity we are modifying an existing state, so, to properly apply changes, we must have the original state. Both possible storage points of this original state are the server (database or distributed cache) or the client. We

www.EBooksWorld.ir

opted to store this original state in the client, so we sent a serialized copy of the original data to the page, using HtmlHelper, which we have designed for that purpose and we can see in the edition view:

```
<%: Html.SerializedHidden(Model) %>
```

We have finished the action responsible for displaying the form with data to be edited, but once we have made the changes, how are such changes applied to the entity? If we know a little about the HTTP protocol, we know that when we "post" a form, we are sending name=value pairs in the request. This is far different from the parameters of the action responsible for saving changes in the entity that directly receives a client object. The process of translation between the request elements and the action parameters is called ModelBinding. ASP.NET MVC provides a ModelBinder by default, in charge of performing this translation based on conventions.

However, since we are using self-tracking entities, we have created our own ModelBinder to simplify the update process. As can be expected, at some point a ModelBinder needs to create an instance of the class to which it is mapping the request parameters. Thanks to the extensibility of ASP.NET MVC, we precisely extend this point, and if we have serialized the original entity, we return it instead of creating a new copy. By mapping the parameters, the entity can keep track of what parameters have been changed, simplifying the update process:

```
   public class SelfTrackingEntityModelBinder<T> :
        DefaultModelBinder where T : class, IObjectWithChangeTracker
{
    protected override object CreateModel(ControllerContext controllerCo
ntext, ModelBindingContext bindingContext, Type modelType)
    {
       string steName = typeof(T).Name+"STE";
       if(bindingContext.ValueProvider.ContainsPrefix(steName)){
           var value = bindingContext.ValueProvider.GetValue(steName);
           return new SelfTrackingEntityBase64Converter<T>()
                           .ToEntity(value.AttemptedValue);
       }
       return base.CreateModel(controllerContext, bindingContext,
   modelType);
    }
   }
```

www.EBooksWorld.ir

## 7.4.- Other Aspects of the Application

An MVC application is an ASP.NET application sub-type, so we have a global.asax file that is the access point for our application. Typically, all the necessary initializations are made in this file. In the case of ASP.NET MVC these initializations are often the following:

- Registration of URL mapping routes / (Controller, Action).
- Registration of Dependencies in the container.
- Registration of specialized controller factories.
- Registration of alternative view engines.
- Registration of specific model binders.
- Registration of alternative value providers.
- Registration of areas.

An MVC application can be divided into areas in order to improve its organization. Each area should be responsible for making all the indicated records, as necessary. This code should first be in the global.asax file (within the Application_Start method) but we can assume that in a large application this method can reach a large number of code lines. Therefore, although we have not created areas, we have delegated the responsibility of each "area" (we consider the application as the only defined area) in a BootStrapper responsible for initializing each area (Registering dependencies in the container, Model Binders, etc.)

To be more specific, each area is responsible for registering its dependencies in the dependency container (mainly the controllers). In the framework extensions, a factory is defined that uses the dependency container to create the controllers and automatically inject its dependencies.

www.EBooksWorld.ir

CHAPTER 9

# Cross-Cutting Infrastructure layers

## 1.- CROSS-CUTTING INFRASTRUCTURE LAYERS

Most applications have common functionality used by the different logical layers and even in different physical *Tiers*. In general, this type of functionality covers operations like authentication, authorization, cache, exception management, logging, tracking, instrumentation and validation. **All these aspects are commonly called "Cross-cutting aspects" or "Horizontal aspects" because they affect the entire application, and we should be able to reuse these components from the different layers**. For example, the code that generates traces in the application log files is probably used from many different points of the application (from different layers and even tiers). If we encapsulate that code in charge of performing traces tasks (or any another action) we will be able to change this aspect behavior in the future by changing the code only in a single specific area.

This chapter intends to help you understand the role played by these cross-cutting aspects in the applications, and learn to solve the typical problems when designing and implementing cross-cutting aspects.

There are different approaches in order to design and implement these functionalities, starting from simple class libraries as '*building blocks*', or even going further by applying **AOP (*Aspect Oriented Programming*)** techniques, where metadata



www.EBooksWorld.ir

is used to inject cross-cutting aspects code during compilation time or during runtime (using **calls interception**).

## 2.- CROSS-CUTTING INFRASTRUCTURE LOCATION IN THE N-LAYERED ARCHITECTURE

The following diagram shows how these cross-cutting aspects typically fit within our Architecture:

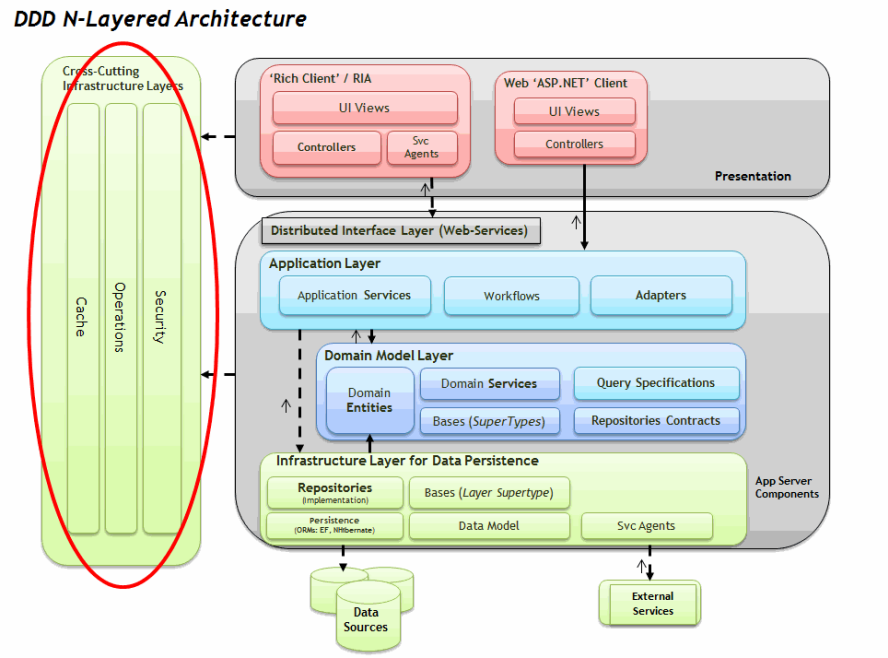



**Figure 1.- Cross-cutting aspects location in DDD N-Layered Architecture**

**Following the general architectural guidelines in DDD, we definitely call it "*Cross-cutting Infrastructure Layers*", because these components are also part of the Infrastructure Layers or layers associated with specific technologies** (specific Security APIs, specific instrumentation APIs, *logging*, etc.).

www.EBooksWorld.ir

## 3.- GENERAL DESIGN CONSIDERATIONS

These guidelines will help us understand the main factors to be considered in this work area:

- Examine the functions required by each layer, and search cases when this functionality can be abstracted to common general purpose components for the entire application that can be configured depending on the specific requirements of each layer. These components can probably be reused by several applications.

- Depending on how the components and application layers are physically distributed, we might need to install cross-cutting components in more than one physical *Tier*. However, even then, we benefit from reusing components and we gain development cost/time reductions.

- Consider the use of abstractions (interfaces) and *"Dependency Injection"* techniques to inject dependencies to cross-cutting components. Doing this will make it easier to change the infrastructure technology in the future with a lower impact on our application.

- Consider using third party libraries (these are usually highly configurable), for implementing common cross-cutting aspects (Cache, logging, etc.).

- Consider AOP (*Aspect Oriented Programming*) if you are looking for a clear and elegant cross-cutting aspects injection so you will not need to mix cross-cutting code with your own application logic from each layer (for example, using attributes as aspects implementation). Many of the IoC containers used for "Dependency Injection" also offer "Call Interception" features, and therefore, a possible way to implement 'AOP aspects'.

www.EBooksWorld.ir

**Table 1.- Architecture Framework Guide**

**Rule # D29.**

**Application Cross-cutting areas have to be identified and implemented as Horizontal/Cross-cutting Aspects, reusable by the different layers.**

**Rules**

- It is important to identify the areas of the application that will be using cross-cutting aspects and we should avoid copying/pasting that code on every method. We must extract and locate cross-cutting aspects in specific reusable projects (Building blocks).

**Advantages of using cross-cutting components**

- Components reuse
- Homogenization of cross-cutting aspects existing in several application layers with subsequent code optimization.

**References**

*'Crosscutting concerns' in '.Net Application Architecture Guide' of Microsoft Pattern & Practices:*
**http://www.codeplex.com/AppArch**

www.EBooksWorld.ir

## 4.- CROSS-CUTTING ASPECTS

The following areas or aspects are generally considered as part of these cross-cutting infrastructure layers:

- Security
  - Identity
  - Authentication
  - Authorization
  - *Claims*-Oriented Security Architecture
- Cache
- Configuration Management
- Exception Handling
- Instrumentation
- *Logging*
- State Management
- Input Data Validation

### 4.1.- Application Security: Authentication and Authorization

The sole consideration of architecture security and application development could be enough to fill an entire book, because there are many security concepts, and it depends on the working area (secure code and security holes, secure communications, encryption, electronic signature, authentication, authorization, cryptographic technologies, etc.). In this guide, we do not intend to discuss all the possibilities concerning application security because we would generate an excessive volume of documentation about something that is not the core of our DDD N-Layered Architecture.

www.EBooksWorld.ir

However, there are two important areas with regard to securing applications which should always be considered because they are necessary for most applications and have to be dealt with by end users. The first area is **Identity**, used to identify the users accessing the application. The second area is **Authorization**, used to check access and, if applicable, to grant access to the users over the resources (functional areas) of the application.

At a technological level there are several possibilities associated with authentication and authorization. For authentication, we can use the well-known user-password credentials, corporate authentication based on *Windows Active Directory*, other LDAP directories, client X.509 Certificates, etc. Regarding authorization to access application resources, there are also many technical options such as *.NET Windows Roles, .NET Custom Roles, Membership-Roles, Authorization Manager (AzMan), WIF (Windows Identity Foundation) and Claims based security*, etc.

Due to the great amount of existing variables, we do not intend to explain all the architecture and technical options of authentication/authorization. Rather, we have chosen to select a single Security option that is suitable for the context we are addressing herein (complex enterprise applications).In most cases these **applications must be able to be integrated with existing corporate security in the organization (or even, outside, in 'The Cloud')**. Therefore, the security type we selected is the new security wave named *"Claims-based Security"*, which is supported by all the companies in the industry (Microsoft, IBM, SAP, Oracle, etc.). We will explain it in detail, both at a theoretical/logical level and at a Microsoft technology implementation level (technologies like WIF: *Windows Identity Foundation* and *ADFS 2.0*).

## 4.1.1.- Authentication

Designing a correct authentication is fundamental for application security. Otherwise, the application may be vulnerable to spoofing attacks, dictionary attacks, session hijacking and other types of attacks. Consider the following guidelines concerning authentication:

- Identify trust boundaries and authenticate users and calls crossing such trust boundaries.
- If you use user/password authentications, the passwords should be "strong", meaning that they should meet the minimum complexity requirements (alphanumeric, minimum length number, number inclusion, etc.).
- If you have multiple systems in the application, or **if users should access multiple applications using the same credentials (common corporate**

www.EBooksWorld.ir

**requirement), consider using a '*single sign-on*' strategy related to a "Claims based Security".**

- Never transmit passwords in plain text through the network and do not store these passwords in plain text in a database or store. Instead, it is a better approach to save '*hashes'* originated from these *passwords*.

## 4.1.2.- Authorization

Designing a correct authorization is essential for application security. Even after having designed a correct authentication, if we do not design and implement the authorization system properly, the Authentication process itself does not suffice, and the application would be vulnerable to privilege increase, information discovery and unauthorized data manipulation. Consider these general guidelines related to authorization:

- Identify trust boundaries, authorized users and authorized callers to cross through such trust boundaries.

- Protect the resources by applying authorization to the callers, based on their identity, groups, roles or even special *claim* types. Minimize the number of roles whenever possible.

- 'Permissions & Roles' versus 'Roles': If the application authorization is complex, consider using a fine-grained system instead of simply making use of groups/roles. This means using the required *permissions* to access a resource. In turn, these permissions will be assigned to application roles and, in parallel, users have to be assigned to such application roles. The use of permissions is very powerful because users and roles are decoupled from the application implementation. This, in turn, is because resources will only be coupled to the required permissions and not to any specific user role in the application code.

- Consider using authorization based on resources to perform system audits.

- Consider using **authorization based on *Claims*** when a federated authorization should be supported in an information combination such as identity, role, rights/permits and other factors. The authorization based on Claims provides additional abstraction levels, which helps to decouple Authorization Rules from the authentication mechanisms. For example, a user may be authenticated with a certificate or username and password, therefore he has to provide this set of claims (within a security token granted by the STS) to the application, which determines the access to the resources. The advantage of the authorization based on Claims is that we leave our application authorization decoupled from the users´ authentication type, being able to potentially accept any authentication technology type, thanks to the intermediary role called STS

www.EBooksWorld.ir

(*Security Token Service*). **These latest authentication trends: '*Claims based Security and STS*' are precisely the options chosen by this Architecture Guide as a preferred system.** Its implementation and design will be explained in detail, as well.

## 4.1.3.- Security Architecture Based on 'Claims'

Identity management is, from any point of view, a real challenge. There are thousands of applications, business applications, and Websites on the Internet, so there are also thousands of credential types. However, the summary from the user's point of view, could be: "I don't want to be re-writing my passwords over and over again to use my enterprise applications". In addition, they do not want to provide multiple credentials for multiple applications, which the user could state like this: "I don't want to have a different user and password for each application I have to use".

In short, the objective is to simplify the user's experience when dealing with identification in applications. This is known as '*Single Sign-on*'.

A clear and very well-known example of '*Single Sign-on*' is the one provided by the *Windows Active Directory*. When your user belongs to a Domain, the password is written and provided by the user only once a day, at the beginning of the day (when the computer is switched on and he logs-in). This ID provides access to the different resources of the internal network, like printers, file servers, proxy to access the Internet, etc. Surely, if we had to write our password every time we accessed any of the resources above, we would not be happy with it. We are used to the transparency provided by "*Windows Integrated Authentication*".

Ironically, the popularity of *Kerberos* has decreased over time because it does not offer a real flexible solution and inter-organization through the Internet on its own. This is because the Domain controller has all the passwords for all the organization resources but it is closely protected by *firewalls*. If we are outside the office, we are often required to connect through the VPN (*Virtual Private Network*) to access the corporate network. Also, *Kerberos* is inflexible in terms of information supplied. It would be useful to be able to extend the *Kerberos* ticket to include **arbitrary attributes (*Claims*)** such as the email address or application roles. Currently, however, this is not one of *Kerberos'* abilities.

At a generic level, and without being linked to any platforms, *Claims* were designed to provide the flexibility we do not have in other protocols. The possibilities are limited only by our imagination and the policies from each IT department.

The standard protocols to exchange *Claims* have been specifically designed to cross security boundaries, such as perimeter securities, *firewalls* and different platforms, even from different organizations. In conclusion, the purpose is to make secure communication easier between different environments and contexts.

www.EBooksWorld.ir

**Claims**:
Claims-based identity promotes separation of concerns.
Claims decouple applications from the *Identity* and *Authentication* details. Through this approach, the application itself is no longer responsible for authenticating users.

**Real life example**: In order to simplify and completely understand the 'Claims Orientation concept', we will compare it with real life, where we are also surrounded by *Claims*. A very suitable analogy is the "authentication protocol" we follow every time we go to the airport to take a plane.

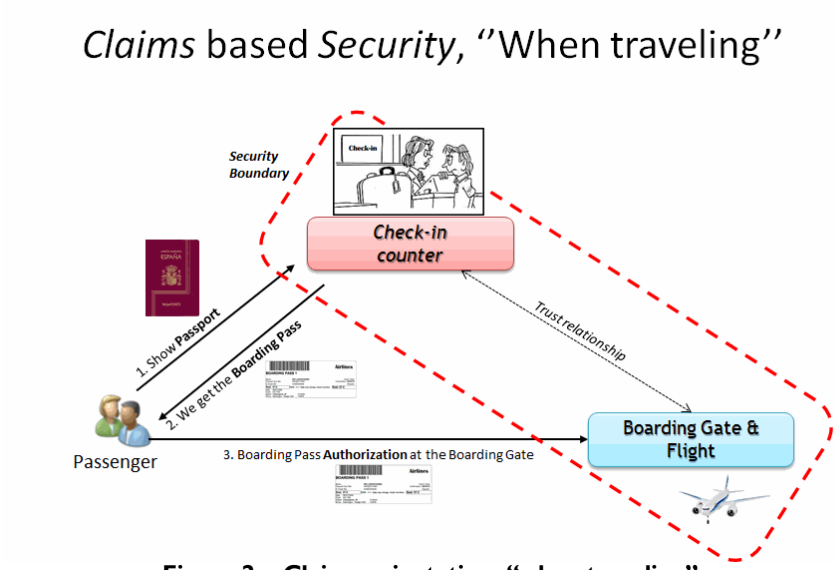



**Figure 2.- Claims orientation, "when traveling".**

We cannot simply reach the boarding gate showing our ID or Passport. Instead, we are required to go to the Check-in counter (comparable to the "Granter" or *Security Token Service*) where we must check in and check in our luggage, if applicable. At this desk or check-in counter we are required to show our initial credentials depending on where we are traveling (similar to the identity credentials used by the Organization, for example, Active Directory). If we are traveling to a destination within our country, we need our ID document at least. If the trip is an international flight, our passport will be required. When traveling with kids, we are required to provide their names, which will be added to our flight information (More data, 'we add other types of *Claims*'). Once our initial credentials are checked (ID document/Passport) by simply examining our faces and checking that they match the document picture (Authentication) and if everything is ok, we will receive a boarding pass exclusively valid for our flight (granting of the security token and set of claims for my application).

www.EBooksWorld.ir

A boarding pass provides a lot of information. The boarding gate personnel will know our name, if we are "Frequent Flyers" with special distinction, (authorization in the application and customization), our flight number (our application), and our seat number (authorization to access a resource, which is the seat). And most importantly, once we go through the boarding gate (security boundary of our application) we are usually only required to show our boarding pass (application security token with a set of claims) while in the airplane.

There is also some very particular information on the boarding pass that is encrypted with a barcode or magnetic strip. That is the evidence that the ticket was issued by an airline and it is not a fake one (this is comparable to an electronic signature).

Essentially, a boarding pass is a set of signed claims (*Security Token*), prepared for us by the airline. It states that we are allowed to get on a certain flight, at a certain time, on a certain seat.

It is also interesting to point out that there may be different ways to obtain the boarding pass (set of claims). We can buy it through the Internet or in a self-service machine at the Airport. The boarding gate staff does not care about the method we used, they will simply authorize us to go in.

In software applications, **this set of claims granted by a STS is called**, as we mentioned before, the **SECURITY TOKEN**. Each token is signed by the "granter"/STS that created it.

**An application using Claims based Security will be able to authenticate users if they show a valid security token issued by a trusted STS.**

Translating this into an application scenario using claims based security, we have the following diagram. In this case though, the diagram items are replaced by software components:

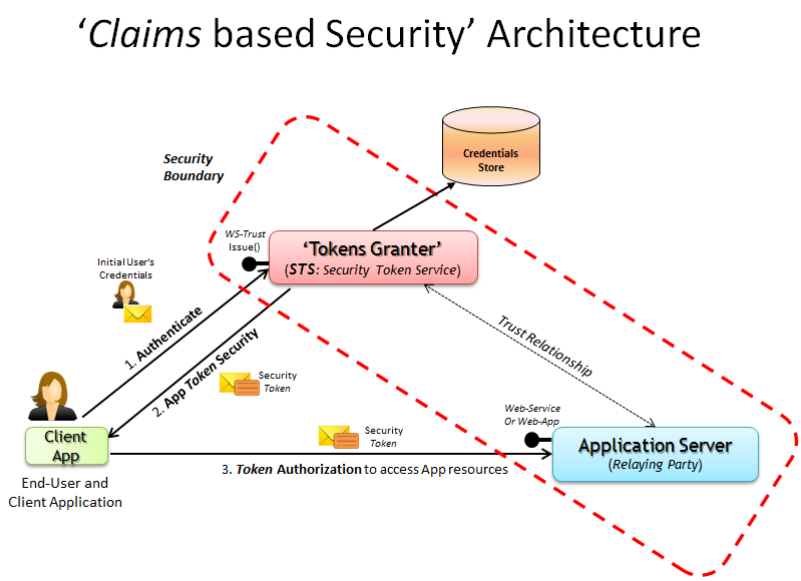



**Figure 3.- Claims based Security process**

www.EBooksWorld.ir

All our application needs in order to authorize the users is a security token provided by the STS (token issuer or *Security Token Service*).

If the IT department decides to update/change the security platform and identity, our application will not be invalidated and the design or implementation will not have to be changed. For example, if a 'smart-card' with a X.509 certificate is now required instead of a username and password, or any other type of identity, since our application is decoupled from the authentication process we would not need to change our application implementation/code or our application configuration at all.

**Advantages when decoupling the authentication process from our application:** For the application development, the advantage is clear: the application itself does not need to worry about the credentials type initially provided by the user. The IT department of the company/organization will have already taken care of that. **Our application will only work** with an equivalent to a Boarding pass, **an application security *token***, and our programming code will only be related to these application tokens, regardless of the initial credentials provided by the user to the STS (Whether they are Kerberos-AD, User-password, Certificates, WLID, FacebookID, GoogleID, etc.).

Undoubtedly, the Domain Controllers (in a Microsoft private network) or any other types of technology (LDAP directories, etc.) will still have to store their credentials in their data source. Trust relationships between identity systems and authentication systems will also still depend on political aspects. The Claims-based Security will not change any of those issues. However, when placing a 'claims layer' on our systems and by performing such decoupling, we can achieve the final purpose of the '*Single Sing-on*'. Furthermore, we will be open to any Identification and Authentication technology (not just Microsoft Active Directory, but also any other LDAP directory, Certificates, Internet authentications, etc.).

**Note**:
Claims-based Security integrates with existing security systems to allow higher compatibility between such systems and our applications. **Claims-based security provides security interoperability to our applications, open to any type of Identity technology.**

### Types of Claim-Oriented Archetypes

Depending on the type of application we are implementing (Web, Rich-Client, etc.) the architecture approach may be slightly different. For example, in a traditional Web

www.EBooksWorld.ir

application (HTML based on the browser) a slightly different technique will be used, as compared to an N-Tier application (*Rich* or *RIA*), in the way Claims are communicated from the STS to the application.

In conclusion, the purpose of these architectures is to allow a federation with a rich client or a browser (IE or any other browser).

- **Rich Client:** federation with a rich client is based on the advanced SOAP specifications of WS-*. We are specifically based on ***WS-Trust*** and ***WS-Federation Active Requestor Profile***. These protocols describe the communication flow between rich clients (such as Windows client applications) and Web services (such as WCF Services) to request a security token from the "issuer" (STS) and then to pass that token to the application web-service, so that it performs the authorization process.

- **Web Client**: federation with a Web client is based on ***WS-Federation Pasive Requestor Profile***, which describes a similar communication flow between the browser and the Web application in the Server. In this case, it is based on a browser redirection, HTTP GET and HTTP POST to request and pass security *tokens*.

**Table 2.- Security Rule**

**Rule # D30.**

**Use "Claims-based Security" as the preferred system for complex business applications that should be integrated into corporate identity systems or several Internet-ID systems.**

**Rules**

- Complex business applications should usually be transparently integrated into the corporate Identity systems. This means using a '*sign-on*' available in the organization and not forcing users to have different/specific credentials for our application.

**Advantages of using Claim-Oriented security**

- Transparency and corporate credentials propagation
- The Authentication system will be decoupled from the Application Authorization system.

www.EBooksWorld.ir

**References**

*A Guide to Claims–based Identity and Access Control*
**http://msdn.microsoft.com/en-us/library/ff423674.aspx**

## 4.2.- Cache

A *Caching* system can drastically increase application performance. All we need to know is at which points of the application we can or cannot use a caching system. You should also keep in mind that using a caching system incorrectly can deteriorate application scalability.

We should use the cache to optimize data search, avoid remote communications and in general to avoid duplicate processing. When implementing cache we should decide when to load data in the cache and when to eliminate expired data. .

It is better to pre-load frequently used data in an asynchronous way or use batch processes that avoid delays to the client.

Consider these guidelines when designing an application cache.

- **Cache location**: it is essential to choose the cache location correctly. If the application is deployed in a *Web-Server-Farm*, avoid using local cache for each farm node (e.g., using *in-proc* ASP.NET sessions in the Web Server processes memory space). This will cause the session system to fail if the load balancing is made in a pure way (no affinity). Instead, consider using distributed caches synchronized between different cache servers.

- **Cache in prepared format**: When using cache, consider using data in a prepared format. For instance, instead of simply caching simple text-data, cache serializable objects that simultaneously act as entities.

- Do not cache all volatile data and never cache sensitive/critical data unless encrypted.

- For long life operations, do not depend on the existence of certain data in the cache, because it may have been eliminated. Implement a mechanism to manage cache failure, for example re-loading the element from its original source, etc.

- Special care should be taken when accessing the cache from multiple threads. In this case, ensure all the accesses to cache are '*thread-safe*' to maintain consistency.

www.EBooksWorld.ir

**Table 3.- Cache rule**

|  Rule # D31. | **Use of CACHE in the application whenever is possible** |
|---|---|
| **Rules**<br>• You should use cache for continuous access to static data or data that is not constantly changing.<br>• The access to a database server is expensive insofar as connection creation, access or data transportation are concerned; however, by using intermediate cache this performance can be improved. | |

## 4.3.- Configuration Management

Designing a suitable configuration mechanism is important for the flexibility of our application. Consider the following guidelines concerning Configuration Management:

- Consider what characteristics should be configurable. Check that there is a real business need for each configurable characteristic and simplify the configuration, exposing the minimum possible configuration options. Excessive complexity in the configuration may cause a system to be too complex as regards management and maintainability, generating malfunctioning and even security holes due to incorrect configuration.

- Decide if the configuration will be saved centrally and if it will be applied to users upon application start-up (for example, based on Active Directory policies). Consider how access to the configuration information will be restricted, and use processes running with the minimum privileges level using controlled service accounts.

- Encrypt sensitive information in the configuration store. For instance, encrypt sensible sections within the .config file.

www.EBooksWorld.ir

- Categorize the configuration elements in logical sections if the application has several physical tiers. If the application server is run in a '*Web-Farm*', decide which parts of the configuration are shared and which ones are specific for each node/machine where the application is executed. Then, choose the proper store for each section.

- Provide a User Interface especially made for Administrators, through which they can edit the configuration information.

## 4.4.- Exception Handling

Designing a suitable exception handling strategy is important when dealing with application security and stability. Otherwise, it may be very hard to diagnose and solve application issues. Correct exception handling is especially important when the application is suffering attacks such as DoS (*Denial of Service*) and sensitive information could be shown because of internal errors/exceptions generated.

A suitable approach is to design a centralized exception management mechanism, and consider providing exception management publishing points (such as WMI events and SNMP traps) in order to support enterprise monitoring systems such as *Microsoft System Center*, *HP OpenView*, *IBM Tivoli*, etc.

Consider the following general guidelines about exception handling:

- Design a proper strategy for exceptions propagation that encapsulates or replaces exceptions (internal errors) or adds extra information as required. For instance, allow exceptions to bubble up to the higher layers until they reach the "boundary layers (such as Web Services or ASP.NET Web Presentation Layer) where such exceptions will be recorded (*logs)* and then transform them as necessary, before passing them to the next layer (usually, before they reach the UI or presentation layer).

- Consider including a context identifier so that related exceptions can be identified and associated throughout the different layers and identify the root cause of the errors and faults. Also, make sure the design considers the unhandled exceptions.

- Do not 'Catch()' errors/exceptions unless you are going to manage them or it is really necessary to add more information.

- Never use exceptions to control the application flow.

- Design a proper critical error notification and logging strategy that stores enough information about the problem so that the application administrators

www.EBooksWorld.ir

can reproduce the scenario. At the same time, the system must not reveal confidential information to the UI (in messages the end-user might receive).

## 4.5.- Record/Logging and Audits

Designing a suitable Logging and Instrumentation strategy is important when dealing with application security and diagnosis. Otherwise, the application may be vulnerable to threats like repudiation, where users disavow their actions and the log files/records can be required to provide legally recognizable proof of such actions. We must Audit and log the key points of the application activity on most layers, which may help to detect suspicious activities immediately and provide instantaneous notification when a serious attack occurs.

**Audits** are considered to be better implemented if they are generated at the exact moment when the resource is being accessed and explicitly coded within the same code method used to access the resource (AOP can be a very elegant way to implement aspects but might not be well accepted for implementing Audits because when using AOP the how and when the audit is being performed is not clear).

Consider the following general guidelines:

- Design a centralized logging system that collects the most critical business events. Avoid generating a too fine-grained log (this would generate very large operations volumes) but consider the possibility of changing the configuration during runtime and then generating a detailed log, when needed.

- Create log management security policies. Do not save sensitive information about non-authorized accesses within the log files. Consider how to securely access and pass log and audit data between the different layers.

- Consider allowing different types of tracing (*trace listeners*), so that it can be extensible to other file types or logs, even changing it during runtime.

## 4.6.- Instrumentalization

Instrumentalization can be implemented based on performance counters and events, providing information about the application state, performance and condition to the administrators.

www.EBooksWorld.ir

## 4.7.- State Management

State Management (Sessions, etc.) is related to data persistence that represents the state of a component, operation or step in a process. The state data may be persisted in different formats and multiple forms. The design of a state management mechanism may affect application performance and scalability. We must store and persist the minimal amount of required states, and all the state management possibilities should be taken into account. Consider these global guidelines about state management:

- Maintain the state management as "clean" as possible; persist the minimum amount of required data.
- Make sure that your state data is serializable if it needs to be persisted or shared between different processes and network boundaries.
- Choose an appropriate state store. Saving states in the process memory space is the technique that offers the best performance, but only if the state does not need to survive process or server re-starts. States should be persisted to a local disk or to a Database if these states need to be available after a process dies, or even after re-starting the Server(s).
- Regarding the technology to be used when sharing states between the different servers, the most powerful are probably:
    - Using a Distributed Cache/State system that supports Web-Farms and automatic synchronization of the cache data between the different servers.
    - Using a central store based on a Database; although this option lowers the performance by having the data physically persisted.

## 4.8.- Input Data Validation

Designing a system of input data validation is essential for usability and stability of the application. Otherwise, the effect on our application may be data inconsistencies, business rule violations and a poor user experience. Additionally, we may have security holes such as '*Cross-Site Scripting*' attacks, SQL injection attacks, etc.

Unfortunately, there is no standard definition to differentiate the valid input data from the malicious entries. Also, how the application uses the input data will completely affect the risks associated with vulnerability exploitation.

Consider these global guidelines when dealing with the data entry validation design:

www.EBooksWorld.ir

- **"*All the input data is malicious, until proven otherwise*".**

- Validate input data for allowed length, format, data types and permitted ranges.

- **List of Allowed Options vs. Blocked List**: Whenever possible, design the validation system to allow a list that specifically defines what is admitted as data entry, instead of trying to define what is not admitted or may compromise the system. It is much easier to subsequently open the scope range of a permitted value list than to reduce a block list.

- **Validation in Client and Server**: Do not trust only the data entry validations exclusively made on the client side. Instead, use client validations to provide the user with an immediate response in order to improve the user's experience. However, the validation should also be implemented on the server side to check incorrect input data or malicious entries that have "skipped" the validation in the client tier.

- Centralize the validation approach in separate components if logic can be reused, or consider using third party libraries. Thus, validation mechanisms will be consistently and homogeneously applied throughout the application.

- Make sure the user input data is restricted, rejected and/or cleaned.

# 5.- CROSS-CUTTING ASPECTS IMPLEMENTATION USING .NET

## 5.1.- Implementing Claims-based Security in .NET

Implementation in .NET is performed with several new development and infrastructure technologies. The main development foundation (related to .NET Framework) is a new pillar in .NET named **WIF** (*Windows Identity Foundation*). During its beta timeframe it was known as 'GENEVA FRAMEWORK'.

WIF provides the necessary API to work with security tokens for the application and its internal sets of claims. This API framework also provides the capacity to create our own custom STS (*Security Token Service*). However, the latter will usually not be necessary because at the infrastructure level several vendors usually provide ready to use STSs. For instance, Microsoft already provides a ready to use STS for Windows AD authentication (**ADFS 2.0**) or another STS in the cloud (WA AppFabric **Access Control**) which is open to many Internet authentication types (WLID, OpenID, FacebookID, GoogleID, Yahoo, etc.). However, if we want to authenticate against custom credential stores, as mentioned, we can create our own STS using the WIF API.

www.EBooksWorld.ir

## 5.1.1.- STS and ADFS 2.0

As discussed before, we can develop our own STS with WIF in order to support "n" applications. However, it is generally most effective to use a STS that is a finished product 'ready to be used'.

If we have a ***Windows Server 2008 R2 Enterprise Edition*** server, we may use a new Windows Server service that is a STS. This service, called **Active Directory Federation Services (ADFS) 2.0**, provides logic to authenticate the user against Active Directory and each instance of the ADFS may be customized to authenticate against KERBEROS, FORMS-AUTHENTICATION or X.509 CERTIFICATES, with the *Windows Active Directory* (AD) itself as the user's final store.

We can also request the ADFS STS to accept a security token from other "grantor" (STS) belonging to another system or authority (*realm*). This is known as **Identity Federation** and this is how we get single *sign-on* between several independent infrastructures (Several isolated AD Domains/Forests with no trust relationship between them).

The following diagram shows the tasks performed by STS, in this case, Windows Server ADFS 2.0.

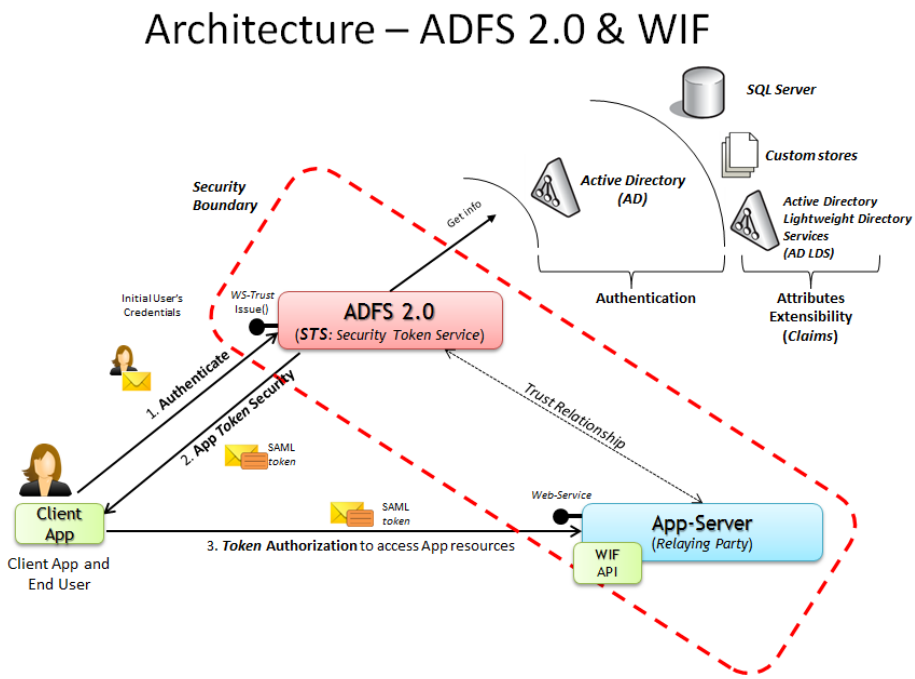



**Figure 4.- ADFS 2.0 process**

www.EBooksWorld.ir

If the authentication type is not based on *Windows Active Directory*, then we would have to use an STS other than ADFS 2.0, either by developing a custom STS using WIF or getting another STS from the market. *Windows Active Directory Lightweight Directory Services* (the evolution of the old ADAM, or *AD Application Mode*) is only supported for attributes extensibility, not for authentication (in the current ADFS 2.0 version).

When using ADFS 2.0, once the user is authenticated against AD, ADFS creates a set of claims about the user (*claims* may be converted from AD attributes or as new extended attributes defined on the SQL Server custom stores). Eventually, the ADFS grants the security token to the user, which will include that set of claims.

ADFS has a Rule engine that simplifies extracting LDAP attributes from AD or AD-LDS. It also allows us to add rules that include SQL sentences so that user data may be extracted from a database in the SQL Server with extended user attributes. Another option is to perform this attribute extensibility using custom stores (suitable when we need to access any other data source). This attribute extensibility in stores external to AD is essential because in most cases the user data is fragmented in many places all over the organization. ADFS can hide this fragmentation. Also, if we need to add attributes/claims in large organizations that have restricted policies regarding AD schema, it is much more feasible to extend these attributes using an external store (such as SQL Server) than requesting extension of the user data scheme in the Active Directory to the organization's IT department.

Also, thanks to this user data composition, if we decide to move the store of certain user's properties, this will be completely transparent for ADFS 2.0 consumers.

As is logical, the security applications based on claims hope to receive claims about the user roles, application permissions or even personal data), but **our application does not care about the origin of such claims and this is one of the advantages of decoupling the authentication from the application, based on a STS (ADFS 2.0 in this case)**.

Architecturally, ADFS 2.0 is fundamentally based on WIF (*Windows Identity Foundation*) and WCF (*Windows Communication Foundation*) framework.

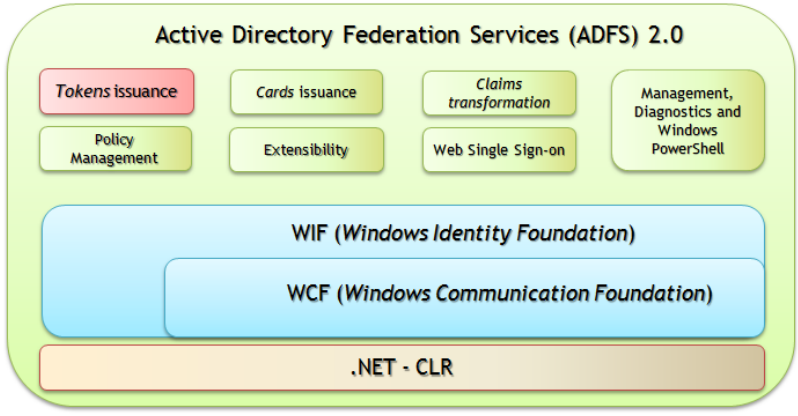



**Figure 5.- ADFS 2.0 Architecture**

www.EBooksWorld.ir

A fundamental part of ADFS 2.0 is the STS (*Security Token Service*) that authenticates against AD (Active Directory) as its identity store based on AD-LDAP and SQL Server, AD-LDS or custom stores as extended stores of the user's properties.

The ADFS 2.0 STS issues security *tokens* based on several protocols and standards, including WS-Trust, WS-Federation and SAML 2.0 (*Security Assertion Markup Language* 2.0). It also supports *tokens* in SAML 1.1 format.

ADFS 2.0 is designed with a clear separation between the communication protocols and internal mechanisms for issuing *tokens*. The different communication protocols are transformed into a standard object model in the access to the system while internally ADFS 2.0 uses the same object model for all protocols. This separation of concerns or decoupling enables ADFS 2.0 to offer a very extensible model, regardless of the peculiarities of each protocol, as shown in the diagram.

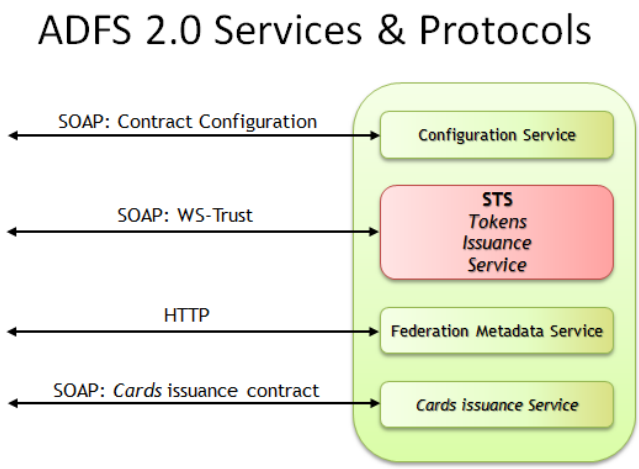



**Figure 6.- ADFS 2.0 Services & Protocols**

## 5.1.2.- Steps to implement Claims-based Security using WIF

There are some steps that we must usually follow in order to implement Claims-based security using WIF:

**Step 1 – Add code supporting WIF and claims to our application**

Our application needs to be able to validate the incoming security tokens as well as to extract *claims* from the security token. Therefore, WIF (*Windows Identity Foundation*) provides us with an API and programming model to work with claims that can be used both from **WCF** services (*Windows Communication Foundation*) and

www.EBooksWorld.ir

**ASP.NET** Web applications. If, for example, we are familiar with the .NET Framework API like **IsInRole()** or properties like **Identity.Name**, this new WIF API is very similar and extends the .NET API. We now have one more property at the Identity level: **Identity.Claims**. This property set provides access to the claims issued by the STS (ADFS 2.0), identifying them and also providing information about who granted them and what they contain.

Of course, there is much more to learn about the WIF programming model, but for the time being keep in mind that we would need to add the WIF assembly (Microsoft.IdentityModel.dll) as a reference to our WCF service or ASP.NET application in order to be able to use the WIF API.

**Step 2 – Buy or Build a token issuer (STS)**

For most scenarios, the safest and quickest option will be using the ADFS 2.0 as **'Token issuer' (STS)** for our organization applications or ***Windows Azure AppFabric Access Control*** as the Internet-Cloud STS.

If ADFS 2.0 and AC do not meet our authentication requirements (for example, it is required to authenticate against a custom store such as databases tables), we may create our own STS using the WIF API. However, building an STS with high Production environment quality is logically far more complicated than simply using WIF in our application; therefore, unless we have some level of experience in advanced security issues, we recommend acquiring an STS on the market.

In any case, it is perfectly feasible to develop a *Custom STS* using WIF. In some cases the effort may be worth considering since an STS is reusable as infrastructure for "n" consumer applications and not only for a single application.

Finally, as mentioned, keep in mind that ADFS 2.0 allows customization through several extensibility points, such as the addition of attributes/claims in external stores, such as SQL Server or custom stores.

**Step 3 – Configuring the Application so that it can trust the "Token issuer" (STS – ADFS 2.0)**

In order for our application to be able to work with the security tokens issued by the ADFS 2.0 STS, we need to establish a trust relationship between both. The application should be able to trust the STS to identify and authenticate users and create the corresponding claims (Roles and/or personal data) as well as creating a security toke service.

There are several items to consider about the **'Tokens issuer' (STS)** when we establish the trust relationship:

- Which claims are going to be offered by the **'Tokens issuer'** (STS)?
- Which cryptographic key will be used by the application in order to validate the tokens signatures?

www.EBooksWorld.ir

- What URL should be accessed by the users to be able to request a token to the STS?

The claims may be any data we imagine about a user, but, as is logical, there are certain typical claims usually provided by an STS. In conclusion, usually common pieces of information are offered, such as **name, last name, email, application roles/groups, etc.**

Each **'Tokens issuer'** (STS) can be configured so that it offers a different number and type of claims. This can be adapted to the application needs and vice versa, adapting the application to the claims already established by the company/organization in their corporate STS.

All the questions above may be answered by "asking" the STS about FEDERATION METADATA, which is, in short, an XML document the STS provides to the application. It includes a serialized copy of the STS certificate that provides the public key to the application so that it can check the incoming tokens signatures. It also includes the list of claims offered by the STS, the URL where the client application will obtain the token and other technical details, such as the token format (generally **SAML**). WIF has a wizard that automatically configures the identity properties of the applications, based on these metadata. We only need to provide the STS URL to the wizard, which will then obtain the metadata and configure our application properly.

**Step 4 – Configuring the 'Tokens issuer' (STS – ADFS 2.0) so that it recognizes our application**

The STS also needs to have certain information about our application before it can issue any token.

- What URI (*Uniform Resource Identifier*) identifies the application?
- Of the claims offered by the STS, which ones are mandatory for the application and which ones are optional?
- Should the STS encrypt the tokens? If so, what key should it use?

Each application is different and not all need the same claims. One application may need to know the application roles while the other may only need the first and last name. So, when a client requests a token, part of such request includes an identifier of the application (***Relaying Party***) that is trying to access. This identifier is the URI. Usually, the easiest thing to do is to use the same URL of the application or web service; for example, *http://www.mycompany.myapp*.

If the application we are building has a reasonable degree of security, it can use SSL (HTTPS) both for the STS and for the application itself. This will protect all the communications information.

If the application has security requirements that are even stronger, they may also request the STS to encrypt the tokens; in this case, the application will have its own X.509 certificate (and private key). The STS will need to have a copy of this certificate

www.EBooksWorld.ir

(without the private key, only with the public key), to be able to encrypt the tokens issued to the application users.

Once again, the "federation metadata" enables this information exchange. **WIF** includes a tool named **FedUtil.exe** that generates a "federation metadata" document for our application, so we do not need to manually configure the STS with all these properties.

### 5.1.3.- Benefits of Claims-Based Security, WIF and ADFS 2.0

The claims decouple the authorization from the authentication so that the application does not need to include specific authentication logic. It also decouples the authorization logic roles and even enables using more fine-grained permissions than those provided by the typical application (roles/groups).

We may grant security access to users that would have been impossible before, because they were in isolated AD Domains/Forests, or used an identity system of other platforms and technologies that were not Microsoft (This case is possible using STSs other than ADFS 2.0).

It improves efficiency of IT tasks by eliminating duplicated user accounts at an application level or Domain level and prevents the critical information store of the users from crossing the security boundaries of the organization (systems controlled by IT).

## 5.2.- Cache implementation in .NET platform

Cache is something that may be implemented at different physical levels (Tiers) and may even be placed in different logical layers. However, implementation at different physical levels may be very different depending on whether cache is implemented in the client side (Rich and RIA applications) or in the application server side.

### 5.2.1.- Server Cache Implementation Using Microsoft AppFabric-Cache

Many scalable applications (Web applications and N-Tier applications) usually have an N-layered architecture like the one discussed in this guide, while most entities and persistent data are stored in data sources that are usually databases. This type of application allows horizontal escalation at a web server level and in a business component server tier (through *web-farms*). We may even extend the same architecture paradigm not only to traditional servers but also to a new paradigm such as 'Cloud-Computing' over Windows Azure, etc.

However, a logical N-Layered architecture with certain physical levels (*N-Tier*) has critical points when dealing with scalability, the most critical of which is usually the

www.EBooksWorld.ir

relational Database Server (SQL Server or any other DBMS). This is because the Database Server usually **escalates vertically** (larger server, increase of processors and memory), but a relational DBMS cannot usually escalate horizontally (scale-out), especially for writing, because they are connection-oriented systems (e.g. in SQL Server, a TCP connection through port 1433).

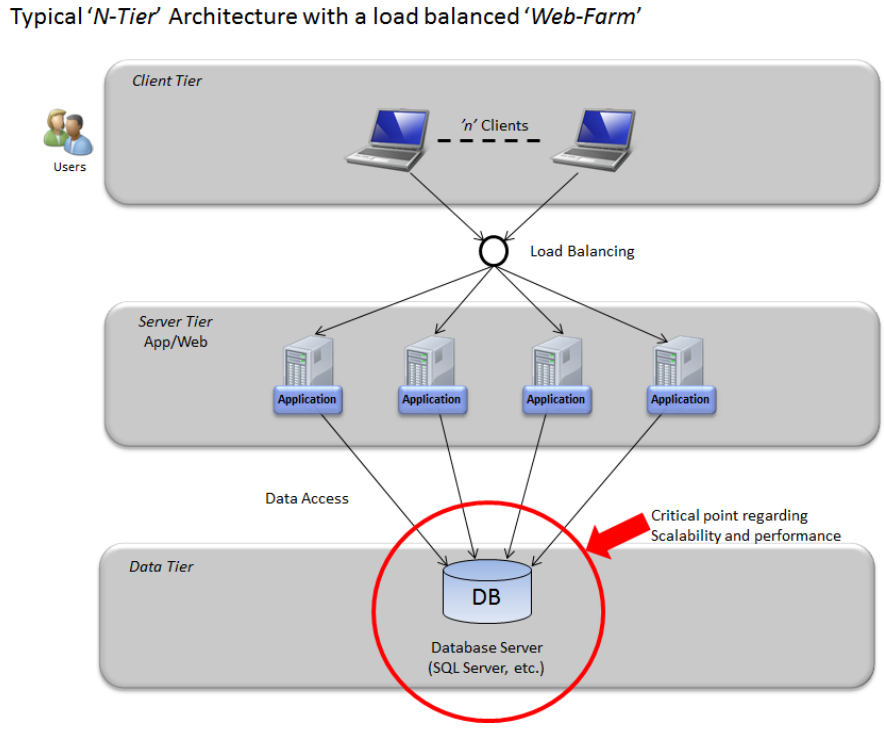



**Figure 7.- Typical 'N-Tier' architecture critical point**

Usually, each time a web page is loaded or data is loaded to a control from a Rich/RIA application, the database should be accessed a number of times. As the application load grows (when increasing the concurrent users number), this frequency against the database may offer serious scalability issues and even performance bottlenecks due to a great increase of resource containment, physical limitations when obtaining persisted data in disk (database) and latency related to remote access to the database server. In conclusion, the weakest point of this architecture when scaling significantly is the database server (we could even have a database hardware cluster but this would not solve our problem in this case since it only offers high availability, not high scalability).

If the application uses traditional cache approaches at the Web level (e.g., ASP.NET sessions) to reduce pressure against the database, then a second challenge appears because 'in-proc' ASP.NET sessions are limited to each IIS server. We would therefore have to use tricks, such as configuring load balancing using affinity in order to link users to the server that they initially accessed with. As a result, we would not have

www.EBooksWorld.ir

optimum load-balancing and potential disproportions in the load balancing may appear. In addition, these 'in-proc' sessions would not have high availability. If one of the servers *goes down* its ASP.NET sessions would be lost.

Finally, we can have a single ASP.NET 'Session Server coordinator', but this means having a single point of failure. Therefore, if these sessions are persisted in the database, we would return to the initial problem of having the bottleneck situated in the database server.

We need to have a memory-based cache, but it should be a distributed cache; in other words, automatically distributed and synchronized between different servers of a cache server tier.

***Microsoft Windows Server AppFabric-Cache*** (its beta name was '*Velocity*') provides an 'in memory' distributed cache, which allows us to create scalable, highly available applications with great performance.

AppFabric-Cache exposes a unified view of the distributed memory to be used by the client applications (in this case, the cache used by the N-layer layers of the ASP.NET web applications or N-Tier applications with WCF services).

Through ***AppFabric-Cache***, the applications can drastically improve performance, since we are placing the data "closer" to the logic that uses it (N-layer application layers) and therefore reducing the pressure on the database server.

The ***AppFabric-Cache*** cluster also offers high availability to avoid loss of data in the applications. The great upside of this type of distributed cache is that it can flexibly increase its scalability by simply adding more cache servers.

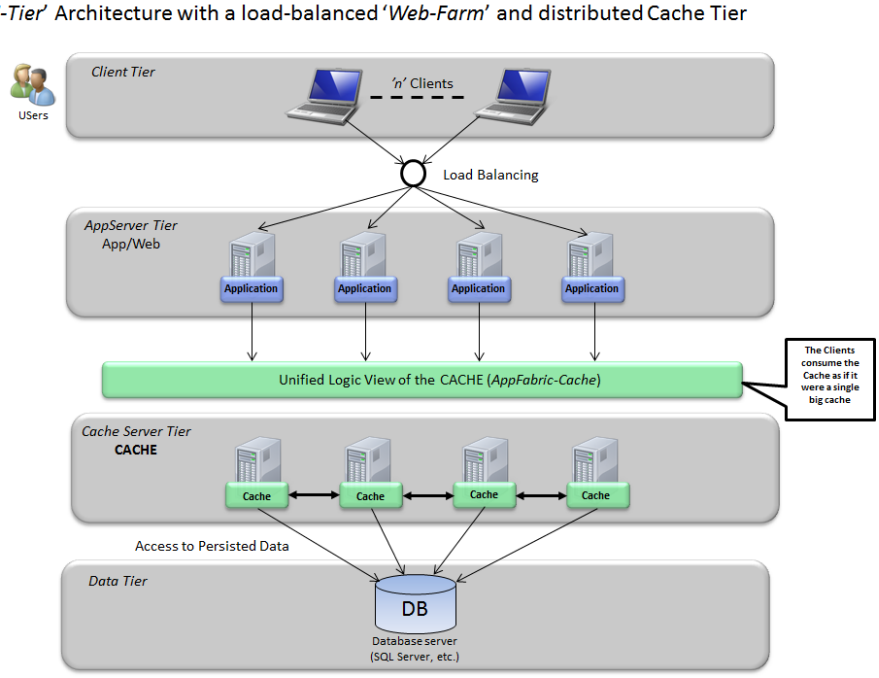



**Figure 8.- 'N-Tier' architecture with a load-balanced 'Web-Form' and distributed Cache**

www.EBooksWorld.ir

The ***AppFabric-Cache*** architecture consists of a cache-server ring where a Cache Windows service is being executed in each server. On the other hand, the client applications of the cache use a .NET client library to communicate with the "Unified Logical View of the Cache."

Therefore, the ***AppFabric-Cache*** allows us to create a new cache tier. This enables **new scalability**, performance and availability levels. Specifically, a high degree of scalability can be obtained precisely by minimizing the access to persisted data in the database. When adding this new level of flexibility to the **scalability**, the physical architecture does not need to perform load balancing with affinity (as when using 'in-proc'ASP.NET sessions). Regarding **performance,** it is also improved because we are 'getting the data' closer to the application logic (Application server), therefore improving the response and latency times (the access to a database server will always be slower than the access to a cache in memory).

We also have **high availability** due to the availability of a cluster with redundancy, mitigating data loss and load peaks that may exist at the data physical level (*Cluster-Hardware* of the database in the event that a node/server of the database goes down).

Of course, if we do not need this much scalability or a very large cache used by several applications, we may also have a more simplified architecture by running the cache Windows service within each 'web-farm' application servers, as shown in the diagram.

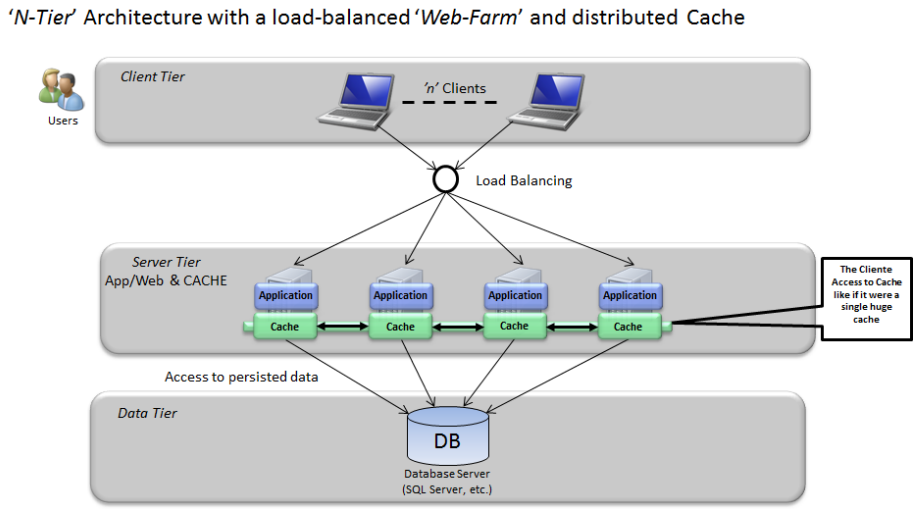



**Figure 9.- 'N-Tier' architecture with a balanced 'Web-Farm' and distributed Cache**

Client applications (our application layers) can access the cache level of ***AppFabric-Cache*** to save or obtain any CLR object that is serializable, through simple operations like 'add/put' and 'get', as shown in the following source code.

www.EBooksWorld.ir

```
C#
//Standard AppFabric-Cache code

//Get default cache
DataCacheFactory _cacheFactory = new DataCacheFactory();
DataCache defaultCache = _cacheFactory.GetDefaultCache();

//Save object/value in cache
defaultCache.Put("Prod101", new Product("Libro N-Layer DDD"));

//Get data/value from cache
Product p = (Product) defaultCache.Get("Prod101");
```

Using a similar API code, we can implement a server cache in an N-Layer DDD application, locating each type of object in the right architecture layers (such as Infrastructure layer for Cache-Factory classes and Application layer for explicit use of cache (Put/Get) API with specific operations related to the domain entities). Later in this chapter we will show a cache implementation approach in the N-Layer DDD architecture.

**Data classification in AppFabric-Cache**

To be able to use AppFabric cache properly, it is important to understand the type of data that is usually cached. This type of data can be classified as **reference data, activity data and resources data**.

The **Reference data** are data mostly used simply in read-only mode, such as user profile data, or product catalogue data. This type of data is not frequently updated: e.g., only once a day or once a week. However, scalability requirements of these reference data usually require a great amount of reading requests against these small data pieces. If this were always done against the database directly, scalability would be limited.

In e-commerce, for example, as the number of visitors increase, the number of queries to the product catalogue may increase drastically. As product data usually do not change very much (the price may change, but not too often) this type of data (product catalogue) is a very suitable candidate to be included in the reference data cache. Therefore, it will greatly alleviate the load against the database server as compared to accessing the database for every catalogue query.

**"Activity" type data** are data involved in business activities and are therefore usually transactional data. A good example in e-commerce would be a "shopping cart". Since they represent activity data, these data is usually for a lot of reading and some writing (From and to the cache). After the lifetime of an activity (in this case, when the purchase is paid), the activity data is removed from the cache and persisted in the persistent data source (database). In this example, the data of the shopping cart would become an Order already persisted in the database. In an older scenario (with no AppFabric Cache), if ASP.NET sessions had been used for a "shopping cart", the e-commerce would have required load balancing with affinity, partially damaging scalability and performance because we would have stored session-data into SQL Server (central point), probably. Now, with AppFabric-Cache, you can store the

www.EBooksWorld.ir

shopping cart in the distributed cache, and the load balancing can be pure, maximizing scalability of the available servers.

**Resource type data** is data constantly read and written, such as product stock or a bank account balance. During an order process, the levels of the stock may require monitoring to ensure stock levels. However, as the orders are processed, these data need to be concurrently updated to reflect changes in the stock. Sometimes, the coherence levels of these data are relaxed to improve performance and scalability,. For example, the order process may oversell items while purchases or manufacture of new items can be generated in separate processes in order to maintain stock levels. However, these processes imply further risks.

### Logical Hierarchy of the AppFabric-Cache Architecture

The logical hierarchy of the **AppFabric-Cache** consists of **machines**, **hosts**, **named caches**, **regions and cache elements**. Machines may execute multiple services of **AppFabric-Cache**, and each service is considered as a cache host. Each cache may contain multiple 'named caches' and these named caches will be defined in a configuration throughout the different machines.

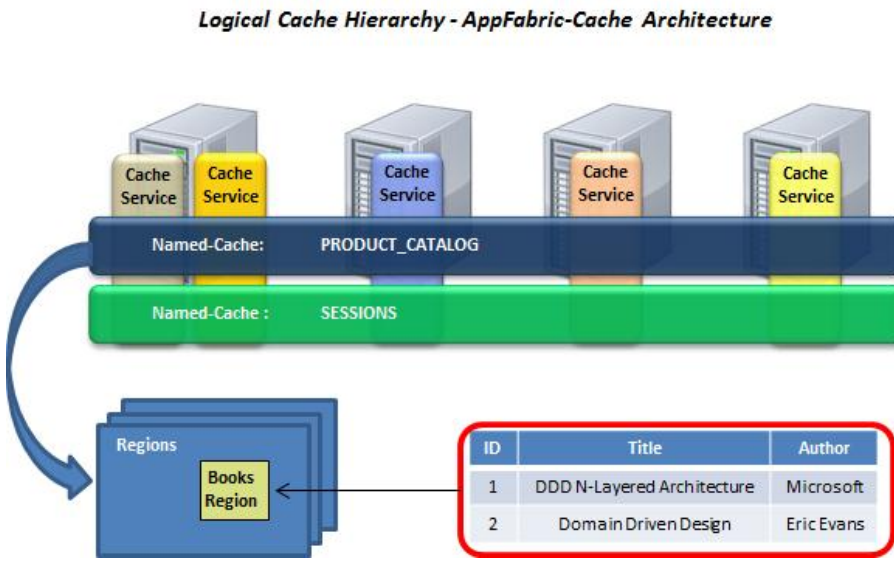



**Figure 10.- AppFabric-Cache Logical Hierarchy**

Each named-cache keeps a logical data group, such as 'User Sessions' or 'Product Catalog'. Named-caches will also set up policies on data expiration, availability, etc.

The explicitly created regions (physically as data containers) may also exist within each named cache. As data is queried, regions perform well if the application needs to get many elements from a single region. However, the explicit creation of these regions

www.EBooksWorld.ir

is optional. AppFabric-Cache will implicitly create regions by default if no region is explicitly specified.

Finally, the lower level cache elements are located within the regions (whether explicit or implicit), and they are responsible for storing keys, serialize objects, *tags*, *timestamps*, versions and expiration data.

**Other Server Cache Implementations**

Finally, it is important to point out that there are other possible cache implementations (other than AppFabric Cache), such as the following:

- *Danga Interactive Memcached*
- *Microsoft P&P Enterprise library* Caching block

## 5.2.2.- AppFabric-Cache Implementation in DDD NLayerApp Sample Application

There is not only one way of implementing cache in an N-layered application. The main options are probably as follows:

- 1. **Explicit use of cache in the Application Layer**: This option clearly distinguishes the application services and related domain entities that use cache from the ones that don't. It is probably the most surgical and clearest option because it uses the cache API very explicitly in the Application Layer.

- 2. **Use of cache Repositories that replace normal repositories**: these cache repositories, using IoC (Unity, etc.), can transparently replace the standard repositories that directly access the database (e.g., with EF). It is important that they implement and meet the same Repository Interface. The advantage is that the use of cache can be changed for certain entities in a transparent manner and even during runtime (using the Unity configuration XML) as desired. The disadvantages are two: it should be used for all the Repository operations (and related aggregate); it is not very explicit for a code reviewer if cache is being used or not, since the only difference is in the repository class being registered in Unity and mapped from the repository interface. This last point (transparency) can be seen as a disadvantage or as an advantage, depending on the context.

- 3. **Use of AppFabric-Cache as Entity Framework second-level cache**: this option is the most transparent because it can make cache of all the entities used with EF without 'doing anything' in the own code of our application. It is actually an intermediate and transparent layer (infrastructure) between our application and EF. It requires developing a library /framework that is reusable, probably external to the code of our application. The first thing the

www.EBooksWorld.ir

ORM does when an entity is being loaded from the database is to transparently check if it exists in the second level cache; if so, the data are returned from the cache without the need to query the database. Both NHibernate and Entity Framework can use AppFabric-Cache as a second level cache. For NHibernate there is a project called nhibernate.caches.velocity (*sourceforge.net/projects/nhcontrib/files/NHibernate.Caches/*) and for Entity Framework EFCachingProvider by Jaroslaw Kowalski. (*code.msdn.microsoft.com/EFProviderWrappers*). In the future, the ORMs themselves may implement this functionality.

- 4. **Use of AppFabric-Cache as *provider* for ASP.NET sessions:** this use is complementary and can be used as long as ASP.NET sessions are used. This new possibility eliminates the 'in-proc' ASP.NET sessions issue when load-balancing Web-Farms with no affinity.

In our AppFabric-Cache implementation of the NLayerApp sample application, we have chosen the first option, as it is the most flexible, explicit and therefore the most educational. Also, we think it is important to be able to cache some aspects/actions of an entity but maybe not any other. For example, we may want to cache queries/lists of an entity and not to cache the specific query of a specific instance of an entity. In conclusion, it is probably the most flexible, surgical and explicit way of using the *cache*.

In addition, and only for the ASP.NET client, we can use the AppFabric-Cache to maintain the sessions in their distributed cache.

### Implementing a Cache Manager class in the infrastructure layer

Following the design lines of our architecture, it is important for the technical elements (in this case, cache) to be located in the infrastructure layer and always used through declared abstractions (contract/interface). Thus, and through the use of IoC and Dependency Injection, we can replace the implemented cache infrastructure transparently with almost no impact at all (replacement of implementation with AppFabric-Cache by other future technology). In addition, this decoupling also enables running unit tests against mocks/stubs that actually are not using the ***cache***.

We show the simplified code of our **CacheManager** class below.

```
C# - Infrastructure cross-cutting elements layer
```

Interface for abstraction and instantiation through IoC container (Unity)

```
public sealed class CacheManager:ICacheManager,IDisposable
{
DataCacheFactory _cacheFactory;

   /// Is recomended using "singleton" life time in the selected IoC
   public CacheManager()
   {
   //configuration for this cache factory is delegated in application
configuration file
```

www.EBooksWorld.ir

```
    _cacheFactory = new DataCacheFactory();
    }

    public bool TryGet<TResult>(CacheItemConfig cacheItemConfig, out
TResult result)
    {
    if (cacheItemConfig != null)
        {
    //get default cache
    DataCache defaultCache = _cacheFactory.GetDefaultCache();

    string cacheKey = cacheItemConfig.CacheKey.GetCacheKey();
    //get object from cache and check if exists
    object cachedItem = defaultCache.Get(cacheKey);

    if (cachedItem != null)
    {
             result = (TResult)cachedItem;
    return true;
    }
                    else
                    {
                        result = default(TResult);

                        return false;
                    }
                }
                else
                    throw new ArgumentNullException("cacheItem");
            }

            public void Add(CacheItemConfig cacheItemConfig, object
value)
            {
                if (value != null
    &&
                    cacheItemConfig != null)
                {
                    //get default cache
                    DataCache defaultCache =
_cacheFactory.GetDefaultCache();

                    string cachekey =
cacheItemConfig.CacheKey.GetCacheKey();
                    TimeSpan expirationTime =
cacheItemConfig.ExpirationTime;

                    defaultCache.Put(cachekey, value,expirationTime);
                }
            }

            public void Dispose()
            {
                if ( cacheFactory != null)
                    _cacheFactory.Dispose();
    }

        }
```

We obtain named-cache by default from AppFabric-Cache

We try to obtain the value requested from the cache

If it exists in the cache, it is returned. Otherwise we return a false.

www.EBooksWorld.ir

## Using the Cache manager class from the Application layer

Below we show an example of an Application SERVICE class implementation that controls everything related to the Customer entity. We use AppFabric cache in several specific methods:

```
C#

public class CustomerManagementService : ICustomerManagementService
{
ICustomerRepository _customerRepository;
ICountryRepository _countryRepository;
ICacheManager _cacheManager;

                                    Constructor with required dependencies

public CustomerManagementService(ICustomerRepository customerRepository,
ICountryRepository countryRepository,ICacheManager cacheManager)
        {
             _customerRepository = customerRepository;
            _countryRepository = countryRepository;
  _cacheManager = cacheManager;
        }
                                    Application logic with cached data for the 'Customer' entity.


public List<Customer> FindPagedCustomers(int pageIndex, int pageCount)
{
            //Implementing cache-aside pattern

            List<Customer> customerResults = null;
            CacheKey key = new CacheKey("FindPagedCustomers", new
{PageIndex=pageIndex,PageCount = pageCount });
            CacheItemConfig cacheItemConfig = new CacheItemConfig(key,
new TimeSpan(0, 0, 30));
                                    We try to obtain the data from the cache.


            if (_cacheManager.TryGet<List<Customer>>(cacheItemConfig,
out customerResults))
                return customerResults;
            else
            {
bool enabled = true;
                Specification<Customer> onlyEnabledSpec = new
DirectSpecification<Customer>(c => c.IsEnabled == enabled);
                                    If it does not exist in the cache, we obtain it from the DB
                                    and save it in the cache for future queries
                customerResults =
_customerRepository.GetPagedElements(pageIndex, pageCount, c =>
c.CustomerCode, onlyEnabledSpec, true)
                        .ToList();
_cacheManager.Add(cacheItemConfig, customerResults);
                return customerResults;
            }
}
}
```

www.EBooksWorld.ir

### Using AppFabric Cache as an ASP.NET Session provider

It can be used as long as ASP.NET is used, logically. This new possibility of using AppFabric Cache with ASP.NET sessions eliminates the problem of 'in-proc' ASP.NET sessions in load-balanced Web-Farms.

This problem starts when a balanced http request is initially accepted. States (values in sessions) will usually be created at this time within the memory of one of the web-farm servers. However, if the next http request is redirected to another server within the cluster (because the load-balancing is not using 'affinity') the values obtained from the ASP.NET sessions will be wrong.

Up to now, the ASP.NET session provider model offered three kinds. Now we add a fourth one:

1. ***InProc Provider***: in the memory of the web server, related to the problem previously described.

2. ***StateServer Provider***: a single server with all the sessions in its memory. This provider is problematic because, to begin with, it is not scalable. It may only be a single server offering the service of the session values. Also, it is a single point of failure. If the service of this session server "is down", all the sessions of our ASP.NET web-farm will stop working.

3. ***SQLServer Provider:*** this solves the problem of multiple sessions in pure web-farm balancing and also solves the problem of having a single failure point (as long as we have a SQL server cluster). However, this provider has the disadvantage of decreasing performance and even scalability of our application, since we will be adding pressure to the SQL Server tier.

4. ***AppFabric-Cache Provider:*** This new ASP.NET session provider uses the AppFabric distributed cache and its high availability as an ASP.NET session provider. This system is implemented in a transparent manner without requiring changes to our ASP.NET application code. The only thing to be changed is the XML definition of the session provider in web.config.

The ASP.NET session provider based on the AppFabric cache enables sessions to keep going even when one or more of the ASP.NET front servers is down or off, because these sessions are saved 'out-of-process' in the AppFabric Cache Web-Farm.

Once the AppFabric cache is installed and configured, you should create a "named-cache" to store the ASP.NET sessions and subsequently enable the ***DataCacheSessionStoreProvider***, modifying the Web.config, as shown below.

www.EBooksWorld.ir

**Web.config**

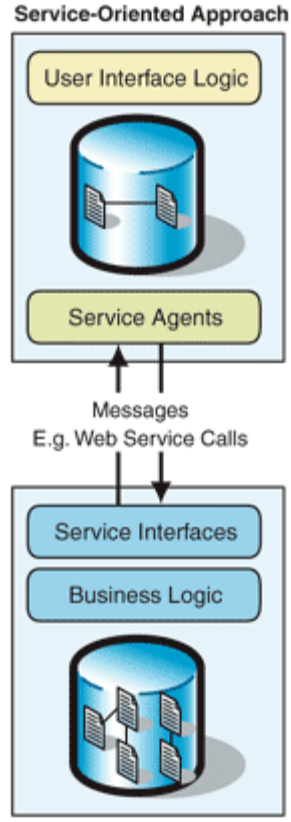
```
<?xml version="1.0"?>
<configuration>
<configSections>
<section name="dataCacheClient"
      type="Microsoft.Data.Caching.DataCacheClientSection,
CacheBaseLibrary"
      allowLocation="true" allowDefinition="Everywhere"/>
<section name="fabric"
      type="System.Data.Fabric.Common.ConfigFile, FabricCommon"
      allowLocation="true" allowDefinition="Everywhere"/>
<!-- AppFabric Cache -->
</configSections>
<dataCacheClient deployment="routing">
<localCache isEnabled="false"/>
<hosts>
<!--List of services -->
<host name="localhost" cachePort="22233"
        cacheHostName="DistributedCacheService"/>
</hosts>
</dataCacheClient>
<fabric>
<section name="logging" path="">
<collection name="sinks" collectionType="list">
<!--LOG SINK CONFIGURATION-->
<!--defaultLevel values: -1=no tracing;
            0=Errors only;
            1=Warnings and Errors only;
            2=Information, Warnings and Errors;
            3=Verbose (all event information)-->
<customType
          className="System.Data.Fabric.Common.EventLogger,FabricCommon"
          sinkName="System.Data.Fabric.Common.ConsoleSink,FabricCommon"
          sinkParam="" defaultLevel="-1"/>
<customType
          className="System.Data.Fabric.Common.EventLogger,FabricCommon"

sinkName="System.Data.Fabric.Common.FileEventSink,FabricCommon"
          sinkParam="DcacheLog/dd-hh-mm" defaultLevel="-1"/>
<customType
          className="System.Data.Fabric.Common.EventLogger,FabricCommon"
          sinkName="Microsoft.Data.Caching.ETWSink, CacheBaseLibrary"
          sinkParam="" defaultLevel="-1"/>
</collection>
</section>
</fabric>
<appSettings/>
<connectionStrings/>
<system.web>
<sessionState mode="Custom" customProvider="
AppFabricCacheSessionProvider">
<providers>
<add name="AppFabricCacheSessionProvider"
        type="Microsoft.Data.Caching.DataCacheSessionStoreProvider,
ClientLibrary"
        cacheName="session"/>
</providers>
</sessionState>
```

www.EBooksWorld.ir

## 5.2.3.- Implementing Client Tier Cache in N-Tier applications (Rich-Client and RIA)

'*Rich-Client*' applications (also known as '*Smart-Client*' using technologies such as *WPF, VSTO or WinForms*) and RIA applications (implemented with *Silverlight*) are loosely coupled with the application server tier, thanks to the "consumption" of Web services, which are "loosely coupled", by design.

Precisely, one of the main advantages of '*Smart-Client*' applications is to simultaneously be able to work in an '*off-line/on-line*' mode. This offline mode (working disconnected in the client PC) promotes the power to perform ***cache*** in the '*Smart-Client*' presentation layer.




The cache in '*N-Tier*' distributed applications that access a Web-Service and WCF-Service is extremely important because this ***cache*** can drastically reduce the load and number of requests to the Web-Services and greatly increase the global performance of the application as well as the response times offered to the client.

The '*candidate*' data to be cached in the client layer of a '*Rich-Client*' and *RIA* application are those data that do not change very often but yet, have a lot to do with the interrelated operation of the forms. For example, master entities like "Countries" and "Provinces", etc. should always be cached in the global memory of the client process, being loaded on the start-up of the application or any similar way.

If we want to cache data that changes more often, it is important to use a system that is capable of detecting changes and/or refreshing the cache more often (short *time-outs*).

In this case (Rich or RIA application), the cache should be located in some global point of the client application (.exe) in the main process memory space.

Locating business entities cache management in the ***Service Agent*** classes allows us to create a "smart" layer that in some cases accesses online to the data (online access through Web-Services/SOA) and in other cases may obtain them from the local client cache.

www.EBooksWorld.ir

## 5.3.- Logging Implementation

There are public libraries, such as *Microsoft Pattern & Practices Enterprise Library*, that are very useful to implement complex logging, with different possibilities.

Some interesting implementations are:

- *Microsoft Enterprise Library Logging Building Block*
- *NLog*
- *Log4net*

## 5.4.- Validation implementation

There are general libraries, such as Microsoft Pattern & Practices Enterprise library, that are very useful to implement a reusable system of data entry VALIDATION (*Microsoft Patterns & Practices Enterprise Library Validation Block*)

www.EBooksWorld.ir

CHAPTER 10

# Architecture and Patterns for PaaS Cloud-Computing and Windows Azure

Currently, there is a lot of literature dealing with 'Cloud Computing' and the different types of *Cloud*, either based on a Service model (*IaaS, PaaS, SaaS*) or a deployment model (Public clouds, private clouds, hybrid clouds, etc.). We also have a great amount of information about each platform and the technology offered by the different cloud providers/manufacturers.

In our case (Application Architecture Guide), we are especially interested in PaaS and IaaS clouds, since this deal with defining architecture, design and implementation of the applications. We will not explain finished products/services (SaaS) in this book. Given these premises, we would like to emphasize that the objectives of this chapter are not focused on defining what 'Cloud Computing' is, or listing and explaining the elements of Microsoft *Cloud-PaaS* technology (*Windows Azure* Service Platform). This would not provide any additional value, since that type of content is already available in many other sources.

However, we think it is convenient to identify the elements and **patterns of an application architecture** that can, and even must be different in an **application that is to be deployed in "the cloud".** We will therefore focus exclusively on identifying Architecture and Design patterns that fit well with the Cloud Computing philosophy and subsequently analyze the possibility of their implementation with Microsoft technology (.NET and Windows Azure service platform).



www.EBooksWorld.ir

# 1.- APPLICATION ARCHITECTURE IN THE CLOUD

As *Cloud-Computing* is a consolidated reality and several manufacturers (like *Microsoft*, *Amazon*, *Google*, *IBM*, *Salesforce.com* and many more) seriously rely on it, organizations are currently wondering and trying to evaluate if *cloud computing* services and capabilities can be really helpful for them.

To that end, companies and all types of organizations obviously want to analyze the current state of their infrastructure technologies, especially their applications, and to identify in which cases *cloud-computing* would be useful or not (evaluating public clouds and private cloud options separately).

In sum, we are dealing with identifying the applications that can substantially benefit from cloud characteristics, such as the following aspects:

- High scalability on demand
- Elasticity in this scalability (both increasing and decreasing), ideal for applications with concurrent user peaks, with any frequency (seasonal, monthly, weekly, and even daily, at certain hours).
- Quick and simple deployments.

Once these application cases are identified (new applications or current application migrations), and after the *cloud-computing* technology to be used is selected (IaaS or PaaS), (such as the Windows Azure platform), organizations will obviously want to adopt it. To that end, the following should be done:

- Application Migration analysis
- Defining and planning the migration strategy

The most important point to consider regarding Cloud-Computing is that change in favor of it does not usually occur due to technical or architectural aspects. The cause normally comes from the "business side". If the *cloud-computing* capabilities (e.g., less TCO in peak-type applications) are aligned with business needs (cost reduction, short time-to-market and quick deployments), then a change to the cloud will probably happen.

Once we arrive at this point (specific applications identified as part of the Cloud strategy of an organization), it becomes important to think about it in from the Architectural point of view, since applications that fit into certain types of architecture styles will be the ones to fit in the desired migration patterns.

Therefore, identifying applications that fit well into *cloud-computing* is essential when we want our migrations to be successful.

www.EBooksWorld.ir

From the Architecture point of view, there are many professionals who currently have several interesting questions. Especially the following:

***'Are the Application Logical Architectures different in the cloud?'***

***'Should I use the same logical architecture in the cloud (e.g. my N–Layered architecture, DDD, etc.) that I use in my current applications? '***

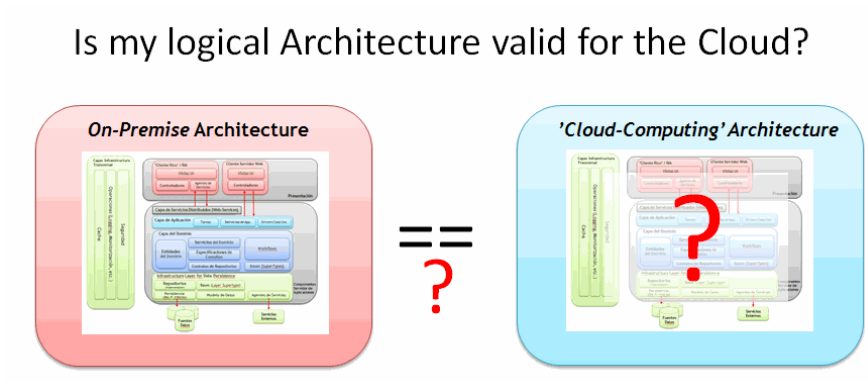



**Figure 1.- Current Logical Architecture Validity for the Cloud**

Actually, the answer is similar to what happens most of the time in our sector… our answer is: "It depends".

Certainly, we can migrate an '*On-Premise*' application (in our own servers) to the cloud and maintain 100% of the same architecture (for instance, our DDD N-Layer Architecture) and we can even virtually maintain the entire implementation we had in .NET when migrating it to Windows Azure and SQL Azure. However..., **should we migrate directly or should we have different logical architectures and patterns and even different implementation and data persistence technologies?**

Well, the answer is still "it depends". '*Cloud-Computing*' is not another dimension or galaxy when designing an application; in the end, the architecture to be designed also depends on the objectives and requirements of our application.

We also should think about why we want to deploy our application in the cloud. "The cloud" (in this case, PaaS) is nothing more than a new deployment environment that offers on demand elasticity for our application scalability, while greatly simplifying tasks and, therefore, deployment costs. However, due to the essence of '*Cloud-Computing*', in many cases we will want to "go to the cloud" because we have nearly unlimited new and ambitious scalability requirements. This is a different matter.

We can summarize the answer to the questions above in the following table:

www.EBooksWorld.ir

**Table 1.- Difference in Architecture and Patterns in Cloud vs. On-Premise**

| Premises | Architecture and Patterns |
|---|---|
| The application requirements are similar to our application deployed *On-Premise*.<br>It does not require high scalability<br>Reasons for deploying in the cloud are similar to the following:<br>• Quick deployments<br>• PaaS Services<br>• Pay per use (OPEX vs. CAPEX) | The logical architecture, and therefore most of the implementation (technologies) may be very similar to what we would do with a standard application in our servers. Minor incompatibilities may always appear at the level of technologies that should be changed, but the architecture and most of the technology (.NET) will be very similar to building an On-Premise application. |
| If high scalability is required | Logical architecture should be different to achieve a much higher scalability dimension. To a considerable degree, implementation (required technologies) will probably be different and we should use certain native cloud technologies that, in many cases, are not available in our current servers. |

It is important to make no mistake about this. The physical architecture will be necessarily different in the PaaS cloud than in our servers, since there will be different physical elements and even the application deployment will be different. However, **logical architecture should not be affected by the deployment environment** (Cloud vs. On-Premise). If we decide we need certain architecture and design patterns to be able to deal with high scalability objectives, that should be so whether the deployment is *on-premise* (traditional servers) or in the cloud. The technologies required to accomplish this do not need to be the same, but the architecture patterns will be.

So, in most cases when we want to change our logical architecture and design patterns for an application "in the cloud", these architecture changes are not actually caused by "the cloud" itself, but by new scalability requirements. The key point is that these high scalability requirements coincide with the fact that "the cloud" (PaaS) simplifies implementation and deployment for high scalable applications.

www.EBooksWorld.ir

## 2.- ARCHITECTURE SCENARIOS IN THE CLOUD

The scenarios above are presented from the two points of view commonly used throughout this guide; first at an architecture and pattern level and then at the implementation and specific technologies level, showing their mapping/relationship.

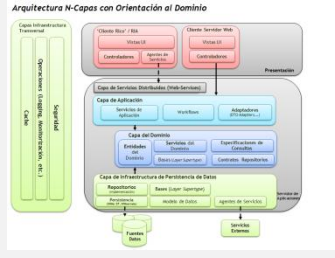
## 3.- BASIC SCENARIO: DIRECT MIGRATION FROM ON-PREMISE APPLICATION TO THE CLOUD

This scenario allows us to have very high compatibility of our application between an '*on-premise*' deployment environment (standard servers in *datacenters*) and '*cloud-computing*' platforms. However, this scenario may not be valid for certain levels of high scalability (very high number of concurrent users).

### 3.1.- Logical Architecture (Basic Scenario)

When the reason for using the cloud is simply for ease of deployment, pay per use (OPEX vs. CAPEX) or other reasons not related to scalability, the logical architecture of our application can be virtually the same when deploying our application in the cloud. Moreover, the implementation (technologies, .NET and data persistence) will probably be almost the same. In our case (complex applications with a large volume of business logic), the logical architecture would be the same as the one analyzed in this guide, that is, our DDD N-layer architecture.

Since this is the same architecture as the one used in '*On-Premise*', we will not analyze it. In this case (Basic Scenario), this would be our same DDD N-layer architecture.




www.EBooksWorld.ir

## 3.2.- Reasons for using Windows Azure

Before analyzing the technology, we would like to explain some business reasons for using Windows Azure. To align business needs with cloud-computing capabilities, organizations must research and assess the Windows Azure cloud characteristics, but from the business objectives point of view:

**Table2.- Business reasons for using Windows Azure**

| Business reasons for using Windows Azure |
|---|
| • **Cost reduction**: organizations should be able to migrate applications and database assets to Windows Azure and while lowering TCO (Total Cost of Ownership) to be paid only for the necessary resources and the ones effectively used (avoiding paying high investments up-front for something that may be or may be not needed in the future).<br><br>• **Lower *'Time to Market'***: organizations can use the quick deployment capabilities offered by Windows Azure (not only because of the simplified deployment but especially because of the immediate availability of the Windows Azure infrastructure on the Internet. We do not have to buy, install and configure servers). Windows Azure gains time, has extraordinary better time to market and, finally, much shorter response times to business requirements.<br><br>• **To be efficient** and mitigate the consequences of maintenance of assets that are not used or underused, organizations can use Windows Azure to increase or reduce used resources (depending on needs) and thus improve efficiency.<br><br>• **To correctly manage future high use demands** that are unpredictable now and continue to provide users with the right experience (system scalability), organizations can use Windows Azure's elasticity to dynamically make use of system scalability as the demand grows.<br><br>• **To move "clandestine" applications to the cloud**: many organizations have "hidden" applications, that is, applications created without supervision of the IT department, in "clandestine" servers. These, therefore, have no service level assurance (This is because a business department may have contracted it without the IT department supervision or because these are applications without enough entity to be under the IT SLA).<br><br>• **Moving these applications to Windows Azure** locates them in a consistent and secure Datacenter environment that provides a minimum and guaranteed SLA (starting at 99.5+). This also makes it possible to scale these applications in the future, decrease the resources on demand if they are under- used or even simply remove them, all of which involve virtually no operation costs. |

www.EBooksWorld.ir

## 3.3.- Brief Introduction to the Windows Azure Platform

We will provide a brief introduction to *Windows Azure* here in an attempt to prevent anyone who does not know *Windows Azure* from having to search for information from other sources in order to understand the following pages.

The *Microsoft® Windows® Azure™* platform provides processing on demand "in the cloud". This cloud is a set of interconnected resources/servers in one or more *datacenters*. *Windows Azure* is therefore a public and hybrid cloud, as well as a **PaaS** (Platform as a Service) and **IaaS** (Infrastructure as a Service).

Currently, the *Windows Azure* platform is available in Microsoft *datacenters* in the United States, Europe and Asia. Developers and IT personnel can use the cloud (in a very simplified way) to deploy and execute their applications and even store data. *On-premise* applications (in *datacenters* of the organization itself) can also use remote resources located in the *Windows Azure* cloud, because any application can consume remote Windows Azure web services or even just data resources (SQL Azure or Azure-Storage).

*Windows Azure* platform distances us greatly from the hardware resources through virtualization. Each application (Web/Web Services) deployed to *Windows Azure* is executed in one or more virtual machines (VMs). These deployed applications behave as if they were in a dedicated machine. Nonetheless, they can share physical resources such as disk space, network bandwidth, CPU cores, etc. with other VMs within the same physical server/host (as commonly happens with virtualization).

A key benefit of the abstraction layer on the hardware is portability and scalability. The virtualization of a service allows it to be moved/ cloned to any number of physical servers of the *datacenter*. By combining virtualization technologies with increasing hardware, multi-organization and aggregation on demand, Microsoft can handle volumes of "scale economy". This generates a more efficient use of the *datacenters*.

Virtualization in *Windows Azure* also provides us with vertical and horizontal scalability.

**Vertical scalability** means that, as the user demand for our application is increased, we can increase the number of virtual machines processing resources, such as the number of CPU cores or memory assigned to each VM.

**Horizontal scalability** means that we can increase/decrease the number of VM instances (these will be copies/clones of the services of our application). All the instances are balanced by ***Windows Azure*** (load balancing) at the network level, so the incoming requests are homogeneously distributed among the number of instances.

Currently, the main components of the ***Windows Azure platform*** are:

- *Windows Azure*
- *SQL Azure.*
- *Windows Azure platform AppFabric*

www.EBooksWorld.ir


**Figure 2.- Windows Azure technology pillars**

- ***Windows Azure*** provides the following capabilities:
    - A virtualized application execution environment based on Windows Server (as "guest" operative system).
    - Persistent storage both for structured data (Azure-Storage) and for non-structured data (Blobs) and asynchronous messaging for scalable architectures.
    - A management portal in the Internet.
- ***SQL Azure*** is basically an SQL Server provided as a server in the cloud, although there is an important value added such as high availability 'out of the box'. In addition it provides:
    - ***Reporting***: Microsoft Reporting Server in the cloud.
    - ***SQL Data Sync***: This is a cloud-based data synchronization service built on Microsoft Sync Framework technologies. It provides bi-directional data synchronization and data management capabilities allowing data to be easily shared across SQL Azure databases within multiple data centers.
- ***Windows Azure platform AppFabric*** provides:
    - ***Service Bus*** on the Internet. This helps to connect applications that are being executed in our datacenters, in the Windows Azure cloud or in other clouds, regardless of the network topology (regardless of the existence of firewalls).
    - ***Access Control Service***: This handles authorization and authentication aspects for Web Services with security tokens. It follows the trend of Claim orientation.
    - ***Caching***: This is a distributed cache, in virtual servers memory, implemented as a service in the cloud.
    - **Composing App & Integration**: These are next assets to be released, but at the moment of this writing they are still in an early CTP state.

www.EBooksWorld.ir

The platform also includes several management services that allow control of the resources above through a web portal or through a program using the API.

Finally, a ***SDK*** and tools integrated in Visual Studio are available, so we can develop, test and deploy everything locally in development PCs (Windows Azure simulated environment is called '*Local Azure Compute Emulator*' and 'Storage Emulator') independently and isolated from the cloud. The deployment to the cloud would be the final step, but not required most of the time at the development phase.

Windows Azure is designed to abstract most of the infrastructure that is normally necessary in traditional servers (servers, operating systems, web services, load balancers, etc.) so that the project teams can focus solely on building the application. A simplified vision of Windows Azure (taking into account the administration and not only the application to be deployed) may be similar to the following:

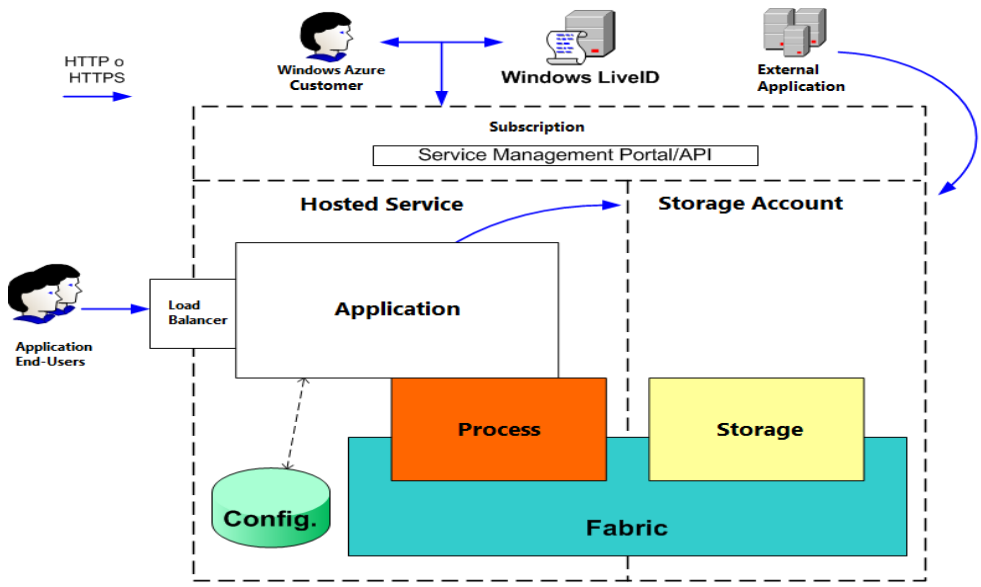



**Figure 3.- Simplified Windows Azure vision**

The Windows Azure customers (Administrators) manage their applications and data storages through a subscription associated to a Windows Live ID account. This account is exclusively for Windows Azure management. On the other hand, our application can use any other authentication system, such as 'custom', Windows Live ID, ASP.NET Membership, OpenID, FacebookID, GoogleID, YahooID, Windows Active Directory, etc. (all authentication types based on claims based security provided by Windows Azure *AppFabric Access Control* and ADFS 2.0).

www.EBooksWorld.ir

## 3.3.1.- Compute Environments in Windows Azure

An application executed in Windows Azure is known as a "Hosted Service". Normally, a Hosted Service has several resources that process information and interact with the outside world (clients or other applications). The Host Services are comprised by roles. We currently have two PaaS-Role types: ***Worker Role*** and ***Web-Role***.

**Note about Roles and other possibilities:**
We also have another role type, which is more IaaS focused (*Infrastructure as a Service*) known as ***VM-Role***. This is basically a virtual machine template we should install from scratch, including the operating system (Windows Server 2008 R2), where we can install all the base software we need/want. However, we have to take into account that, in this case, we are responsible for the entire maintenance, patches updates, new versions of the Operating System or Service Packs, etc. In contrast, when using the Web-Role and Worker Role, all those tasks are automatically performed by Microsoft datacenter operations.
Finally, for traditional applications deployment, there will be an additional possibility called ***'Server App-V' (Microsoft Server Application Virtualization)*** for Windows Azure (deployed over Worker Roles). It is currently (March 2011) in CTP state, but when available it will be a very powerful option when we cannot build our app as a regular Web/Worker role.

However, in order to achieve a PaaS productivity level, we need to use the high level roles (Web role and Worker role), where we do not have to handle base software administration and installation tasks.

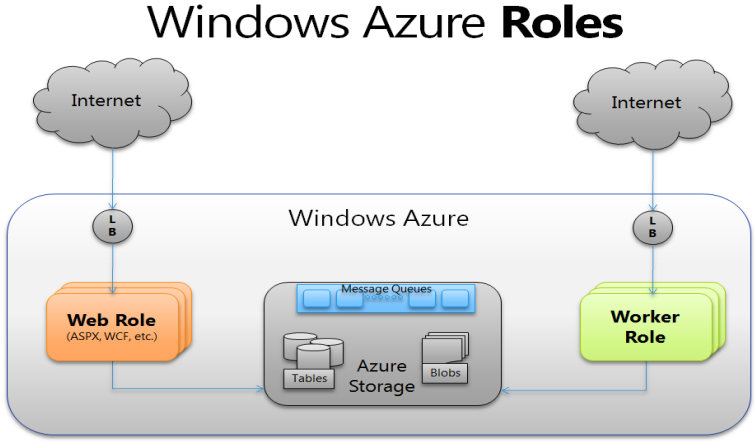



**Figure 4.- Roles in Windows Azure**

www.EBooksWorld.ir

Worker Roles are general purpose processes (each instance is actually a VM). They are normally used for long execution tasks, not interactive (similar to how a Windows Service works). We can even host full application platforms such as the *Java* virtual machine and *Apache Tomcat* in a Worker role.

*Windows Azure* starts *Worker-Roles* in the similar way to how Windows Server runs and starts Windows Services. They run in background.

*Web-Roles* may be seen as a specific case of *Worker-Role* but with IIS installed and available for use.

Normally, a web-role instance accepts incoming HTTP or HTTPS requests through the 80 and 443 TCP ports. These public ports are known as public endpoints. All the public endpoints are automatically balanced at the network level. The applications deployed in *Web-Roles* can be implemented using **ASP.NET**, **WCF** or other 'Non-Microsoft' technologies that are compatible with **IIS**, such as *PHP*, since IIS supports *FastCGI*.

## 3.4.- Implementing of a Basic Scenario in Windows Azure Platform

Business applications based on IIS (Web, RIA) and SQL Servers are very common these days and fulfill different missions, from critical mission applications to small department applications. In any case, these physical architectures can be like any of the following (in a simplified way):

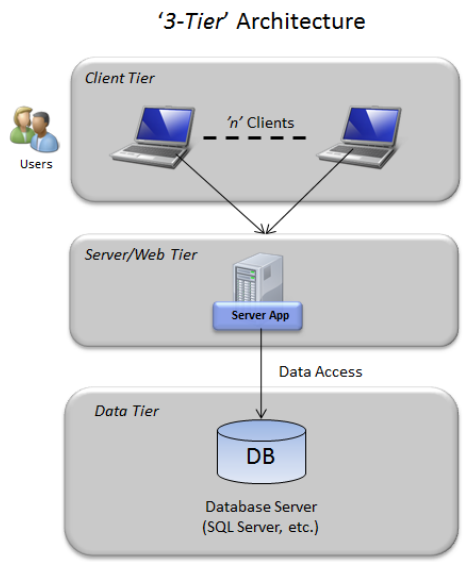



**Figure 5.- Web/RIA/Rich 3-Tier Application (a single web application server level)**

www.EBooksWorld.ir

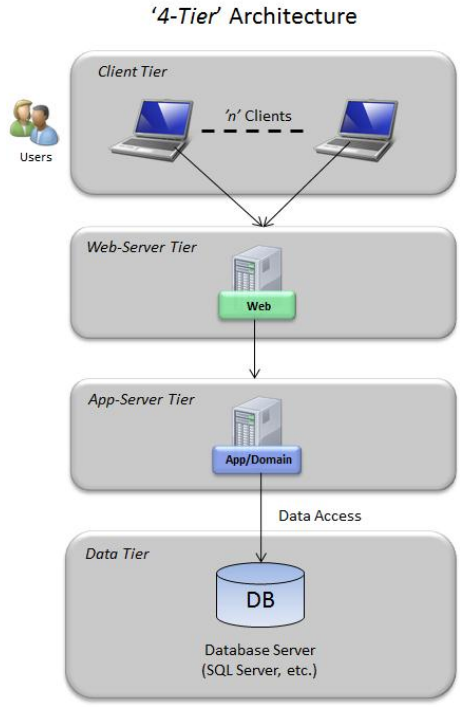



**Figure 6.- Web/RIA/Rich N-Tier application (Business and Web server levels)**

The applications that have these architecture patterns are strong candidates to migrate to Windows Azure. Their migration can be very simple, and we get the benefits of Windows Azure, such as quick deployment and resources optimization when adjusting cloud-resources to actual application needs.

It is worth mentioning that "direct migration" to Windows Azure does not necessarily mean we will have "magically" unlimited scalability. Scalability will depend on how the application architecture is designed and how it is developed. If all we do is directly migrated it, we may have immutable scalability limits, unless the application architecture or the technologies are changed (e.g., sometimes we will need to use Azure Storage instead SQL Azure, etc.).

In any case, in this scenario we are speaking of simple/direct migrations of current On-Premise architectures by direct deployment to Windows Azure. In this guide, this is known as "*Basic application scenario in Windows Azure platform*".

Normally, the objective of this scenario is to migrate current applications to Windows Azure requiring the minimum possible number of changes to be made on our application.

So, in a direct migration scenario from '*on-premise*' applications to Windows Azure applications, technical changes will not be very relevant, and the necessary changes will be minor and very transparent.

www.EBooksWorld.ir

This simple scenario allows us to have very high compatibility between the On-Premise version application (to be deployed in a ***Windows Server*** environment) and our cloud-application in ***Windows Azure***.

We can map between each technology from each environment:

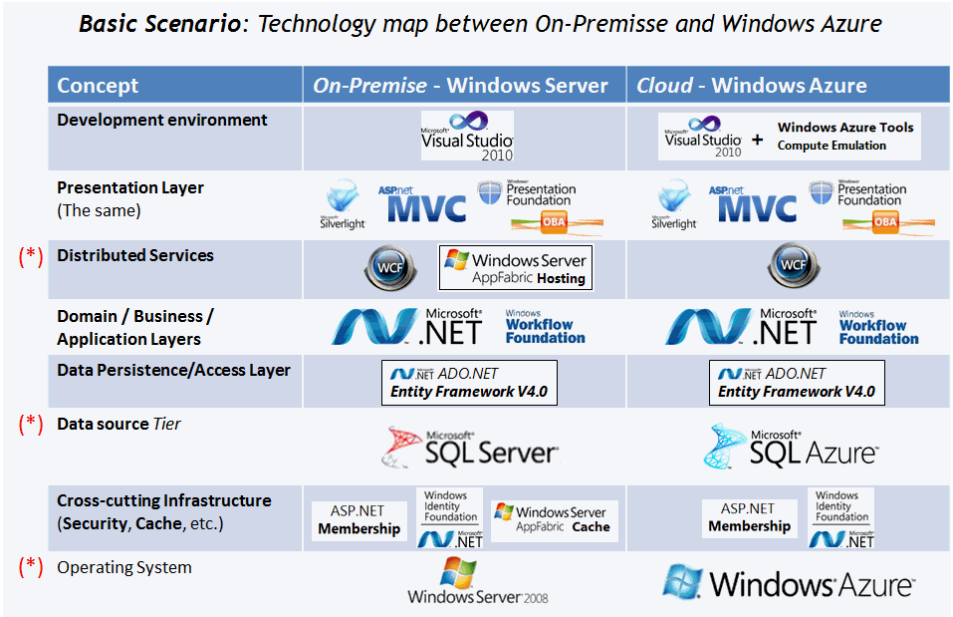



**Figure 7.-Relationship between 'Windows Azure' and 'On-Premise' technologies in a Basic Scenario**

We have placed an "(*)" next to the technologies or areas that will need minor changes.

www.EBooksWorld.ir

## 3.5.- Steps to migrate the sample NLayerApp Application to Windows Azure (Basic Scenario in the Cloud)

In this section we show the steps required to migrate our sample *NLayerApp* application so that it can be executed in the *Windows Azure* platform.

In this case, this migration coincides with the basic scenario mentioned above. This means we use the same DDD N-Layer architecture and virtually the same technologies, since the only required changes are the following:

**Changes required for basic migration to Windows Azure**

- ***SQL Server* database to *SQL Azure* migration**
    - It is clearly transparent when dealing with development and .NET components of data access.
    - In our code we only need to change the connection string by pointing at the DNS name of the new SQL Azure Server.
- **Adding a Windows Azure configuration project to the Visual Studio solution.**
    - This project only has two configuration XML files. Therefore, impact is very small.
- **Adding code for Diagnostics and instrumentation of Windows Azure Roles.**
    - This is the only intrusive change to our original on-premise code. However, there is not much code to be added.
    - It is necessary to organize/add our hosting projects within a Windows Azure role (Web Role or Worker Role).

The detailed steps for basic migration of our sample NLayerApp application to Windows Azure are explained in detail in the following sections.

www.EBooksWorld.ir

## 3.5.1.- Migrating SQL Server Databases to SQL Azure

This step is quite simple. We can replace our database in an SQL Server with a similar database in SQL Azure. The process for migrating a database schema and data from SQL Server to SQL Azure is quite easy (Although there are several possible ways).

**Note**:
There are some minor internal incompatibilities between the *SQL Server* and *SQL Azure* databases. In most cases, this will not affect us. However, we should perform the pertinent tests to ensure no problems arise. For more information about compatibility between databases and *Transact-SQL* from *SQL Server* and *SQL Azure*, see:

**http://msdn.microsoft.com/en-us/library/ff394115(v=MSDN.10).aspx**

### Database Schema Migration

The options available for the migration of a database schema from SQL Server to SQL Azure are:

- **Generating SQL statement Scripts using '*SQL Server Management Studio*' from *SQL Server 2008 R2*** which generates the required Transact-SQL statements compatible with SQL Azure. In the "Export-Scripts" task there is a new option specifying that we want to generate compatible SQL-Azure statements:

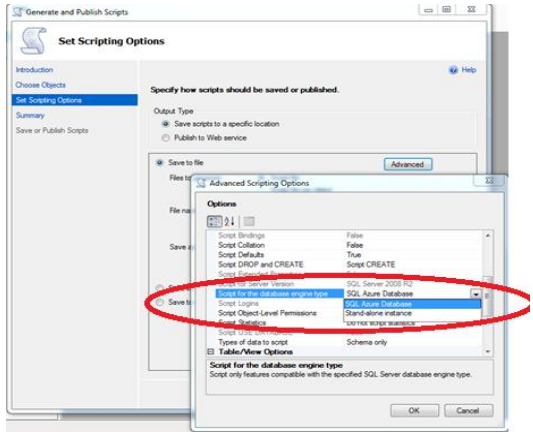



**Figure 8.-We can specify that we want to generate SQL-Azure compatible scripts**

www.EBooksWorld.ir

- **DACPAC (Data Tier Packages) package export** using *'SQL Server Management Studio'* from *SQL Server 2008 R2* or *Visual Studio 2010*. We will then be able to deploy it using the '*SQL Server Management Studio*' (SQL Server R2) connected to a SQL Azure logical database:

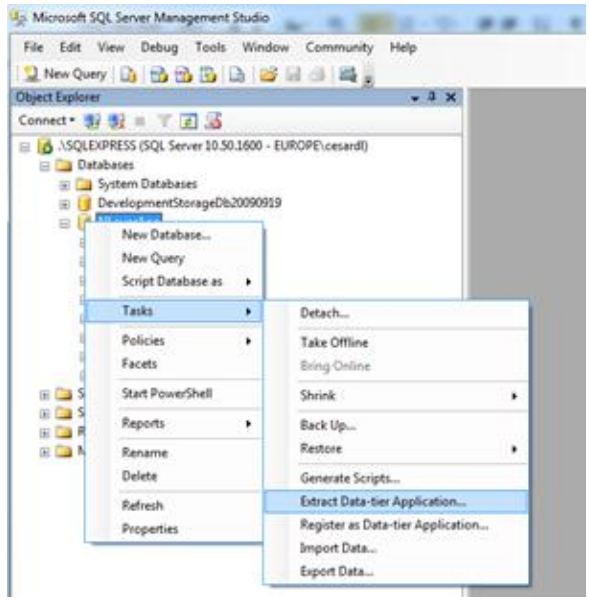



**Figure 9.-DACPAC package exportation from the SQL Server DB**

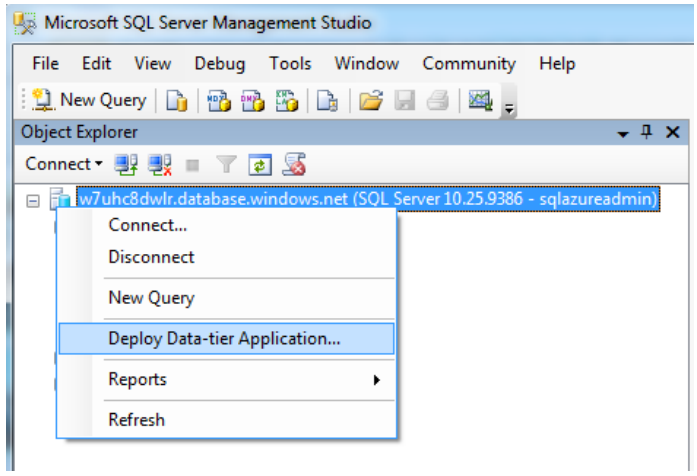



**Figure 10.-DACPAC package import/deployment to SQL Azure**

www.EBooksWorld.ir

### Database Schema and Data migration

- **Creating a Database and initial SQL Azure configuration**

The first step we need to take is to create an empty database in SQL Azure. The easiest way to accomplish this task is to create it from the SQL Azure portal:

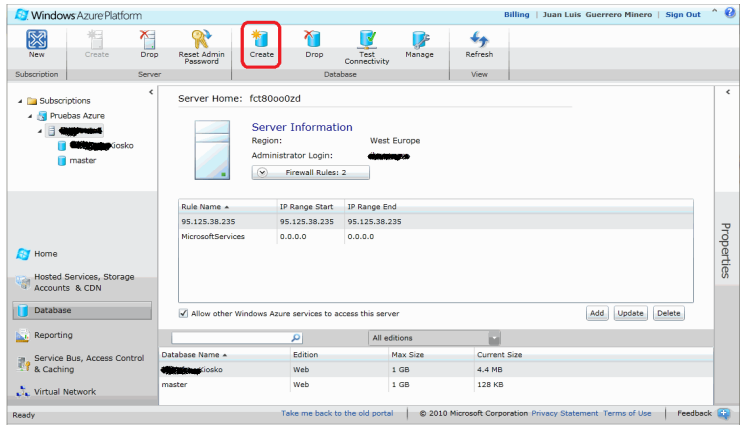



**Figure 11.-The easiest way is creating the DB from the SQL Azure portal**

In our case (for 'NLayerApp') a 1GB database is more than enough:

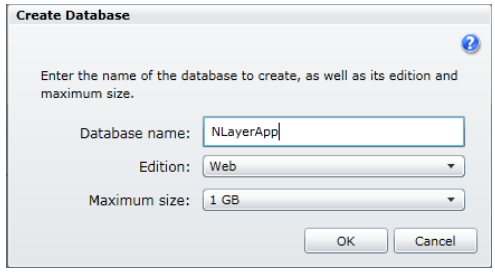



**Figure 12.-We define the name and size of the DB**

So it will appear similar to the following:

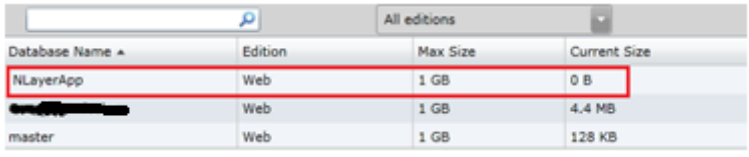



**Figure 13.-The DB is finally created**

www.EBooksWorld.ir

If we want to be able to access the database from a remote client application outside the Windows Azure datacenter (e.g. our development PC), then we should open the SQL Azure firewall to allow external access to any IP, adding a firewall rule in SQL Azure, as follows:

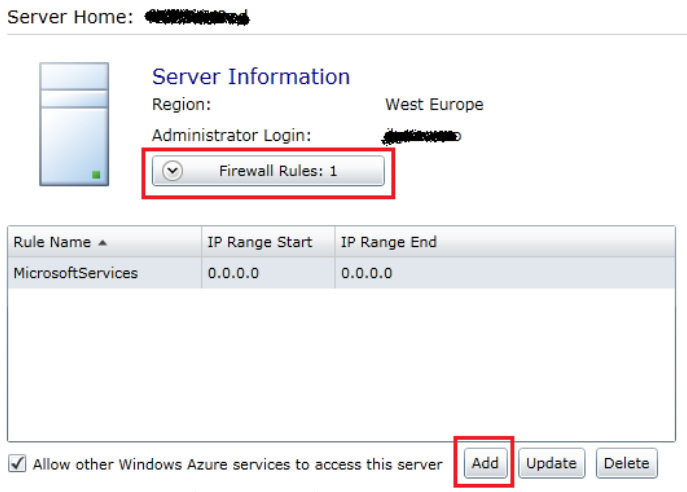



**Figure 14.-Firewall rules**

We usually create this kind of firewall rule only for development and testing time. We will probably need to remove this firewall rule during production time; otherwise, we will be allowing any IP (external machine) to access remotely to our SQL Azure logical server and database:

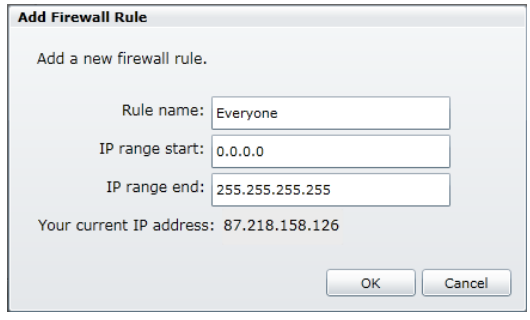



**Figure 15.-Firewall rule Configuration allowing any IP**

If we don't create such a firewall rule, we will not be able to remotely connect to our SQL Azure database from our *SQL Server Management Studio* or when *debugging* our application. In that case, we will obtain the following error/exception:

www.EBooksWorld.ir

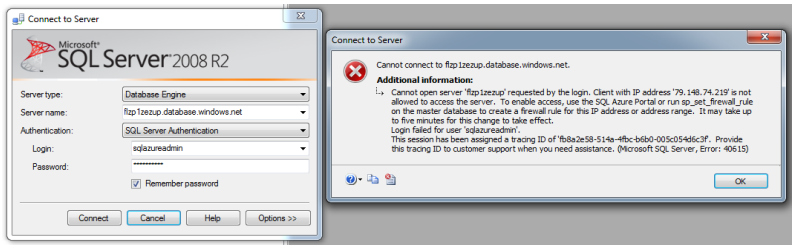



**Figure 16.-Error we would get if we do not open the firewall**

### Migrating a SQL Azure database

When we are migrating data from a SQL Server database to a SQL Azure database, we also have several options:

- **BCP** (*Command line* utility). We can manually migrate data using BCP. It is useful in systems where we want to launch a data migration script regularly. For more information on how to perform this data migration with BCP, check out this *post*:

  **http://blogs.msdn.com/b/cesardelatorre/archive/2010/06/04/importing-exporting-data-to-sql-azure-databases-using-bcp-and-sql-scripts.aspx**

- **SSIS** (*SQL Server Integration Services*): SSIS has much more advanced capabilities, such as ETL (*Extract, Transform, Load*). It is very simple to perform an export/import package against *SQL Azure*, similar to the following:

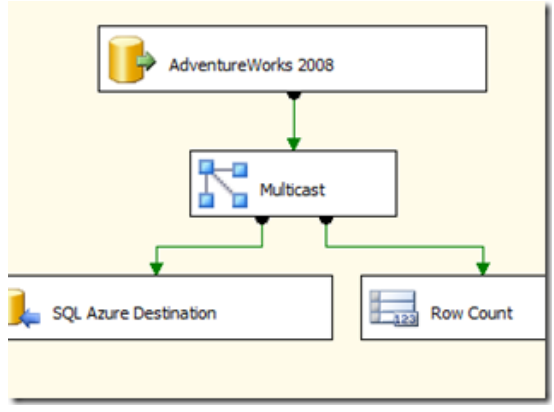



**Figure 17.- SSIS package sample**

The downside regarding the use of SSIS for simple but large migrations is that it always works by processing record by record, so it could be too slow for certain databases.

www.EBooksWorld.ir

- **SQL Azure Data Sync:** This is based on *Microsoft Sync Framework SDK* and *Microsoft Sync Framework Power Pack for SQL Azure*. However, this technology goes beyond simple migrations, as it offers complex data synchronization capabilities and not just migration functionality.

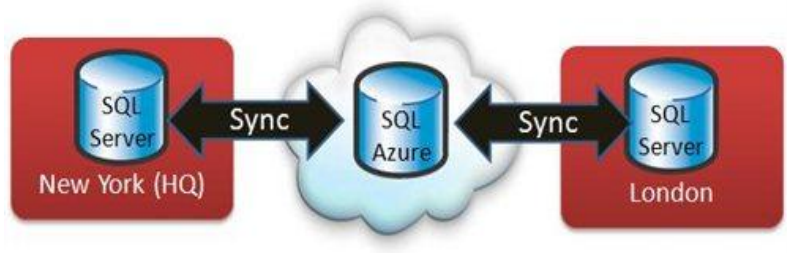



**Figure 18.-SQL Azure Data Sync scenario**

- **SQL Azure Migration Wizard**: Currently, this is probably the most simple and most frequently used option to perform simple database migrations because it can perform database schema and data migration at the same time:

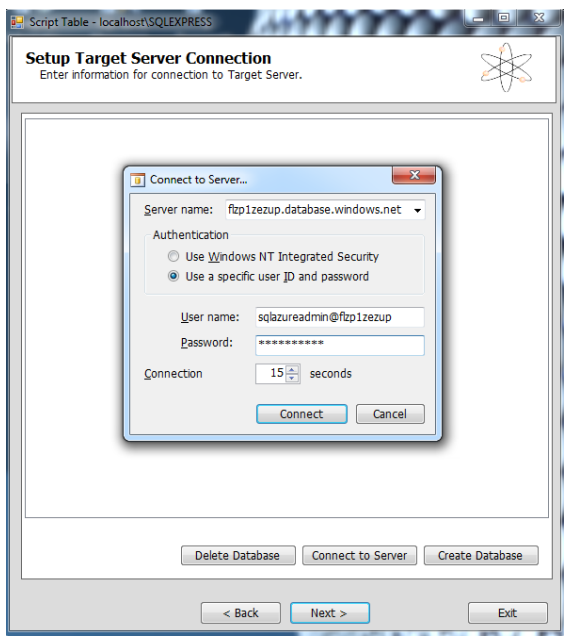



**Figure 19.- SQL Azure Migration Wizard**

*SQL Azure Migration Wizard* is available at: **http://sqlazuremw.codeplex.com/**

www.EBooksWorld.ir

We can migrate our '**NLayerApp**' *SQL Server* database to *Windows Azure* using any of the approaches explained above. As mentioned, however, the simplest and easiest way is probably using ***SQL Azure Migration Wizard***.

### SQL Azure Database Management

We currently have two ways to manage our databases in the SQL Azure:

- ***SQL Server 2008 R2 SQL Server Management Studio***:

This option is the most powerful. All we need to do is to install the '***SQL Server 2008 R2 SQL Server Management Studio***' in our development/management machine. Once installed, it allows us to connect remotely to our SQL Azure database, as follows:

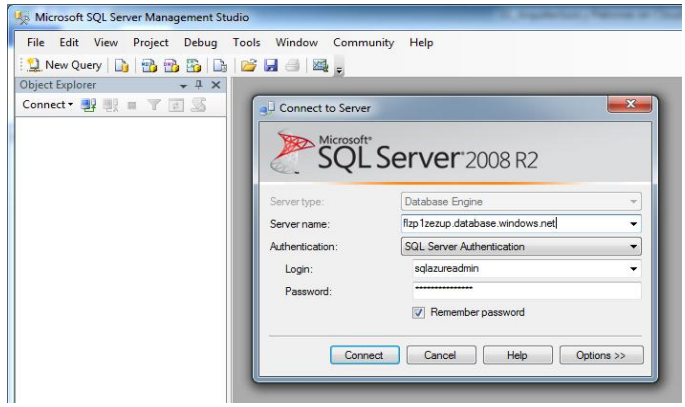



**Figure 20.- Connecting to SQL Azure from SQL Management Studio**

We can then manage our database in a familiar way:

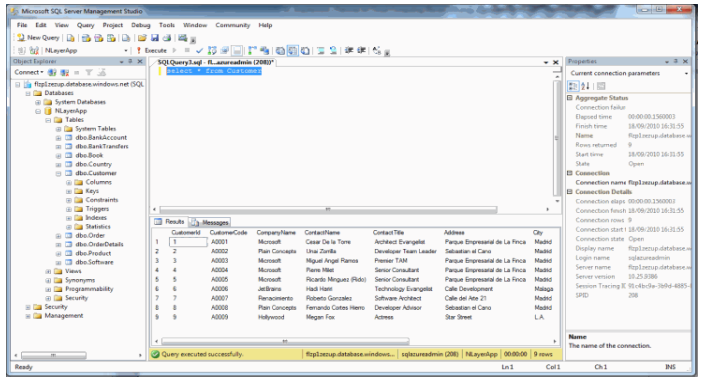


**Figure 21.- Managing SQL Azure databases using the SQL Management Studio**

www.EBooksWorld.ir

- ***Microsoft SQL Azure Database Manager (Web/RIA admin- app)***:

The second option is quite innovative and consists in using a new SQL Azure management application (aka. *Project Code-Named "Houston"*). The new feature in this tool is that it is a Web/RIA application (Silverlight client). Therefore, as our database is on the Internet, we can manage it from this tool using any PC and browser, without the need for previously installed SQL Server management software. Any machine in the world with a simple browser and the ***Silverlight*** plug-in installed is good enough to use it. Images of the Manager are included below:

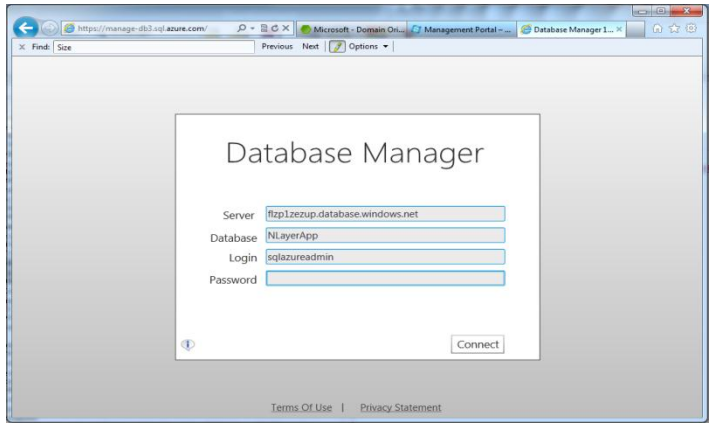



**Figure 22.- SQL Azure Database Manager**

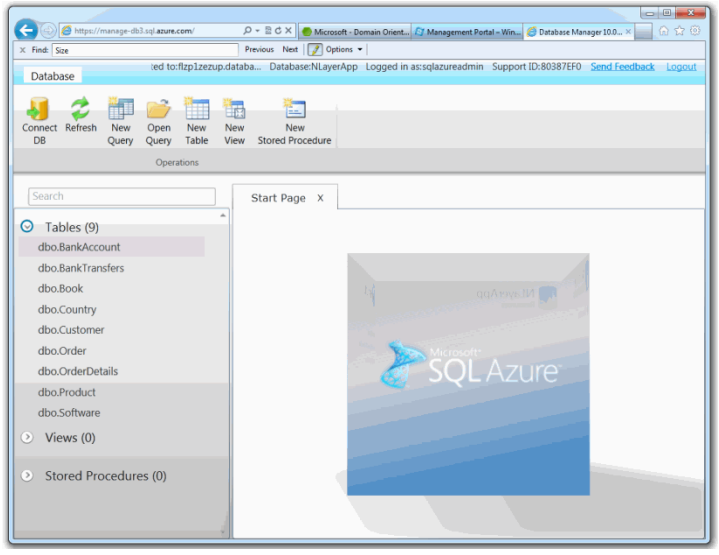



**Figure 23.- Listing tables using the browser**

www.EBooksWorld.ir

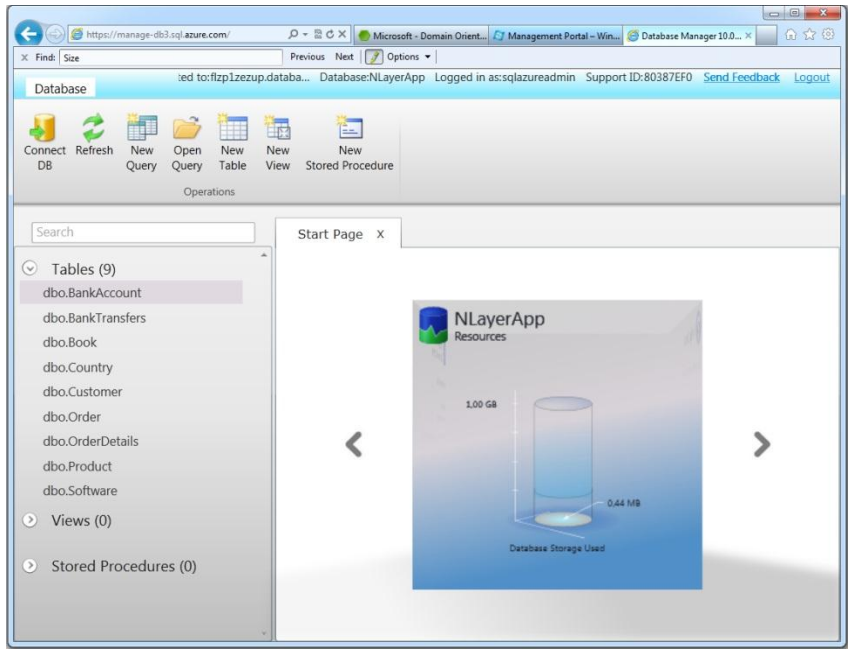



**Figure 24.- Database usage charts**

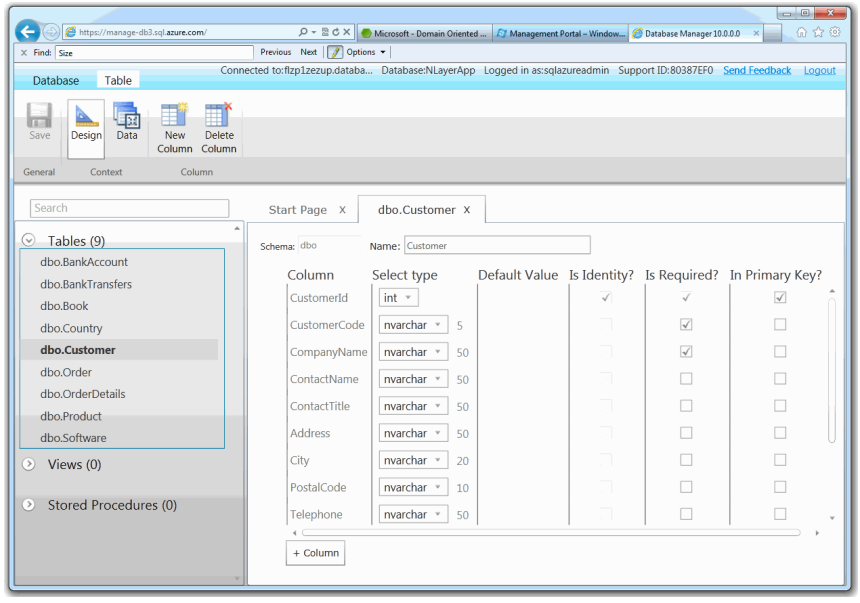



**Figure 25.- Managing SQL Azure Tables using a browser**

www.EBooksWorld.ir

## 3.5.2.- Changing the ADO.NET EF Connection String

This step is extremely easy. Since our database is situated in SQL Azure, we should change the connection string used by ADO.NET / EF.

For example, we should change the following code in the .config configuration file of the WCF services hosting project in our *NLayerApp* application:

```
EF CONNECTION STRING FOR SQL SERVER

<add name="MainModuleContext"
connectionString="metadata=res://*/Model.MainModuleDataModel.csdl|res://
*/Model.MainModuleDataModel.ssdl|res://*/Model.MainModuleDataModel.msl;p
rovider=System.Data.SqlClient;provider connection string=&quot;Data
Source=.\SQLEXPRESS;Initial Catalog=NLayerApp;Integrated
Security=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" />
```

We should then use the same connection string but with a different server name and credentials:

```
EF CONNECTION STRING FOR SQL AZURE

<addname="MainModuleContext"
connectionString="metadata=res://*/Model.MainModuleDataModel.csdl|res://
*/Model.MainModuleDataModel.ssdl|res://*/Model.MainModuleDataModel.msl;p
rovider=System.Data.SqlClient;provider connection
string=&quot;Server=tcp:KKuhc8dwlr.database.windows.net;Database=NLayerA
pp;User
ID=sqlazureadmin@KKuhc8dwlr;Password=mipass@word1;Trusted_Connection=Fal
se;Encrypt=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" />
```

As shown, **basically, the only change to be made is to put the DNS name of the SQL-AZURE server** as well as the SQL Server standard credentials.

If we were not using EF, but only ADO.NET, the change would be similar, simply the server name and security credentials.

www.EBooksWorld.ir

**NOTE:**
The user and password to be used (standard SQL Server credentials) are necessary because *SQL Azure* currently supports only *SQL Server* standard security and does not support AD integrated security. This is logical, by the way, because we do not actually have AD support within *Windows Azure* cloud (At the moment of this writing. We might have it in the future).

## 3.5.3.- Migrating Hosting Projects from IIS to Azure

**IMPORTANT:**
In the following steps we are migrating each IIS site to a new Windows Azure WebRole. But starting on WA SDK 1.3, we can also have several sites running within the same WA WebRole. This is possible because staring on WA SDK 1.3, Windows Azure is running Full-IIS instead of HWC (Hosted Web Core). In many cases this new approach will be cheaper regarding WA 'pay per use' costs.

**Migration of WCF Service *Hosting* from IIS/Cassini to Azure WCF WebRole**

Migrating the sample application hosting WCF Service code to a new project created in the Windows Azure WCF WebRole type is quite simple.

The steps are as follows:

- Within our solution, we create a Windows Azure project (e.g., '**AzureServices'**) with a role of the **WCF ServiceWebRole type** (e.g., '**WCFWebRole'**). We should have something similar to the following with the Azure role definition:

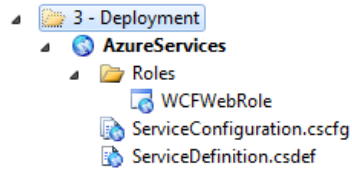



**Figure 26.- Azure configuration project**

www.EBooksWorld.ir

- As regards the WCF project, we should create an empty/standard *WCFWebRole* project, *so* we should have something similar to the following:

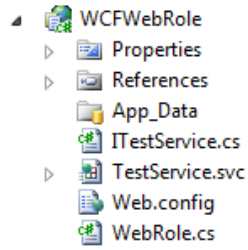



**Figure 27.-**

- For the time being, this WCF project may be located in any part of our *NLayerApp* application Solution. Eventually, we would move it into the '1.2 DistributedServices' folder.

- Copy the definition files of our WCF hosting project and paste it into the web.config owned by the new WebRole; that is, the following files:

    - **crossdomain.xml** (to allow Silverlight accesses from other DNS domains)

    - **MainModule.svc** (endpoint file for accessing the WCF service)

    - Original NLayerApp WCF service **web.config**, changing the XML code of '<system.diagnostics>' to the following code which uses Azure.Diagnostics instead of the standard .NET-Windows listeners:

www.EBooksWorld.ir

```
<system.diagnostics>

   <sharedListeners>
     <add type="Microsoft.WindowsAzure.Diagnostics.DiagnosticMonitorTrac
  eListener, Microsoft.WindowsAzure.Diagnostics, Version=1.0.0.0, Cultur
  e=neutral, PublicKeyToken=31bf3856ad364e35"
           name="AzureDiagnostics">
               <filter type="" />
       </add>
     </sharedListeners>

     <trace>
       <listeners>
         <add name="AzureDiagnostics" />
       </listeners>
     </trace>

     <sources>
       <source name="NLayerApp" switchValue="Error">
         <listeners>
           <add name="AzureDiagnostics" />
         </listeners>
       </source>
     </sources>

  </system.diagnostics>
```

Add references to the assemblies used by the WCF service of our module:

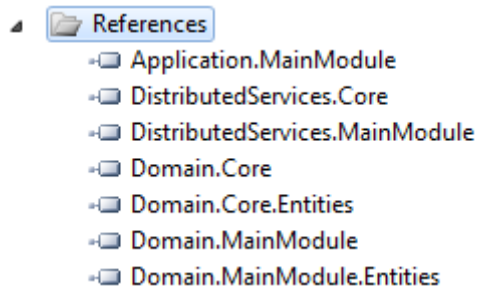



**Figure 28.- References needed by our NLayerApp WCF Service**

Eliminate the "`<serviceDiscovery/>`" section in the new web.config, because we will not use WS-Discovery in Azure as it is not running in a LAN with local broadcast possibilities.

Add the following line within the `<system.serviceModel>` section

```
<serviceHostingEnvironmentmultipleSiteBindingsEnabled="true" />
```

www.EBooksWorld.ir

Eliminate the following line also related to WS-Discovery:

```
  <endpoint name="DiscoveryEndpoint" listenUriMode="Explicit"
kind="udpDiscoveryEndpoint" />
```

- **Check the Database EF Connection String change:** if we haven't done it before, we should do it now. In the Entity Framework connection string (Web.config of our WCF hosting project in Azure) we should simply change the name of the SQL Server to the name of the SQL Azure logical server, and specify the access credentials to SQL Azure (user and password).

```
<?xml version="1.0"?>
<configuration>
      <connectionStrings>
    <!-- (SQL AZURE) -->
    <addname="MainModuleContext"
connectionString="metadata=res://NLayerApp.Infrastructure.Data.MainM
odule/Model.MainModuleDataModel.csdl|res://NLayerApp.Infrastructure.
Data.MainModule/Model.MainModuleDataModel.ssdl|res://NLayerApp.Infra
structure.Data.MainModule/Model.MainModuleDataModel.msl;provider=Sys
tem.Data.SqlClient;provider connection
string=&quot;Server=tcp:w7xxxxxxx.database.windows.net;Database=NLa
yerApp;User
ID=sqlazureadmin@w7xxxxxxx;Password=mipassword;Trusted_Connection=F
alse;Encrypt=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" />
    <!-- (SQL Server Express) -->
    <!-- <add name="MainModuleContext"
connectionString="metadata=res://NLayerApp.Infrastructure.Data.MainM
odule/Model.MainModuleDataModel.csdl|res://NLayerApp.Infrastructure.
Data.MainModule/Model.MainModuleDataModel.ssdl|res://NLayerApp.Infra
structure.Data.MainModule/Model.MainModuleDataModel.msl;provider=Sys
tem.Data.SqlClient;provider connection string=&quot;Data
Source=.\SQLEXPRESS;Initial Catalog=NLayerApp;Integrated
Security=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" /> -->
      </connectionStrings>
```

**Migrating the Web-Silverlight Hosting project from IIS/Cassini to AzureWebRole**

This migration will require us to modify our original Silverlight project or create a new WebRole Azure project and move the Silverlight code to this new project. We have chosen the second option (although the other option is also feasible). To that end, we create a new WebRole project similar to the following:

www.EBooksWorld.ir

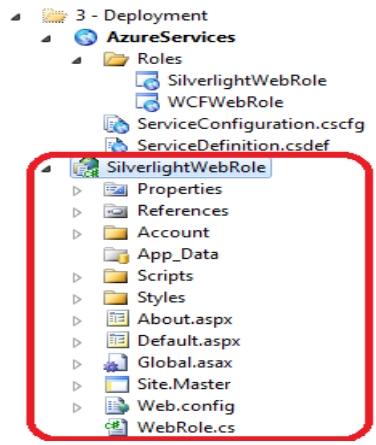



**Figure 29 New Webrole for the Silverlight project**

- **Changes in Silverlight project files:** We can delete all example pages of the SL project, Styles, Scripts, Accounts, Global.asax, etc. and add the original (NLayerApp) Silverlight hosting/IIS files from the project '*Silverlight.Client.Web*':

    - **Web.config**: Add the Web.config of our original NLayerApp Silverlight project (actually, this web.config is almost empty) and the only thing we have to add is the XML related to the Windows Azure Diagnostics:

```
   <system.diagnostics>
    <trace>
      <listeners>
        <add type="Microsoft.WindowsAzure.Diagnostics.DiagnosticMoni
torTraceListener, Microsoft.WindowsAzure.Diagnostics, Version=1.0.0.
0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
          name="AzureDiagnostics">
          <filter type="" />
        </add>
      </listeners>
    </trace>
  </system.diagnostics>
```

We will have a Web.config similar to the following:

www.EBooksWorld.ir

```
<?xml version="1.0"?>
<configuration>
    <system.web>
        <compilation debug="true" targetFramework="4.0" />
    </system.web>
  <system.diagnostics>
    <trace>
      <listeners>
        <add type="Microsoft.WindowsAzure.Diagnostics.Diagnostic
MonitorTraceListener, Microsoft.WindowsAzure.Diagnostics, Versio
n=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
          name="AzureDiagnostics">
          <filter type="" />
        </add>
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

- Copy the **Silverlight.Client.Web.html** and **Silverlight.js** files and establish the **Silverlight.Client.Web.html** page as the start-up default page (to be tested during *debugging*, etc.).

- Add the relationship between our Azure hosting WebRole project and our Silverlight application project. For that purpose, open the properties of our **SilverlightWebRole** project in the "Silverlight Applications" tab and add our existing project, '***Silverlight.Client***':

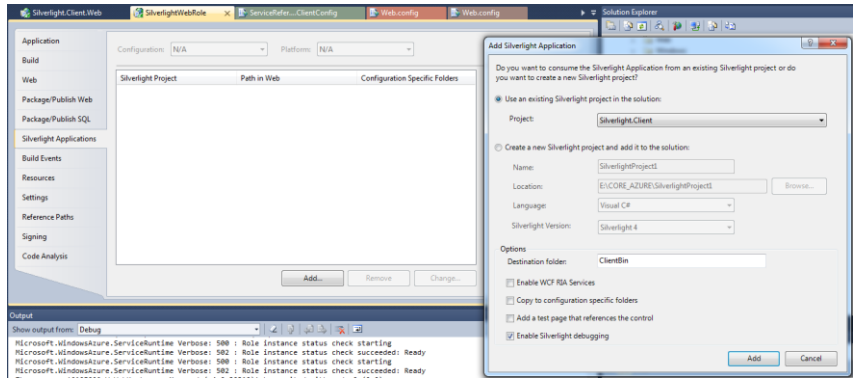



**Figure 30.- Establishing Silverlight relationship**

www.EBooksWorld.ir

This is how it should be displayed:

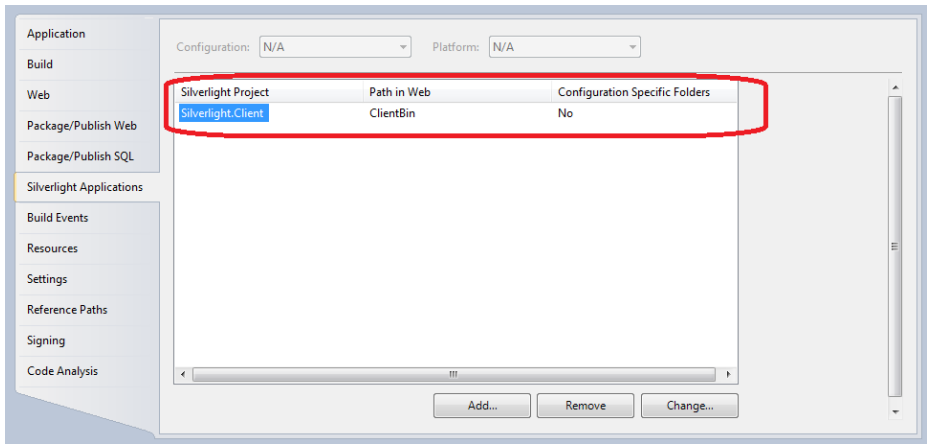



**Figure 31.- Silverlight relationship**

This action causes Silverlight .XAP to be included in our hosting WebRole project:

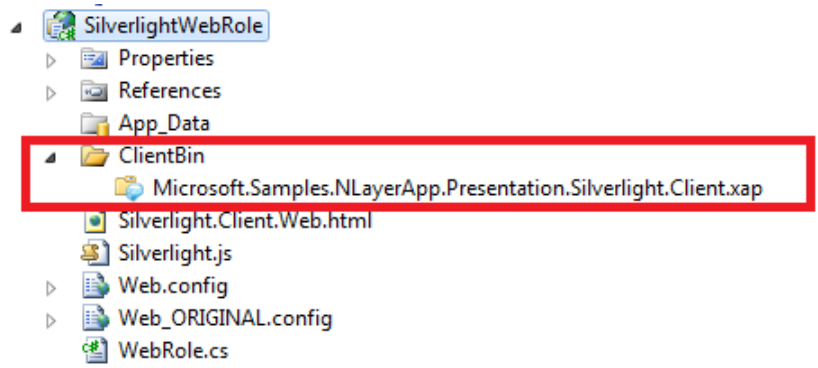



**Figure 32.- Silverlight .XAP added to our WebRole**

- **Change all the WCF Services URLs to consume:** In the Silverlight project, named **Silverlight.Client** (not in the Azure WebRole hosting we were talking about before), modify the WCF client bindings configuration (in the **ServiceReferences.ClientConfig** file) so that it now consumes the WCF services hosted in Windows Azure (the URL will be different).

When executing the WCF service in test mode with Windows Azure (AzureDevelopmentFabric/Compute Emulator), it appears as follows:

www.EBooksWorld.ir

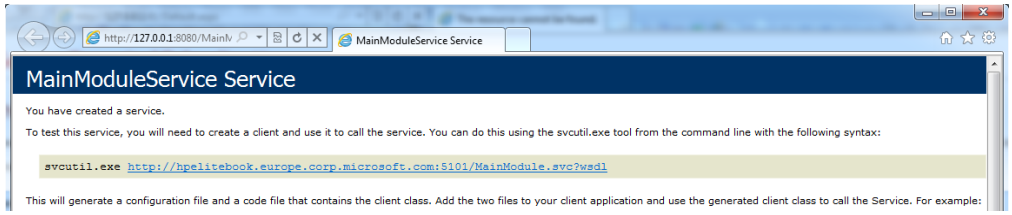



**Figure 33.- Testing our WCF Service**

We see that it is running on the **http://127.0.0.1:8080** address (or the TCP port chosen in the Web properties of the WcfWebRol project), so we should change it in the **ServiceReferences.ClientConfig** file as follows:

```
   <configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicBindingForSilverlightClients" maxBufferSize=
"2147483647"
          maxReceivedMessageSize="2147483647">
          <security mode="None" />
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <!-- ADDRESS FOR LOCAL AZURE DEVELOPMENT FABRIC-->
      <endpoint address="http://127.0.0.1:8080/MainModule.svc/basic"
        binding="basicHttpBinding" bindingConfiguration="BasicBindingFor
SilverlightClients"
        contract="ServiceAgent.IMainModuleService" name="BasicBindingFor
SilverlightClients" />
    </client>
    <extensions />
  </system.serviceModel>
</configuration>
```

Remember that, when we want to upload it to the real Windows Azure cloud on the Internet, we should change the IP and port for those actually used in Windows Azure on the Internet (Production environment).

- So, at this point, if we execute our application in debugging mode (having the Windows Azure configuration project as start-up by default), our application will be deployed and executed in the local '**Windows Azure Compute Emulator**' environment (aka. **Development Fabric**) (although we will be accessing the real SQL Azure DB), as we can see below:

www.EBooksWorld.ir

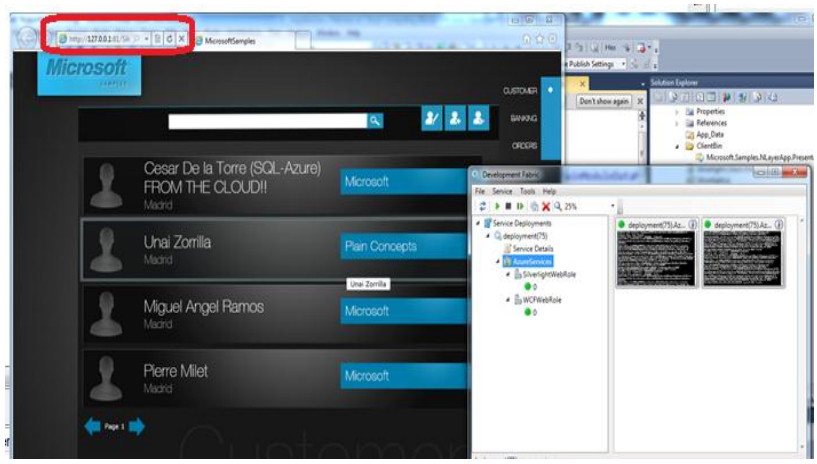



**Figure 34.- Running NLayerApp on Azure Compute emulator**

Checking the IP address (127.0.0.1). Note that this execution of our application is made locally, running on the '*Windows Azure Compute Emulator*' simulation.

## 3.5.4.- Deploying our Application in the Production Windows Azure Cloud (Internet)

This execution environment change from the WA simulated environment (*Windows Azure Compute Emulator*) to the real Windows Azure cloud is extremely simple.

**Configuring TCP ports to be used in the cloud**

- Before deployment, we should confirm/configure the TCP ports to be used in the cloud:
    - Confirm/change the execution port of our Silverlight WebRole. In this case we specify the default Http port (TCP port 80). Check this within the WebRole properties in the Endpoints tab:

www.EBooksWorld.ir

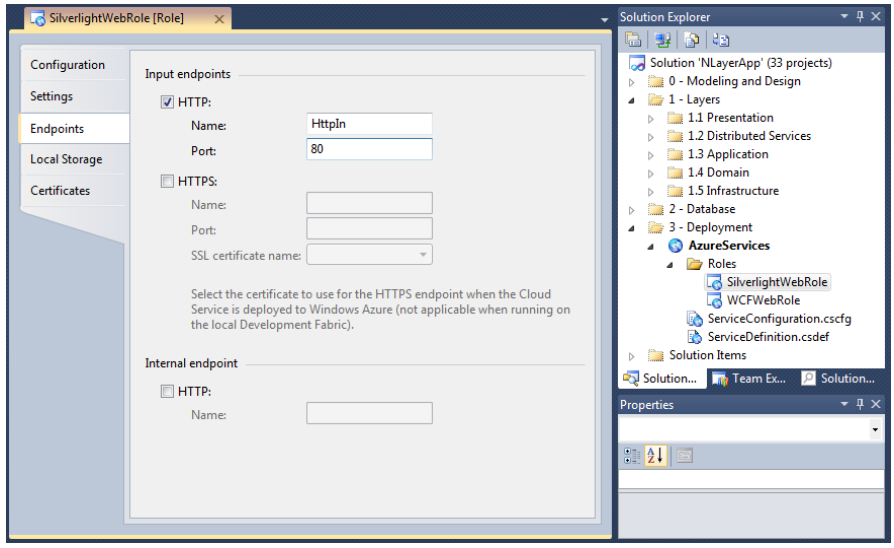



**Figure 35.- Checking/Changing host project port**

- Also confirm the TCP port that is going to be used by our WCF service in the WCFWebRole project. In our case, we will be using port 8080.

- **Change the WCF service consumption URL.** After deploying our WCF service to the real Windows Azure cloud, our WCF service URL will be based on the URL of our Windows Azure service. In this case, it wasI created as '**http://nlayerazure.cloudapp.net/**', so the binding URL to consume the WCF service should be **http://nlayerazure.cloudapp.net/:8080** instead of **http://127.0.0.1:8080/** the one we were using before in the local simulated environment (*Azure Compute Emulator*):

```
<!-- AZURE-INTERNET-CLOUD -->
<endpoint address="http://nlayerazure.cloudapp.net:8080/MainModule.svc/basic"
          binding="basicHttpBinding" bindingConfiguration="BasicBindingForSilverlightClients"
          contract="ServiceAgent.IMainModuleService" name="BasicBindingForSilverlightClients" />
```

This client binding definition should be specified in the '***ServiceReferences.ClientConfig***' file of the Silverlight project (in our sample application, it is called '***Silverlight.Client***' project).

www.EBooksWorld.ir

**Silverlight Website default page in Azure Web-Role**

If our application start-up page is not called default.aspx, we will get an error when we try to access it in the Windows Azure application URL, because our default page in the Silverlight project is called '*Silverlight.Client.Web.html*'.

When "following the PaaS path", the Windows Azure portal does not allow modifications in the IIS console. However, as most of the IIS 7.x configurations can also be made through the XML configuration files, we do not need any IIS console to do it.

In this case, we can establish the default page of the Windows Azure WebRole application by changing the web.config of our application (**the web.config** of the **SilverlightWebRole** project). We simply have to add the following XML section, within the '**System.Web.Server**' section:

```
<system.webServer>
    <modules runAllManagedModulesForAllRequests="true"/>

    <defaultDocument>
      <files>
        <clear/>
        <add value="Silverlight.Client.Web.html"/>
      </files>
    </defaultDocument>

     </system.webServer>
```

If there are several folders in our application, we could even specify a default page for each sub-folder. Instead of specifying it by "System.WebServer", we specify it by "location". For example:

```
      <location path="webFolder1">
           <system.webServer>
               <defaultDocument>
                   <files>
                       <add value="myDefalutPage.aspx" />
                   </files>
               </defaultDocument>
           </system.webServer>
       </location>
       <location path="webFolder2">
           <system.webServer>
               <defaultDocument>
                   <files>
                       <add value=" myDefalutPage.aspx" />
                   </files>
               </defaultDocument>
           </system.webServer>
            </location>
      <location path="webFolder1">
```

www.EBooksWorld.ir

### Windows Azure *Diagnostics-Storage* Connection-String

To export the diagnostics information (Trace, performance counters, etc.) in Windows Azure we should do so from our WebRole/WorkerRole applications and save it in an 'Azure Storage' database. Once there, we should be able to query it through API or by using different third party tools (***Cerebrata Azure Diagnostics Manager***).

To that end, it is important to change the connection string of our Windows Azure Role configuration file. This means changing the default string for the simulated environment. In other words, the following line in the '**ServiceConfiguration.cscfg**' file of the Windows Azure configuration project:

```
<Setting  name="DiagnosticsConnectionString"
value="UseDevelopmentStorage=true" />
```

We should specify, instead, a connection string against an 'Azure-Storage' that we have available for storing these diagnostic data, for example:

```
<ConfigurationSettings>
      <Setting name="DiagnosticsConnectionString"
      value="DefaultEndpointsProtocol=https;AccountName=cesardldiagnostics;AccountKey=hgm
      u0lpsPCpysCHANGED0OjRfR32XHGCI4pY6lOI/EvoS3yJp/9d7YJXzAAMVIzKLkyRLhKt
      //XNqp+CHANGED==" />
</ConfigurationSettings>
```

The *AccountName* and *AccountKey* values will depend on each Azure Storage subscription available. The following two steps in the Windows Azure Portal, shown in the following figures, should be taken:

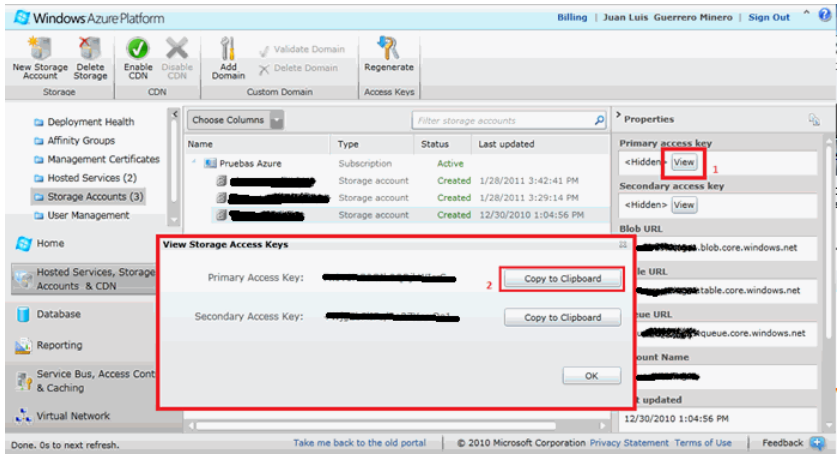



**Figure 36.- Windows Azure Storage account info**

www.EBooksWorld.ir

The keys to be copied & pasted are the ones that appear crossed out (obviously for security reasons).

It can also be changed by using the Visual Studio interface:

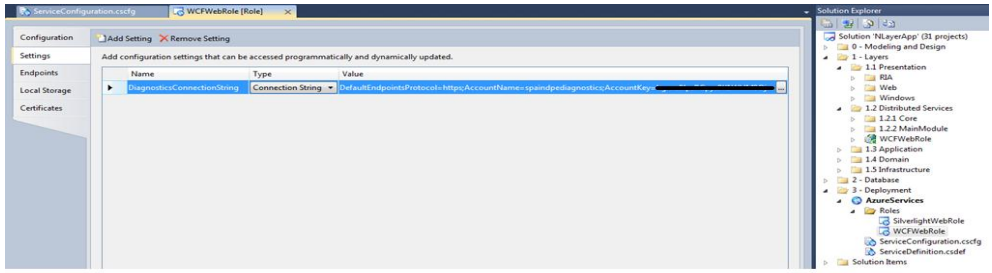



**Figure 37.- Changing WA Storage account info using VS.2010 UI**

The artifact that enables collecting the 'Diagnostics' information is the Windows Azure role '***start-up code***', situated within the '**WebRole.cs**' file for each WebRole or WorkerRole project.

This is the only change that is slightly more intrusive as compared to our initial code running in Windows Server. However, as mentioned, there is not very much code to deal with.

When we start the Diagnostic Monitor, we use the URL of our Azure-Storage to store diagnostic information.

It is a very simple code, like the following:

```
public class WebRole : RoleEntryPoint
{
        public override bool OnStart()
        {
                                        Use the Diagnostics connection string
DiagnosticMonitor.Start("DiagnosticsConnectionString");

            // For information on handling configuration changes
            // see the MSDN topic at http://go.microsoft.com/fwlink/?LinkId=1
66357.
            RoleEnvironment.Changing += RoleEnvironmentChanging;

            return base.OnStart();
        }

        private void RoleEnvironmentChanging(object sender, RoleEnvironmentCh
angingEventArgs e)
        {
            // If a configuration setting is changing
            if (e.Changes.Any(change => change is RoleEnvironmentConfiguratio
nSettingChange))
            {
                // Set e.Cancel to true to restart this role instance
                e.Cancel = true;
            }
```

www.EBooksWorld.ir

```
            }
}
```

For a better understanding of the Windows Azure instrumentation system better, check out this information:

**Take Control of Logging and Tracing in Windows Azure**

**http://msdn.microsoft.com/en-us/magazine/ff714589.aspx**

**Note1:**

After **Windows Azure SDK 1.3** release, we now use full IIS instead HWC (*Hosted Web Core*) in Windows Azure WebRoles. By using IIS, we can have several WebSites per WebRole, so now the internal architecture and processes are different. This is why we have to change the way we use the API regarding Windows Azure configuration settings.
For more info, read **http://bit.ly/hsMzhf**

**Note2:**

Also, for quick monitoring, staring on that mentioned release, we can also connect through Terminal Services and directly watch and analyze Performance counters in any VM (Role, etc.)

## 3.5.5.- Web image management: Change from local storage (disk) to Windows Azure Blobs

Specifically as regards Web applications such as CMS systems that use dynamic images storage (.JPG, .PNG, etc.) mostly stored in the local hard disk, we recommend to migrate the system to a centralized storage system to store these images in ***Blobs*** within *Windows Azure Storage or Windows Azure CDN*. This is very important if the images are dynamically changed by administration users, etc., such as photo images for a product catalogue. On the other hand, if images are static images that will not change, they can stay in the local directory for the web resource files.

For more information about 'Windows Azure Blob Storage', check out the SDK of Windows Azure at:
**http://www.microsoft.com/downloads/details.aspx?FamilyID=21910585-8693-4185-826e-e658535940aa&displaylang=en**

www.EBooksWorld.ir

## 3.5.6.- Windows Azure Security

This subject is most likely the most complex.Since our application is on the Internet and outside the security domain of our organization, the application security system will probably have to change, at least partially.

Depending on the security type (authentication and authorization) that we used before in our *on-premise* application, we might perform some partial changes or maybe no changes at all:

**Table 3.- Windows Azure Security**

| On-Premise App Security | App Security in Windows Azure |
|---|---|
| **Membership** as an authentication system against isolated providers (SQL Server DB tables, of the application itself) | We can continue using **Membership**. This does not require changes impacting the application development, because the Membership provider for SQL Azure is the same as for SQL Server. Only the tables required in SQL Azure have to be created. To that end, the Script SQL is a bit different. See:<br>**http://code.msdn.microsoft.com/KB2006191**<br>**http://support.microsoft.com/kb/2006191/** |
| **Windows Security** integrated with the **Active Directory** authentication and framework or **AzMan** authorization API (***Authorization Manager***). | Change to '**Claims based Security**' using **ADFS 2.0** and **WIF** (*Windows Identity Foundation*) and integrating with corporate AD through the ADFS 2.0 STS Server published on the Internet.<br>Even if users are not corporate users, currently it is not possible to create an isolated AD in Windows Azure. Therefore, authentication must be delegated to a corporate AD through ADFS 2.0 STS (*Security Token Service*).<br><br>***NOTE***: to make changes related to WIF in our application and to establish trust relationships between our application and the STS (ADFS 2.0), it is most convenient to use the **FedUtil** tool, provided as part of WIF.<br><br>**http://msdn.microsoft.com/en-us/library/ee517284.aspx** |

www.EBooksWorld.ir

| | |
|---|---|
| Several Authentication systems published/accessible through the Internet, such as **Windows Live ID**, **OpenID, FacebookID, GoogleID, YahooID**, and even **Windows Security** integrated with **Active Directory** authentication published with ADFS 2.0 on the Internet. | Use the Windows Azure ***AppFabric Access Control.*** Especially in cases when we want to have several simultaneous authentication systems for our application, the best choice is to use WA Access-Control. This decouples our application from the different authentication systems we want to use. For more information, check out the sections of this guide related to *Claims Based Security* (Cross-Cutting concerns chapter) that are closely related to AC (AC is actually a STS in the cloud). |

## 3.5.7.- Other items to be considered when migrating applications to Windows Azure

The following are some simple but useful migration tips:

- **Assembly name length**: be careful with the ***assemblies name*** length. Long names that may not have any problems in a Windows Server solution can cause trouble in Windows Azure by exceeding the limit when Windows Azure adds more information to the path.

- **Use the Azure Diagnostics Monitor TraceListener**: when using listeners to perform tracing and logging, it should be changedto **Microsoft.WindowsAzure.Diagnostics.DiagnosticMonitorTraceListener** as the default listener. We recommend using a 'sharedListener' that points to the one pertaining to Windows Azure. On top of this, we can still use our previous system.

- **ASP.NET sessions**: we should not use '*in process*'ASP.NET sessions stored in the IIS pool memory space, because when we have several load-balanced Web-Roles and the session system will not work properly. This also happens in balanced systems when using Windows Server if the load-balance is pure (with no affinity).

  Options in Windows Azure:

  - ***ASP.NET Session provider forAppFabric-Cache*** (*codename.'Velocity'*). This is the most highly recommended option.

  - *ASP.NET Session provider for Azure Storage (Sample)* :**http://code.msdn.microsoft.com/windowsazuresamples**

www.EBooksWorld.ir

- *ASP.NET Session provider for SQL Azure* (currently, this provider does not work in SQL Azure because it uses SQL Agent, which is not supported in SQL Azure)

- **Retries in SQL Azure connections**: Connections are often closed in SQL Azure in order to obtain high availability. Therefore, the logic of our application has to control connection errors and try to re-connect. See:

  **http://blogs.msdn.com/b/sqlazure/archive/2010/05/11/10011247.aspx**

# 4.- ADVANCED SCENARIO: HIGH SCALABLE APPLICATIONS IN CLOUD-COMPUTING

This scenario will require changes in our logical architecture, defining new architecture and design patterns for applications that need high scalability requirements (such as the *CQRS pattern– Command and Query Responsibility Segregation* and other patterns) and sometimes implementing them with native/exclusive technologies of our selected Cloud-Computing platform.

High scalability requirements may also have an impact on persistence and data access technologies and even on the data sources themselves.

## 4.1.- Logical Architecture (Advanced Scenario in the Cloud)

When using the cloud due to high scalability requirements, the logical architecture of our application will most likely be affected, since it is highly recommended that you use certain Architecture and Design patterns that will favor such high scalability.

The patterns and architectures we recommend to evaluate and implement these advanced scenarios are the following:

*CQRS – Command and Query Responsibility Segregation*
*Event Sourcing and Event Stores*
*EDA – Event Driven Architecture*

Before covering this section in detail, it is important to point out that the following patterns are not exclusive for Cloud applications. In fact, its original definition has been identified and defined independently, even before the cloud computing phenomenon came about. However, these architecture and design patterns fit very well with the cloud-computing objectives, especially with high scalability requirements.

www.EBooksWorld.ir

However, take into account that these patterns can also be implemented and used for '*On-Premise*' applications.

## 4.2.- CQRS Pattern (*Command and Query Responsibility Segregation*)

The **CQRS** pattern (*Command and Query Responsibility Segregation)* actually extends and structures the idea mentioned previously (Chapter 3: Dual Data Access).

CQRS is a way of designing business application architectures that require:

- Massive scalability.
- A focus on business and not on technology.
- The ability to grow and manage complex scalability problems without increasing development costs.
- The ability to adapt to changing business requirements
- A good fit in Cloud-Computing

One of the best exponents of this pattern is ***Greg Young***, who stated the following phrase:

*"A single model cannot be suitable to perform reports, searches and transactional behaviors".*

The main concept proposed by CQRS is the concept of separation between query tasks and transactional/persistence tasks (edition/insertion/deletion, etc.).

The following diagram shows the CQRS pattern in its simplest form:

www.EBooksWorld.ir

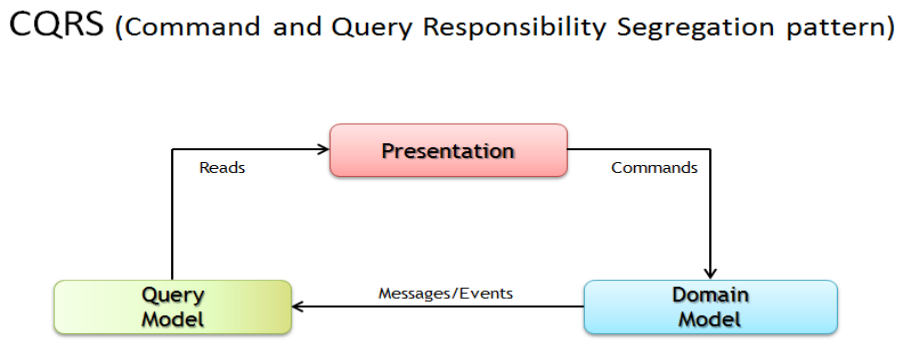



**Figure 38.- Simplified CQRS diagram**

CQRS proposes the use of commands, implemented as **messages** as a triggering mechanism and way to start Domain processes and "Data Writes" actions. These operations are closely related to business and transaction logic. The idea is to separate and locate the Domain and persistence actions in a specific storage specially made for updates (normally a transactional storage) and having another data storage and model dedicated only to queries/reads.

CQRS is an approach that provides solutions to the following typical problems in application development:

- Scalability and bottlenecks in performance.
- Concurrency conflicts, resolution and prevention.
- Design complexity, development and maintainability.

A key point is that CQRS applications must accept the *Stale data* ('*staleness*') concept in order to be able to have separate read & write data sources.

This is the summary of the **fundamental CQRS subjects:**

- All state changes are articulated with Messages.
- Application services accept **Commands** from the UI and publish **Messages/Events.**
- Data source for queries (Reads, Reports, etc.) are updated as a result of the Events publication.
- All the Queries requested by the Presentation layer go directly against the Queries system. The domain is not involved in any way, in this case.

www.EBooksWorld.ir

This is a simple example.

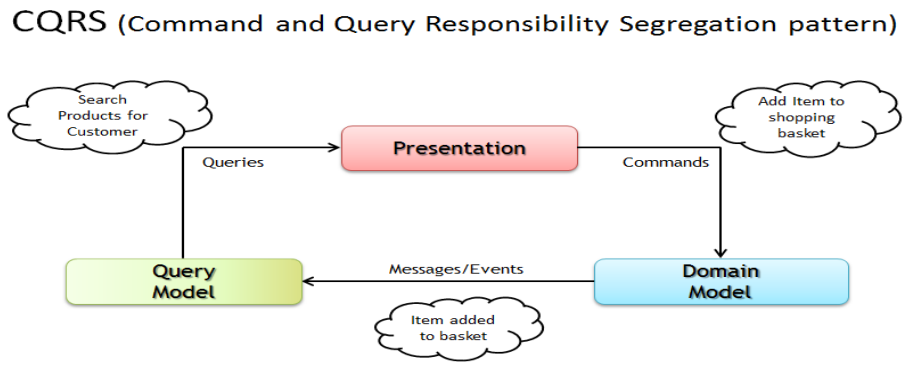



**Figure 39.- Use Case sample implemented following the CQRS pattern**

As regards ***Commands*** management, we could finally store actions in a regular transactional database with a regular database schema, or going to a more advanced system we could also be using '*Event-Sourcing*'. However, the CQRS pattern is actually independent from '*Event-Sourcing*'.

A more detailed CQRS diagram is the following:

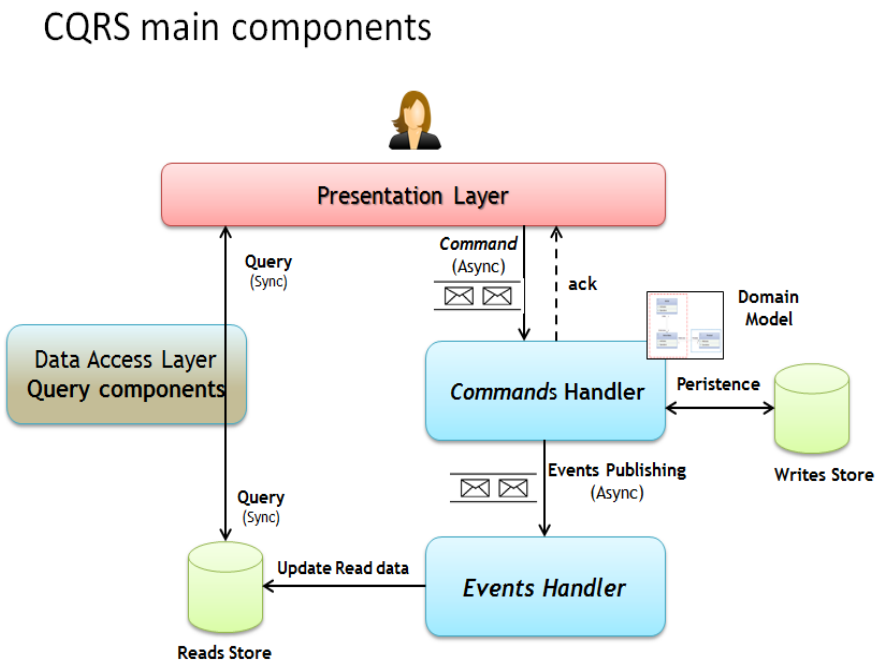



**Figure 40.- CQRS main components**

www.EBooksWorld.ir

Of course, in the diagram above the remote invocation mechanisms and other intermediate components that may be necessary do not appear. We only show essential components of the idea.

Again, how the commands are processed is a detail of the final design and implementation. They may be implemented with events or messages (message queues from *MSMQ* or *Azure-Storage-Queues*), etc.

**CQRS** fits great for **high scalability scenarios** that are closely related to '*Cloud-Computing*', where decoupling between a transactional storage and query storage can be an advantage and where even the nature of the transactional storage may be very different from the nature of the storage dedicated to queries only.

## 4.2.1.- CQRS Essential Benefits

Finally, we summarize the **CQRS essential benefits:**

- 'Domain isolation': This is completely encapsulated and only exposes behaviors
- Queries do not use the 'Domain Model'
- No impedance mismatch between objects and relational data
- Perfect system for historical traceability and audits
- Easy integration with external systems
- Good Performance and especially **high scalability**

www.EBooksWorld.ir

APPENDIX A

# Conclusions

In the pages above we have attempted to define the main challenges that companies face in the development of complex and long-term business applications. Although this book presents the primary difficulties of these types of developments and offers their corresponding solutions, there are many aspects that could have been delved into more thoroughly. This, however, would have engendered another book of approximately the same size. We therefore recommend that the reader use this book as reference for the construction of business applications and suggest that he expand his knowledge of the contents herein with other specialized publications, such as '*Domain-Driven Design Tackling Complexity in the heart of Software*' by Eric Evans.

It is our firm belief that the development of high-quality software requires thorough knowledge of the construction system domain. We encourage the reader to modify his *modus operandi* and focus on the construction of good domain models and to use the technologies developed by Microsoft to facilitate system construction. The value of the software resides in the domain that is constructed, which allows us to satisfy the needs of specific clients. The technology exists to facilitate the development of applications based on the domain models we construct, not to obstruct said development. This is one of the points we have wanted to make as clear as possible.

We also want to make it clear that the future of IT is found in the cloud, where we will be facing a series of new problems. We encourage the reader to do research on '*Event Sourcing'* and '*Command and Query Responsibility Segregation'*, and to explore this set of models, which are ideal for high scalability scenarios that fit perfectly into '*Cloud Computing*' and Microsoft *PaaS*, that is, *Windows Azure*.

Finally, we would like to thank all those readers that have followed the development of this project (which we will continue to work on, since it is ongoing and far from finished), as well as the collaborators and developers who have used our



www.EBooksWorld.ir

sample application in CODEPLEX (*NlayerApp*) as a model and reference for their own applications or products. There is no doubt that your contributions, whether in the form of questions or feedback, have led us to adopt a fresh approach on many occasions and, in sum, to improve the quality of our proposal. We are aware that it is very difficult to be completely in agreement on certain issues, especially when dealing with software architecture. However, we hope this publication has enriched the perspective of a significant number of software architects and developers as regards application design and development.

Sincerely,

"The team". ☺

www.EBooksWorld.ir


www.EBooksWorld.ir