



CHAPTER

5

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-Layered 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 which is within a specific BOUNDED CONTEXT), and also to the data exposed outside the boundaries of our system, such as Web services of external systems. Therefore, it has components like "*Repositories*" that provide such functionality to access the data hosted within the boundaries of our system, or "Service Agents" that will consume Web 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:

DDD N-Layered Architecture

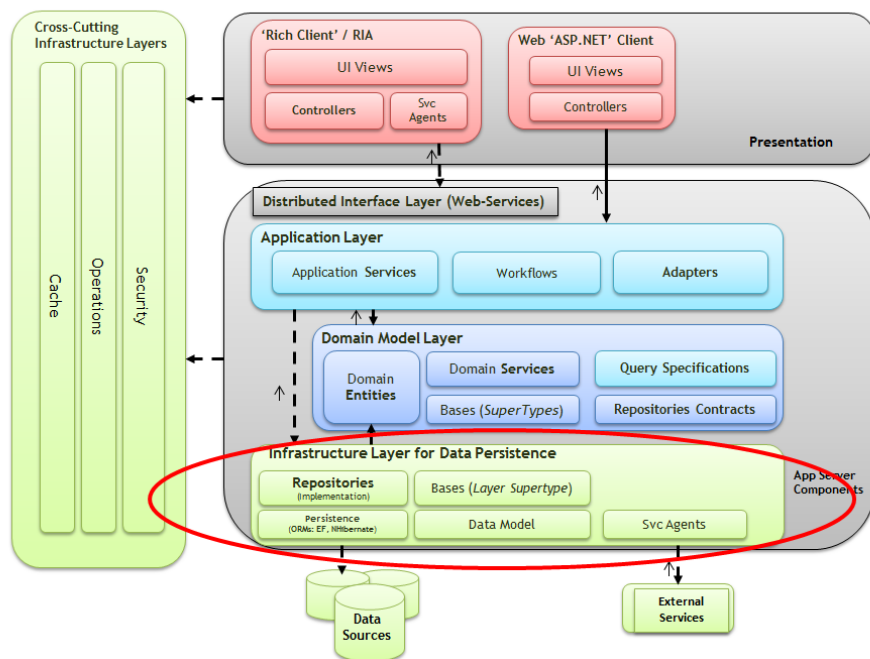


Figure 1.- Data Persistence Layer location within the N-Layered Architecture



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.



2.1.1.- Repositories (Repository Pattern)

In some aspects, these components are very similar to the *Data Access Layer* (DAL) components of traditional N-Layer architecture. But, in many other aspects, a Repository is quite different to a DAL class. 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 highly 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 (DAL Classes 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 update operations it intends to perform against the storage (but these updates will not be performed immediately). These persistence operations will be really performed at a later time, commanded 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 usually does not group update actions against the storage. Therefore, the selected O/RM will provide 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 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* had 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 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 each AGGREGATE (in DDD that means we have to create one Repository for each main root ENTITY in our domain). The AGGREGATE pattern will be explained later on, but, to simplify it for now, we will not create a Repository for each table in a data source.

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

Table 1.- Repository Rule

 Rule #: D?.	Design and implement Repository classes within the data persistence layer
<p> <u>Rules</u></p> <ul style="list-style-type: none"> - To encapsulate the data persistence logic, you should design and implement Repository classes. Repositories are easily implemented over O/RMs. <p> <u>Advantages of using Repositories</u></p> <ul style="list-style-type: none"> → 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 objects life cycle.

- It de-couples the APPLICATION and DOMAIN layers from the persistence technology, multiple-database strategies, or even multiple data sources.
- They can be easily replaced by fake data access implementations which are particularly useful when *testing* (especially unit-testing) 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 unit tests.



References

- ‘Repository’ Pattern. By Martin Fowler.
<http://martinfowler.com/eaCatalog/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’ (called *AGGREGATE 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 Aggregates (1:1)

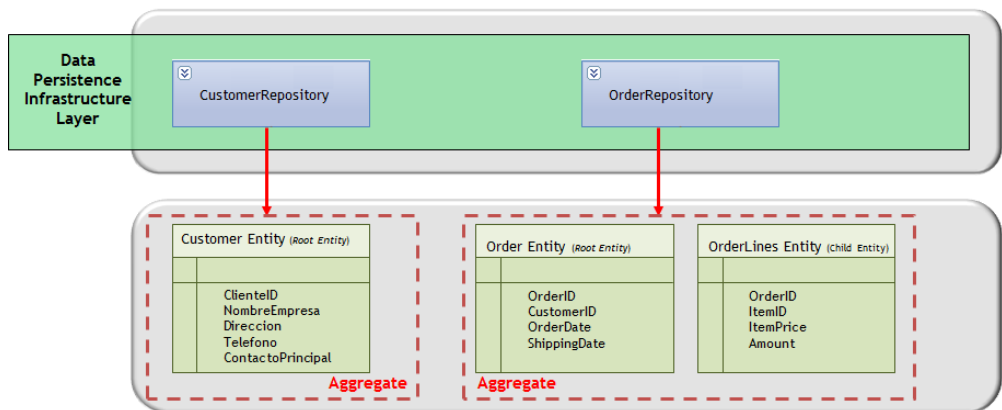


Figure 2.- Relationship between Repository Classes and Entity Classes

For example, in the diagram above, the ‘Order’ entity is the entity-root for the Order and OrderLines entities aggregation.

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




 Rule #: D?.	Repository classes (data access and persistence classes) as a unique channel for data storage access
<p>○ <u>Rule</u></p> <ul style="list-style-type: none">- In a project with a DDD architecture, the only interlocutors with data storages (typically relational databases or other type of storages) will be the Repositories. <p>This does not mean that in systems outside the <i>Domain Oriented</i> architecture, there cannot be parallel access to these storages through other paths.</p> <p>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 when accessing data could be CQRS architectures for very high scalable applications (Read the last part of the chapter 10 for a CQRS introduction).</p>	

Table 3.- Layer Supertype Rule

 Rule #: D?.	Implement a “Layer Supertype” pattern for the Repositories Sub-layer
<p>○ <u>Recommendations</u></p> <ul style="list-style-type: none">- 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 “<i>Layer SuperType</i>”.- It is particularly useful when we have similar data access code for different domain entities. <p> References</p> <p><i>‘Layer Supertype’ Pattern by Martin Fowler.</i> http://martinfowler.com/eaCatalog/layerSupertype.html</p>	

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 is previously 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 / Data Mapping

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 a visual entity data model or even to create it from an existing database). Within that ‘visual data model’, the ORM will have all the mappings in order to relate our Domain Entity Model to our database tables.

If we don’t have such a visual ‘data model’, like when using *EF 4.1 Code First approach* or when using *NHibernate*, we still will need an area where we will have to place code related to our **mappings from our Domain Entity Model to our physical database model**.



2.1.3.- Persistence Technology (ORM, etc.)

The repository and Unit-Of-Work 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” in the second part of this chapter.



2.1.4.- UNIT OF WORK pattern

The UoW (Unit of Work) pattern is mostly used/consumed from the Application layer Services, but, it is usually implemented and defined within the Data Persistence Infrastructure Layer, therefore we have to make a review about the UoW pattern within this chapter.

The UNIT OF WORK pattern concept may be closely linked to the use of REPOSITORIES. Although some people use UoW with no associated Repositories, our approach consists on using a UoW as coordinator of Repositories' updates.

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 'in memory' (Context) the operations it "wants to perform". Therefore, in order effectively perform those changes into 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, 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.

Usually, when using an O/RM, our UoW would be simply the O/RM Context/Session. Simplifying, it would be as simple as defining our UoW interface wrapping the O/RM context/session implementation. We will check how to do this at the second part of this chapter when explaining the .NET implementation for the Persistence Infrastructure Layer.



<http://martinfowler.com/eaCatalog/unitOfWork.html>

The following scheme shows the operation of traditional data access classes (DAL), **without using any UoW**:

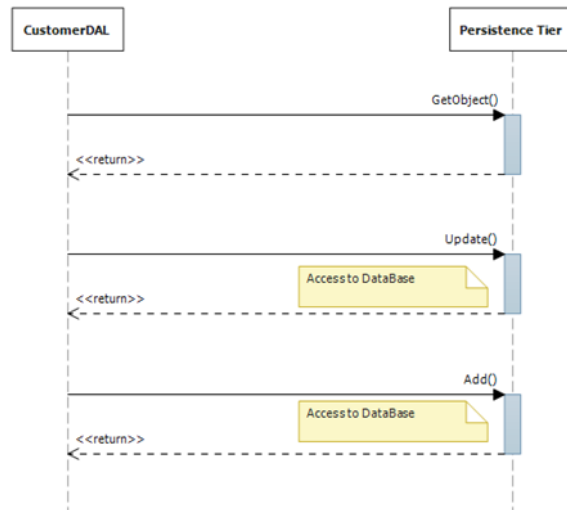


Figure 4.- Scheme of data access classes (DAL)

The following scheme shows the operation of a REPOSITORY class along with a UoW, which is what we recommend in this Architecture guide:

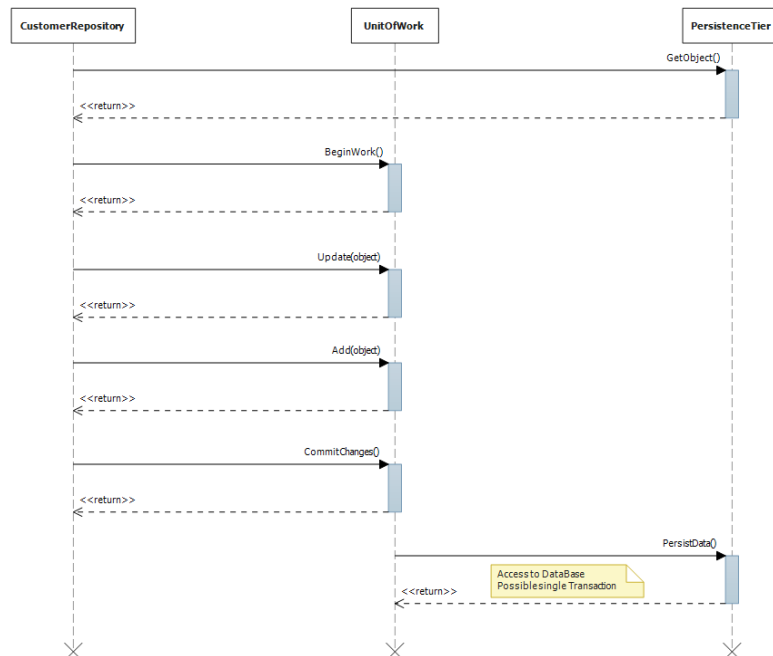


Figure 5.- Operation of UoW and REPOSITORY classes



2.1.5.- Service Agents for External Services

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.

Service Agents are also very useful for mocking the external web services consumption so unit testing can be run in an isolated way regarding external dependencies.



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 if used in a wrong way. 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 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.



2.2.2.- Table Data Gateway

This pattern, initially 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.



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
<ul style="list-style-type: none">• Active Record• Data Mapper• Query Object• Repository• Row Data Gateway• Table Data Gateway• 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/eaCatalog/>*



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 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.



Rule #: D?.

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 app-settings) 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/>



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.

- ‘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:
- ORM tools may allow design of an entity-relation model and generate a real database schema (this approach is called ‘*Model First*’ in EF) 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.

- In general, whether stored procedures are used or not largely depends 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.

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 – En <http://msdn.microsoft.com>



5.- IMPLEMENTING DATA PERSISTENCE LAYER WITH .NET 4.0 AND ENTITY FRAMEWORK 4.1

The explanation and logical definition of this layer (design and patterns) is given in the first half of this chapter. Therefore, the purpose of this section 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':

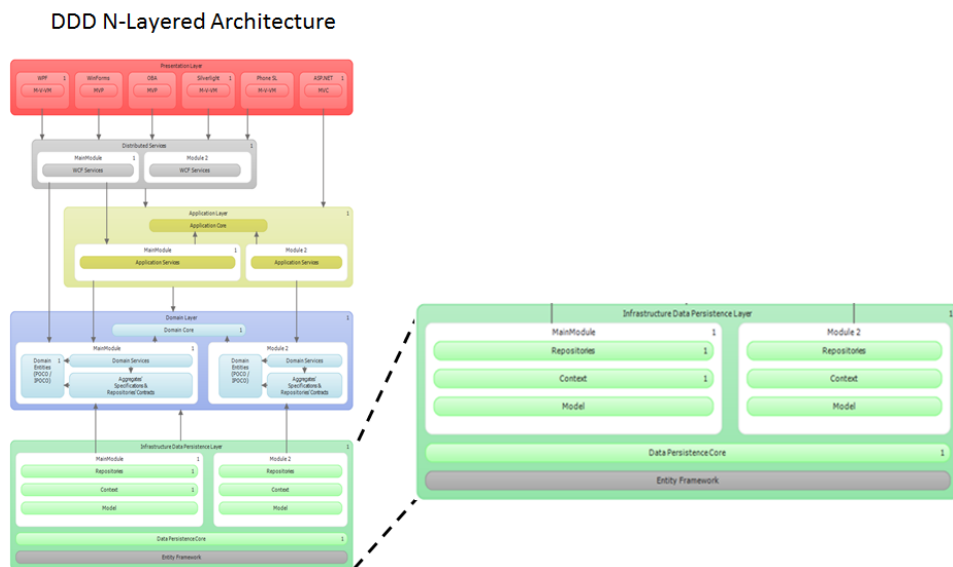


Figure 3.- Data Persistence Layer Diagram in VS2010

Steps to be followed:

- 1.- The first step will be to identify the type 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. 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 and their mapping to persistence should be taken at this point (Persistence Layer implementation).

- 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. The main 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 (reusable services or existing Data Access Building Blocks) implemented using ADO.NET, naturally.
- **Microsoft P&P Enterprise Library Data Access Application 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 '*Application Block*' is a technology older than EF. The 'Microsoft P&P' product team itself recommends EF, if possible, instead of this library.

- **Microsoft 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 (O/RMs like NHibernate, etc.) which are not provided and supported by Microsoft but they also help with a DDD approach.



5.1.2.- Other technical considerations

- If low level support is required for queries and parameters, use the plain ADO.NET objects, but even then, you could implement Repository patterns.
- 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**.

Table 5.- Data persistence layer default technology



Rule #: I?.

When using relational databases, the preferred default technology for implementing Repositories and Data Persistence Layer should be an O/RM such as Microsoft 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 is a much more straightforward and easier approach 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 be 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..



References

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



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.
- A common pattern associated to DDD is modeling domain entities with domain classes/objects. This has been logically explained in the Domain Layer chapter.
- 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*, but we can implement it in our own code.
- When working with Web applications or Web Services, sometimes entities should be grouped and transformed to DTOs (Data Transfer Objects) so you can return just the required data 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, is **Microsoft Entity Framework**, but specifically, we are going to use **EF 4.1**.



5.2.1.- What does Entity Framework 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 visual 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 and 4.1 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 and 4.1 offers the possibility of implementing our Domain Entities as POCO Entities, STEs (*Self Tracking Entities*) or *POCO Code-First 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, T4 POCO or STE templates or 'POCO Code-First approach'). This analysis is explained in the Domain Layer chapter, so we recommend to read that chapter first and to learn about the pros and cons of each type of possible EF entity type before moving forward in the current chapter.

In our case, we selected the '*POCO Code-First approach*' available since EF 4.1, because it fits better with DDD architecture and design patterns, as explained .



5.3.- What is new in EF 4.1

EF 4.1 includes includes two new main features:

- **DbContext API**
- **Code-First**

We already explained most of the features regarding to 'Code-First' in our 'Domain Model Layer' chapter, as it is very related to the Domain Entities. On the other hand, the **DbContext** API is a simplified abstraction over the existing **ObjectContext** type.

5.3.1.- DbContext API

The **DbContext** API is mostly related to data infrastructure topics like database connections and mappings from domain entities towards database tables. The DbContext API surface is optimized for common tasks and coding patterns. Common functionality is exposed at the root level and more advanced functionality is available as you drill down through the API.

DbContext is really a UNIT OF WORK implementation, a context for the database operations.

DbContext could be defined as a light version of EF ObjectContext, and it allows us to work in an easier way through its '*Configuration*' and '*Database*' properties. In fact, DbContext is really an ObjectContext adapter declared by the IObjectContextAdapter.

```
public class DbContext : IDisposable, IObjectContextAdapter
{
    //Omitted
}
public interface IObjectContextAdapter
{
    ObjectContext ObjectContext { get; }
}
```

As previously stated, having any domain entity defined in our code (like the Customer or Order classes we defined in the previous chapter), we could create a simple EF DbContext and run the following code and it simply would work. No need to create the database-tables or elaborate any kind of pre-established mapping.

```
// EF DbContext
// This code is still an isolated example, it is not part of the SampleApp
public class MyModelContext : DbContext
{
    public IDbSet<Customer> Customers { get; set; }
    public IDbSet<BankAccount> BankAccount { get; set; }
}
```

DbContext

Figure 18.- Simple DbContext

That is all the code we need to write in order to start storing and retrieving data. Obviously there is quite a bit going on behind the scenes and we will take a look at that in the following sections. Also, in our DDD Architecture we won't use a simple DbContext but we will extract a 'Unit of Work' interface and use Dependency Injection, etc.

What this means is that code first will assume that your classes follow the default conventions of the schema that EF uses for a conceptual model. In that case, EF will be able to work out the details it needs to do its job. The following code will simply work. (Take into account that we are executing this code from a 'main' console app, just as an example. Usually this kind of code would be within the Application and Domain layers).

```
// Using 'Code First'- Simple isolated example. It is not part of the SampleApp
class Program
{
    static void Main(string[] args)
    {
        using (var db = new MyModelContext())
        {
            // Add a Customer
            var customer = new Customer {CustomerId = "ALFKI", Name = "Joe Smith"};
            db.Customers.Add(customer);
            int recordsAffected = db.SaveChanges();

            Console.WriteLine(
                "Saved {0} entities to the database, press any key to exit.",
                recordsAffected);

            Console.ReadKey();
        }
    }
}
```

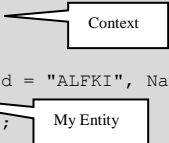


Figure 19.- Using 'Code-First' based on 'default conventions'

It is important to highlight that we have started working with entities and an EF DbContext with default conventions, so, even with no explicit/manual database creation and connection string definition, it is working. As we explained in the previous chapter, this is because we are using default conventions in order to access the database, but we are able to change most of those conventions using either Fluent API or Data annotations.

Even though in our selected approach we will be using 'Code-First', DbContext can also be used on a 'Model First' and 'Database First' approach, even using a visual .EDM model.



5.4.- Database Connections

In the former example data access works even when we did not specify any specific database connection. This is because, by default, EF Code First uses a default connection based on SQL Server Express and a name of catalog which will be the same as our context class. In this case, the default connection string really is the following:

```
"Server=.\SQLEXPRESS;Initial Catalog=[Namespace].MainModuleUnitOfWork; Integrated Security=true"
```

This default behavior is defined by a new class named **Database**, which is available within the namespace **System.Data.Entity**.

```
public class Database
{
    public static IDbConnectionFactory DefaultConnectionFactory { get; set; }

    // Ommited
}
```

This class has a static property named **DefaultConnectionFactory** which can be used to specify our connection creation factory from a name or connection string.

```
public interface IDbConnectionFactory
{
    DbConnection CreateConnection(string nameOrConnectionString);
}
```



5.4.1.- Modifying Connection parameters

In many cases, we would need to modify and customize our connection strings. In order to do that, we have several choices, depending on what we want to achieve. First of all, we can see how we can modify the catalog name we want to connect to. To do so, we only need to use one of the constructors in our Unit of work. The following code is an example of that.

```
public class MyModelContext
    : DbContext
{
    public MyModelContext()
        : base("NLayerApp")
    {
    }

    public IDbSet<Customer> Customers { get; set; }
    public IDbSet<Order> Orders { get; set; }
}
```

In the constructor parameter we could also have specified a whole connection string, like this other example.

```
public class MyModelContext
    : DbContext
{
    public MyModelContext()
        : base("Server=.;Initial Catalog= NLayerApp; Use=user;
```

```

        Password=password")
    {
    }

    public IDbSet<Customer> Customers { get; set; }
    public IDbSet<Order> Orders { get; set; }
}

```

Finally, we could also use our XML ‘Configuration Settings’ using the connection string section. The only requirement is to set as connection string name the whole qualified path of our working context.

```

<?xml version="1.0"?>
<configuration>
  <connectionStrings>
    <add name="[Namespace].MyModelContext"
    connectionString="Server=.\SQLEXPRESS;Initial
    Catalog=NLayerApp;Integrated Security=true"
    providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>

```



5.5.- Changing Conventions and Mappings

We saw how our domain entities definitions (like Customer and Order) can be used to generate a related database schema. That is based on the concept of ‘Conventions’ which we already mentioned and explained in our ‘Domain Model Layer’ chapter. Those conventions are things like what is going to be the name of a table, or what property is going to be the primary key, etc., and usually they are based on convention names (like ‘a property ending with the prefix ‘Id’ will be the primary key’).

Therefore, there are many EF Code-First conventions and most of them can be changed either using ‘*Data Annotations*’ or ‘*Fluent API*’. ‘*Data Annotations*’ must be used on the entity model itself (this is why we mentioned that it is a more intrusive way). On the other hand, ‘*Fluent API*’ is a way we can choose to change most conventions and mappings within our ‘Data Persistence Infrastructure Layer’, so the Entity Model will be cleaner.

In order to change conventions and mappings, we can use the method ‘**OnModelCreating()**’ from the **DbContext** class. In our Domain Model Layer we already explained it a little bit, but because of that method is part of our data persistence infrastructure method, we are going to explain it in a deeper detail. Next code shows that specific method.

```

public class MyModelContext
    : DbContext
{
    // Code Omitted
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
    }
}

```

As we can see, this method has the **DbModelBuilder** parameter which represents the main mapping artifact. In fact, this class has the ‘**Conventions**’ property which we can use in order to **customize** any convention or even **eliminate** any convention, using the property *ConventionsConfiguration*.

```
public class DbModelBuilder
{
    public virtual ConventionsConfiguration Conventions { get; }
}
```

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
}
```



5.6.- Mappings in the Data Infrastructure Layer

In the ‘Domain Model Layer’ chapter we mentioned that we can use either ‘Data Annotations’ or ‘Fluent API’ to customize our mappings from our Domain entity model to our specific database infrastructure. In the current layer we are analyzing, we would use the ‘**Fluent API**’ for this matter.

Note: In software engineering, a **fluent interface** (as first coined by Eric Evans and **Martin Fowler**) is a way of implementing an object oriented API in a way that aims to provide for more readable code.

A fluent interface is normally implemented by using method chaining to relay the instruction context of a subsequent call (but a fluent interface entails more than just method chaining). Generally, the context is:

- Defined through the return value of a called method
- Self-referential, where the new context is equivalent to the last context
- Terminated through the return of a void context.

This style is marginally beneficial in readability due to its ability to provide a more fluid feel to the code[citation needed] however can be highly detrimental to debugging, as a fluent chain constitutes a single statement, in which debuggers may not allow setting up intermediate breakpoints for instance.

Source: Wikipedia - http://en.wikipedia.org/wiki/Fluent_interface

For instance, we can use ‘Fluent API’ to specify the primary key of any of our domain entities.

```
public class MainBCUnitOfWork
    : DbContext, IMainBCUnitOfWork
{
    public IDbSet<Customer> Customers { get; set; }
    //Omitted Code (Other Entity sets, etc.)
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
```

```

{
    modelBuilder.Entity<Customer>()
        .HasKey(c => c.CustomerId);

    modelBuilder.Entity<Customer>()
        .Property(c => c.CustomerId)
        .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
}

```

Example modification:
PrimaryKey specification



5.6.1.- Modifying Database Table and column attributes

In many cases, we would need to modify our domain entity mappings so they fit to different table names. For that matter we can use the methods **ToTable()** and **HasColumnName()**.

```

public class MainBCUnitOfWork
    : DbContext, IMainBCUnitOfWork
{
    public IDbSet<Customer> Customers { get; set; }
    //Ommitted Code (Other Entity sets, etc.)

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Customer>()
            .ToTable("tblCustomers");

        modelBuilder.Entity<Customer>()
            .Property(c => c.CustomerId)
            .HasColumnName("customer_code");

        modelBuilder.Entity<Customer>()
            .Property(c => c.FirstName)
            .HasColumnName("GivenName");

        modelBuilder.Entity<Customer>()
            .Property(c => c.LastName)
            .HasColumnName("Surname");
    }
}

```

Specifying a custom table name

Specifying custom column names

We can also set specific data types and requirements using the methods **HasColumnType()**, **IsRequired()** and **IsOptional()**.

```

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
        .HasKey(c => c.CustomerId);

    modelBuilder.Entity<Customer>()
        .Property(c => c.Age)
        .HasColumnType("bigint")
        .IsRequired();

    modelBuilder.Entity<Customer>()
        .Property(c => c.LastName)
        .IsOptional();
}

```




5.7.- Concurrency management

When using the traditional EF EDM we were used to the *ConcurrencyMode.Fixed* property. Now, using Code-First approach, we can use the 'ConcurrencyCheck' attribute (using 'Data annotations') or the **IsConcurrencyToken()** method when using Fluent API, like shown below.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
        .Property(c => c.FirstName)
        .IsConcurrencyToken(true);

    modelBuilder.Entity<Customer>()
        .Property(c => c.LastName)
        .IsConcurrencyToken(true);
}
```

Concurrency management



5.8.- Improving and structuring ‘OnModelCreating’ code with ‘TypeConfiguration’ classes

As the reader might have thought, adding so many customizations to the ‘OnModelCreating()’ method when evolving towards a large project could turn our customization code in a ver long and crazy method.

In order to structure this code in a convenient way, we can use the ‘TypeConfiguration’ classes, either for entities (using **EntityTypeConfiguration**) or for Complex-types and value-objects (using **ComplexTypeConfiguration**).

Using this more structured way we could, for instance, create a type configuration class (*CustomerEntityTypeConfiguration* class) for the Customer entity, like shown below.

```
class CustomerEntityTypeConfiguration
    :EntityTypeConfiguration<Customer>
{
    public CustomerEntityTypeConfiguration()
    {
        this.HasKey(p => p.CustomerId);

        this.Property(p => p.FirstName)
            .HasMaxLength(400);
    }
}
```

Our CustomerEntityTypeConfiguration class has to derive from EntityTypeConfiguration<>

Specific entity mapping customization

Once we have defined our ‘EntityTypeConfiguration’ class, we have to apply it into the ModelBuilder configuration, adding it to the configuration list.

```
public class MainBCUnitOfWork
    :DbContext, IMainBCUnitOfWork
{
    public IDbSet<Customer> Customers { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new CustomerEntityTypeConfiguration());
    }
}
```

Adding our CustomerEntityTypeConfiguration to the Configuration list

Doing so we could have a list of many configuration classes added to the list, but the code would be much better structured and readable.

For a ‘Complex-type’ and Value-Object customization example using a **ComplexTypeConfiguration** you can look for it in the ‘domain Model Layer’ chapter where we previously showed code about it.



5.9.- Entity Validations using 'Fluent API'

In the 'Domain Model' Chapter, we already analyzed how to implement Entity-Validations using *Data-Annotations*, although we also stated that many data-annotation attributes are quite intrusive against the domain model, especially when those attributes are defined within EF DLLs so in that case we'd need to have a direct dependency from the Domain Model to the EF DLL. That is not good. This is why, in many cases, implementing entity validations using 'Fluent API' within the persistence-infrastructure layer is a better approach regarding DDD and Domain Model isolation from infrastructure assets like EF. The other possibility is doing validations within the Domain Layer but using basing our validations on the 'IValidatableObject' interface, as we do in most of our domain entities in our SampleApp.

Typical examples of entity model validations are 'properties *max-length*', *nullables*, required properties, etc.

For instance, in the next example we implement code on how to set *required* properties and *maxlength*, within the mapping code (infrastructure layer).

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
        .Property(p => p.FirstName)
        .IsRequired()
        .HasMaxLength(10);

    modelBuilder.Entity<Customer>()
        .Property(p => p.LastName)
        .IsRequired()
        .HasMaxLength(20);
}
```

If we try to change any of those properties and we do not comply with those validations, we would get a `DbEntityValidationException` which has an `EntityValidationErrors` collection thrown by our Unit of Work (EF Context) `SaveChanges()` method.

If we want to catch and manage those errors, we should manage the former exception and flatten the possible multiple errors we could get, like in the following simple example.

```
try
{
    //Ommitted
    //...
    unitOfWork.Commit();
}
catch (DbEntityValidationException ex)
{
    DumpErrors(ex);
}
```

Once we have caught the exception, we need to flatten all the errors. Then we could publish them, store them or simply show them like in the following example.

```
private static void DumpErrors(DbEntityValidationException ex)
{
    foreach (var item in ex.EntityValidationErrors)
    {
        Console.WriteLine("Validation error for entity with this state:{0}",
            item.Entry.State);

        Console.WriteLine("Validation errors are the following:");

        foreach (var ve in item.ValidationErrors)
        {
            Console.WriteLine("\tProperty:{0} Error:{1}",
                ve.PropertyName,
                ve.ErrorMessage);
        }
    }
}
```

Additionally, if we don't want to wait until the EF context (UoW) is trying to save the data, we can check these validations 'on-demand'. We can do so in two possible ways.

1. Invoking the `GetValidationErrors()` method, part of the EF context.

```
var validationErrors = unitOfWork.GetValidationErrors();
```

2. Validating entity by entity, accessing the `DbEntityEntry` for each entity and invoking the `GetValidationResult()` method.

```
var vresult = unitOfWork.Entry(customer)
    .GetValidationResult();
```



5.10.- Entity Relationships using EF 4.1 Code-First

EF 4.1 and Code-First support all the possible relationships and cardinality that were supported in EF 4.0 EDM. From 0,1 cardinality up to * or even single-directional and bi-directional associations, and of course, independent associations and foreign key associations.

5.10.1.- One to One relationships

If we have two entities, like **Customer** and **RewardsInfo** and we want to relate them to each other, we can establish a relationship either using '*Data annotations*' or '*Fluent API*'.

Like previously said, '*Fluent API*' is less intrusive regarding our Domain entities code, but because of that, we are going to compare both ways.

If using '*Data annotations*', we can establish entity associations using the **ForeignKeyAttribute** in a way similar to the following code. Afterwards, we will see why we don't like this way of establishing relationships.

```
public class Customer : Entity
{
    //...
    //Ommitted
    //...
    public Guid CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public decimal CreditLimit { get; set; }
    public Guid CountryId { get; set; }
    public virtual Country Country { get; private set; }
    public virtual Address Address { get; set; }
    public virtual Picture Picture { get; set; }

    //Methods with Entity logic
    //...
    //Ommitted
    //...
}

public class RewardsInfo
{
    //...
    //Ommitted
    //...
    [ForeignKey("Customer")]
    public Guid RewardsInfoId { get; set; }
    public bool IsPremiumCustomer { get; set; }

    public virtual Customer Customer { get; private set; }

    //Methods with Entity logic
    //...
    //Ommitted
}
```

Relationship established using the attribute ForeignKeyAttribute which is defined within the EntityFramework.dll assembly!!! This is against PI principle!!

Set as **virtual** so we facilitate the Dynamic Proxies generation

But, as mentioned, this way (*Data Annotation*) is quite intrusive, especially when this specific attribute (**ForeignKeyAttribute**) is defined within the **EntityFramework.dll**, so we wouldn't be complying with the '*Persistence Ignorant principle*', as we would need to add a direct reference to the EF assembly from our Domain Layer where we have defined our domain entities.

But there is a solution, we have '*Fluent API at rescue!*'. We have a few methods for configuring associations, like shown in the table.

Method	Description
HasMany()	It allows to configure a one to many relationship (1..*)
HasOptional()	It allows to configure a relationship to an optional element (1..0,1).
HasRequired()	It allows to configure a one to one relationship (1..1)

Regarding our former example, we could write the following Fluent-API code.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
        .HasOptional(c => c.RewardsInfo)
        .WithRequired(c => c.Customer);
}
```

There are many possible combinations using those methods, therefore we recommend checking them out on the following URLs:


EF 4.1 - Managing Relationships

[http://msdn.microsoft.com/en-us/library/hh295848\(v=VS.103\).aspx](http://msdn.microsoft.com/en-us/library/hh295848(v=VS.103).aspx)

EntityTypeConfiguration Class information:

[http://msdn.microsoft.com/en-us/library/gg696117\(v=vs.103\).aspx](http://msdn.microsoft.com/en-us/library/gg696117(v=vs.103).aspx)

Table X.- Fluent API is preferred for establishing entity relationships

 Rule #: I?.	<p><i>The preferred way to establish Domain Entity relationships is 'Fluent API' because it complies with the PI principle .</i></p>
<p>○ <u>Rule</u></p>	<ul style="list-style-type: none"> - Following the <i>Persistence Ignorant principle</i>, our Domain Model Layer should not have any direct reference to Entity Framework assemblies, therefore we should not use any 'Data Annotation' with attributes defined within <i>EntityFramework</i> assemblies, but on the contrary, we should better use 'Fluent API' in our Data Persistence Infrastructure Layer to specify all

of our entity relationships.



References

<http://thinkddd.com/glossary/persistence-ignorance/>

5.10.2.- One to Many relationships

This is a very typical scenario, like the relationship between **Customer** and **Order**. The following code shows a simple scenario with an implicit one to much relationship.

```
public class Company : Entity
{
    public Guid    CompanyId { get; set; }
    public string  Name { get; set; }
    public string  CompanyType { get; set; }
    //...
    //Ommitted
    //...

    public virtual ICollection<Customer> Customers { get; set; }
}

public class Customer
{
    public Guid    CustomerId { get; set; }
    public string  FirstName { get; set; }
    public string  LastName { get; set; }
    //...
    //Ommitted
    //...

    //Methods with Entity logic
    //...
    //Ommitted
    //...
}
```

Relationship established implicitly

In this case the association is established even when there are not any explicit foreign key and the association in this case is not bidirectional. But, underneath, in the database tables, we would have a *CompanyId* foreign key within the Customers table.

In case we want a bi-directional relationship, we would update the **Customer** class like shown below.

```
public class Customer
{
    public Guid    CustomerId { get; set; }
    public string  FirstName { get; set; }
    public string  LastName { get; set; }
    //...
    //Ommitted
    //...
```

```

    public virtual Company Company { get; set; }
}

```

Finally, in case we need to explicitly specify which property is our foreign key, we can also do it.

```

public class Customer
{
    public Guid    CustomerId { get; set; }
    public string  FirstName { get; set; }
    public string  LastName { get; set; }
    //...
    //Omitted
    //...

    public Guid    CompanyCode { get; set; }

    [ForeignKey("CompanyCode")]
    public virtual Company Company { get; set; }
}

```

Again relationship established using the attribute `ForeignKeyAttribute` which is defined within the `EntityFramework.dll` assembly!!! **This is against P1 principle!!**

But, again, if we use that kind of attributes, we will need to add a direct reference to EF 4.1 assembly from our Domain-Model Layer. So if you need to customize your relationships, 'Fluent API' would be a better way, like shown in next code.

```

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Company>()
        .HasMany(c => c.Customers)
        .WithRequired(m => m.FirstName)
        .HasForeignKey(m => m.CompanyCode)
        .WillCascadeOnDelete(false);
}

```


5.10.3.- Many to Many relationships

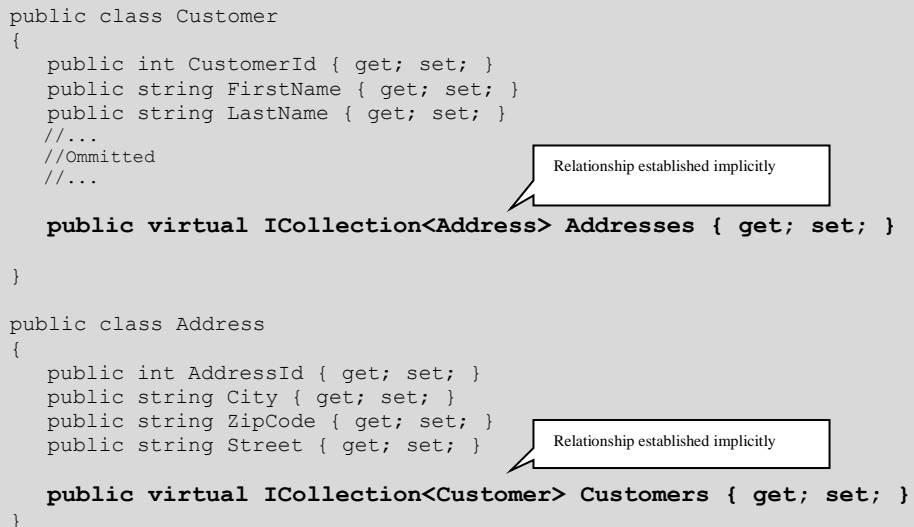
In this scenario is when a O/RM gets powerful and clean, as the developer does not have to deal with tables in between for relations (even though in database they will be), but from an object oriented point of view, we will deal only with associations. Here we can see a simple many to many association case.

```
public class Customer
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    //...
    //Ommitted
    //...

    public virtual ICollection<Address> Addresses { get; set; }
}

public class Address
{
    public int AddressId { get; set; }
    public string City { get; set; }
    public string ZipCode { get; set; }
    public string Street { get; set; }

    public virtual ICollection<Customer> Customers { get; set; }
}
```



Relationship established implicitly

Relationship established implicitly

If we would like to change the table mappings instead of using the default names convention, we could change it using 'Fluent API', as follows.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
        .HasMany(c => c.Addresses)
        .WithMany(c => c.Customers)
        .Map(c
            =>
            {
                c.ToTable("CustomerAddressRelationTable");
                c.MapLeftKey("CustID");
                c.MapRightKey("AddID");
            });
}
```



5.1.1.- Other Entity Framework 4.1 topics

There are many other EF 4.1 topics we could explore, but because of this is not an EF book and we need to cover many other layers, we must stop here. Although, we recommend exploring the following subjects:

- Entity Inheritance
- Table Splitting
- Entity Splitting
- DbSetEntry
- DbSet – IDbSet
- References management



5.12.- Implementing Repositories using EF and LINQ to Entities

As discussed in the section about the design of this layer, these components are in many aspects similar to the "Data Access" (DAO/DAL) components of traditional N-layered 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 Layer.




By separating the responsibilities of data retrieval into our Repositories we allow our domain objects to stay focused on representing the domain. We also allow our applications to become more de-coupled which help us to more easily test our applications.

If you use any O/RM technology as we will do using 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.).

A Repository initially "records" the data it is working with into the memory (a storage context). It also "marks" operations it intends to perform against the storage (usually, a database) although these operations will not be performed until those "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' onto the real persistence storage is usually based on the Unit of Work pattern used in the Application layer.

Therefore, as a general rule, for DDD N-Layered applications we will implement the Repositories with Entity Framework.

Table 6.- Repositories implementation rule

	<i>Implementing Repositories and Base Classes with Entity Framework.</i>
Rule #: I?.	
 <u>Rule</u>	
<ul style="list-style-type: none">- 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 AGGREGATE. As a general rule and for our sample Architecture we will implement the repositories using Entity Framework.	
	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	



5.13.-Repository Pattern Implementation

At the implementation level, a repository is simply a class with data access code (but coordinated by a Unit of Work when performing updates), which can be the following simple class:

```
c#  
  
Public class CustomerRepository  
{  
    ...  
    // Data Access and Persistence  
    // Linq to Entities, etc.  
    ...  
}
```

So far, there is nothing special in this class. It will be a regular class and we will implement methods like “Customer→GetCustomerById(int customerId)” by using ‘LINQ to Entities’ and POCO domain entities.

In this regard, the persistence and data access methods should be placed in the proper Repositories, usually related to the data or entity type that will be returned by a method, i.e. following this rule:

Table 7.- How to distribute methods in Repositories



Rule #: I?.

Place 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, *CustomerRepository*. Placing it into the *CompanyRepository* would be wrong).
- If the types being used are sub-entities within an AGGREGATE, the method should be placed in the Repository related to the Aggregate 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 ‘*OrderRepository*’.
- In update methods, the same rule should be followed but depending on the main updated entity.



5.13.1.- Base Class for Repositories ('Layer Supertype' Pattern)

Before seeing how to develop specific methods of Repositories using .NET and EF, we will implement a base class for all the Repository classes.

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 the following code would not make sense:

```
C#
// USELESS-CODE
// Base Class or Layered-Supertype for 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 the Repository base class methods, because after that we may want to inherit the “*CustomerRepository*” class which was not initially related to *Products*.



5.13.2.- Using 'Generics' for the Repository 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>
    : IRepository<TEntity>
    where TEntity : Entity
{
    //Base methods for all Repositories
    //Add(), FindAll(), Add(), Modify(), etc...
}

Public class CustomerRepository : Repository
{
    ...
    // Specific methods of Data Access and Persistence
    ...
}
```

Base class for Repositories

Inheriting from *Repository* base class

'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 use in the N-Layered application example:

```
C#

public class Repository<TEntity> :IRepository<TEntity>
    where TEntity:Entity
{
    //EF Context (Unit of Work)
    IQueryableUnitOfWork _UnitOfWork;

    public Repository(IQueryableUnitOfWork unitOfWork)
    {
        //Validations omitted
        UnitOfWork = unitOfWork;
    }

    public IUnitOfWork UnitOfWork
    {
        get
        {
            return _UnitOfWork;
        }
    }

    public virtual void Add(TEntity item)
    {
        ...
    }
}
```

Repository Base class Repository, using generics for multi-entity use

Dependencies in constructor, to be created by the IoC container

Common Repository method

```

if (item != (TEntity)null)
    GetSet().Add(item); // add new item in this set
else
{
    LoggerFactory.CreateLog()
        .LogInfo(Messages.info_CannotAddNullEntity,
            typeof(TEntity).ToString());
}
}

}

Common Repository method

public virtual void Remove(TEntity item)
{
    if (item != (TEntity)null)
    {
        //attach item if not exist
        _UnitOfWork.Attach(item);

        //set as "removed"
        GetSet().Remove(item);
    }
    else
    {
        LoggerFactory.CreateLog()
            .LogInfo(Messages.info_CannotRemoveNullEntity,
                typeof(TEntity).ToString());
    }
}

}

Common Repository method

public virtual void TrackItem(TEntity item)
{
    if (item != (TEntity)null)
        _UnitOfWork.Attach<TEntity>(item);
    else
    {
        LoggerFactory.CreateLog()
            .LogInfo(Messages.info_CannotRemoveNullEntity,
                typeof(TEntity).ToString());
    }
}

}

Common Repository method

public virtual void Modify(TEntity item)
{
    if (item != (TEntity)null)
        _UnitOfWork.SetModified(item);
    else
    {
        LoggerFactory.CreateLog()
            .LogInfo(Messages.info_CannotRemoveNullEntity,
                typeof(TEntity).ToString());
    }
}

}

Common Repository method

public virtual TEntity Get(Guid id)
{
    if (id != Guid.Empty)
        return GetSet().Find(id);
    else
        return null;
}

}

Common Repository method

public virtual IEnumerable<TEntity> GetAll()
{
    return GetSet().AsEnumerable();
}

}

```

```

    public virtual IEnumerable<TEntity> AllMatching(ISpecification<TEntity>
specification)
    {
        return GetSet().Where(specification.SatisfiedBy())
            .AsEnumerable();
    }

    public virtual IEnumerable<TEntity> GetPaged<KProperty>(int pageIndex,
        int pageCount,
        Expression<Func<TEntity, KProperty>> orderByExpression,
        bool ascending)
    {
        var set = GetSet();

        if (ascending)
        {
            return set.OrderBy(orderByExpression)
                .Skip(pageCount * pageIndex)
                .Take(pageCount)
                .AsEnumerable();
        }
        else
        {
            return set.OrderByDescending(orderByExpression)
                .Skip(pageCount * pageIndex)
                .Take(pageCount)
                .AsEnumerable();
        }
    }

    public virtual IEnumerable<TEntity> GetFiltered(Expression<Func<TEntity,
        bool>> filter)
    {
        return GetSet().Where(filter)
            .AsEnumerable();
    }

    public virtual void Merge(TEntity persisted, TEntity current)
    {
        UnitOfWork.ApplyCurrentValues(persisted, current);
    }

    IDbSet<TEntity> GetSet()
    {
        return UnitOfWork.CreateSet<TEntity>();
    }
}

```

Common Repository method

Common Repository method

This illustrates how to define certain common methods that will be reused by different *Repositories* of different domain entities. Even when a Repository class can be very simple at the beginning, with no explicit methods implementation, however, it would already inherit many methods implementation from the Repository base class.

For example, the initial implementation of '*ProductRepository*' could be as simple as the following code:

```

C#
//Class Repository for Product entity
public class ProductRepository
    :Repository<Product>, IProductRepository
{

```

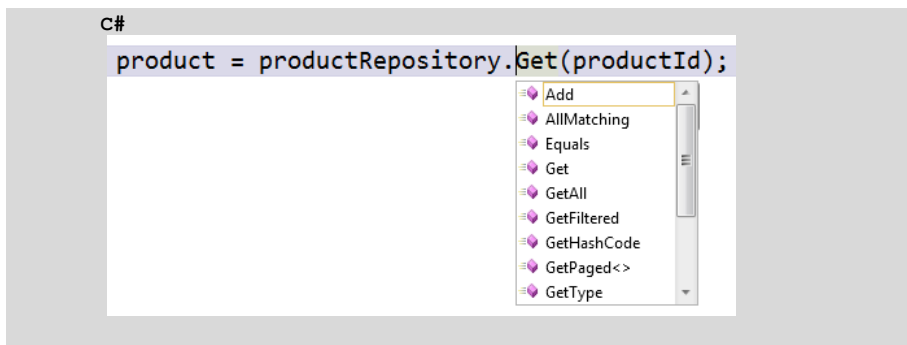
Inherits Common methods


```

public ProductRepository(IMainBCUnitOfWork unitOfWork)
    : base(unitOfWork)
{
    //You can add specific Repository methods
    //using Linq to Entities, etc.
}
}

```

As you can see, we have not implemented any explicit method in the *ProductRepository* class, however, if we instantiate an object of this class, the following would be the methods that we could invoke "without doing anything".



We would therefore have basic query, addition, and deletion methods for the mentioned 'Product' entity without having implemented them specifically for this entity.

In addition, we can add new explicit methods for the Product entity within the *ProductRepository* class itself.

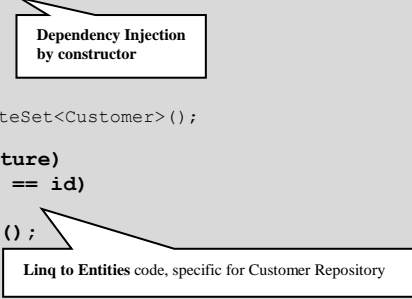
Implementing specific methods in Repositories (additional to base class methods)

An example of specific implementation of a particular Repository method would be the following:

```
C#
//Class Repository with specific methods
public class CustomerRepository
    : Repository<Order>, ICustomerRepository
{
    public CustomerRepository(IMainBCUnitOfWork unitOfWork) : base(unitOfWork) { }

    public override Customer Get(Guid id)
    {
        if (id != Guid.Empty)
        {
            var set = unitOfWork.CreateSet<Customer>();

            return set.Include(c => c.Picture)
                       .Where(c => c.Id == id)
                       .Select(c => c)
                       .SingleOrDefault();
        }
        else
            return null;
    }
}
```



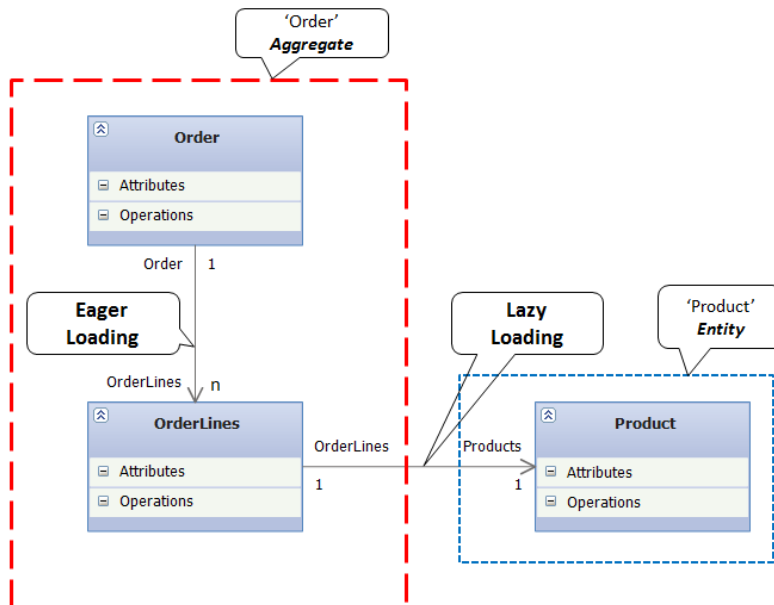
The diagram consists of two callout boxes. The first box, labeled "Dependency Injection by constructor", has an arrow pointing to the constructor parameter `IMainBCUnitOfWork unitOfWork` in the `CustomerRepository` constructor. The second box, labeled "Linq to Entities code, specific for Customer Repository", has an arrow pointing to the Linq query chain in the `Get` method.



5.14.- Aggregates, relationships and Eager/Lazy Loading

Aggregate boundaries are a good way to determine guidelines for lazy loading, meaning that **usually, eager loading should only be allow inside the boundaries of an Aggregate.**

Aggregates and Eager/Lazy Loading



By default, establishing relationships between entities (like previously shown), they are set as lazy loading. If you want to do an Eager loading, you should specify it within your Repository code writing code similar to the following:

```
//Repository method code for Eager-Loading other internal Aggregate entity
...
...
var d = from Order in uow.Orders.Include("OrderLines")
      select order;
...
//Another way of doing it using a SPECIFICATION
public override IEnumerable<Order> AllMatching(ISpecification<Order> specification)
{
    var set = _currentUnitOfWork.CreateSet<Order>();
    return set.Include(o => o.OrderLines)
               .Where(specification.SatisfiedBy())
               .AsEnumerable();
}
```

Eager Loading w/ Linq to Sql

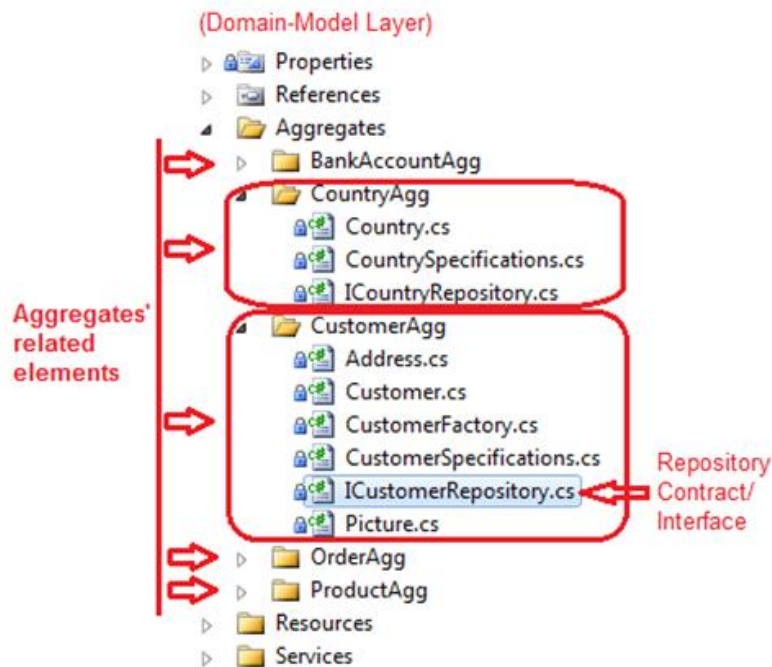
Eager Loading using a specification



5.14.1.- Repository Interfaces and the Importance of Decoupling Layers Components

As introduced in the Domain-Model Layer, for a correct de-coupled design, the use of Interface-based abstractions is essential. So, for each Repository we define, we should also have 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 should have about Repositories. 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.

Additionally, these abstractions (Repository interfaces) must be defined within the **Domain Layer**, usually in a folder that will group the contracts related to each Aggregate-Root entity, like shown in the image.



This would allow us to fully replace the data persistence infrastructure layer, or repositories through abstractions/interfaces without impacting on the Domain and Application layers, and without having to change Domain dependencies.

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.

At the interface implementation level, the following would be an example for **ICustomerRepository**:

```
C#
namespace
Microsoft.Samples.NLayerApp.Domain.MainBoundedContext.ERPModule.Aggregates.CustomerAgg
...
...
//Interface/Contrat ICustomerRepository
public interface ICustomerRepository : IRepository<Customer>
{
    //Additional methods specific for CustomerRepository
    IEnumerable<Customer> GetEnabled(int pageIndex, int pageCount);
}
```

Specific method for ICustomerRepository

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 is like this:

```
C#
namespace Microsoft.Samples.NLayerApp.Domain.Core
...
...
public interface IRepository<TEntity>
    where TEntity : Entity
{
    IUnitOfWork UnitOfWork { get; }

    void Add(TEntity item);
    void Remove(TEntity item);
    void TrackItem(TEntity item);
    void Merge(TEntity persisted, TEntity current);
    TEntity Get(Guid id);
    IEnumerable<TEntity> GetAll();
    IEnumerable<TEntity> AllMatching(ISpecification<TEntity> specification);
    IEnumerable<TEntity> GetPaged<KProperty>(int pageIndex, int pageCount,
    Expression<Func<TEntity, KProperty>> orderByExpression, bool ascending);
    IEnumerable<TEntity> GetFiltered(Expression<Func<TEntity, bool>> filter);
}
...
...
```

Base IRepository Interface

Common methods for ICustomerRepository

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 Dependency Injection constructors. All this is explained in the chapter on Application Layer, which is where the Repositories are mostly used.



5.15.- Unit of Work implementation in the Data Persistence Infrastructure Layer

In the 'Domain Model Layer' chapter we introduced the concept of UoW and we even defined a UoW contract within the Domain Layer so we will be able to comply with the PI principle. Now we need to define the real implementation of the UoW pattern.

Usually, the easier way to implement a UoW is using the Context/Session available in any O/RM, as they are similar concepts.

In our case, we will run the 'EF Context' when using a Unit of work. Specifically, we use the new **DbContext** available in EF 4.1. You could create your own UoW implementation instead of using an O/RM, but creating that asset is a quite complex task and you usually won't have the same potential than when using an O/RM, like EF or NHibernate.

The important point here is to wrap the EF Context with an Interface/Contract so we can use it from the Application and Domain Layers with no direct dependencies to EF thanks to our selected IoC/DI container and abstractions (the mentioned interfaces). Additionally, if we make this abstraction (UoW interfaces), we will also be able to mock the EF Context with an in-memory unit of work 'simulating the EF context accessing to the database'. This mocking will be very useful in order to make proper and faster Domain and Application Unit Testing isolated from the database persistence.

First, we need to add another UoW interface/contract in the Infrastructure-Persistence Layer which will contain the specific methods only available in EF context, like the following interface.

C#

UoW contract in Infrastructure Layer tied to EF

Implements the generic UoW methods

```
public interface IQueryableUnitOfWork : IUnitOfWork, ISql
{
    Returns IDbSet instance for accessing the underlying store related to a specific entity

    IDbSet<TEntity> CreateSet<TEntity>() where TEntity : class;

    Attaches this item into the "ObjectStateManager"

    void Attach<TEntity>(TEntity item) where TEntity : class;

    Sets object as modified

    void SetModified<TEntity>(TEntity item) where TEntity : class;

    Attaches original and set current values of a specific entity

    void ApplyCurrentValues<TEntity>(TEntity original, TEntity current)
        where TEntity : class;
}
```

Then, we need to have the Unit of Work implementation, which is really based on the EF 4.1 DbContext, as shown below.

C#

UoW Implementation

Based on EF 4.1 DbContext

```
public class MainBCUnitOfWork : DbContext, IMainBCUnitOfWork
{
    IDbSet<Customer> _customers;
    public IDbSet<Customer> Customers
    {
        get
        {
            if (_customers == null)
                _customers = base.Set<Customer>();

            return _customers;
        }
    }

    IDbSet<Product> _products;
    public IDbSet<Product> Products
    {
        get
        {
            if (_products == null)
                _products = base.Set<Product>();

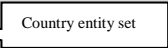
            return _products;
        }
    }

    IDbSet<Order> _orders;
    public IDbSet<Order> Orders
    {
        get
```

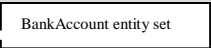
```

    {
        if (_orders == null)
            _orders = base.Set<Order>();

        return _orders;
    }
}

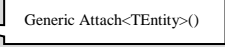
IDbSet<Country> countries;
public IDbSet<Country> Countries 
{
    get
    {
        if (_countries == null)
            _countries = base.Set<Country>();


        return _countries;
    }
}

IDbSet<BankAccount> bankAccounts;
public IDbSet<BankAccount> BankAccounts 
{
    get
    {
        if (_bankAccounts == null)
            bankAccounts = base.Set<BankAccount>();

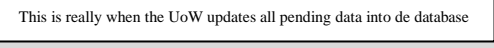
        return _bankAccounts;
    }
}

public IDbSet<TEntity> CreateSet<TEntity>() 
    where TEntity : class
{
    return base.Set<TEntity>();
}

public void Attach<TEntity>(TEntity item) 
    where TEntity : class
{
    //attach and set as unchanged
    base.Entry<TEntity>(item).State = System.Data.EntityState.Unchanged;
}

public void SetModified<TEntity>(TEntity item) 
    where TEntity : class
{
    //this operation also attach item in object state manager
    base.Entry<TEntity>(item).State = System.Data.EntityState.Modified;
}

public void ApplyCurrentValues<TEntity>(TEntity original, TEntity current)
    where TEntity : class
{
    //if it is not attached, attach original and set current values
    base.Entry<TEntity>(original).CurrentValues.SetValues(current);
}

public void Commit() 
{
    base.SaveChanges();
}

public void CommitAndRefreshChanges()
{
    bool saveFailed = false;

    do
    {

```



```

        try
        {
            base.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            saveFailed = true;

            ex.Entries.ToList()
                .ForEach(entry =>
                {
                    entry.OriginalValues.SetValues(entry.GetDatabaseValues());
                });
        }
    } while (saveFailed);
}

```

```

public void RollbackChanges()
{
    // set all entities in change tracker
    // as 'unchanged state'
    base.ChangeTracker.Entries()
        .ToList()
        .ForEach(entry => entry.State =
System.Data.EntityState.Unchanged);
}

public IEnumerable<TEntity> ExecuteQuery<TEntity>(string sqlQuery, params
object[] parameters)
{
    return base.Database.SqlQuery<TEntity>(sqlQuery, parameters);
}

public int ExecuteCommand(string sqlCommand, params object[] parameters)
{
    return base.Database.ExecuteNonQuery(sqlCommand, parameters);
}

```

Rollback data changes

Fluent API customizations based on **EntityTypeConfiguration** classes

```

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    //Remove unused conventions
    modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
    modelBuilder.Conventions.Remove<IncludeMetadataConvention>();

    //Add entity configurations in a structured
    //way using 'TypeConfiguration' classes
    modelBuilder.Configurations.Add(new BankAccountEntityTypeConfiguration());
    modelBuilder.Configurations.Add(new ProductEntityTypeConfiguration());
    modelBuilder.Configurations.Add(new SoftwareEntityTypeConfiguration());
    modelBuilder.Configurations.Add(new BookEntityTypeConfiguration());
    modelBuilder.Configurations.Add(new CountryEntityTypeConfiguration());
    modelBuilder.Configurations.Add(new CustomerEntityTypeConfiguration());
    modelBuilder.Configurations.Add(new OrderEntityTypeConfiguration());
    modelBuilder.Configurations.Add(new OrderLineEntityTypeConfiguration());
    modelBuilder.Configurations.Add(new PictureEntityTypeConfiguration());
}
}

```

Finally, we need to map from the UoW interface to the real implementation class. That should be done in our UNITY container registry code, like it is implemented in the following code.

```

C#
public static class Container
{
    //Omitted
    //Other methods
    static void ConfigureContainer()
    {
        _currentContainer = new UnityContainer();

        //-> Unit of Work and repositories
        currentContainer.RegisterType<IMainBCUnitOfWork,
            MainBCUnitOfWork>(new PerResolveLifetimeManager());

        //Other types registration and mapping
        _currentContainer.RegisterType<IBankAccountRepository,
            BankAccountRepository>();
        _currentContainer.RegisterType<ICountryRepository,
            CountryRepository>();
        currentContainer.RegisterType<ICustomerRepository,
            CustomerRepository>();
        _currentContainer.RegisterType<IOrderRepository,
            OrderRepository>();
        _currentContainer.RegisterType<IProductRepository,
            ProductRepository>();

        //-> Adapters
        currentContainer.RegisterType<ITypeAdapter, TypeAdapter>();
        _currentContainer.RegisterType<RegisterTypesMap,
            ERPModuleRegisterTypesMap>("erpmodule");
        _currentContainer.RegisterType<RegisterTypesMap,
            BankingModuleRegisterTypesMap>("bankingmodule");

        //-> Domain Services
        _currentContainer.RegisterType<IBankTransferService,
            BankTransferService>();

        //-> Application services
        currentContainer.RegisterType<ISalesAppService,
            SalesAppService>();
        _currentContainer.RegisterType<ICustomerAppService,
            CustomerAppService>();
        currentContainer.RegisterType<IBankAppService,
            BankAppService>();

        //-> Distributed Services
        _currentContainer.RegisterType<IBankingModuleService,
            BankingModuleService>();
        currentContainer.RegisterType<IERPModuleService,
            ERPModuleService>();
    }
    //Other methods
    //...
}

```

Types registration and Mappings

UoW Registration & Mapping

The EF UoW uses **PerResolveLifetimeManager** so it can be reused by all objects within each DI Objects graph



5.16.-Integration Tests for Repositories

This functionality is offered by most testing frameworks and also of course by the Visual Studio Unit Testing Framework.

If we want to run tests on a certain repository, such as **ICustomerRepository** the code could be similar to the following:

```
C#
[TestClass()]
public class CustomerRepositoryTests
{
    [TestMethod()]
    public void CustomerRepositoryGetMethodReturnCustomerWithPicture()
    {
        //Arrange
        var unitOfWork = new MainBCUnitOfWork();
        ICustomerRepository customerRepository = new CustomerRepository(unitOfWork);

        var customerId = new Guid("43A38AC8-EAA9-4DF0-981F-2685882C7C45");
        Customer customer = null;

        //Act
        customer = customerRepository.Get(customerId);

        //Assert
        Assert.IsNotNull(customer);
        Assert.IsNotNull(customer.Picture);
        Assert.IsTrue(customer.Id == customerId);
    }

    [TestMethod()]
    public void CustomerRepositoryGetMethodReturnNullWhenIdIsEmpty()
    {
        //Arrange
        var unitOfWork = new MainBCUnitOfWork();
        ICustomerRepository customerRepository = new CustomerRepository(unitOfWork);

        Customer customer = null;

        //Act
        customer = customerRepository.Get(Guid.Empty);

        //Assert
        Assert.IsNull(customer);
    }

    //Other Test Methods
}
```



Rule #: I?.

Implement Integration Tests for Repositories

○ Note

- As long as we are accessing to the database, we will be running integration tests.

5.17.- 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 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 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 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

- o <http://blogs.msdn.com/b/cesardelatorre/archive/2010/12/20/handling-sql-azure-connections-issues-using-entity-framework-4-0.aspx>



5.17.1.- 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":

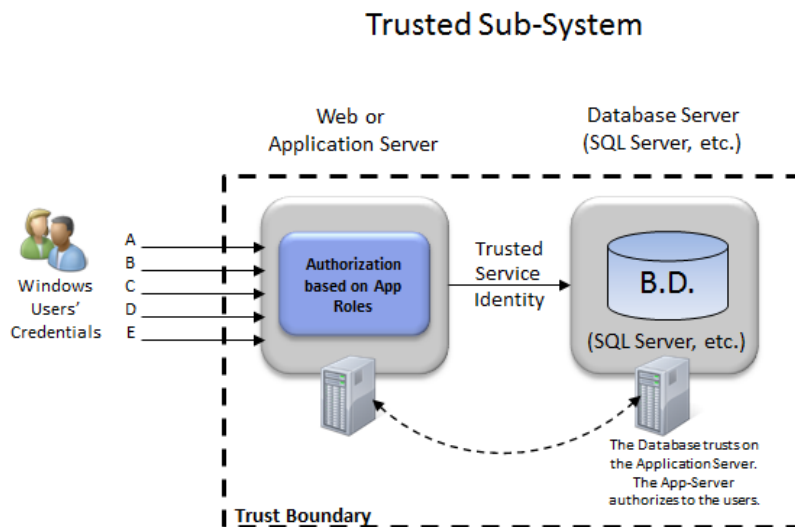


Figure 4.- "Trusted Sub-System" diagram

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.



5.18.- 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.
- Design and implement a logging system and error notification system for critical errors and exceptions that do not show confidential information.



5.19.- Implementing Service Agents for External Services (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.



5.20.- References of Data Access Technologies

"T4 and code generation" - [http://msdn.microsoft.com/en-us/library/bb126445\(VS.100\).aspx](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](http://msdn.microsoft.com/en-us/library/bb896304(VS.100).aspx)

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

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

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