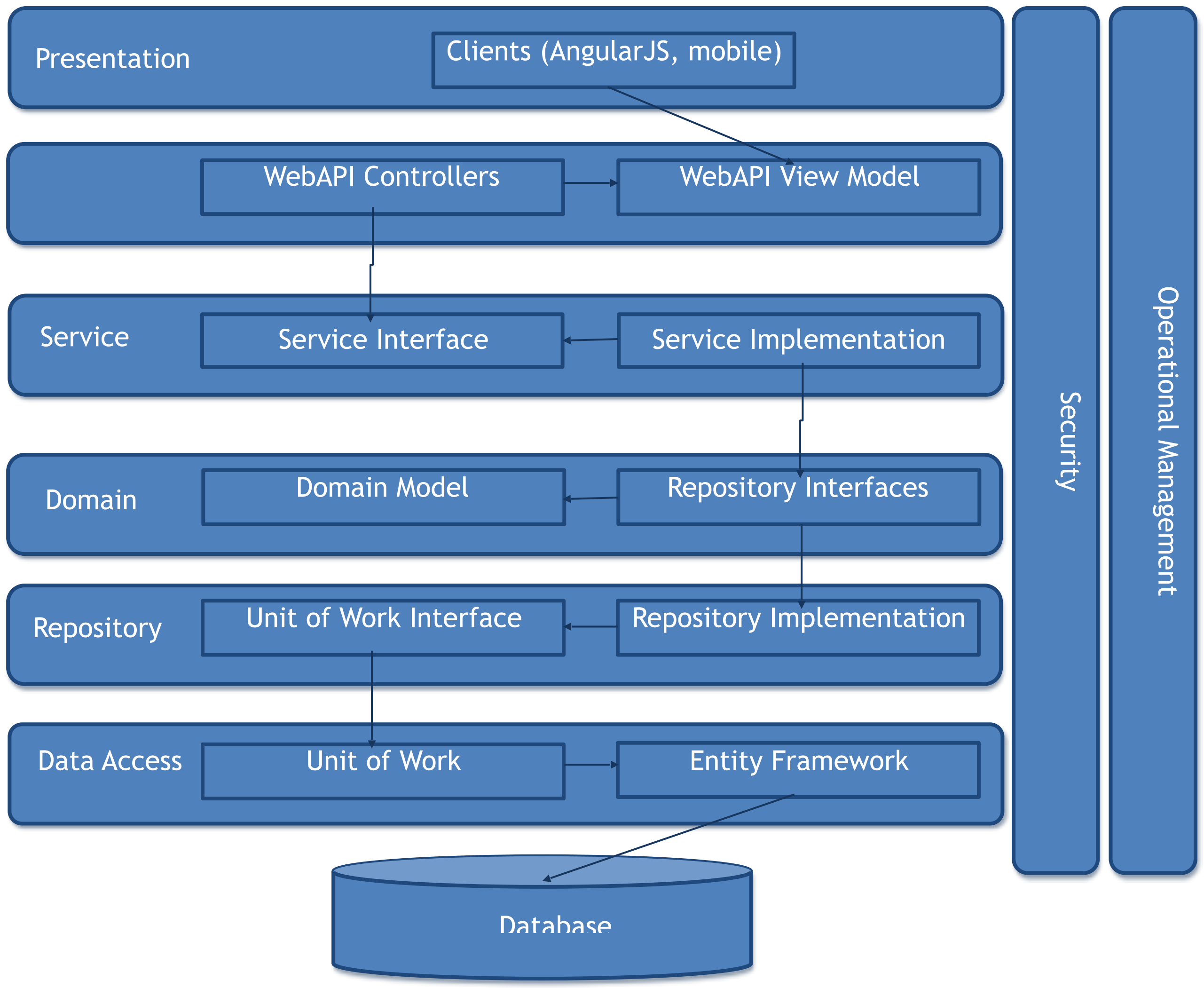
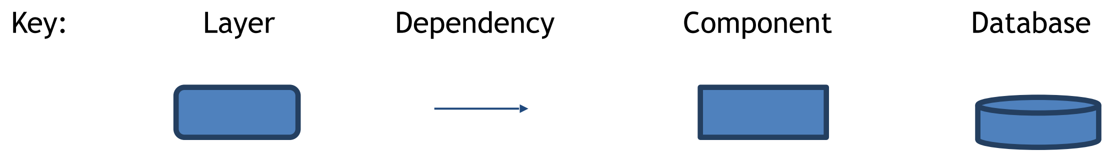
Architecture Logical Diagram





Comments

- Each layer is dependent on the layer below it.
- Each layer contains an interface that describes how the layer below it must communicate with it. With this design the implementation of the lower layer can be changed so long as the new implementation abides by the parent interface.
- The subordinate layer returns objects that are defined in the parent layer.

2.4.2 Application Layer

2.4.2.1 Issues

- Orderly development of the application with significant code reuse.

- Long term maintenance and enhancement. To develop a design that articulates where new code should go and where to find existing code.

2.4.2.2 Solution and Rationale

The primary purpose of the Application Layer is to:

- Orchestra the processing of requests.
- Map objects between the Domain Model and the View Model.
- Provide server-side validation.

Object Interaction Diagram

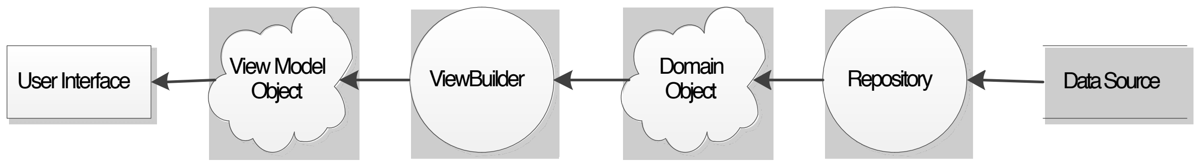
An introduction to the primary object classifications in the Application Layer is warranted and useful when visualizing the use of this layer's classes. (Examples of Application Layer classes being: View Model, ViewBuilder, Factory and Repository classes.)

Definition

- A **ViewBuilder** takes in Domain objects and produces View Model objects that are sent to the client. It uses repository to obtain persisted Domain objects.

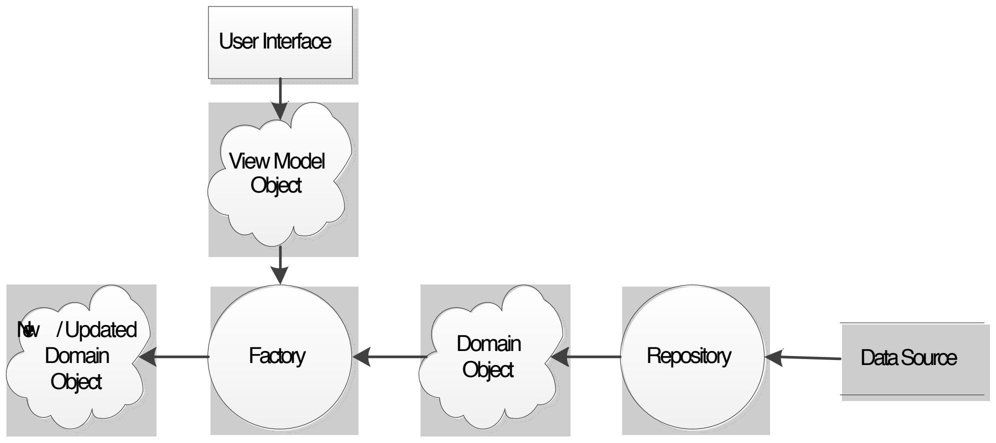
- A **Factory** takes in View Model objects to create Domain objects.

The *Object Interaction Diagrams* show the interaction of objects within the Application Layer. A Controller object, which is not shown in the diagrams, orchestras all the tasks performed during a request operation.



ViewBuilder Object Interaction Diagram

- To provide content to the user interface a ViewBuilder object reshapes Domain objects to create View Model (VM) objects that map cleanly to a View that is sent to a user interface (UI). The ViewBuilder relies on a Repository object to obtain Domain objects from the datasource.



Factory Object Interaction Diagram

- When responding to a UI request, the MVC framework populates a VM object with the UI's data. This VM object is then consumed by a Factory object. The Factory knows how to transform the VM object into a Domain object. It also instructs Domain objects to validate themselves before they can be made available to the rest of the application. (See Entity Validation Framework for details.)

- The Factory object also merges previously saved Domain objects with updated data contained in a corresponding VM object returned from the client. It instructs a Repository object to retreive presisted data from the data source, updates the retreived Domain object with the data contained in the VM object, and then has the updated Domain object validate itself.

Controller Design Guidelines

- Controllers should be thin and orchestrate the tasks that are to be performed during a request. The actual work should be performed by previously mentioned objects within the application.

- A programmer should be aware of situations where a Controller is performing a set of tasks on behalf of a set of Domain objects. For example, in another application the userId is stored in two different objects with different encryption. Which object should responsible for the updating of the other object? The Controller could orchestra this updating but this places domain logic in the Controller. Instead, a Domain Service object should oversee the operation. The Controller would call a method on the Domain Service and the service would know how to update the objects.

- Having the Controller focus on calling methods on different objects instead of actually performing the work promotes the reuse of code and simplifies the maintenance of the application.

View Model Design Guidelines

- Only View Model objects will be sent to a View. This decouples the View for the Domain. It promotes the unit testing of view content and enables the View to be developed independently of the Domain. In other words, Domain Model objects should never be used directly within a View.

- There will be a one-to-one relationship between a View and a View Model object for the following reasons:
    - It simplifies the maintenance of this application since only the data required for a single View is contained in a View Model object. Having data related to multiple views in a single View Model object complicates the maintenance of the Views and the View Model object.
    - It improves the performance of the application since only the required data is transmitted to the client.
    - It simplifies testing of a view's content. A View Model object decouples the Application Layer from the Presentation Layer. This enables the View to be tested with mock data. It also enables ViewBuilders, which generate View Model content, to be unit tested irrespective of the View.

View Model Design Format

Frequently, when working with Views certain fields are used to display data while other fields are used to capture and return data. In these situations the following format should be followed:

- Separate display data from input data.
- Define the input properties in a separate class.

ViewBuilder Guidelines

- A `ViewBuilder` class should be associated with a `Controller` class. Each method on the `ViewBuilder` should be associated with a View and return a View Model object for that View. For example:

    - A `VehicleController` contains an `Edit()` action.
    - By the MVC3 convention, the `return View()` statement will return a view named Edit.chtml.

  Adhering to the MVC convention over configuration principle, all Vehicle View related View Models are placed in a Vehicle folder. For example, that folder that could contain:

    1. An `EditViewModel` class for the `Edit` view.
    2. A `VehicleViewBuilder` class that contains a `CreateEditViewModel` method and a `ReCreateEditViewModel` method. The first method is used to create an `EditViewModel` object on the initial request from the client. On subsequent requests, the data entered by the client must be preserved while drop down list related data must be repopulated in the `EditViewModel` object. Because of the previously mentioned View Model Design Format, the code needed to recreate a view is very clean and easy to understand.

Such an approach simplifies application maintenance. Each method has a single responsibility. Therefore, there would be only one reason to change the method.

- `ViewBuilders` are stateless objects that act as services. Any dependencies they need should be injected into them. In addition, these parameters should be defined as interfaces. This decouples the ViewBuilders from the implementation of the interfaces.

Factory Design Guidelines

- A Factory encapsulates the knowledge needed to construct new non-trivial Domain objects and a valid object graph from an aggregate root. It returns these objects in a valid state with respect to themselves and other Domain objects. (See Aggregate Roots for details.)
- Factories should be aligned with aggregate roots and return a reference to the root of that Domain object's graph.
- Factory methods (operation) must be atomic.
- They should return complete objects that are ready to be saved by the unit of work.
- In this application, the Factories reside in the Application Layer. They take View Model objects as an input parameter and return Domain objects. Placing the Factories in the Domain Layer would cause the Domain to be dependent on the Application Layer since ViewModel classes are defined in the AL.
- A Factory does not perform the actual validation of a DM object. This task is handled by the DM object's `IsValid` property when called by the Factory. (See Entity Validation Framework for details.)
- A Factory assists in ensuring that a domain object is in a valid state with respect to another domain object. For example, suppose there is a rule stating each Vehicle object must have a unique license plate number. After the Vehicle object is created, the Factory would use an associated Repository object to provide the Vehicle object with the information it needs to perform the plate validation rule. This approach ensures the decoupling of the Domain object from the Factory. It enables unit testing of the Domain object's IsValid() method irrespective of the Factory since the data provided by the Factory could be mocked.
- A Factory ensures that all validation is performed by all objects in the aggregate.

Loosely Coupled Classes: Dependency Injection

This application will use Dependency Injection (DI). DI consists of software design principles and patterns that promote the development of loosely coupled classes. It addresses the issue of object interacting with each other such that changes to an object will have no or minimum effect on a dependent object. Under DI:

- Object dependencies are to be injected into the object that uses them. Dependencies are not to be instantiated from within that object. For example, a ViewBuilder uses repositories to handle retrieval of objects from the database. The repository objects are to be injected into the ViewBuilder either through constructor injection (parameter in the constructor) or method injection (parameter in the method).

- The type of the input parameter being injected should always be an interface. For example, the constructor for the `VehicleViewBuilder` could be

`VehicleViewBuilder(IVehicleRepository repository)`. It should not be an implementation of the `IVehicleRepository` interface. By using an interface the implementation can be changed and the `VehicleViewBuilder` code remains the same.

- When an object contains some functionality that assists in an operation that does not seem to logically fit the description of that object, that code should be placed in a separate class and injected into the dependent class. For example, an employee object may contain an encrypted UserId property. There should be an object that provides encryption services and it should be injected into the employee object.

- This application will use StructureMap, a DI container that facilitates the centralized "wiring up" of an application's objects. The use of interface based parameters with a DI container makes it possible to have a centralized location were the implementation of an interface can be defined. In addition, the definition is adhered to throughout the application.

2.4.3.2.2 Aggregate Roots

The aggregate root is a concept that has proven to be very useful in another application and will be implemented in this application[1].

"An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each AGGREGATE has a root and a boundary. The boundary defines what is inside the AGGREGATE. The root is a single, specified ENTITY contained in the AGGREGATE. The root is the only member of the AGGREGATE that outside objects are allowed to hold references to, although objects within the boundary may hold references to each other." (Evans, pg. 126-127)

The aggregate root concept:

- Simplifies the design of the application.
- Promotes the data integrity of the application. (See Data Integrity for details.)
- Enhances the performance of the application. The persisting/retrieval of an object via the aggregate root cause all the necessary data to be saved to or retrieved from the database in a single trip.
- Takes advantage of the features offered by ORMs such as Entity Framework. EF provides the means to load specific parts of an object graph in a single query. In addition, EF is capable of persisting the entire graph when the root is persisted. (See Why the Unit of Work Pattern for details.)

The guideline for aggregate roots is as follows:

- The root entity has global identity.

- When changes to any object within an aggregate are committed all invariants[2] of the whole aggregate must be validated. (This is why Factories will be aligned with aggregate roots.)

[1] Using the aggregate root concept the time to display an Order Index View was reduced from approximately 11 seconds to approximate 0.5 seconds.

[2] Consistency rules that must be maintained whenever data changes.

- Entities inside the boundary have local identity unique only within the aggregate.
- An object outside the boundary cannot hold a reference to an object inside the boundary. These objects are obtained via the root.
- Only aggregate roots can be obtained directly from the database. That means repositories should be aligned with aggregate roots. There should not be a repository for every object in the Domain Model.
- Objects within the aggregate can hold references to other aggregate roots.
- It's up to the developers of the Domain Model to determine where an aggregate begins and ends.
- A delete operation must remove everything within the aggregate boundary at once. This can be accomplished in two ways. The `Clear()` method can be called on the child collection or cascading deletes can be setup in the database. The objective is to preserve the integrity of the database by not having orphaned children objects or deleting objects that are referenced by other others. For example, when a vehicle is deleted the status object should not be deleted.

2.4.4 Repository Layer

2.4.4.1 Issues

- Data integrity. Repositories should be designed in accordance with the defined relationships between DM objects in the Domain Model.
- Reuse of code. Repository classes perform the same CRUD operations against different tables. However, there are situations where queries unique to a domain object are required. The design should consolidate the code needed for basic CRUD operations while being able to accommodate unique situations.
- Decoupling. It should be possible to change the implementation of the repositories without affecting the Domain Layer.
- Performance. Minimize database retrieval time. Lazy Loading is designed to improve the response time for stateless type applications by retrieving the minimum amount of information during a trip to the database. However, there are situations where it is known in advance what related data will be needed. In these situations, all relevant data should be retrieved during the initial trip to the database. The repositories should be designed to provide a standard way of doing this.
- Performance. The processing of queries should be performed as close to the data source as possible. In addition, only data that meets the query criteria should be returned from the data source.

2.4.4.2 Solution and Rationale

- Repositories reduce the complexity of an application by encapsulating the code needed to perform CRUD operations.

- To further reduce complexity repositories should reflect domain model concepts. For example, an acceptance letter has no meaning unless assigned to a vehicle. This relationship can be thought of as a logical unit within the domain. An aggregate root depicts a set of objects that form a logical unit. A repository should return logical units. Therefore, repositories should be aligned with aggregate roots.

The following patterns are implemented in this layer:

- Repository
- Unit of work
- Dependency Injection

The use of these patterns along other design techniques that will be presented, address the issues raised for this layer.

2.4.4.2.1 Data Integrity

Repositories will be designed and used to take advantage of the DDD concept of an aggregate root. For example, an Acceptance Letter has no meaning unless it is associated with a vehicle. Therefore, when creating an acceptance letter a valid vehicle object is a prerequisite and when persisting an acceptance letter it is the vehicle that is persisted. An outcome of this logic is that an acceptance letter repository is not needed since the root is the vehicle. CRUD operations should be done via a vehicle repository.

This sort of design promotes data integrity since an aggregate root defines what constitutes a complete domain object, thus reducing the likelihood of orphaned children in the database. In addition, validation begins at the root so not only is an object's property data validated but parent-child validation rules can also be validated in factory objects.

As per data integrity the outcome of using the aggregate root concept is a multi-layered integrated scheme to ensure the correctness of the application's data. The design promotes higher cohesion and looser coupling.

2.4.4.2.2 Decoupling: Unit of Work Interfaces

The unit of work interface consists of two interfaces. The base interface is `IUnitOfWork` that contains one method, `Save()`. The implementation of this method calls the save method associated with the ORM being used.

```
public interface IUnitOfWork : IDisposable
{
        void Commit();
}
```

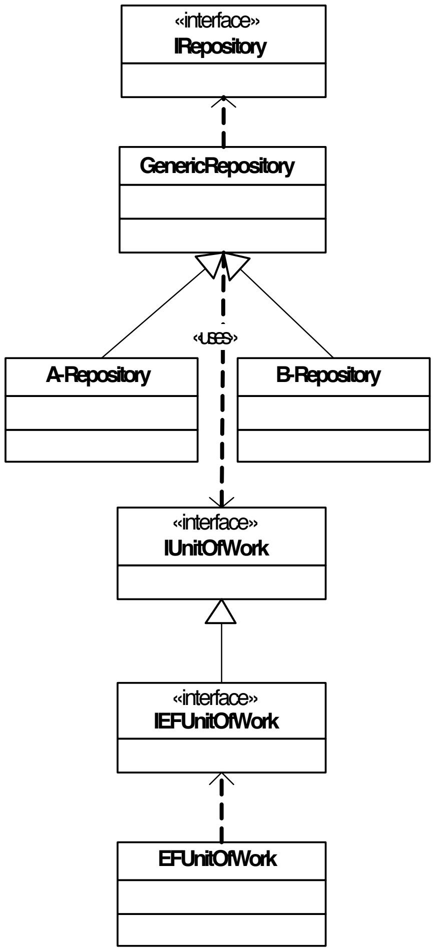
Unit of Work Base Interface

One level above `IUnitOfWork` is an interface that is used to house the context object implemented by the particular ORM. A description of the interface for Entity Framework is shown below. "T" represents the type of the particular unit of work; for Entity Framework that would be the `ObjectContext` object. These interfaces consist of one method named `GetContext()` that is used by repositories to extract the context to perform CRUD operations against it.

```
public interface IEFUnitOfWork<T> : IUnitOfWork
{
        T GetContext { get; }
}
```

EF Interfaces

Repository – Unit of Work Class Diagram



Relevant Interface Description

`IQueryable<T> GetQuery():`

- Returns an `IQueryable<T>` object containing all the objects in the specified entity type. It is used to build customized queries against the entity.

`IList<T> GetAll(Func<IQueryable<T>, IOrderedQueryable<T>> orderBy = null):`

- This function returns a list of entities in the specified sort order. An example of how it might be used is: `GetAll(q => q.OrderBy(x => x.FirstName)).`

`IList<T> Find(Expression<Func<T, bool>> whereFilter, string[] includeList ):`

- This method executes a Where method for the lambda expression methods provided by Entity Framework. It returns a list of objects that meet the search criteria provided in the `whereFilter` parameter which is a lambda expression. If objects are not found, an empty list is returned.

`T GetSingle(Expression<Func<T, bool>> expressionFilter, string[] includeList):`

- This method is used in situations where one and only one object should satisfy the query condition. If an object is not found, null is returned. If more than one object is found the meets the query condition an exception is thrown.

`T GetFirst(Expression<Func<T, bool>> expressionFilter, string[] includeList):`

- This method returns the first element in a set of elements that meet the search criteria. If objects are not found, null is returned.

2.4.4.2.4 Extensibility: Specialized Repository Example

The listing below shows how the `GetQuery()` method from the `GenericRepository` base class can be used. In this case, `Where()` clauses are attached to the `IQueryable` object that is returned by the `GenericRepository.GetQuery()` method. The query is not executed until the `ToList()`method is called.

The second method, `GetSpecIdBySeriesId()`, uses the `GenericRepository.GetSingle()` method. This method takes two parameters. The first parameter accepts a lambda expression that is used to filter the data being requested. The second parameter accepts a lambda expression indicating how the returned objects are to be sorted.

```
public class OrderRepository : GenericRepository<OrderModel>, IOrderRepository
{
   public OrderRepository(IUnitOfWork UoW) : base(UoW) {}

   public IList<OrderModel> GetOrders(string order, int customerNo, int lineId)
   {
      var query = GetQuery();

      if (!String.IsNullOrEmpty(order))
         query = query.Where(x => x.Orders.OrderType == order);

      if (!String.IsNullOrEmpty(customerno))
         query = query.Where(x => x.Customers.ContractNo == customerNo);

      if (equiptype.HasValue && equiptype != 0)
         query = query.Where(x => x.Order.LineItem.LineId == lineId);

      return query.ToList();
   }
```

```
    public long GetOrderIdByLineId(long id)
    {
       var obj = GetSingle(x => x.LineItem.LineId == id, null);
       return (obj != null) ? obj.OrderId : 0;
    }
}
```

Specialized Repository Implementation

The Benefits of Generics and Lambda Expressions

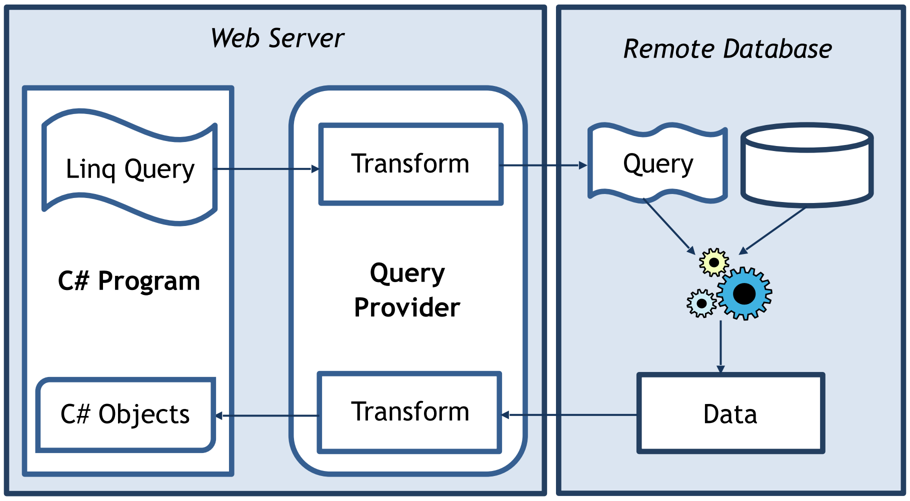
The use of generics and lambda expressions promotes reuse and significantly reduces written code. Generics allow the same code to be used against different types of Domain Model objects. Lambda expressions allow the same method to be used to select different objects sorted in different ways.

2.4.4.2.5 Performance

Why IQueryable<T>?

The method `GenericRepository.GetQuery()` returns an object of type `IQueryable<T>`. It could have returned an object of type `IEnumerable<T>` since `IQueryable<T>` is derived from `IEnumerable<T>`. The following provides the reasoning for this decision.

Using the diagram below as a guide, `IEumerable<T>` processes data on the WebServer. This means it first retrieves all the data from the remote database and then applies the conditions outlined in a lambda expression. In contrast, `IQueryable<T>` provides a lambda expression that is transformed by a query provider into the logic format of the appropriate provider. In this case, a sql query that is sent to the database. That query is then executed by the database engine and only the data that satisfies the query is returned. As a consequence, in many situations `IQueryable<T>` will process data faster than `IEnumerable<T>`.



In the diagram above a query provider translates an IQueryable object into a sql format that can be used by the database. The query is executed by the database engine. The resultset is sent back to the Query Provider where it is transformed into C# objects.

Minimize Data Retrieval Trips: The Extensions Class

The `Include()` method used in the `GenericRepository` class methods is actually an extension method. Its implementation is shown below. These methods are used to "eager load" the requested data along with specified data in its graph . They are used in situations where it is known that the related data will be needed and enhance performance since a single join query is sent to the database to be processed.

NOTE: The default behavior of EF is lazy loading. In this case, the root element is returned and graph objects are retrieved when required. For example, when a `return` or `foreach` statement is encountered. This results in multiple trips to the database.

```
   public static class Extensions
   {
       public static IQueryable<T> Include<T>(this IQueryable<T> source, string
path)
       {
           var objectQuery = source as ObjectQuery<T>;

           if (objectQuery != null && !string.IsNullOrEmpty(path))
               return objectQuery.Include(path);

           return source;
       }

       public static IQueryable<T> Include<T>(this IQueryable<T> source, string[]
paths)
       {
           var objectQuery = source as ObjectQuery<T>;

           if (objectQuery != null && paths.Length > 0)
           {
               return paths.Aggregate(objectQuery,
                                      (current, path) => current.Include(path));
           }
           return source;
       }
   }
```

Include() extension method implementation

`Include()` is an overloaded method in the `Extensions` class.

`IQueryable<T> Include<T>(this IQueryable<T> source, string path)`

- An extension method that takes a string containing the name of the child objects that are to be returned with the query. Notice that Linq provides an Include method. However, the Include methods cannot be chained to return different object types in the graph when using POCOs. For example, `_repository.GetQuery().Include("tblSery.Series").Include("POItem")` could not be used.

`IQueryable<T> Include<T>(this IQueryable<T> source, string[] paths)`

- This method is used when multiple object types from the graph is required. The second parameter is an array that contains the object types to be returned with the root object. The Aggregate method calls the Include method for each element in the array.

2.4.5 Data Access Layer

2.4.5.1 Issues

- How to design this application so that it can take advantage of the features offered by an object relational mapper (ORM) and be able to change the ORM implementation with minimum impact to the rest of the application.
- How to fulfill the decoupling requirement outlined in the Architecture Logical Diagram Description.
- Security: securing connection strings and putting up safe guards again hacker attacks.
- Performance.
- Exception management and logging: identify exceptions that should be handled by this layer with all others bubbling up through the application. Design a component that logs exceptions and notifies the appropriate individuals when critical errors occur.

2.4.5.2 Solution and Rationale

- The Data Access Layer will be implemented as a separate component that implements a single interface defined in the Repository Layer.
- Performance gains will be realized through the use of the IUnitOfWork Save() method. ORMs track the state of domain objects and persist the state of those objects as a single transaction when the Save() method is called. Repositories will not contain a Save() method.
- The use of a unit of work also promotes data integrity and simplifies the coding process.

Performance

- ORMs provide the means to execute stored procedures that historically have better performance than queries generated by a data source client. However, today's ORMs perform significantly faster. In situations where performance gains are desired, tests should be run to compare query processing speeds.

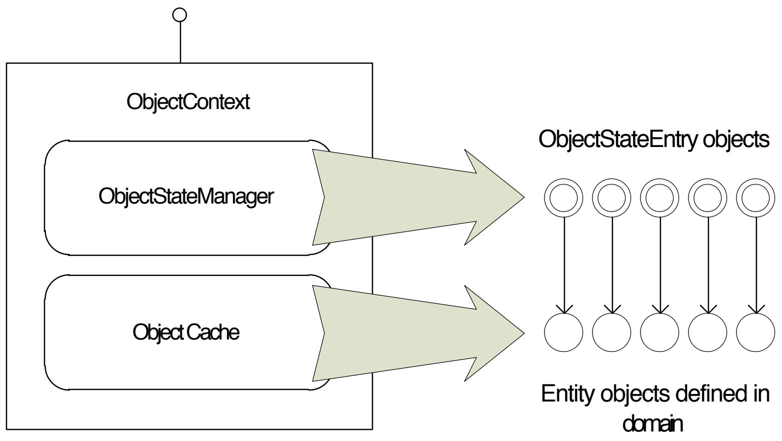
2.4.5.3 DataAccess Layer Interface

Previously it was stated that using unit of work interfaces reduces the effect changing the ORM would have on the Repository Layer. Since this application will use EF, the implementation of IEFUnitOfWork is presented. In addition, the rationale for the implementation of a unit of work is provided.

2.4.5.3.1 Why the Unit of Work Pattern?

EF is designed to emulate a unit of work. EF simplifies programming and promotes the integrity of the data it handles. The diagram below serves to explain how EF works.

Programmers communicate with EF through the `ObjectContext` class. This class contains methods for CRUD operations and a `SaveChanges()` method. Internally, the ObjectContext contains a cache and ObjectStateManager object. All entity objects that have gone through CRUD operations are contained in the cache. The ObjectStateManager tracks the state of entities by creating ObjectStateEntry objects for each entity. Those states can be: Added, Unchanged, Deleted or Modified.



When deemed appropriate by the programmer the objects in the cache are explicitly persisted to the data source when the `DbContext.SaveChanges()` method is called.

The benefit of this design is that CRUD operations can be performed without concern for the state of the database due to these operations. If a sequence of steps is to be performed during an http request and one of these steps fails, the entire operation can be negated by disregarding the unit of work. Since the SaveChanges() method was not explicitly called, programmers do not have to be concerned with what was or was not saved to the database at the point of failure.

To take advantage of this capability offered by the DbContext the application must implement the unit of work pattern.

2.4.5.3.2 Unit of Work Implementation

The DataAccess Layer contains a single interface that implements the IUnitOfWork interface defined in the Repository Layer. The code in Figure 4 shows the Entity Framework implementation of the interface. It contains a:

- Constructor that takes in the `DbContext`.
- `GetContext` property that can be used by repositories to obtain a reference to the `DbContext`.
- `Save()` method that causes the `DbContext` to persist all CRUD operations to the database.

```
        public class EFUnitOfWork : IEFUnitOfWork<DbContext>
        {

                private readonly DbContext _context = null;
                private bool disposed = false;

                public EFUnitOfWork(DbContext context)
                {
                        if (context == null)
                                throw new ArgumentNullException("Unable to initialize
");
                        _context = context;
                }

                public DbContext GetContext
                {
                        get { return _context; }
                }

                public void Commit()
                {
                        _context.SaveChanges();
                }
        }
```

Entity Framework implementation of unit of work

2.4.5.3.3 Using the Unit of Work

The instantiation and use of the Unit of Work follows:

```
        IUnitOfWork UoW = new EFUnitOfWork(new ObjectEntities());
         GenericRepository<Rule> target = new GenericRepository<Rule>(UoW);
```

Unit of Work instantiation.

In the preceding code the ObjectContext is housed in the ObjectEntities object and it is passed into the constructor of the unit of work during its creation. The unit of work is then passed into the constructor of a repository.

With this design the same instance of the unit of work is passed into different repositories. This connects all the repositories to the same `ObjectContext`. Consequently, any combination of operations across different repositories can be persisted as a single transaction.

Here is how the Save method would be used:

```
UoW.Save();
```