Work Orders Architecture Document

Project Management

Michael Walfall
Version 1.0

Contents

1. Introduction

One of the primary goals of the Architecture was to develop a base architecture/framework that could be used by future projects. With the envisioned design, application developers would send very little time on infrastructure and the vast majority of their time writing code for the Presentation and Application layers of applications.

That proved to be the case with the NameHere Management System. It took 2-3 days to set up the infrastructure. From that point on all development resided in the Presentation layer, Application layer and adding behavior to domain objects. Tools, such as StructureMap and AutoMapper, significantly reduce the amount of written code and proved to facilitate the decoupling of the application.

With each project we attempt to refine different aspects of the base architecture. With the Work Orders System, attention will be focused on developing:

- Reusable client-side JavaScript code that address the recurring one-to-many relationship between data displayed in a view.
- Robust error logging tailored to providing useful production environment error diagnostics.
- Refining interfaces between different layers to speed up application development.

Because the Work Order application is small and needs to be developed quickly, the goal is to get significant reuse from the existing base architecture. Identify everything that proved beneficial and make use of it. This document discusses the architecture of the application while pointing out things that will facilitate its rapid development.

1.1 Purpose

- Describe the architecture of this application.
- To facilitate the rapid development of the application.
- Identify the application's core components and their structure.
- Identify risks.
- Assist in the application's project management.

1.2 Scope

1.2.1 Definitions, Acronyms and Abbreviations

- AL – Application Layer
- CRUD – Create, Read, Update and Delete
- DAL – Data Access Layer
- DI – Dependency Injection
- DL – Domain Layer
- DM – Domain Model
- EF – Entity Framework

- MVC3 – The third release of ASP.NET MVC
- ORM – Object Relational Mapper
- PL – Presentation Layer
- POCO – Plain Old CLR Object
- Representation – the XML or JSON document returned by a web service during a successful response. It contains data and can also contain links to logical next states and directions on how to transition to those states.
- RL – Repository Layer
- UI – User Interface

1.2.2 References

- *"Richardson Maturity Model",* Martin Fowler, http://matinrfowler.com/articles/richardsonmaturitymodel.html
- *"REST in Practice";* J.Webber, S. Parastatidis, I. Robinson
- *"Fleet Architecture Document", DSNY-BIT*
- *"Microsoft Application Architecture Guide, Second Edition"*
- *"Domain Driven Design: Tackling Complexity in the Heart of Software",* Eric Evans
- *"Don't Create Aggregate Roots",* Udi Dahan, www.udidahan.com/2009/06/29/don't-create-aggregate-roots
- *"Programming Entity Framework",* 2nd Edition, Julia Lerman
- *"Dependency Injection in .NET",* Mark Seeman
- *"Cohesion and Coupling",* Jeremy Miller, msdn.microsoft.com/en-us/magazine/cc94717.aspx
- "*Performance Considerations for Entity Framework Applications",* msdn.microsoft.com/en-us/library/cc853327(v=vs.90).aspx

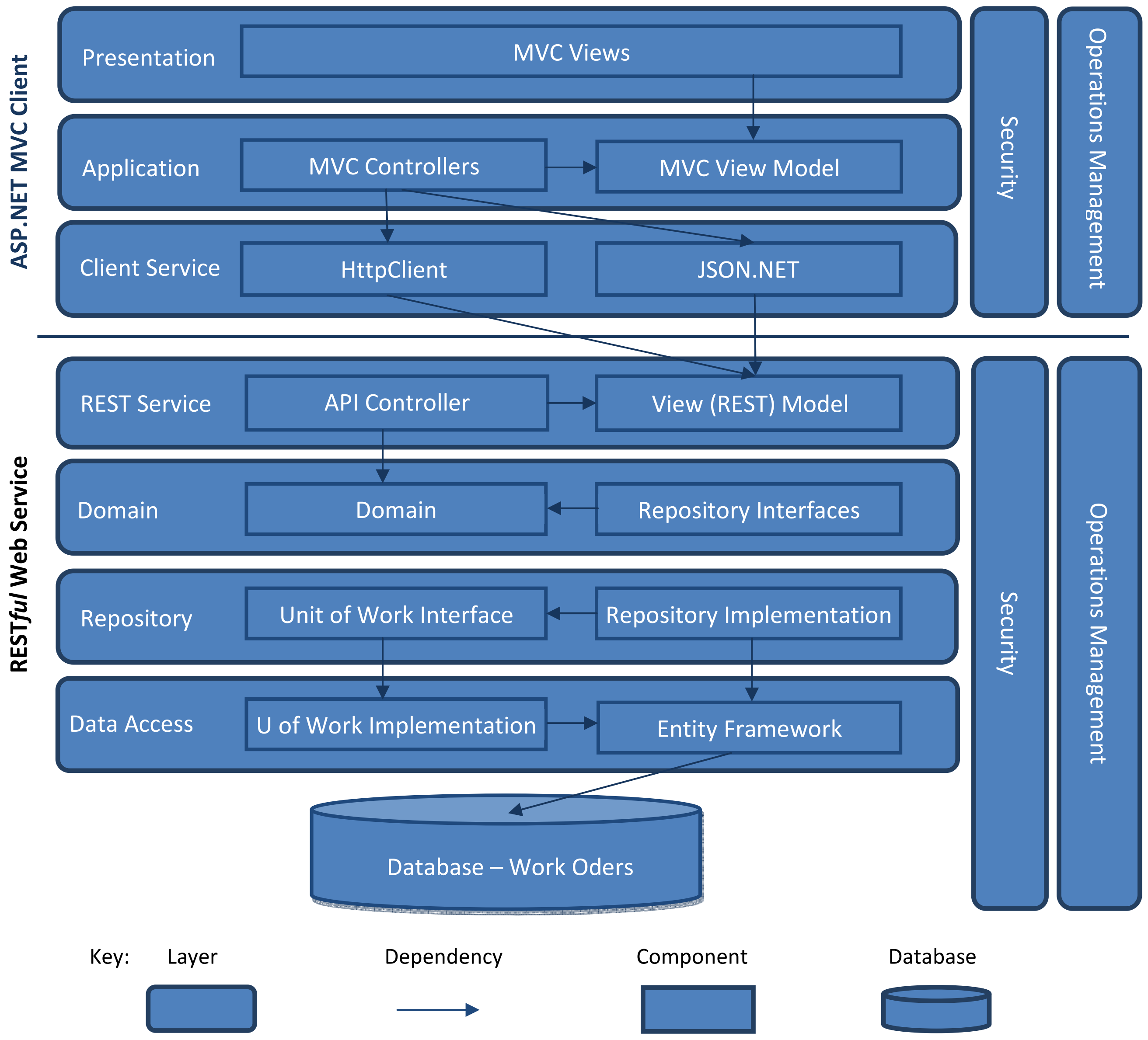
2. Architectural Solution and Representation

2.1 Overview

This section:

- Presents the architecture of the Work Orders Application.
- Discusses the rationale for the chosen architecture.
- Discusses alternatives considered and technical risks associated with implementing this architecture.

2.2 Architecture Logical Diagram



2.3 Architecture Logical Diagram Description

As shown in the Architecture LogicalDiagram, the Work Orders Application will consists of two applications: an ASP.NET MVC client application and a REST*ful* Web Service application.

ASP.NET MVC Client Application

This is a thin client that focuses on the presentation of the information obtained from the web service and packaging information to return to the web service.

The **Presentation Layer (PL)** houses Views that are sent to a web browser.

The **Application Layer (AL)** consists of ASP.NET MVC Controllers and the View Model. Views depend on View Model objects for their content. Within this architecture Views and View Models have a one-to-one relationship with each other. This design simplifies the maintenance of this layer.

Controllers are thin classes that orchestrate tasks performed during a request. They do not perform the actual work. Instead, they hand off the work to objects defined within this layer.

This layer also contains classes that translate the Representations (XML or JSON documents) returned from the web service into View Model objects. These classes also know how to extract the links contained in the Representations for use by the Views.

The **Client Service Layer (CS)** consists of libraries that know how to communication with the Web Service via HTTP/ HTTPS. It removes these concerns from the other layers of the client application.

RESTful Web Service

A RESTful application attempts to make use of a well defined interface and other built-in capabilities of the network protocol while minimizing the use of application specific features. In particular, this service will use the well defined HTTP protocol and its vocabulary of methods.

The **Domain Layer (DL)** contains the domain model that is an abstraction of the target domain. It acts as the heart of the application and it is concerned with mimicking physical business procedures. It is designed irrespective of the interface. That is, the domain model is not concerned with how the UI interacts with it.

The DL contains entities and services. Entities contain business logic and state. Services contain business logic but have no state.

The domain classes are implemented as POCOs. This decouples the domain classes from the Object Relational Mapper (ORM) and makes it possible to house the domain in its own layer. Consequently, the Domain Model (DM) is not dependent on any other layer or component in this application.

The DL also houses the repository interfaces that serve as a means for other layers to communicate with the DM in a decoupled manner.

The purpose of the **Repository Layer (RL)** is to implement the repository classes. This layer relieves the DL of all CRUD operations thus allowing the domain model to focus exclusively on modeling the domain.

This layer also contains the unit of work interface that serves as a contract depicting how ORMs are to communicate with this layer.

The **Data Access Layer (DAL)** houses the ORM, Entity Framework. This layer also contains the implementation the unit of work for the ORM.

Operational Management Layer (OML) contains the applications logging mechanism.

2.4 Layered View

2.4.1 Presentation Layer

2.4.1.1 Issues

- Develop the interface using a "Code First" approach where the interface is aggressively refined before major work starts on other parts of the application.
- Identify frequently used display patterns to develop code that can be reused in this application and in future applications.
- Design an intuitive, responsive, easy to use interface.

2.4.1.2 Solution and Rationale

Following the approach used in previous applications, the UI will be built first and continuously reviewed by the users. Such an approach should:

- Save time due to the continuous confirmation from the users pertaining to the look and feel of the application and the data being captured.
- Significantly decrease the need for later changes to the UI.

Initial analysis revealed that the application consists of many one-to-many relationships between objects on different levels. For example, a Work Request consists of many Work Orders. Work Orders consist of many Job Items. A previous application had a similar pattern that was handled by JavaScript code. That code will be enhanced into a library that can be used by different applications. (See Appendix 1 for the JavaScript code that will be modified.)

Telerik MVC Extensions will be used to develop grids and menus. (See Appendix 2 for details and examples.)

2.4.2 Application Layer

2.4.2.1 Issues

- Enable the simultaneous development of the frontend and backend.
- 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

View Models

View Models will pay a critical role in the development of this application. They will:

- Enable the UI to be populated with data without the need for a backend data source.
- Serve as the basis of the Data Model.
- Facilitate the simultaneous development of the frontend and the backend. With an agreed upon UI whose data content is housed in the View Models work can be done refining the UI while the Data Model, based on those View Models, is developed.

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.

This approach proved to simplify the development of the Vehicle view in the previous application. In that particular view many AJAX calls were used to obtain data. This approach clearly identified the data that needed to be focused on during updates.

The code snippet below provides an example. It shows the display data defined separately from the input content. The input properties are defined in the `InputViewModel` class.

```
public class CreateViewModel
{
    public IList<SelectListItem> WorkLocation { get; set; }
    public IList<SelectListItem> RequestCause { get; set; }
    public IList<SelectListItem> ResponseCategory { get; set; }

    [DisplayName("Work Req Number: ")]
    public string WorkReqNumber { get; set; }

    public InputViewModel Input { get; set; }
 }

 public class InputViewModel
 {
    [DisplayName("Shop Number: ")]
    public string ShopNumber { get; set; }

    [DisplayName("Work Location: ")]
    public int WorkLocation { get; set; }
 }
```

CreateViewModel

This approach:

- Makes the View Model class easy to understand and easier to maintain.
- Promotes re-use: If there is another View Model class with identical input properties the `InputViewModel` class can be housed in its own file and reused.
- During validation, it clearly identifies what properties need to be updated before returning the View Model to the View. In this example, all the attributes that are not contained in `InputViewModel` are to be updated.

AutoMapper: Code Reduction and Shortened Maintenance

The **AutoMapper** library will be used to promote the reduction and reuse of code written to map Domain Model objects to View Model objects and populate dropdown list. Its use in the Fleet Management project proved to significantly reduce the amount of code needed for mapping. In addition, it helps shorten the time needed for debugging since the definitions of the mappings are located in one compact source file.

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.
- The type of the input parameter being injected should always be an interface. By using an interface the implementation can be changed and the `VehicleViewBuilder` code remains the same.
- 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.

2.4.4 REST Service Layer

HTTP Mehtods Definitions

GET

Used for getting data from the server. This method should not change the state of the server.

POST

Used to update a resource or create a new resource.

2.4.3 Domain Layer

2.4.3.1 Issues

- Code reduction and promoting data integrity.

2.4.3.2 Solution and Rationale

The Domain, Repository, Data Access layers and the database are decoupled yet have overlapping functionality. Understanding and leveraging this functionality will:

- Improve the integrity of the data.
- Simplify the coding.
- Enhance the performance of the application.

This section discusses how the concept of the aggregate root fits into the architecture of the application and concludes with guidelines that can be used by our developers.

2.4.3.2.1 Aggregate Roots

The aggregate root is a concept that has proven to be very useful in the previous 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. Its the means to reference objects inside the boundary.

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

- 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.)
- 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.

These guidelines will be used to determine which entities will serve as aggregate roots. It chosen correctly programmers will find that they have all the information they need to process a request with a single retrieval from the database. In addition, when doing updates to the database because the information is being persisted from the root, the likelihood of persisting data in an invalid state is reduced. For example, the chances of persisting an orphaned child are removed.

The design of the aggregate root significantly influences the design of the repository interface classes. It will reduce the number of repository classes and promote the use of the generic repository base class.

2.4.4 Repository Layer

2.4.4.1 Issues

- Code reduction, reduction in complexity and promoting data integrity.

2.4.4.1 Solution and Rationale

- Repositories reduce the complexity of an application by encapsulating the code needed to perform CRUD operations. The solution will use a generic base class repository to handle most required features. The Unit of Work pattern helps to ensure the integrity of the data.

The following is discussed in regard to this layer:

- Data Integrity
- Code Reuse: The Generic Repository Implementation
- Loose Coupling: The Unit of Work Interface

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

- Performance: Why IQueryable<T>?

2.4.4.2.1 Data Integrity

Repositories will be designed around the concept of an aggregate root in accordance to the interfaces defined in the Domain layer.

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.

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

2.4.4.2.2 Code Reuse: Generic Repository Implementation

The design of the repository layer consists of a generic repository and a set of specialized repositories. The generic repository performs operations that are common for all repositories, such as: update, add and delete. The following code shows the implementation of the generic repository.

```
public class GenericRepository<T> : IRepository<T> where T : class
{
    private readonly ObjectContext _context;
    private IObjectSet<T> _objectSet;
    private readonly IEFUnitOfWork<ObjectContext> _unitOfWork;

    public GenericRepository(IUnitOfWork uoW)
    {
        if (uoW == null)
            throw new ArgumentNullException("GenericRepository: UoW not Provided");
        _unitOfWork = (IEFUnitOfWork<ObjectContext>)uoW;
        _context = _unitOfWork.GetContext;
    }

    public IQueryable<T> GetQuery()
    {
        return ObjectSet;
    }

    public IList<T> GetAll(Func<IQueryable<T>, IOrderedQueryable<T>> orderBy = null)
    {
        if (orderBy == null)
            return ObjectSet.ToList();
        else
            return orderBy(this.ObjectSet).ToList();
    }

    public IList<T> Find(Expression<Func<T, bool>> whereFilter, string[] include)
    {
        return include != null
            ? ObjectSet.Include<T>(include).Where(whereFilter).ToList()
```

```
                : ObjectSet.Where(whereFilter).ToList();
        }

        public T GetSingle(Expression<Func<T, bool>> expressionFilter,
                           string[] includeList)
        {
            return (includeList != null)
                ? ObjectSet
                      .Include<T>(includeList)
                      .SingleOrDefault(expressionFilter)
                : ObjectSet.SingleOrDefault(expressionFilter);
        }

        public T GetFirst(Expression<Func<T, bool>> expressionFilter,
                          string[] includeList)
        {
            return (includeList != null)
                ? ObjectSet.Include<T>(includeList).FirstOrDefault(expressionFilter)
                : ObjectSet.FirstOrDefault(expressionFilter);
        }

        public void Add(T entity)
        {
            if (entity == null)
              throw new
                  ArgumentNullException("Entity not provided for addition to Context.");
            ObjectSet.AddObject(entity);
        }

        public void Delete(T entity)
        {
            if (entity == null)
              throw new
                 ArgumentNullException("Entity not provided for deletion from Context.");
            ObjectSet.DeleteObject(entity);
        }

        public void Save()
        {
            this._context.SaveChanges();
        }

        private IObjectSet<T> ObjectSet
        {
            get
            {
                if (_objectSet == null)
                {
                    _objectSet = this._context.CreateObjectSet<T>();
                }
                return _objectSet;
            }
        }
    }
```

The GenericRepository class

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.

Minimize Data Retrieval Trips: Relevant Parameter Description

Some methods contain the **`string[] includeList`** parameter which is used to identify which objects in the object graph are to be explicitly loaded with a query.

Here are examples of the methods being used. The list of Work Request objects will contain their associated Work Order and Job Item objects. (See the section immediately below for details.)

```
        string[] includeList = { "tblWorkOrder.tblJobItem" };
        var requests = _workRequestRepository.GetQuery()
                                         .Include(includeList)
                                         .OrderBy(v => v.WorkOrderName);
```

Examples of GenericRepository method usage

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 where the root element is returned and graph objects are retrieved when required. For example, the additional retrieval would occur 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.

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.

2.4.4.2.3 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 Save();
}
```

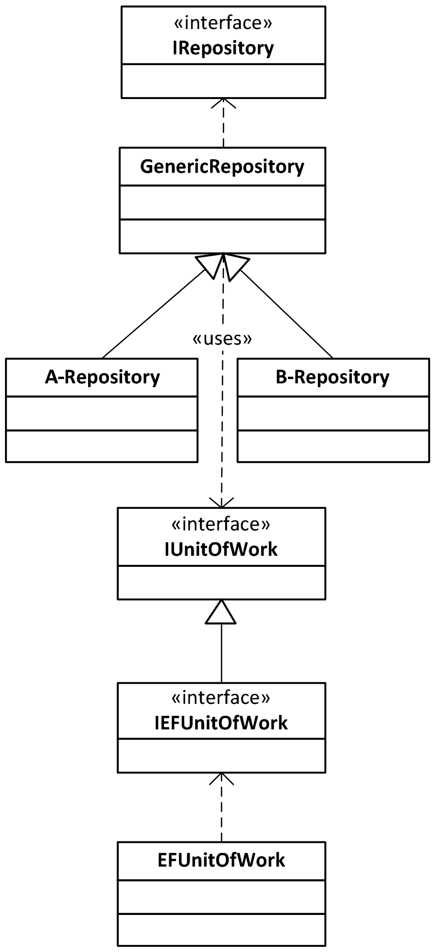
Unit of Work Base Interface

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

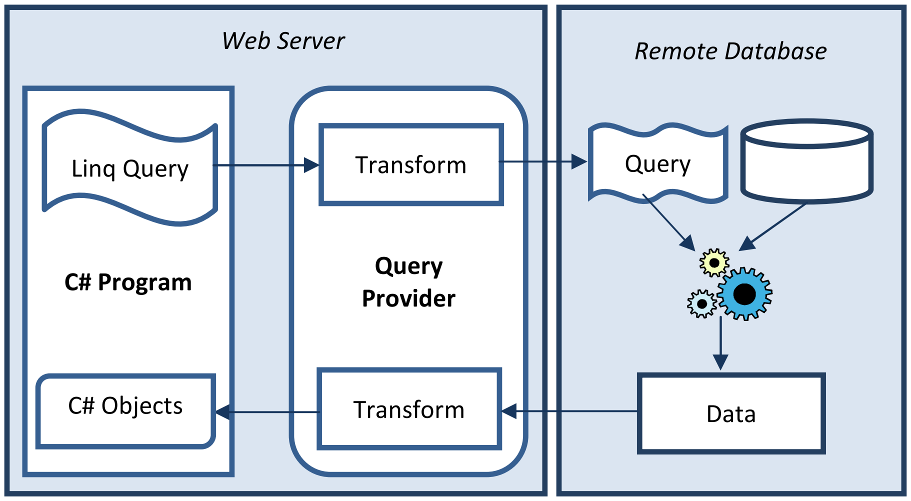


2.4.4.2.5 Performance

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.

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.
- Performance.

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

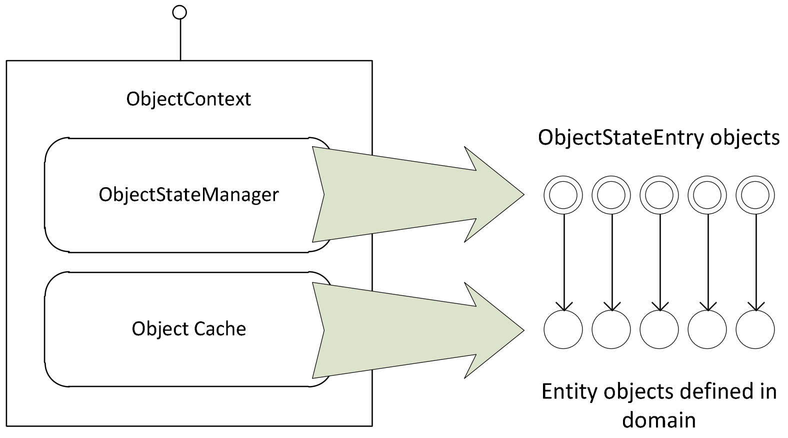
- ORMs provide the means to execute stored procedures that historically have better performance than queries generated by a data source client. However, today's ORMs perform significantly faster. In situations where performance gains are desired, tests should be run to compare query processing speeds.

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

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

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

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

        public ObjectContext GetContext
        {
                get { return _context; }
        }

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

Entity Framework implementation of unit of work

2.4.5.3.3 Using the Unit of Work

Because this architecture uses StructureMap, developers never concern themselves with obtaining a unit of work for a repository to work with. It is done automatically by the DI container.

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.

The Unit of Work Save method is used to execute the transaction:

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

2.4.6 Operational Management

2.4.6.1 Exception Management and Logging

The tracking of bugs that arise when the application is placed in the production environment is a major concern. Developers need the required amount of information to determine the cause of an exception.

Exceptions will be handled by ELMAH (Error Logging Modules and Handlers), an open source library that is widely accepted by the developer community due to its ease of use, ability to log information to various data sources and its ability to send exception notification by different means.

This library provides the following features that are of importance to this application:

- Logging of unhandled exceptions.
- Provides webpage to remotely view log of recorded exceptions.
- In some cases, view the original yellow screen.
- E-mail notification of each exception at the time it occurs.

In this application:

- All unhandled exceptions will be logged.
- Concurrency conflict will be propagated to the user for resolution.

2.5 Selection Criteria for Chosen Architecture

The criteria used to arrive at this architecture are as follows:

- Leverage existing knowledge and experience developing previous applications.
- Speed: the application must be developed quickly.
- Facilitate the isolated design and development of each layer.
- Promote reuse of code and reduce the amount of written code.
- Simplify the maintenance of the code base.
- Reduce the amount of time required to implement future enhancements.
- Ensure the integrity of the data processed by the application.
- Ensure a satisfactory response time by the application.
- Ensure exception handling and logging.
- Promote the availability of the application.

The MVC pattern has proven to be the preferred pattern for web application development. Whereas early web applications essentially returned static documents, present day browser requests actually want a set of tasks performed with every request. The result of the execution of these tasks determines the View that will be returned to the client. The MVC pattern is best suited for this scenario.

Over time the Domain Layer will experience the most change. This layers lack of dependency on other layers significantly benefits the maintenance of the application. Enhancements to business rules may change the way objects interact with each other and yet have no effect on the types of data presented to a user or obtained from the database. Put another way, a change in a business procedure does not mean there will be a change in other layers of the application.

The decoupling of layers via interfaces enables the implementation of layers to change with minimum impact on other layers.

Libraries and layers are designed with the intent of being used by other applications.

2.6 Technical Risks & Mitigation Strategy

ID	Risk Description & Impact	Mitigation Strategy and/or Contingency Plan
1	Errors related to the production environment cannot be tested in advance of the actual production role out.	This risk will be reduced by implementing the changes that were made to the Fleet Acquisition application after it was rolled out to production.
2	Development of the JavaScript library that will handle one-to-many object scenario may prove to be time consuming.	With minimal change different versions of the code could handle particular one-to-many scenarios.
3	Project assignments. It's a small application so the likelihood one programmers work interfering with another programs work increases. This can result in lost time due to work having to be reproduced.	Daily Standup meetings where these concerns are discussed. Daily check-ins to TFS.

3. Enterprise Architecture Models

3.1 As Found State

The Work Order application presently consists of an incomplete Microsoft Access application. Taking this fact into consideration, this Work Order application is based on an architecture that does not take the architecture of the present system into consideration.

4. Architectural Goals and Constraints

- To come up with architecture that facilitates the rapid development with a design that simplifies future maintenance.

4.1 Technology

- Visual Studio 2010
- SQL Server 2008
- SQL Server Reporting Services
- .NET Framework 4
- ASP.NET MVC 3
- Entity Framework 4.1
- C#
- StructureMap

- AutoMapper
- jQuery/JavaScript
- Telerik extensions for ASP.NET MVC (Now a part of Kendo UI)
- HTML5/CSS3
- ELMAH
- IIS 7

4.2 Production Platform

- .NET Framework 4
- ASP.NET MVC 3
- IIS 7
- SQL Server 2008 R2

4.3 Training

- .Net framework 4
- ASP.NET MVC 3
- IIS 7
- SQL Server 2008 R2

4.4 QA Platform

- .Net framework 4
- MVC framework 3
- IIS 7
- SQL Server 2008 R2

4.5 Development Platform

- Visual Studio 2010
- SQL Server 2008
- SQL Server Reporting Services
- .NET Framework 4
- ASP.NET MVC 3
- Entity Framework 4.1
- C#
- StructureMap
- AutoMapper
- jQuery/JavaScript
- HTML5/CSS3
- ELMAH
- Telerik extensions for ASP.NET MVC (Now a part of Kendo UI)

5. Non Functional Requirements

5.1 Reliability

Reliability ensures the integrity and consistency of the application and all its transactions.[3]

Reliability as applied to the Work Order Application relates to the following issues:

- Ensure that database transactions are saved correctly. (See DataAccess Layer Interface for how this is addressed.)
- Ensuring the data entered by users is valid.

5.2 Availability

Availability ensures that a service/resource is always accessible.[4]

All components should be available 24/7.

5.3 Scalability

The Work Order application is a three tier application. Any gains due to scalability will be realized through the use of additional memory processor cycles, and/or bandwidth on the webserver. These issues will be addressed by DOITT.

5.4 Performance

In order to improve performance following techniques will be implemented.

- Apply techniques used by high performance websites.
- Utilize MVC3 feature that enables views to be pre-compile.
- Caching of data used in dropdown list boxes.
- Application will use database connection pooling.
- Hardware tuning will be addressed by DevOps.
- Long running code will be performed asynchronously.
- Database will be tuned by identifying potential indexes, use of SQL profiler's index tuning wizard, query plans will be analyzed using query analyzer to optimize queries.
- Stored procedures will be used for fetching Report data.

5.4.1 Caching Strategy

A design goal of this application is to provide the most current information to a user. Consequently, the caching of data will be administered on an "as needed" basis and will not be applied unless all other options do not provide a satisfactory response time.

[3] Architecture Document Template.
[4] Architecture Document Template.

5.5 Extensibility & Maintainability

The application will be developed using the following design principle and patterns:

- Code to an interface not an implementation: Classes should be designed to meet the contract of an interface. These interfaces are then used as the types of method parameters.
- Loose Coupling/Dependency Injection (DI): Using interfaces as parameter types enables different implementations of the interface to be used by a class without needing to modify that class. It also facilitates isolated unit testing since a mock implementation can be provided to the class being tested. When interfaces are used as parameter types and dependencies are injected into a class, a DI Container can be used to "wire up" the application from a centralized location known as the Composite Root (CR). The CR has the effect of having any change in the implementation of an interface to propagate through the entire application.
- Open-Close Principle: A class should be designed so that it is open for extension and closed for modification. Portions of code that is prone to change should be placed in separate classes that meet the contract of an interface. These objects are then injected (DI) into the objects that use them.
- Single Responsibility Principle: Classes that have many responsibilities should be reviewed to determine which responsibilities are prone to change. Those responsibilities should be removed from the main class and placed in their own class that meets the contract of an interface.
- Aggregate Roots. (See Aggregate Roots for details.)

5.6 Persistence

The Work Order application will use Entity Framework and SQL Server database to implement its data persistence layer.

5.7 Manageability

The *Teams Foundation Server* will be used to promote an organized means for the long term enhancement of this application certain features of. In addition, certain developer best practices related to *Teams* will be identified, documented and implemented. These practices will enhance the integrity of the code base and reduce confusion between developers working on this application.

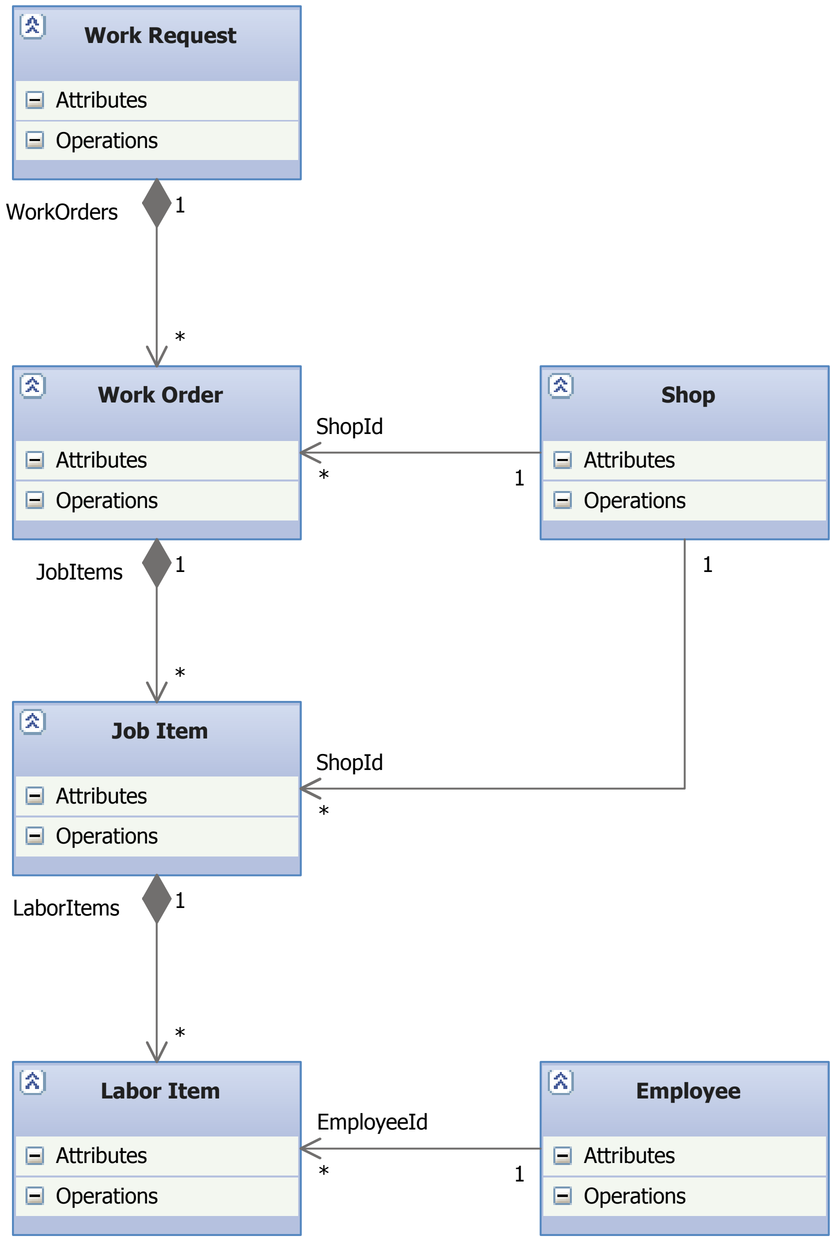
6. Non-Functional Check List

Work in progress. Will be documented upon completion.

7. UML Diagrams

7.1 Class Diagram

The diagram below shows the primary classes the Work Order application will consist of.



7.2 Entity Relationship Diagram

Work in progress. Will be documented upon completion.

7.3 Sequence Diagram

Work in progress. Will be documented upon completion.

8. Implementation View

8.1 Design Patterns

Pattern Name	**Where Used**
Repository	For data access.
Unit of Work	Persisting of data and providing repositories with a single context.
Dependency Injection (DI)	Managed by StructureMap Container and implemented throughout the application.
MVC	Used in the Presentation and Application Layers
Factory	Used to create valid domain objects.

9. Deployment View

The Work Order Application will consist of dynamic linked libraries that will be published to the DevOps Web Server. It is depicted below as the *Work Order Solution Package* that will contain Presentation, Domain, Repository and DataAccess components (dlls) along with other application support dlls.

9.1 Environment Details

9.1.1 Server Side

ID	**Hardware/software**	**Details (e.g. Ports, Setup, Certificated)**
1	External firewall	DevOps firewall
2	Internal firewall	None
3	Load balancer	None
4	Web Server	IIS 7 Servers
5	Application Server	None

ID	Hardware/software	Details (e.g. Ports, Setup, Certificated)
6	Portal Server	None
7	Database Server	Production: Prod Server Development: Dev Server

9.1.2 Client Side

ID	Hardware/Software	Details
1	Browser	Web browsers

9.2 Production Roll out Strategy

Work in progress. Will be documented upon completion.

Appendix 1

As-is source code that handles one-to-many relationship between objects.

```
// Initialization section ------------------------

var rowCnt = 0;
var editRow;
var action = "A";
var applyAll = new Array();
var selectDebit = new Array();
var templateKey = new Array();
var rowToDelete;

// Function section ------------------------------

function ValidateEntries() {
    var sMessage = "The information could not be applied for the following reason(s):
<br\><br\>";
    var validationError = false;
    if ($jq("#SelectedDebitType").val() == 0) {
        sMessage = sMessage + "- A Debit Type must be selected. <br\>";
        validationError = true;
    }

    if ($jq("#DebitAmount").val() == 0 || $jq("#DebitAmount").length == 0) {
        sMessage = sMessage + "- A Debit Amount is required.<br\>";
        validationError = true;
    } else {
        if (!ValidateCurrency($jq("#DebitAmount").val())) {
            sMessage = sMessage + "- The currency is formatted incorrectly.<br\>";
            validationError = true;
        }
    }

    if ($jq("#Description").val() == "") {
        sMessage = sMessage + "- A Description is required.";
        validationError = true;
    }

    if (validationError == true) {
        $jq("#ErrorMessageDialogTxt").html(sMessage);
        $jq("#ErrorMessageDialog").dialog('open');
        return false;
    }
    else {
        return true;
    }
}

function ValidateCurrency(amt) {
    if(/^\$?\-?([1-9]{1}[0-9]{0,2}(\,\d{3})*(\.\d{2})?|[1-
9]{1}\d{0,}(\.\d{2})?|0(\.\d{2})?|(\.\d{2}))$|^\-?\$?([1-
9]{1}\d{0,2}(\,\d{3})*(\.\d{2})?|[1-
9]{1}\d{0,}(\.\d{2})?|0(\.\d{2})?|(\.\d{2}))$|^\(\$?([1-
9]{1}\d{0,2}(\,\d{3})*(\.\d{2})?|[1-
9]{1}\d{0,}(\.\d{0,2})?|0(\.\d{2})?|(\.\d{2}))\)$/.test(amt))
        return true;
    else {
        return false;
    }
}

function FormatCurrencyForTableDisplay(obj) {
    var amt = new String(obj);
    var pos = 0;
    amt = amt.replace("$", "");
    amt = amt.replace("-", "");
    if (amt.indexOf(".") == -1)
        amt = amt + ".00";
    if (amt.length > 6 && amt.indexOf(",") == -1) {
         pos = amt.indexOf(".");
         amt = amt.substr(0, amt.length - 6) + "," + amt.substr(pos - 3);
         if (amt.length > 10) {
             pos = amt.indexOf(".");
             amt = amt.substr(0, amt.length - 10) + "," + amt.substr(pos - 7);
         }
         if (amt.length > 14) {
             pos = amt.indexOf(".");
             amt = amt.substr(0, amt.length - 14) + "," + amt.substr(pos - 11);
         }
     }
     amt = "($" + amt + ")";

    return amt;
}

function SetupDebitSecurityTemplateArrays() {
    var cnt = 0;
    var pos = 0;
    var keys = new String($jq("#TemplateIds").val());
    var ok = true;
    do {
        cnt++;
        pos = keys.indexOf("*");
        templateKey[cnt] = keys.substr(0, pos);
        if (pos + 1 == keys.length)
            ok = false;
        else
            keys = keys.substr(pos + 1);
    } while (ok);
    var templateIdCount = cnt;
    cnt = 0;
    pos = 0;
    ok = true;
    keys = new String($jq("#TemplateDebitTypeIds").val());
    do {
        cnt++;
        pos = keys.indexOf("*");
        selectDebit[cnt] = keys.substr(0, pos);
        if (pos + 1 == keys.length)
            ok = false;
        else
            keys = keys.substr(pos + 1);
    } while (ok)
    rowCnt = cnt;

    cnt = 0;
    pos = 0;
    ok = true;
    keys = new String($jq("#ApplyAll").val());
    do {
        cnt++;
        pos = keys.indexOf("*");
        applyAll[cnt] = keys.substr(0, pos);
        if (pos + 1 == keys.length)
            ok = false;
        else
            keys = keys.substr(pos + 1);
    } while (ok)

    if (templateIdCount != rowCnt) {
        $jq("#ErrorMessageDialogTxt").html("The Security Debit Templates were not set up
correctly. This can lead to the corruption of your entries. <br/><br/>Contact the System
Administrator.");
        $jq("#ErrorMessageDialog").dialog('open');
    }
    return;
}

function ResetFields() {
    $jq("#SelectedDebitType").val(0);
    $jq("#DebitAmount").val("");
    $jq("#Description").val("");
    $jq("#ApplyDebit").attr("checked", false);
}

function ShowAcceptanceLetterView() {
    $jq("#DebitDiv").hide();
    $jq("#AcceptanceDiv").show();
}

function ShowSecurityDebitView() {
    $jq("#AcceptanceDiv").hide();
    $jq("#DebitDiv").show();
}

function ConvertCurrencyToDecimal(obj) {
    var IsDebit = false;
    obj = new String(obj);

    return obj =  "-" + obj.replace("-", "").replace("$", "").replace("(",
"").replace(/\,/g, "").replace(")", "");
}

function ProcessDebitTbl() {
    var rcnt = 1;
    var sStr = "";
    var rowType = "";
    $jq("#SecurityDebitTbl tr[class] td").each(function () {
        var row = new String($jq(this).parents("tr").attr("id")).substr(1);
        if (rcnt == 1) {
            row = new String($jq(this).parents("tr").attr("id")).substr(1);
            rowType = $jq(this).parents("tr").attr("class");
            sStr = sStr + rowType + "|";
        }
        if (rcnt == 2) {
            if (rowType == "t")
                sStr = sStr + templateKey[row] + "|0|";
            else
                sStr = sStr + "0|" + applyAll[row] + "|";
        }
        if (rcnt == 3) sStr = sStr + selectDebit[row] + "|";
        if (rcnt == 4) sStr = sStr + ConvertCurrencyToDecimal($jq(this).text()) + "|";
        if (rcnt == 5) {
            sStr = sStr + $jq(this).text() + "~";
            rcnt = 0;
        }
        rcnt++;
    });
    $jq("#SecDebitResponse").val(sStr);
    $jq("#theForm").submit();
    return;
}

$jq(document).ready(function () {

    // Event handlers ----------------------------------

    $jq("#CreateBtn").click(function (event) {
        event.preventDefault();
        ProcessDebitTbl();
    });

    $jq("#SaveBtn").click(function (event) {
        event.preventDefault();
        ProcessDebitTbl();
    });

    $jq("#AddDebitBtn").click(function () {
        action = "A";
        ResetFields();
        $jq("#ApplyToNotice").show();
        ShowSecurityDebitView();
    });

    $jq("#ShowAcceptBtn").click(function () {
        ShowAcceptanceLetterView();
    });

    $jq("#BackToAcceptance").click(function () {
        ShowAcceptanceLetterView();
    });

    $jq("#CreateDebitBtn").click(function () {
        if (!ValidateEntries())
            return;
        if (action == "A") {
            rowCnt++;
            if ($jq.browser.msie) {
                if ($jq("#ApplyDebit:checked").val() != undefined)
                    applyAll[rowCnt] = "Y";
                else
                    applyAll[rowCnt] = "N";
            }
            else {
                if ($jq("#ApplyDebit").attr("checked") == true)
                    applyAll[rowCnt] = "Y";
                else
                    applyAll[rowCnt] = "N";
            }
            selectDebit[rowCnt] = $jq("#SelectedDebitType").val();
            var classValue = "x";
            if ($jq("#PageType").attr("value") == 'e')
                classValue = "n";
            $jq("#SecurityDebitTbl").append('<tr class="' + classValue + '" id="N' +
rowCnt + '">' +
                '<td class="td-childtable e"><span class="aEdit">Edit</span></td>' +
                '<td class="td-childtable r"><span class="aRemove">Remove</span></td>' +
                '<td class="td-childtable">' + $jq("#SelectedDebitType
option:selected").text() + '</td>' +
                '<td class="td-childtable">' +
FormatCurrencyForTableDisplay($jq("#DebitAmount").val()) + '</td>' +
                '<td class="td-childtable">' + $jq("#Description").val() + '</td></tr>');
        } else {
            if (action == "E") {
                rowCnt = 1;
                var idattr = new String(editRow.attr("id"));
                if ($jq.browser.msie) {
                    if ($jq("#ApplyDebit:checked").val() != undefined)
                        applyAll[idattr.substring(1)] = "Y";
                    else
                        applyAll[idattr.substring(1)] = "N";
                }
                else {
                    if ($jq("#ApplyDebit").attr("checked") == true)
                        applyAll[idattr.substring(1)] = "Y";
                    else
                        applyAll[idattr.substring(1)] = "N";
                }
                selectDebit[idattr.substring(1)] = $jq("#SelectedDebitType").val();
                editRow.children().each(function () {
                    if (rowCnt > 2) {
                        switch (rowCnt) {
                            case 3:
                                $jq(this).text($jq("#SelectedDebitType
option:selected").text());
                                break;
                            case 4:

$jq(this).text(FormatCurrencyForTableDisplay($jq("#DebitAmount").val()));
                                break;
                            case 5:
                                $jq(this).text($jq("#Description").val());
                                break;
                        }
                    }
                    rowCnt++;
                });
                action = "A";
            } else {
                alert("Unable to determine action to be taken.");
            }
        }
        $jq("#DebitDiv").hide();
        $jq("#NoSecurityDebits").hide();
        ResetFields();
        $jq("#SecurityDebitTblDiv").show();
        $jq("#AcceptanceDiv").show();
    });

    // Process a click on the table.
    $jq("#SecurityDebitTbl").click(function (event) {
        if ($jq(event.target).parents("td").attr("class") == "td-childtable r") {
            rowToDelete = event.target;
            $jq("#RemoveDebitDialog").dialog('open');
        }

        if ($jq(event.target).parents("td").attr("class") == 'td-childtable e') {
            var cnt = 1;

            if ($jq(event.target).parents("tr").attr("class") == 't' ||
($jq(event.target).parents("tr").attr("class") == 'x' && $jq("#PageType").val() == 'e'))
                $jq("#ApplyToNotice").hide();
            else
                $jq("#ApplyToNotice").show();

            var idStr = new String($jq(event.target).parents("tr").attr("id"));
            if (applyAll[idStr.substring(1)] == "Y") {
                if ($jq.browser.msie)
                    $jq("#ApplyDebit").attr("checked", "checked");
                else
                    $jq("#ApplyDebit").attr("checked", true);
            } else {
                if ($jq.browser.msie)
                    $jq("#ApplyDebit").removeAttr("checked");
                else
                    $jq("#ApplyDebit").attr("checked", false);
            }
            $jq(event.target).parents("tr").children().each(function () {
                if (cnt > 2) {
                    switch (cnt) {
                        case 3:

$jq("#SelectedDebitType").val(selectDebit[idStr.substring(1)]);
                            break;
                        case 4:
                            var tmp = new String($jq(this).text());
                            if ($jq("#IsMod").val() == "true") {
                                if (tmp.substr(0,1) == "(")
                                    $jq("#DebitAmount").attr("value", "-" +
tmp.substring(2, tmp.length - 1));
                                else
                                    $jq("#DebitAmount").attr("value", tmp.substring(1));
                            }
                            else
                                $jq("#DebitAmount").attr("value", tmp.substring(2,
tmp.length - 1));
                            break;
                        case 5:
                            $jq("#Description").attr("value", $jq(this).text());
                            break;
                    }
                }
                cnt++;
            });

            editRow = $jq(event.target).parents("tr");
            action = "E";
            ShowSecurityDebitView();
        }
    });

    $jq("#RemoveDebitDialog").dialog({
        autoOpen: false,
        title: 'Confirmation Required',
        width: 350,
        height: 160,
        modal: true,
        resizable: false,
        buttons: {
            "No": function () { $jq(this).dialog("close"); },
            "Yes": function () {
                $jq(rowToDelete).parents("tr").remove();
                if ($jq('#SecurityDebitTbl td').val() == undefined) {
                    $jq("#SecurityDebitTblDiv").hide();
                    $jq("#NoSecurityDebits").show();
                }
                $jq(this).dialog("close");
            }
        }
    });
    // Page setup -------------------------------------

    $jq("#DebitDiv").hide();

    $jq("#ErrorMessageDialog").dialog({
        autoOpen: false,
        title: "Input Error",
        modal: true,
        width: 445,
        buttons: { "Ok": function () { $jq(this).dialog("close"); } }
    });

    if ($jq("#SecurityDebitItems").val() == "True") {
        SetupDebitSecurityTemplateArrays();
        $jq("#NoSecurityDebits").hide();
        $jq("#SecurityDebitTblDiv").show();
    }
    else {
        $jq("#SecurityDebitTblDiv").hide();
        $jq("#NoSecurityDebits").show();
    }
});
```

Appendix 2

REMOVED