## Layer Design

### Data Access Layer (DAL)

The data access layer consists of core contracts in the JBRICK.Dal project. Included in this project are repository contracts, Dtos and unit of work contracts. ADO.NET will be the data access technology used in the repository implementations. To support this, an implementation of the Unit of Work contracts were created and live in the project JBRICK.Dal.Sql. The actual data access logic (Repositories) will reside in the specific application projects under the namespace JBRICK.Dal.Sql.{AppName}.

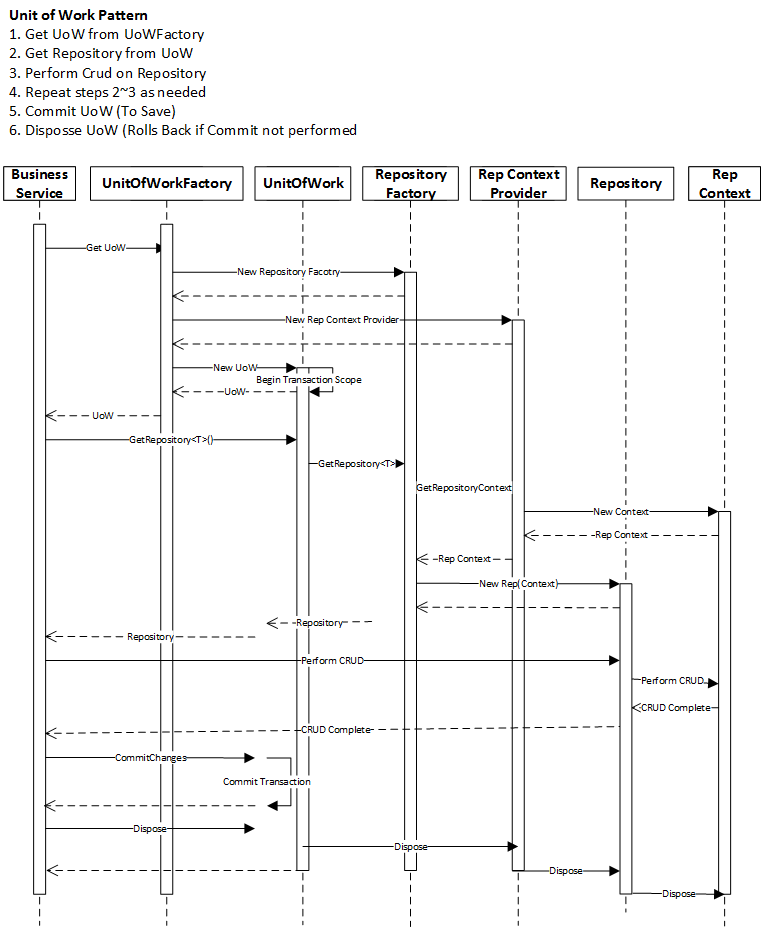
#### Repository Pattern

The Repository pattern is used to isolate the data layer allowing for testable code and decoupling of the data logic from the rest of the system. Typically you will create one repository class for each business entity. Dto objects should be used to pass entity data between the data layer and the business layer. Each repository must have a defined contract and Dto(s) that will reside in the JBRICK.Dal project. The actual repository implementation will reside in a specific project in the JBRICK.Sql namespace such as JBRICK.Sql.EquipmentGuide. A repository class must have a constructor with the following signature as it will be created by a factory object which will inject these dependencies:

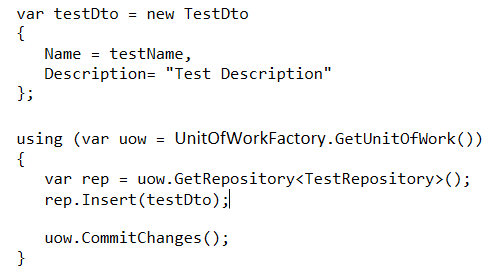
public RepositoryMockWithValidConstructor(IRepositoryContextProvider<IDbConnection> context, ILogger logger)

#### Unit of Work Pattern

The Unit of Work pattern is used to manage business transactions and the data connection context. It accomplishes this by wrapping the data access calls in a transaction and managing the connection string instances required by the repositories. Below is a sequence diagram of the Unit of Work framework components and a code example that would be used in a business service to insert data.



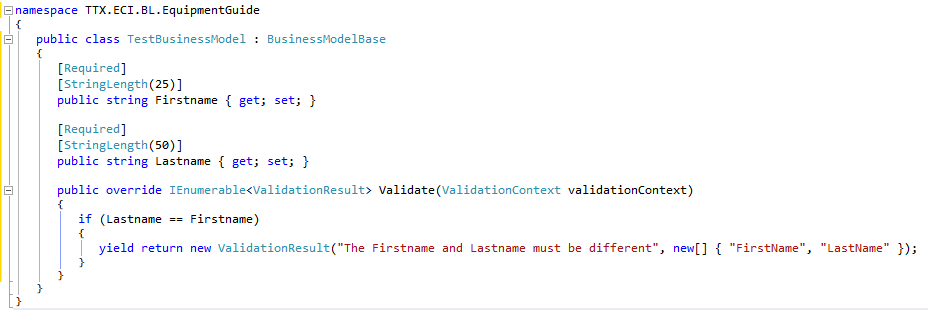
Example Business Service



### Business Layer (BL)

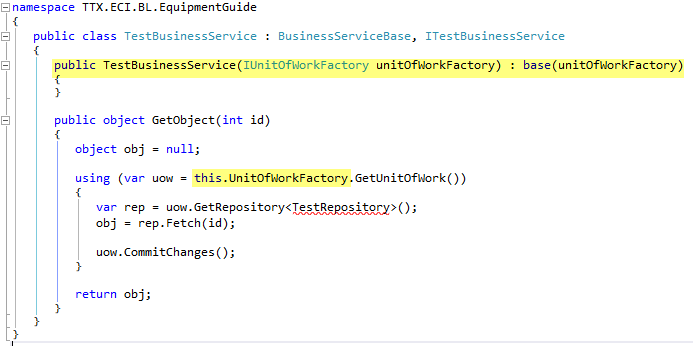
The business layer contains business services and business models. All business services and models must implement a contract defined in the JBRICK.Contracts project. Clients of the business layer should reference the business services by contract and use some form of dependency injection to retrieve instances. Business models must implement the BusinessModelBase class which provides base functionality including data validation support. Microsoft Data Annotations should be used as the main means of business model data validation. If custom validation is required that is not provided by existing data annotations then custom data annotations can be created and reused or business model specific validation logic can be implemented by overriding the base classes Validate method. The base class implements the IValidateableObject interface allowing you to provide custom validation logic through the Validate Method. Both the Data Annotations and the IValidateableObject.Validate implementations are supported by the MVC framework providing out of the box client side validation.

Business model Data Annotations and custom validation example



Even though client side validation will be performed against business models, the business service shouldn’t trust the caller and should always execute business model validation as well. To support this, the BusinessModelBase also provides a ValidateModel method that executes validation checks against both the data annotations and the IValidateableObject.Validate method of the business model.

As explained in the Data Access Layer section, the business services will access the DAL using the unit of work pattern. To support this, the BusinessServiceBase class contains an IUnitOfWorkFactory property and a constructor that accepts an instance of the factory object. When creating your business service you should create a constructor that accepts the IUnitOfWorkFactory passing it to the base constructor which sets the property.



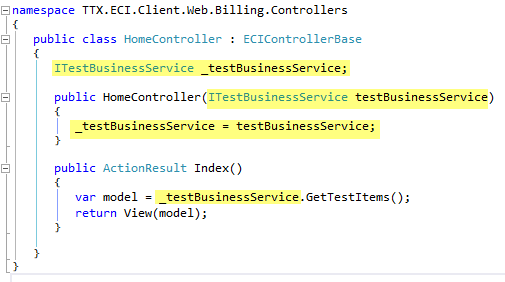
It is the responsibility of the unit of work factory object to create unit of work instances when needed by the business service. When the unit of work instance is created, it starts a transaction using the System.Transactions.TransactionScope. When you are finished calling all required data logic you must call the CommitChanges method to commit the changes. When finished using the unit of work instance you must call its dispose method ensuring all resources, including the Sql transaction and connections, are disposed of. If the unit of work instance is disposed without calling the CommitChanges then the transaction is rolled back and no changes are saved to the database. You should follow the above example of wrapping the unit of work in a using statement ensuring it is disposed correctly. The unit of work object is also used by the business service to get repository instances to execute data logic. Behind the scenes it uses a repository factory object to create the repository instances passing in a repository context provider that manages the Sql connections used by the repositories. This ensures only one instance of each required connection is used in the business transaction.

### Presentation Layer

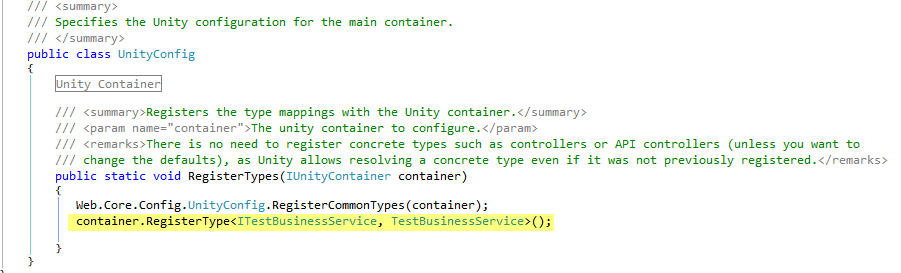
MVC web applications are used in the presentation layer of the ECI applications. This means the presentation layer consists of views, controllers and models. The models passed to the view by the controller can either be business models supplied by a business service in the business layer or a view model, located in the Models folder of the MVC application that will be populated by the controller and passed to the view. If the model required by a view is a single business model then the controller can retrieve it form a business service and pass it to the view. But if the view requires multiple business models or other view specific properties or lists then the controller can retrieve one or more models from a business service and wrap the data in a view model class. An example of this is if a view needs multiple lookup lists as well as a specific business model.

When the application is started an IoC container is registered as the Dependency Resolver that will be used by the MVC framework when it needs to create MVC classes such as controllers, views and filter attributes. This provides controller dependencies, such as business services, to be injected using dependency injection. By providing controller dependencies is this fashion allows the controllers to be tested in isolation and as well as being able to easily change the implementation of the dependencies in one place, the unity container registration. All external controller dependencies should be injected in this manner including the business services used by the controller. The dependencies then need to be registered in the IoC container.

Business service injected into controller constructor



Business service registered with IoC container



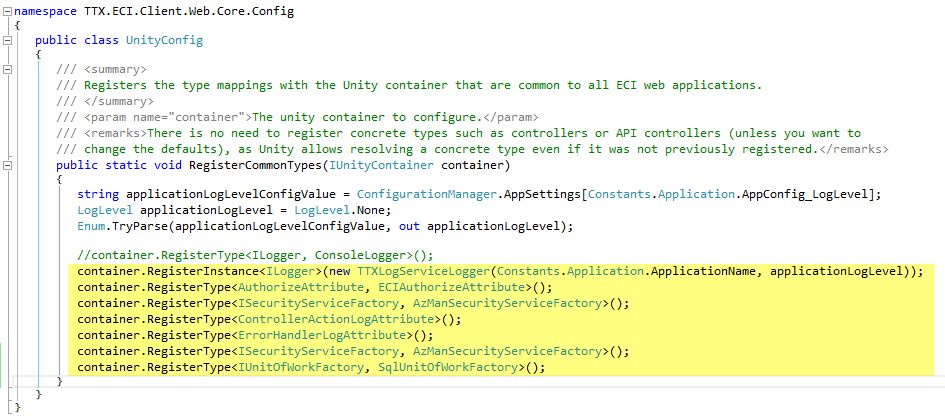
Because the business service instance will be created by the IoC container all of its dependencies will also be resolved by the IoC container. As mentioned earlier, all business services should use a unit of work factory to communicate with the data layer. This dependency should be in its constructor allowing it to be resolved by the IoC container along with all of its dependencies.

Because the ECI solutions consists of multiple MVC web applications all reusable base functionality is located in the JBRICK.Client.Web.Core project. This project contains items such as filter attributes, a base controller and base configuration settings. Controllers should inherit the ECIControllerBase which provides base functionality. Currently this is a Logger property that gets populated automatically by the IoC when the controller instance is created. There are three filters provided in this project that are globally registered in each of the web applications. They are:

* **ECIAuthorizeAttribute** – This attribute checks that the user is authenticated. It also provides the ability to check role and right authorization. The globally registered filter just ensures the user is authentication. The attribute then can be added to any controller or controller action with rights and roles that are required.
* **ErrorHandlerLogAttribute** – This attribute is globally registered to handle all unhandled exceptions, logging them then displaying the error view. This means that you only need to use a try catch block in your controller if you specifically need to handle the exception for other reasons then just logging the exception.
* **ControllerActionLogAttribute** – This attribute is performs debug level logging of all controller action interaction. The application logging level must be set to the Debug level in order for these logs to be stored.

Another area of common functionality provided by this project is common configuration used by all of the web applications. For example, common IoC registration and global filter registration is configured here and is referenced by all web applications. For example, notice that all IoC dependencies are registered here that are common to all ECI web applications.

Common IoC registrations



## Dependency Injection

The ECI solution uses Microsoft Unity as its dependency injection container. The IoC container dependency registration will occur at application startup. Constructor injection will be the primary means of supplying dependencies to objects. A Unity dependency resolver is configured in the MVC applications allowing the Unity IoC container to be used to resolve dependencies when creating MVC objects such as filters, views and especially controllers. When designing your controllers, any external dependencies, such as business services, should be added to the constructor and registered with the IoC container allowing them to be created and injected by Unity when the controller is created. This will ensure your dependencies are decoupled properly allowing isolated unit testing of the controllers.

## Security

All security checks within the application will be performed against these interfaces ensuring that the security implementation can be changed in the future with minimal impact.

### Authentication

Authentication is the process of confirming the identity of the user and establishing a principal object for that user. This application will use Windows authentication provided by IIS.

### Authorization

Roles based authorization will be used to give access to different features of the application. The roles of the application and their corresponding rights within the application will be identified and documented in the technical requirements. TTX will manage the application security roles using AzMan allowing flexibility in managing individual user rights for the application using an environment they are accustom to.

In order to allow the implementation of retrieving the roles for an authenticated user to be easily updated without affecting the rest of the application, the ECI Principal class will perform all role access through an IRoleProvider contract. A provider will be injected into the custom principle object allowing the implementation to easily be updated without affecting the rest of the application. An AzMan security role provider will be implemented at the time of this writing.

Role Provider Injection



### Application Security

Page level security checks needs to be performed for each view of the application to ensure that a user cannot bypass the security implemented in the SharePoint navigation. All other application security requirements will be captured and documents in the requirement documentation.

### Document Retrieval Security

All document such as drawing and specification sheets will be accessed through a single entry point. Currently this entry point is provided by dwgsecurity.asp page. The security logic in this entry point must be updated per the application security roles matrix documented in a separate requirements documentation.

## Logging

All logging within the ECI applications will be performed against a logging contract. An IoC container will be used to provide a concrete implementation. This will allow the logging provider to easily be changed without affecting the rest of the application. For this project, a logging provider will be created that wraps a global logging service hosted by TTX. Here is the logging contract.

/// <summary>

/// An interface that any logging provider can implement

/// </summary>

public interface ILogger

{

void Debug(string message);

void Info(string message);

void Warn(string message);

void Error(string message);

void Error(string message, Exception x);

void Error(Exception x);

void Fatal(string message);

void Fatal(string message, Exception x);

void Fatal(Exception x);

}

Notice that the contract provides the ability to log information at different levels. This allows the log message to be stored with a severity attribute to help with diagnostics and reporting. The logging implementation also accepts a log level attribute to allow the application to be configured to only log messages up to a specific severity. This setting is stored in the web config and used when creating the logging provider. Here is the log levels and their corresponding severity levels. Note that whatever value is passed into the logging provider then it will only log that level and all levels below it. For example, if we pass a log level of “LogLevel.Error” to the logging provider when creating an instance then only Error & Fatal log calls will be stored.

/// <summary>

/// Enumeration to indicate logging severity

/// </summary>

public enum LogLevel

{

/// <summary>

/// With None level Logging will not occur.

/// </summary>

None = 0,

/// <summary>

/// Fatal level is used to log fatal exceptions such as unhandled exceptions

/// </summary>

Fatal = 1,

/// <summary>

/// Error level is used to log all exceptions

/// </summary>

Error = 2,

/// <summary>

/// Warn level is used to log application warning information such as if a configuration settings is missing

/// </summary>

Warn = 3,

/// <summary>

/// Info level is used to log audit or application management type information such as when someone logs into the systems

/// </summary>

Info = 4,

/// <summary>

/// Debug level is used to log debugging information

/// </summary>

Debug = 5

}