# Applications Features

* Dependency Injection - Autofac
* Logging - NLog
* Domain Driven Application – ie uses d domain objects to articulate the business rules
* Allows for growth of application features
* Mocking Framework – Moq
* Testing Framework - Nunit

# Assumptions

* This will only need to run on a medium traffic environment. If this application was required to run on a high traffic environment then less structure would have been implemented
* Importance has been placed on Maintenance – Readability of code is a key consideration
* Although this is a small application I wanted to showcase how I would structure a larger application.

# Explainations

I put together a FAQ to explain some to the decisions that I have made during this application

## What is the approach to TDD?

The goal for conducting TDD on this project is to **test the business rule in isolation not the code**. With this in mind not every class will have a test, a test can cover a number of classes as long as the business value is covered. To facilitate this, the public interface for TDD is the GRM.Service project, it can use as many of the objects in GRM.Domain, GRM.Repository and GRM.ApplicationInfrastructure projects necessary to cover the business value. Mocking is kept to a minimum (repositories only ) as too much mocking exposes underlying implementation which makes the code brittle to change

## How does the project structure work?

The naming convention that I have used is a standard template that I use on all new applications

The project names have been carefully chosen to represent the application hierarchy and also represent the direction of data flows in the alphabetically structure of folder and in the solution explorer.

|  |  |
| --- | --- |
|  | GRM.ApplicationInfrastructure  GRM.Domain  GRM.Repository  GRM.Service  GRM.UI |

|  |  |
| --- | --- |
| GRM.ApplicationInfrastructure | This project holds the library’s for all the cross cutting concerns that are used by all other projects in this solution, e.g. string functions, xml functions, email clients, logging libraries etc. |
| GRM.Domain | These are the domain objects, in some case they are POCO’s but they are also used to house some domain business logic. Test coverage should be as high as possible for this project. |
| GRM.Repository | These are the libraries that connect to data stores eg databases, XML files, NoSQL data stores etc. Theses libraries are often mocked out when doing TDD. Test coverage of this project is mainly covered integration tests as its main purpose is to interact with external sources and it should hold no business logic. |
| GRM.Service | These are libraries in house business logic that does not fit neatly into the domain and also can act as central orgestrator of logic. Test coverage should be as high as possible for this project. It is the public interface for TDD as it is the interface for the business logic. |
| GRM.UI.Console | Anything that is GRM.UI.\* is a front end concern. It leverages services and pushes that data to the front end. There should be as little logic in this project as possible. Unit Test coverage is low as the functionality should be covered by the “End to End” Tests. This can be extended to as many UI’s as needed  Eg  GRM.UI.Website  GRM.UI.WebsiteAdmin  GRM.UI.CustomerApi  The naming convention keeps the UI projects together, and they are free to leverage the logic in the service layer . Test coverage of this project is mainly covered “End to End Tests” tests as its main purpose is to draw the application experience together for a user so there for simulating the user experience is the ideal approach. |

## What techniques are used in the code promote the single responsibility principle?

As much as possible I try to have only one public method, whilst this does not enforce single responsibility it does reduce the confusions caused my having more than one public method

## What techniques are used in the code promote the readability?

Logic is grouped in named private methods that should allow the public method to read like psudeo code. Member variables as much as possible to reduce the size of method signatures and draw focus to the pseoudo code as seen below. This is particularly important in the service project

public static void Main(string[] args)

{

SetDeliveryPartnerName(args);

SetEffectiveDate(args);

ValidateInputs();

if (InputsAreInvalid()) {

OutputErrorsToConsole();

return;

}

GetContracts();

OutputContractsToConsole();

}

## How does the “cache” folder work on the repository folder?

Although there is no requirement in this application if needed we can fill cache folder c with decorator classes that wrap around the repositories. These usually leverage System.Net.Cache to cache the data. The decorators also help maintain the single responsibility principle as the caching code is not mixed up with the data access code.

## How does the logging work?

In a similar fashion the caching proposal above the logging is a decorator that wraps around the service class catching any errors that occur and using Nlog to log the error to file whilst presenting a friendly message to the end user. The decorators help maintain the single responsibility principle as the error logging is not mixed up with the service layer code.

## What goes in GRM.Tests.EndToEnd?

This project contains contain tests that cover UI to database. These tests are run during check in process and can be slow to run

## What goes GRM.Tests.IntegrationTests?

This project contain’s tests which interact with an external source e.g. database, API, No Sql data source, XML file etc. As a result the repositories functionality take up much this project.

## What goes GRM.Tests.UnitTests?

This project houses the unit tests created during the TDD process. They should be small and fast running.

## What goes in the Automation Folder?

These folder contains the build scripts and deployment scripts that are used in automation. These are usually powershell scripts in a framework such as psake.