**ASP.Net Web Dev. 2 Coursework**

THSurveys: Survey System

**Solution Evaluation**

**Table of Contents**

[Scope of Evaluation 3](#_Toc341951010)

[Approach 3](#_Toc341951011)

[General 3](#_Toc341951012)

[MVC as a Framework 4](#_Toc341951013)

[My Experience of MVC 4](#_Toc341951014)

[Notable areas of my learning experience. 5](#_Toc341951015)

[View Models 5](#_Toc341951016)

[Routing Tables 5](#_Toc341951017)

[Model Binding 6](#_Toc341951018)

[Action Filters 6](#_Toc341951019)

[Use of Simple Membership and LocalDB 7](#_Toc341951020)

[Unit Testing in Earnest 8](#_Toc341951021)

[Linq and mocked Entity Framework models 8](#_Toc341951022)

[Injecting Dependencies 8](#_Toc341951023)

[Using Mocking Frameworks 8](#_Toc341951024)

[Future Development 10](#_Toc341951025)

[General Improvements 10](#_Toc341951026)

[Fixes 10](#_Toc341951027)

[Code Refactoring 10](#_Toc341951028)

# Scope of Evaluation

## Approach

The approach to this evaluation is to cover the areas of the application

The MVC framework

My experience of the Framework

Notable areas of the learning experience

Testing

Future Development

## General

I’m reasonably happy with the application I have managed to produce, although I have found the timescales extremely tight. The coursework seems to include so many areas designed to provide opportunities to develop the MVC aspect of the application.

As far as the functional requirements go, each one is addressed by the inclusion of each of the views within the application. Hopefully the non-functional requirements are met in the following ways:

**Responsive and easy to use**: with the inclusion of a custom routing table providing for simple Urls, the use of jQuery within the client layer, the activation of client side validation and the use of Ajax calls to the server side where appropriate results in a responsive application

**Robust and Secure**: using the membership to provide user registration and authentication, using the supplied features to protect against such malicious attacks as XSS, XSRF etc, using error handling to ensure errors are passed to the appropriate error page, keeps the application secure and robust so the any potential errors are handled gracefully

**Designed for Testability**: the inclusion of many unit tests, though not an exhaustive list, designing all components to allow any dependencies to be injected at runtime using the Ninject Dependency Resolver and keeping as close as possible to the Single Responsibility Principle for each component provides for a structure that should be testable for each component.

I have managed to incorporate examples of most of the customisable features of the MVC framework, but still wish there was time to include those I’ve not had time to so far.

I’ve experienced problems with the Entity framework Code first, which on the face of it should have been the simplest way to create the Data model and back end to the MVC application. However, getting this to work properly, involved much more than I’d anticipated so I felt I took much longer on this part of the solution, which is not examinable. However, this is part of the learning experience and I feel I have learned quite a bit to be able to construct this part of an application much easier now.

Therefore there is a large section at the end which covers the future Development of the application should it ever be considered as the basis for a workable solution.

# MVC as a Framework

MVC as a framework is an extremely extensible and customisable framework for the development of browser based intranet and internet applications. Its evolution has been very quick, with the current version of MVC4 only being issued earlier this year.

The basic framework, and templates used within Visual Studio will generate a basic application, from which a fully functional application can be developed fairly quickly and easily. Such an application would conform to exactly the way MVC is intended to work, but would be completely workable as an application.

Some of the points of extensibility are somewhat obscure and not really intended to be customised, such as the **IActionInvoker** which actually launches the action method of the selected controller. However, this extensibility allows the development of robust applications whose functionality, structure and user experience is totally under the control of the designer and developer. Such applications can incorporate both server and client side code, with inbuilt support for jQuery libraries and Javascript libraries that support Single Page applications as well.

A point of caution though, is that as new components and technologies are added to the environment, there is a risk to do with stability as integration of these infant technologies can sometimes cause problems. An example is the Fakes and Shims modules to support automated unit testing, which at the present time do not cooperate with MVC as I found out to my cost and waste of time.

Another example is the speed with which Entity Framework, not strictly MVC, changes and the code first technology changes its way of handing versioning of the underlying database and seeding of information, Version 3.5 had no versioning but included the ability to initialise the database and seed it. EF 4.5 uses Migrations which does have versioning and a completely different way of seeding the database. It’s quite slick, except that for me it didn’t work when used in a different project from the main application. I haven’t spent the time to make it work, which I’m sure it does as other

# My Experience of MVC

Having attended the lectures, worked all the lab exercises and followed the demonstration code, when faced with designing an application from scratch, the thought of customising MVC is quite daunting. There are, in fact, so many places to change the default behaviour and so many different ways to code a solution.

The knowledge of just how customisable MVC framework is has led me along some blind alleys as far as how to design the application is concerned. It demonstrates that the difference between developing an out-of-the-box MVC application, as we did in ASPWD1 course last year, and a customised application such as this coursework, is quite marked.

Development in such an environment has to be even more carefully considered so that only the appropriate customisations are used. Only experience will quicken this design process as we become more familiar with the types of customisation that work and in what circumstances they work. The application will obviously develop quicker the more of the default behaviour that can be incorporated.

This ASPWD2 course has covered some topics that are not specific to MVC such as the guidelines and considerations of designing Urls as part of the UI itself. However, MVC has the ability to customise its routing table, which means that the structure of the Url can be developed, independently of the internal structure of the application, and therefore present a very succinct set of Urls that can be used directly to interact with the application.

Having managed to produce my first attempt at an MVC application, I realise that there are so many features that I have managed to incorporate in it and yet there are so many that I haven’t had time to incorporate. It’s a very large framework to use and its use has to be very carefully considered.

# Notable areas of my learning experience.

## View Models

The use of view models instead of any business models, helps decouple the business layer form the UI layer which is what the MVC is intended to provide. It also means that any logic that iexist in the view model, is very specifically UI logic and not business logic.

It has the benefit of reducing the amount of information that is rendered on the view and therefore the time taken to serve it up to the browser as it contains only the information required to make it function correctly. This was a good principle I adopted in my solution.

## Routing Tables

I tried to design the routing table at the beginning of the project, and failed miserably as I ended up with something that the default routing table would have supported very easily.

This proves that designing such things as this is not an easy task but needs careful consideration.

The actual structure I used for the Url appeared naturally as I progressed with the application development. In fact it was one of the last things I did. Since, by this time, the application was essentially working, it meant I had to develop the routing table independently then plug it into the application. Test Driven Development, ie I wrote the unit tests for the all the Urls first and constructed the routing table within the test project. Once all tests were passed the table was plugged into the application itself.

The following points were observed as I underwent this process

1. You must have an appropriate route for every single controller action method within the application.
2. Careful consideration must be given to the naming the optional parameter as it must be referenced within the code correctly otherwise the routes do not match
3. Making the ‘id’ numeric by setting a constraint will cause a mismatch if you need to pass NULL.
4. The order of the definitions in the table is important and getting the order correct can sometime be hit or miss.
5. When unit testing and using additional parameters, checking for the correct values in the **id** part of the Url can be awkward as it receives **5?questionId=17**, for instance, when testing a Url like http…/Survey/AddQuestion/5?questionId=17.
6. When developing a custom routing table, do not forget to include the necessary routes for the account membership login forms etc. I promptly blocked all login functionality and therefore all secured parts of the application on the first attempt.
7. When using the likes of the Url.Action helper in jQuery, a dummy must be used for the id as the url is calculated by the Razor engine before being rendered and then within the jQuery the dummy must be substituted for the real value required. If the id is constrained to be numeric, a numeric dummy value must be used otherwise the Url will not match the intended route.

**Figure 1** below, taken from the Add Questions view illustrates this.

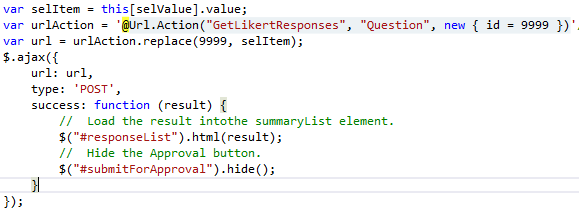


Figure 1 - Using Url.Action within jQuery

## Model Binding

I took a while to get my head round the contents of and responsibility of the custom model binder. Originally I thought of it as taking the fields on the view and transforming them to those necessary for the controller to process, however I ended with mapping between view model and business model which necessitated calls to the repository. This just did not feel right, as this contained more responsibility in addition to binding, model binders should bind and nothing more.

By the time I’d completed the only working custom model binder in the application, I had it performing reformatting of the view model, to ensure the view model contained the correct values. Additionally it has the responsibility for validation and setting the model state errors where validation fails. This seemed a much more appropriate set of responsibilities, which relate only to binding.

## Action Filters

The solution presented a few opportunities for the creation of custom Action filters. The tasks performed within filters are varied and may involve interaction with view data directly, but alternatively may do something completely different.

There are 4 types of filter, which execute in a very strict order: Authorisation, Action, Result and Exception. Therefore, when an Action filter is invoked, it is guaranteed that authentication has already occurred. Within the action filter itself, there are two methods, **OnActionExcuting**, which executes before the action method does and **OnActionExecuted** which occurs immediately after the action method completes.

One task suited to action filters on a Get request, is mapping a business class to a view model class; it would be achieved by overriding the OnActionExecuted method allowing the controller to only reference the business model. Mapping, however, on the OnActionExecuting method, is best avoided as exceptions can happen when accessing data contexts and, more importantly, this is the proviso of the Model binder.

Within this solution, mapping of business model to View Model is consistently performed using custom action filters, for Get requests. They are not used on Post request, however. Two other examples of custom action filters are used:

**AjaxActionOnly**: to ensure the incoming HttpRequest is an Ajax request, overriding the OnActionExecuting method. This is to stop the situation where action methods, there to support ajax call, are prevented from executing should the url be entered directly into the browser address bar.

**AuthorisedForSurvey**: to ensure that the authenticated user owns the survey being accessed, byt overriding the OnActionExecuting method. It is intended to stop one user accessing another user’s survey by entering the url and a surveyId directly into the browser address bar.

Both these action filters enhance the robustness and security of the application.

## Use of Simple Membership and LocalDB

The use of the simple membership provided as part of the MVC4 internet application template, like previously with ASP.Net membership, makes the basic functionality of user registration and authentication quite simple. The benefit of the simple membership is that is it easy to integrate in into the application database simply by including a UserProfile table in the application database and pointing the membership context to this table.

In this solution the database being used is the new LocalDB. This is a fully functional SQL server database, but one which doesn’t require an instance ot SQL Server Express and instead runs as a separate process. However, without being able to say which component is the problem, intermittent errors occurred during application development, where accessing the User.Identity.IsAuthenticated property of the HttpRequest resulted in an Invalid Operation exception. The message was that the database initializer had to be run before accessing the membership stuff. The System Design document shows this error on the example of the error page.

The problem is resolved by logging off. This may be yet another example of the technology being so new and not fully debugged. To handle this error, I used jQuery on the error page to check for the existence of the **logoffForm** element, which is only on the page if the user was logged on, and forcing a logoff to close the error form. This is necessary as accessing the User, Identity properties would result in the same error being re-thrown. **Figure 2** below illustrates the solution.

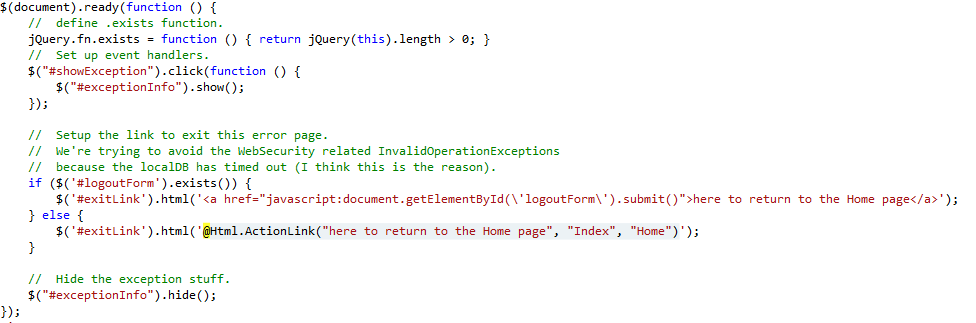


Figure 2 - Handling problems with authentication errors using jQuery

# Unit Testing in Earnest

## Linq and mocked Entity Framework models

When using an Entity framework data context, Linq returns IQueryable<> collections whereas when using POCO classes to mock the data context Linq will return IEnumerable<> colletions. This can cause problems in the unit test when methods expect IQueryable<>. The solution to this is to use the AsQueryable<>() on the IEnumerable<> collection which casts it to the expected IQueryable<>.

An example of which is shown in **Figure 3** below. This is obvious to someone with more experience of EF and myself now that I’ve experienced it too.

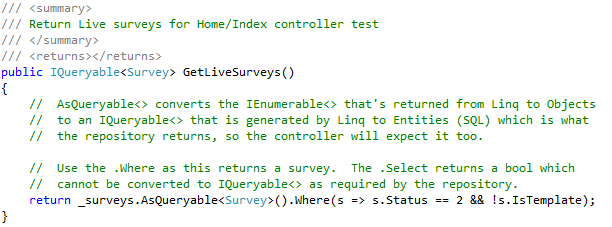


Figure 3 - IEnumerable and IQueryable returns from Linq

## Injecting Dependencies

So that unit testing of all modules can be one in isolation, **all** dependencies should be injected.

Where factories are use, follow the “abstract factory” pattern. This allows the concrete factory instance to be injected, removing the need to “new” the instance of the factory.

This is good practice from the coding point of view anyway, but improves the ability to unit test.

## Using Mocking Frameworks

###### Moq

Using Moq, which was my chosen mocking framework and the class being unit tested requires multiple calls to the dependency with a different response each time, then use the Moq.SetupSequence<> call and include as many .Return<> as required to satisfy the expected number of calls;

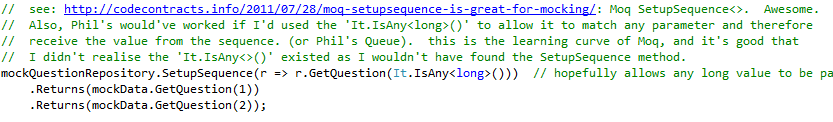


Figure 4 - Setting up Moq for multiple dependency calls

This also shows the use of **It.IsAny<type>()** to allow the mock to use a method that requires a parameter, which obviously isn’t needed as the return value is mocked but allows the test to compile and run. If this **IsAny** parameter was not used, Moq would expect the value supplied as the parameter to match the call and would end up failing.

###### Ms Fakes

Moq does not support the mocking of static or extension methods. I discovered this when trying to unit test the AjaxActionOnlyAttribute custom filter. It came to light as MS in their infinite wisdom extended the HttpRequestBase to include the method IsAjaxRequest() but as an extension method.

This makes mocking very difficult.

MS provide Fakes and Shims as a means to create stubs and mock such methods. However, this technology is very new indeed and currently is not compatible with MVC.

Shims are intended to allow mocking of static or extension methods by rewriting the executable code at runtime. Using a shim, by following examples in the internet, results in a sever error: “Operation may destabilise the Runtime”. Ms acknowledge this as a problem and have a fix but have not, as yet, release the fix.

After taking considerable time to get to this stage, and after a few emails to Dr J. Paterson, course tutor, I gave up deciding that these are the sort of pitfalls encountered with using such new technologies: they are not always stable. In a working environment a set of professionally developed tools would likely be paid for and used. Such issues would then become matters of support provided by the vendor for the vendor to fix.

# Future Development

## General Improvements

###### Templates for Survey Questions and Responses

The ability to add template for surveys, question and Likert responses would be added to the system. This would facilitate quicker and simpler techniques for setting up surveys.

The current implementation includes the ability to store templates within the Domain Model, and populate these with test data. The maintenance of these templates would be reserved for future development.

###### Custom view engine

A custom view engine should be implemented to remove the checking for **vbhtml** files and therefore improve performance.

###### Take Survey Paging

Add paging to this survey so that many questions can be included OR to display a single question or group of related questions and progress to the next part of the survey, showing a progress indicator across the top of the page showing how far through the survey the respondent is.

###### Approve Survey Feedback

When the approve surveys button is clicked with no surveys selected, it returns to the Approve view. There is no indication that something has happened. This is not good from a user perspective. Validation should be included that returns a message saying “no surveys have been selected for approval”. This would be best implemented using a custom model binder, to validate the model.

###### Custom Html Helper

Complete the coding of the RadioButtonListFor helper, but taking account of the prefix for the property name. Also process any additional html attributes added as the last parameter. Provide the standard overloaded signatures to be compatible with the built-in Html Helpers.

## Fixes

###### Entity Framework Migrations

Future development would undoubtedly change the model requiring the EF Migrations to be properly implemented so they worked. For now, the data model should not change and so the fact that the EF migrations is not working doesn’t matter.

## Code Refactoring

###### Survey Controller: Create action method: POST

Refactor the code in the method to a mapping class in the Mapping folder, to reduce the controller responsibility following the Single Responsibility Principle.

###### Interface to Mapping classes (not including custom action filters)

Convert these classes to implement a generic interface as they all use a Map method, but the classes used are different. The generic implementations should be restricted to types of Class.

###### Change of Status of Survey

The code to change the status of the survey from Incomplete, to Approval to Live should be refactored within the Core project to use the State design pattern.

###### Location of Filters, Helpers and Mappings

The folders containing the Action Filters, Html Helpers and Mappings should be moved to the Infrastructure of the UI project. This is neater as all non-standard MVC folders are then in the one place.

###### Models that have moved location since project setup

Some modules have been relocated to different folders since the project was set up. The namespaces for these modules have not been altered to reflect their current location. The code should be refactored to eliminate such namespace mismatches.