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

THSurveys: Survey System

**System Design**

**Table of Contents**

[Application Setup 4](#_Toc341950448)

[Project Solution 4](#_Toc341950449)

[Database setup 4](#_Toc341950450)

[User Id 4](#_Toc341950451)

[Solution Structure 4](#_Toc341950452)

[Description 4](#_Toc341950453)

[Architecture 4](#_Toc341950454)

[Packages 7](#_Toc341950455)

[Application Structure and Logic 8](#_Toc341950456)

[Overview 8](#_Toc341950457)

[Data Model 9](#_Toc341950458)

[Menu 10](#_Toc341950459)

[Styling 10](#_Toc341950460)

[Page Layouts 10](#_Toc341950461)

[Use of the MVC Framework 26](#_Toc341950462)

[Overview 26](#_Toc341950463)

[Dependency Resolver: IoC 27](#_Toc341950464)

[Routing 27](#_Toc341950465)

[Controllers and Actions 28](#_Toc341950466)

[View Models 28](#_Toc341950467)

[Model Binding 28](#_Toc341950468)

[Action Filters 29](#_Toc341950469)

[Mapping Classes 30](#_Toc341950470)

[Html Helper 30](#_Toc341950471)

[Validation 30](#_Toc341950472)

[Division of Functionality 31](#_Toc341950473)

[Client 31](#_Toc341950474)

[Server 31](#_Toc341950475)

[Core 31](#_Toc341950476)

[Infrastructure 32](#_Toc341950477)

[Cross cutting Concerns 32](#_Toc341950478)

[Security: 32](#_Toc341950479)

[Exception Handling 34](#_Toc341950480)

[Code Modules and Patterns 34](#_Toc341950481)

[Unit Testing 35](#_Toc341950482)

[Areas of Testing 35](#_Toc341950483)

[Future Development 39](#_Toc341950484)

[General Improvements 39](#_Toc341950485)

[Fixes 39](#_Toc341950486)

[Code Refactoring 39](#_Toc341950487)

[Acknowledgements 40](#_Toc341950488)

[Use of the MVC Charting tool 40](#_Toc341950489)

# Application Setup

## Project Solution

The solution is delivered as a Visual Studio 2012 project. It has been cleaned so that it contains no executable files. All files have been compressed into ***THSurveys.zip***. The contents of this zip file must be extracted to a folder which is accessible by the instance of Visual Studio.

## Database setup

The system uses Entity Framework (EF), Code First to create the Domain model. When the model is accessed for the first time, through the THSurveys DbContext, EF creates the underlying database. The test data is already loaded into the database, and EF should not attempt to rebuild the schema so no data should be lost.

An instance of **LocalDB** is used as the database. This is a recent addition and functions like a genuine SQL database, but doesn’t need an instance of SQL Server (Express) to manage it, as it runs as a process. It can, however, be a bit unstable as it’s a very new product.

## User Id

The following users are set up.

|  |  |  |
| --- | --- | --- |
| UserName | Password | Role |
| Tim | Password | User |
| Nancy | Password | User |
| Administrator | Password | Admin |

# Solution Structure

## Description

The solution is provided as a series of project, the main one which is an MVC4 Internet application, as provided by the VC2012 template, serving multiple pages to facilitate the requirements specified in the Coursework.

A benefit of this is the automatic inclusion of the Simple Membership provider which can be built into the systems database much easier than before.

Functionality is split between the client and server layers of the application. Some client pages contain jQuery elements to improve the responsiveness of the application and to provide some of the essential functionality. The server side of the application provides the control between pages and the routing of the requests to the necessary resources.

## Architecture

The solution architecture is designed around the “Onion model”; a layered design with the business model at the core and other layers communicating directly with the core. The other layers in the application are the UI layer, which is the MVC project, and the infrastructure project which has the concrete implementations of the Data context and access to them.

The model does not allow communication or reference between components at the same layer, but only towards the core.

This can cause a problem for the location of the IoC container, which needs to reference both Core and Infrastructure layers. This is the single exception and is set up as a project that is allowed access to both layers. Having this as a separate layer, removes the need for any reference across layers in the UI layer.

The solution contains multiple projects to implement this architecture.

### Core Project

This project contains the business model, interfaces, factories and Services used by the application.

|  |  |
| --- | --- |
| Folder | Contents |
| Model Folder | The business model is contained within the Model folder. It has a class for each object in the model, some of which are abstract to facilitate the use of abstract factories to create concrete instances of them. Such classes will also have a concrete version within this folder. |
| Interfaces Folder | This folder contains the interfaces for the various repositories and the UnitOfWork for the data context |
| Factories Folder | This folder contains the factories used to create concrete instances of various objects, used within the UI layer. |
| Services Folder | This contains the Survey Analysis service, which will calculate the various statistics based upon a supplied survey. It is used in the Analyse pages |

The Unit Of Work exists to ensure that, in conjunction with Ninject, the same instance of the data context is used by all the repositories, avoiding exceptions related to the lifetime of the data context.

### Infrastructure Project

|  |  |
| --- | --- |
| Data Folder | This folder contains the Entity Framework data context, which uses Code First based upon the classes in the Core/Model. |
| Migrations Folder | This should contain the migrations for the Code First database generation. However, problems experienced early in the project meant that this feature doesn’t currently work properly. The database schema has settled for this application so making the Migrations feature of Entity Framework function correctly is left for future development. |
| Repositories Folder | This folder contains the concrete repositories used by the application UI controllers. |

### IoC Project

This project exists to allow the Ninject Dependency Resolver bind the interfaces and abstract classes of the Core project with the concrete implementations contained in the Infrastructure project.

This is required as these binding cannot be achieved in the UI project, where the Dependency Resolver is created, since to reference the Infrastructure project from the UI project causes a circular reference.

### Models and Documents Project

This project contains the class and Use case diagrams for the application. It also contains the documentation, of which this document is one.

### THSurveys Project

This is the UI project, which is the MVC application project. It was created from the standard MVC4 Internet Application template and contains the standard folders.

Additional folders added to the project, contain the components which allow the application to functions.

|  |  |
| --- | --- |
| Filters Folder | This folder contains the Action Filters used by the application. Most of them map various classes to other viewModel classes and vice versa. There is an additional AjaxActionOnly filter. |
| Helpers | Within this folder is the only Html helper method. |
| Infrastructure | This is subdivided into the following: |
| Factories | This contains the factory for creating the MVCChart used to display survey results. |
| Interfaces | Contains interfaces for the mapping classes used by the Take Survey view. It allows them to be injected into the controller and facilitates better unit testing of the controller action. |
| ModelBinders | This contains the only custom model binder used in the application. It handles the binding for the Take Survey viewmodel and the **Home** controller **Take** action method. |
| Mappings Folder | This contains the mapping classes used for converting the returning View model classes to the corresponding business model classes. |
| Models Folder | This is the standard models folder for an MVC project but it has split into separate folders to contain view models that relate to the correspondingly names controller. |

All models within these folders are considered to be view models and each view has an associated view model. The restructuring of the folders happened after the coding had started, so some of the namespaces of the view models do not necessarily correspond to the folder they are contained in.

It is a future development refactoring task to correct this situation, but it doesn’t hinder the running of the application itself.

### ThSurveys.Tests Project

This project as the name implies, contains the unit tests for a sample of test for as many of the components used within the application.

The tests included are by no means exhaustive, but hopefully are indicative of possible ways to unit test the individual components.

The tests for each of the components are held in the subfolders of the project.

|  |  |
| --- | --- |
| Folder | Tests Contained |
| Controllers | Test on the Home controller. |
| Filters | Test on the AjanActionOnly filter |
| HtmlHelpers | Test of the RadioButtonListHelper |
| Mappings | Test on the MapTakeSurveyViewModel mappers of the THSurveys project |
| Model | Contains the mock data for the unit tests. |
| ModelBinders | Test on the TakeSurveyModelBinder |
| Routes | Test on the Custom Routing tables for incoming and outgoing Url’s. It also contains the original RouteTable, as this was developed outside the application towards the end of development and then plugged in when it passed all tests. Akin to Test Driven Development. |

## Packages

The following packages are used within the project solution to support the application

|  |  |
| --- | --- |
| **Package** | **Function** |
| Ninject | Dependency Injection |
| AutoMapper | Mapping between business classes and ViewModel classes |
| Glimpse | Installed but not used much to verify routes being taken. |
| MVCChart | Charting tool supplied as part of .Net framework, since .Net 3.5. |
| MSTest | Unit Testing framework, part of VS2012 |
| Moq | Mocking framework for unit testing, object creation. |
| Fakes | MS tool, fake objects and allow Static or Extension method to be ‘mocked’. |

# Application Structure and Logic

## Overview

The application is structured to support the requirements, with as simple a presentation layer or UI as possible. The UI contains pages split into the main sections

**Participate in a Survey**

* View a list of available surveys and select one
* Take a survey:

**Creation of Surveys**

* Create a Survey
* Add questions to a survey
* View a list of surveys and their status, for the registered user
* Submit for Approval, by Administration

**Approve Survey**

* View list of surveys pending approval
* Approve a Survey

**Review Survey Results**

* View the summary of responses to a survey
* View the summary as a Bar Chart

The application is secured so that Creation of surveys and Review Survey Results can only be accessed by registered users of the application. The surveys are restricted to those belonging to the registered user. A system Administrator is solely responsible for the Approve Survey functionality.

Participation in a Survey is unrestricted and available to anyone[[1]](#footnote-1).

To anticipate user interaction with the application via the browser address bar, the applications Urls have been structured to represent the functionality directly.

Some examples:

|  |  |
| --- | --- |
| Url | Purpose |
| http…../Surveys | Allow registered user to view a list of their surveys |
| http…./Survey/Create | Allow registered user to create a new project |
| http…./Survey/Top5 | Allow unregistered user to view a list of the top 5 surveys available for participation |
| http…./Survey/Take/id | Allow an unregistered user to participate in a survey |

## Data Model

The data model is defined as a series of classes, held in the Model folder of the Core project. Figure 1 shows the Class Diagram of the business model.

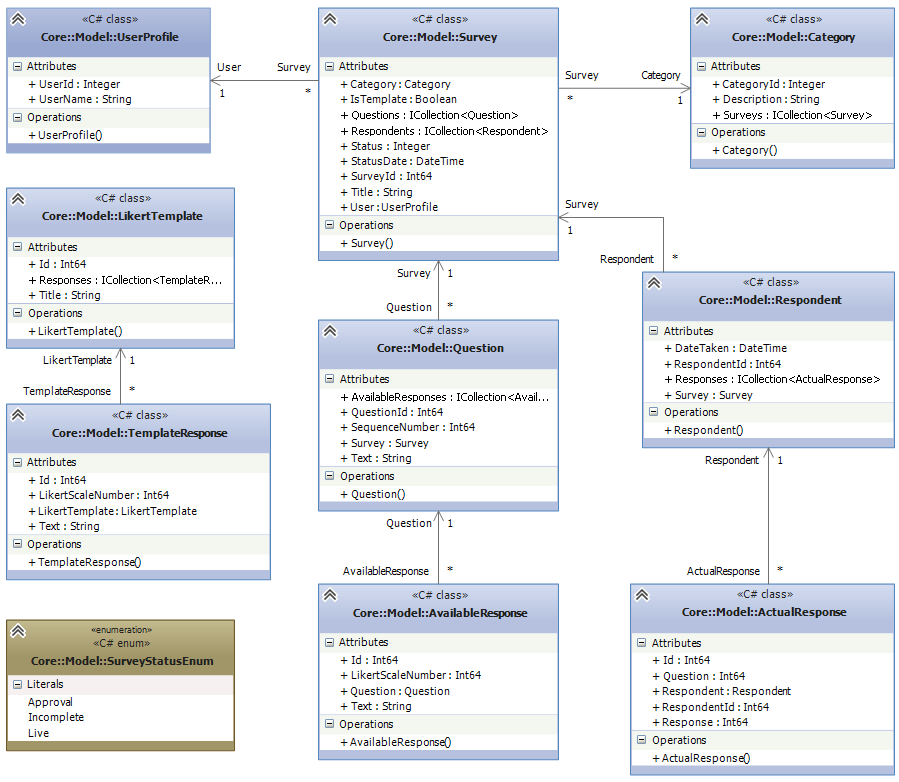


Figure - Business Data Model

The various classes included in the model provide for the following:

|  |  |
| --- | --- |
| **Table** | **Purpose** |
| Survey | Defines the Surveys held on the system |
| Question | Defines the questions associated with a Survey. There is an indicator **IsTemplate**, to allow template surveys to be stored. This is future development |
| Available Response | Defines the possible answers to a specific question. These are Likert responses, held in the Likert Template class. |
| Respondent | Defines the respondents for each survey, they are anonymous so no personal information is held. |
| Actual Response | Defines the answers a respondent placed to a specific question. |
| Category | Defines the categories, which group questions. |
| Likert Template | Defines the available Likert scales for a question. These, along with the Template Responses are used to populate the Available Responses to a question. |
| Template Response | Defines the possible responses for a Likert Template. |
| User Profile | Defines the users on the application, and forms the main user profile table for the integrated Simple Membership functionality provided by the MVC Internet Application template. |

These classes are referenced by the Entity Framework to facilitate its Code First technology to generate the underlying database.

### Business Rules

1. Sequence Number of a question should be unique within each survey. Its intention is to allow the order of questions to be altered. This applies to Question.SequenceNumber
2. Sequence number of a response on the Likert Scale should be unique. This is so that order of the the Likert Scale responses can be preserved. It applies to AvailableResponse, LikerScaleNumber and similarly on the LikertResponse template.

## Menu

An application menu is provided as a series of links which gives access to the parts of the application directly accessible by users.

|  |  |  |
| --- | --- | --- |
| **Menu Item** | | **Description** |
| Home | | Shows the Top 5 Surveys |
| Participation | |  |
|  | Top 5 Surveys | Shows the Top 5 Surveys |
| Choose by Category | Allows surveys for each category to be listed |
| Surveys | |  |
|  | List My Surveys | List surveys for the user who is logged in. |
| Create | Allows a registered user to create a new survey |
| Administration | | Allows the system administrator(s) to approve surveys for release |
| About | | General information: disclaimer and acknowledgements. |

Not all options are included within the menu. Those which are not, are accessed from links within the various pages, when it is appropriate to.

## Styling

Styling of the application making use of CSS is not a primary concern of this project in terms of the coursework requirements. However, a simple standard layout is applied across all pages of the application.

The styles are achieved using CSS. The styles for the menu are based on freely available css templates, specifically <http://cssmenumaker.com/>.

## Page Layouts

This section describes the various pages contained within the application. For each one, there is a description, a sample layout, the navigation links available, a description of its functionality and a list of the MVC components used to provide the page and its data.

### Home Page

This lists the top 5 surveys in terms of the number of respondents.

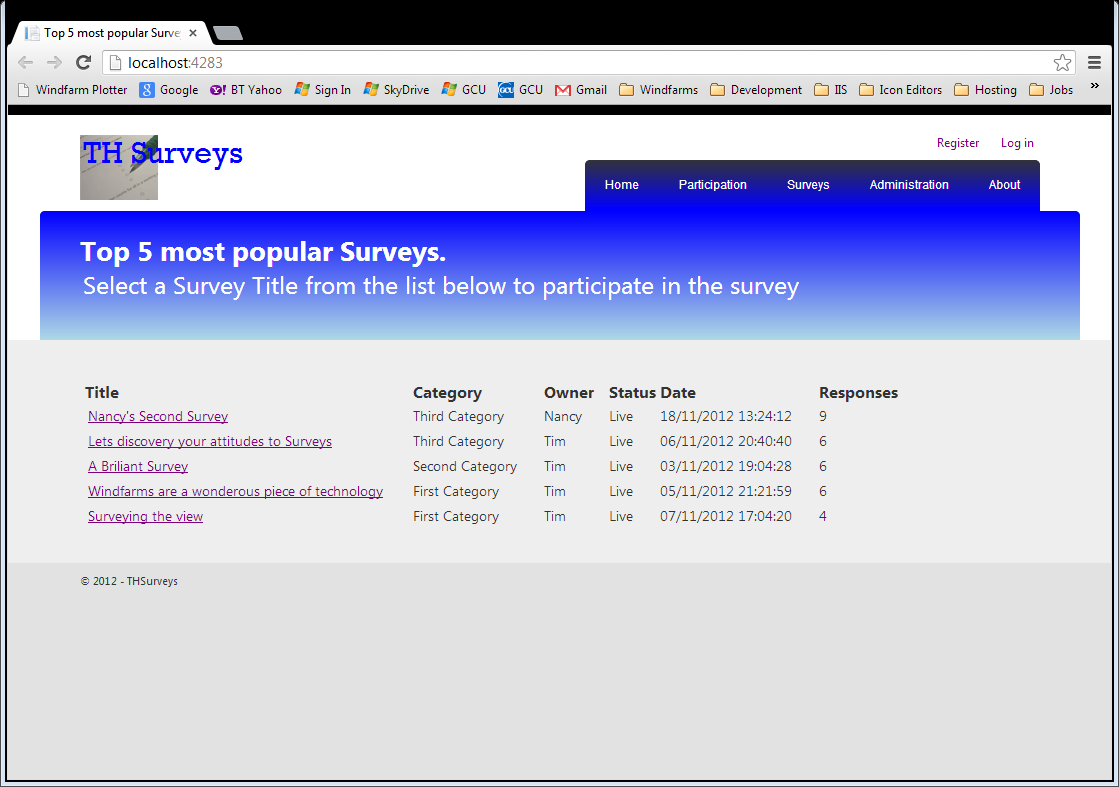


Figure - Home or Top 5 Surveys View

###### Functionality

This is the default page, but is also accessible from the **Top 5 Surveys** option on the Participate menu.

###### Security

This view is unsecured

###### Links

The Title links of each survey navigate to the Take Survey page.

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Home/Index | Controller action method providing the info |
| Index.cshtml | View |
| \_SurveySummaryList | Partial view to show the surveys returned |
| MapSurveyToSurveySummaryAttribute | Custom Action Filter to map the Surveys to the SurveySummaryViewModel |
| SurveySummaryViewModel | View model to support the \_SurveySummaryList partial view |

### Choose a Survey by Category

This shows a list of surveys that can be taken by a respondent. It is accessed from the **Choose** option on the **Participate** menu.

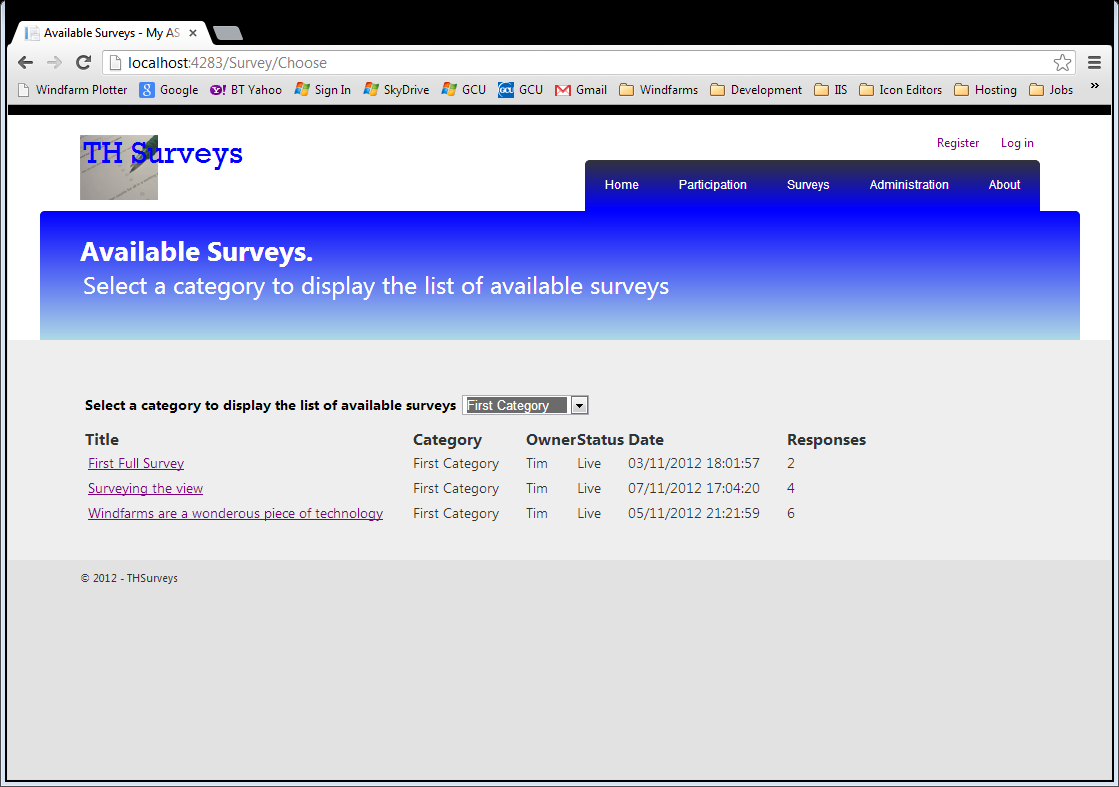


Figure - Choose Survey by Category View

###### Functionality

Selecting a category from the drop down box will redisplay the surveys for that category. There is an option to select all categories.

###### Security

This view is unsecured

###### Links

The Title links of each survey navigate to the Take Survey page.

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Home/Choose | Controller action method providing the info |
| Home/SurveyList | SurveyList which returns partial view, and responds to an ajax call so that only the partial view is updated, improving performance. |
| Choose.cshtml | View |
| HomeChooseViewModel | The model supporting the view. |
| \_SurveySummaryList | Partial view to show the surveys returned. This is the return from the ajax call, submitted when the selected value in the drop down is changed. |
| MapSurveyToSurveySummaryAttribute | Custom Action Filter to map the Surveys to the SurveySummaryViewModel, on the SurveyList Action method |
| SurveySummaryViewModel | View model to support the \_SurveySummaryList partial view |
| AjaxActionOnlyAttribute | Action filter to ensure the HttpRequest is an Ajax request. This stops the action method being called from the browser address bar. |

### Take a Survey

This is the page used to participate in a survey. The desired response to each question is indicated by selecting the appropriate radio button.

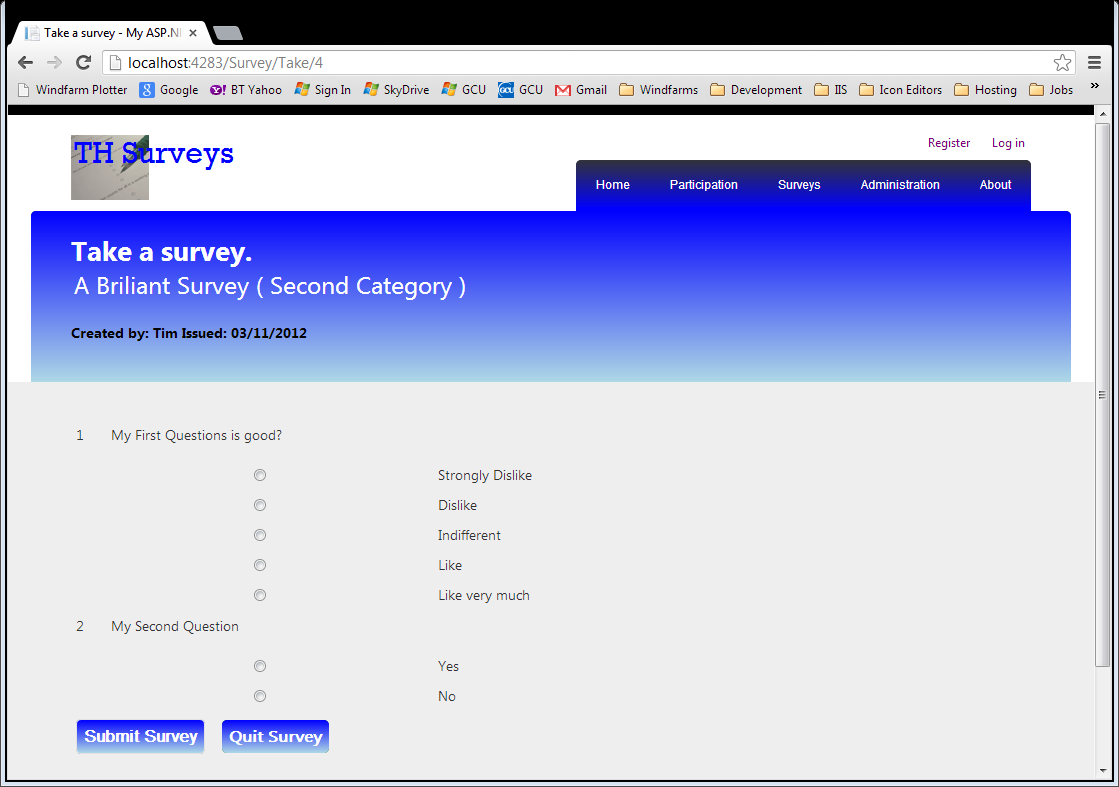


Figure - Take Survey View

###### Functionality

When the Submit Survey button is clicked the responses are validated, missed questions are indicated at the top with a message showing which question has been missed. If all questions are answered the survey results are updated

###### Links

Click Submit Survey returns to this view if validation fails, otherwise in navigates to the **Thank You** view for a survey

Clicking Quit Survey, navigates back to the Top 5 Surveys, Home page.

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Home/Take (GET) | Controller action method providing the info |
| Home/Take (POST) | Action method, responds to the POST request, and handles the survey results updating. |
| Take.cshtml | View |
| TakeSurveyViewModel | The model supporting the view. |
| MapSurveyToTakeSurveyViewModelAttribute | Custom Action Filter to map the Surveys to the TakeSurveyViewModel. |
| \_SurveyQuestions.cshtml | Partial view to show the questions for the survey being taken |
| SurveyQuestionsViewModel | Model supporting the \_SurveyQuestions partial view, displays the question for the survey |
| \_QuestionResponses.cshtml | Partial view to show the responses for a specific question of the survey |
| SurveyResponsesViewModel | View model to support the \_QuestionResponses partial view |
| RadioButtonListFor<> | Custom Html Helper to list the responses as a group of radio buttons. |
| TakeSurveyModelBinder | Custom model binder to bind the responses to the TakeSurveyViewModel. It is responsible for the validation of the responses and set the modelstate.Isvalid to false is a question is missed. |
| ValidateAntiForgeryTokenAttribute | Action filter to ensure built in protection against XSRF attacks is checked. |
| MapTakeSurveyViewModel | Map the TakeSurveyviewModel to the survey prior to updating. Called from within the action method on the POST. |
| ReinstateTakeSurveyViewModel | Reinstate the TakeSurveyViewModel, if the modelstate is invalid. |

This view is by far the most complex in the application and therefore is used as the basis of the sample unit tests, as it covers all of the areas where custom components have been used.

### Take Survey Completed page

This is the response page shown when the survey is successfully completed by the respondent.

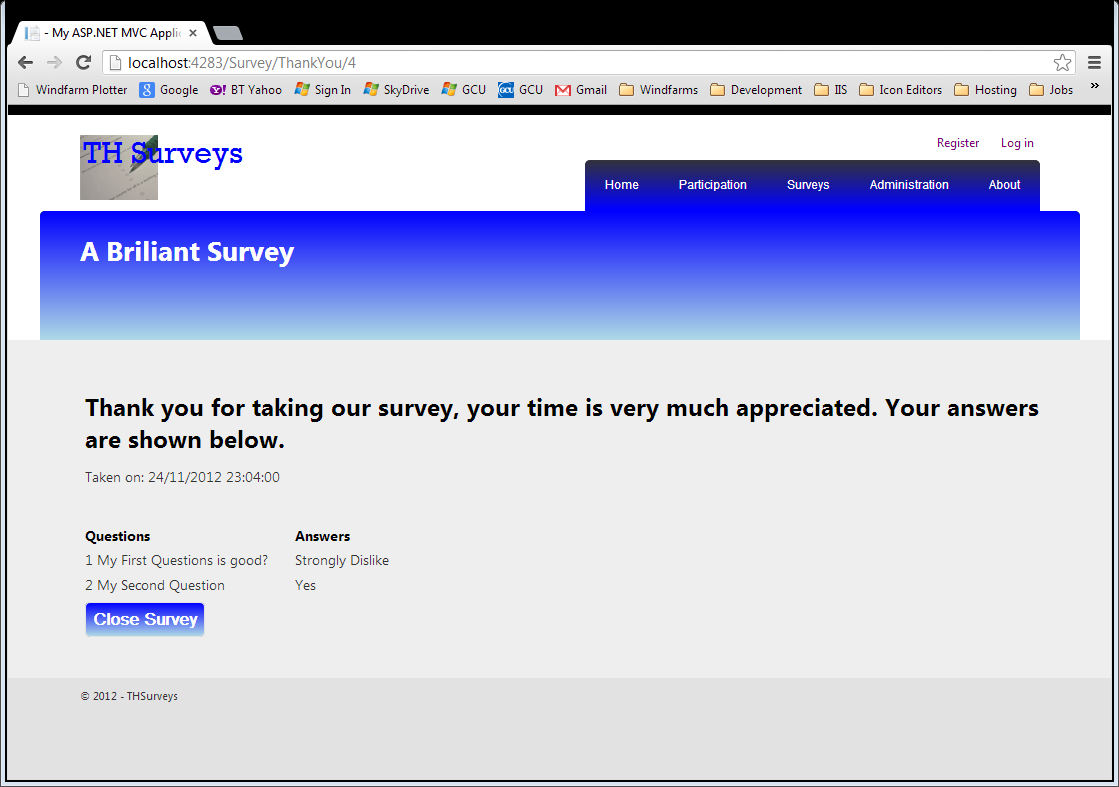


Figure - Take Survey Results View

###### Links

Close takes you back to the Home page.

###### Security

This view is unsecured

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Home/ThankYou (GET) | Controller action method providing the info |
| ThankYou.cshtml | View |
| SurveyResultsViewModel | The model supporting the view. |
| MapSurveyToSurveyResults | Map the latest survey results to the view model, called directly from within the action method. |

### Login View

This is the standard MVC provided form, displayed automatically when the controller and / or action methods are secured. It has been modified slightly to take the styling of the application.

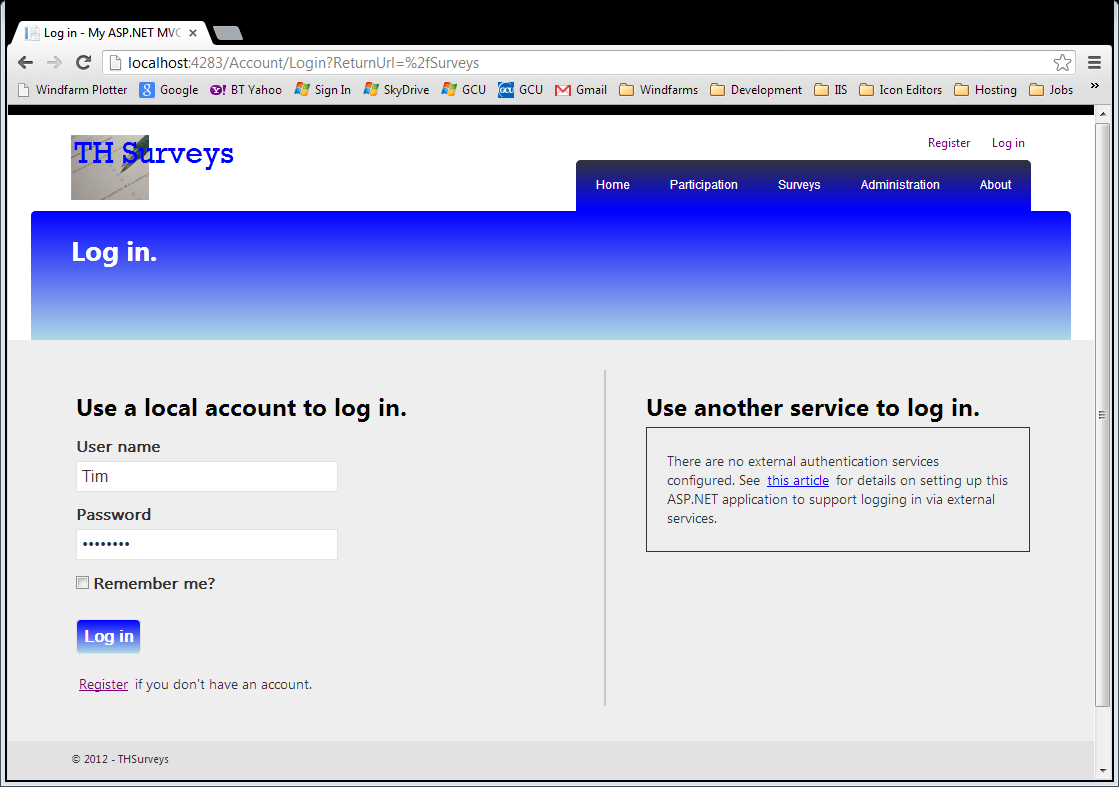


Figure - MVC Provided Log In view

###### Functionality

Use the local account to log in, complete the details click the login button.

###### Links

On successful login, the user is redirected to their intended view.

###### MVC Components

It uses the standard Account Controller action methods, provided by the MVC Internet Application template.

### Register User View

This view has not been shown separately. It functions in the standard way. The only modification is that the user, where registered, is added to the Role “User”. This ensures that the user can access the Survey maintenance functionality. It precludes setting up a new Administrator.

### My Surveys Page

This page shows all surveys for the logged on user and constitutes the users main page. It indicates the status of the survey and provides links to view the results or add question to incomplete surveys

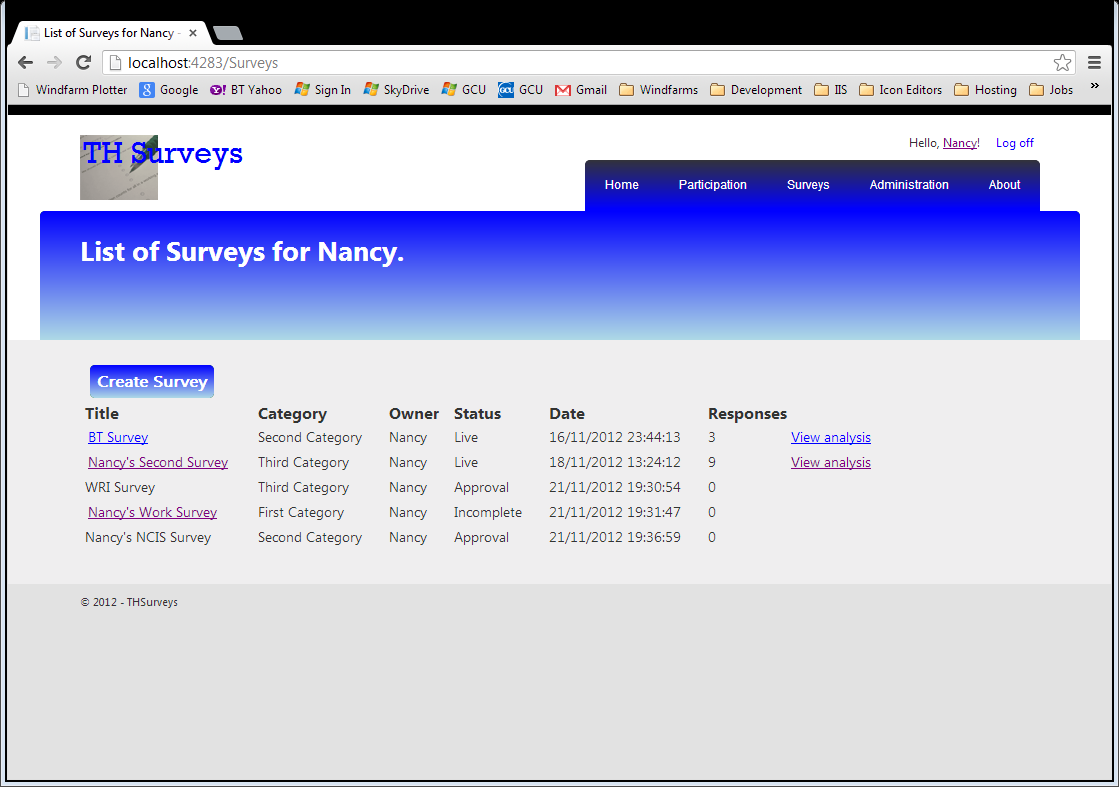


Figure - List My Surveys View

###### Functionality

The view provides a link to view the analysis of a survey, if its status is Live, or to Add Questions if the status is Incomplete. If the status is Approval, no further action is possible until an Administrator has approved the survey, and promoted it to Live status. It also provides a link to create a survey, separate from the menu option.

###### Security

This view is secured and restricted to users in the role USER.

###### Links

Click on **Create Survey**, navigates to the Create Survey view. The Title link links to Add Questions or View Analysis as described in the functionality. The View Analysis link only applies to the Live surveys and links to the View Analysis view.

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Survey/List (GET) | Controller action method providing the info |
| List.cshtml | View |
| \_SurveySummaryList.cshtml | Partial view to list the surveys |
| SurveySummaryViewModel | The model supporting the view. |
| MapSurveyToSurveySummaryAttribute | Custom Action filter to map the surveys for the surveySummary list. |

### Create Survey page

This is the page where surveys are created, by the logged on user.

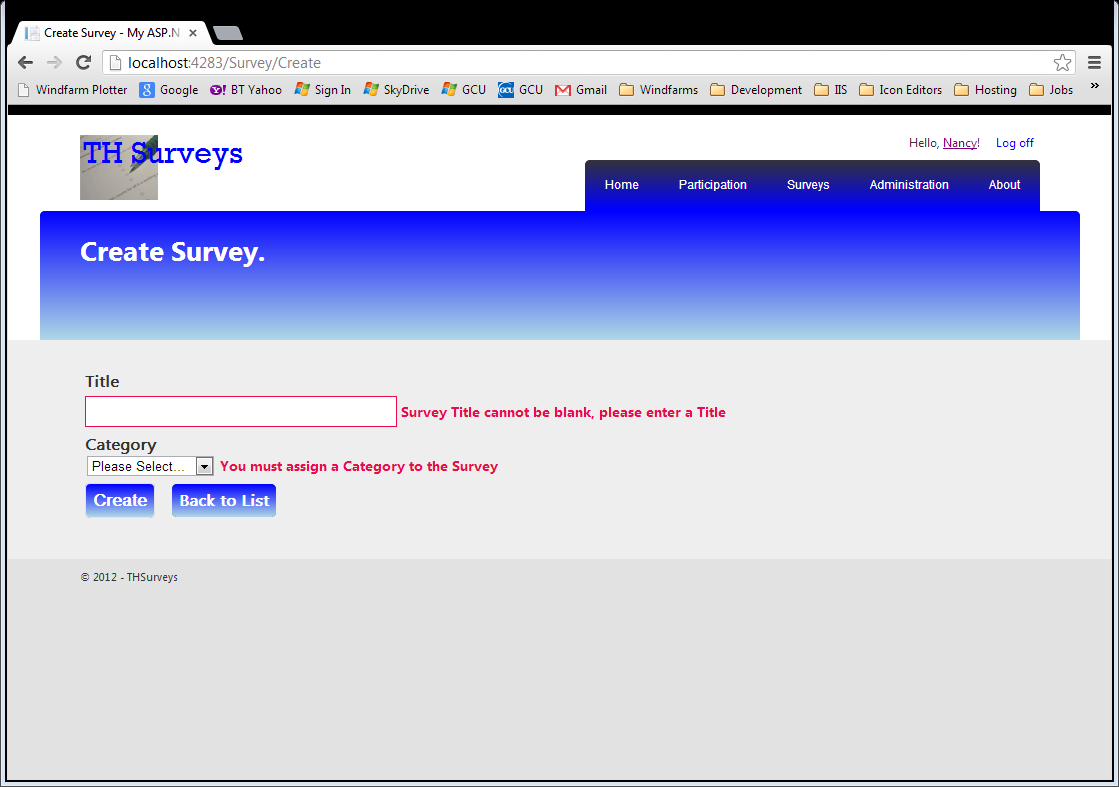


Figure - Create a Survey View

###### Functionality

The validation is performed using the data annotation attributes on the view model. To keep the page responsive, it is client side validation using the jQuery validation supplied by the application template. This same validation is also happening on the server, in case the browser has JavaScripot disabled.

###### Validation

This shows both validation rules broken. The Title cannot be blank, and a category must be selected form the drop down list.

###### Security

This view is secured and restricted to users in the role USER.

###### Links

**Create** button navigates to the Add Questions view after successful validation.

**Back to list** Navigates to the Surveys list.

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Survey/Create (GET) | Controller action method providing the info |
| Create.cshtml | View |
| CreateSurveyViewModel | The model supporting the view. |
| ValidateAntiForgeryTokenAttribute | Action filter to ensure built in protection against XSRF attacks is checked. |

### Add Questions to Survey

This is the main page for setting up the survey. It allows questions to be added, showing them as they are added.

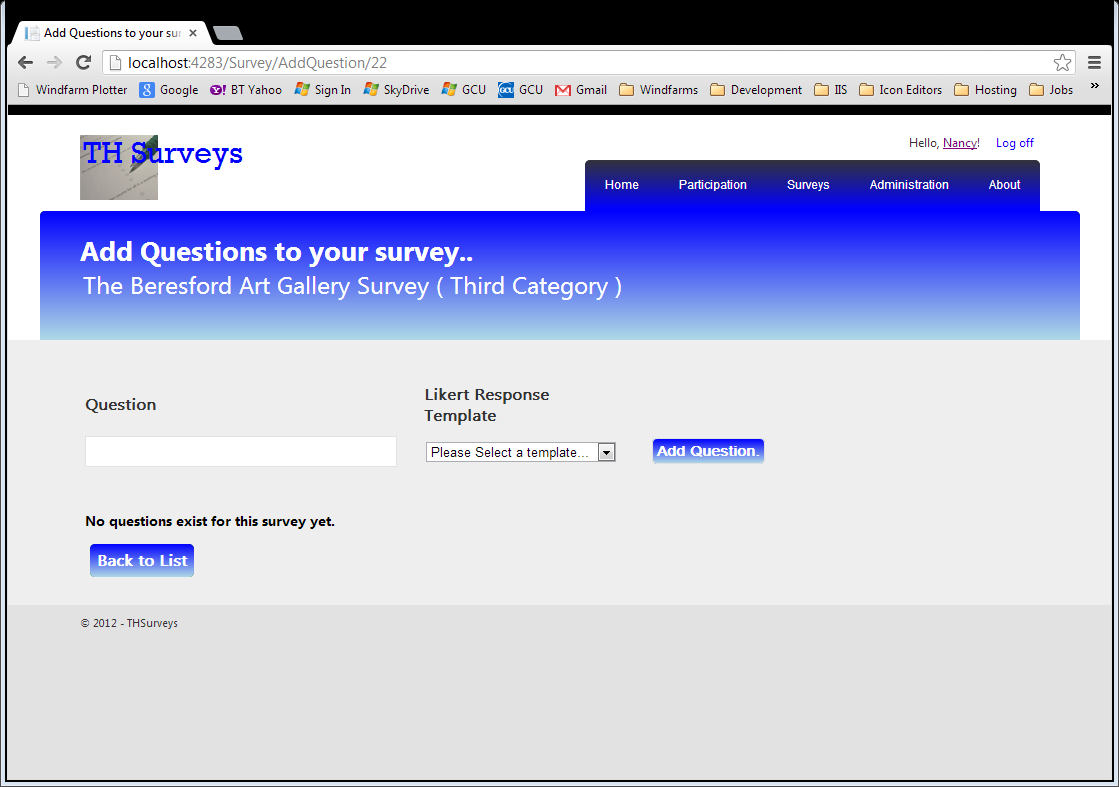


Figure - Add Questions to a Survey View

###### Functionality

It allows questions to be added, shows the questions as they are set up. Allows the possible responses to be selected from a template and allows the survey to be submitted for approval by the Administrators before going Live.

This shows the initialised state, when questions exist they are shown where the “No questions…” message is shown. A button to submit the survey for approval also appears, when there are questions but nothing in the input fields. This stops the survey being submitted for approval if a question is currently being added.

Selecting a response template from the drop down list retrieves the possible answers and displays them below the dropdown.

Validation is in place on both client and server so that the Question must have some text and a response template must be selected from the drop down list.

###### Security

This view is secured and restricted to users in the role USER.

###### Links

**Add Question** adds the question to the survey, and returns to this page redisplaying the updated questions at the bottom of the page.

**Submit for Approval**: updates the status of the survey and displays a message dialog if it is successful. It then navigates back to the Surveys list.

**Back to List**: navigates back to the Surveys list.

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Question/Create (GET) | Controller action method providing the info |
| Create.cshtml | View |
| AddQuestionsViewModel | The model supporting the view. |
| ValidateAntiForgeryTokenAttribute | Action filter to ensure built in protection against XSRF attacks is checked, on the POST action. |
| \_LikertResponses.cshtml | Partial view used to display the responses for the selected Likert Template. |
| MapResponseTemplateToResponsesViewModel | The view model supporting the \_LikertResponses partial view |
| Question/GetLikertResponses | The controller action method, which responds to the ajax request when the Template is selected from the drop down box. |
| \_AddQuestionsList.cshtml | Partial view used to display the questions for the survey. |
| MapQuestionToAddQuestionsListviewMode | The view model supporting the \_AddQuestionsList partial view. |
| Question/GetQuestionsForSurvey | Controller action method, which responds to the ajax request to get the questions for a survey |
| AjaxActionOnlyAttribute | Custom filter to ensure the requests for the GetLikertResponses and GetQuestionsForSurvey are ajax calls, thus preventing them being requested from the browser address bar. |

###### NB.

jQuery is used within this view to control the display of the **Submit for Approval** button, when there are questions, but the input fields are blank (i.e. no question is in the process of being added).

###### Future Development

Code for the Create action method needs to be refactored to abstract the code for model states of valid and invalid to lighten the code in the controller in keeping with the Single Responsibility Principle. Some of the code should be in mapper classes and some should be in the business model directly.

### View Results page

This is the page where the analysis of the responses to a survey can be viewed. The analysis is simple for each question in the survey. The view below shows part of a results page and includes the results shown as a chart.

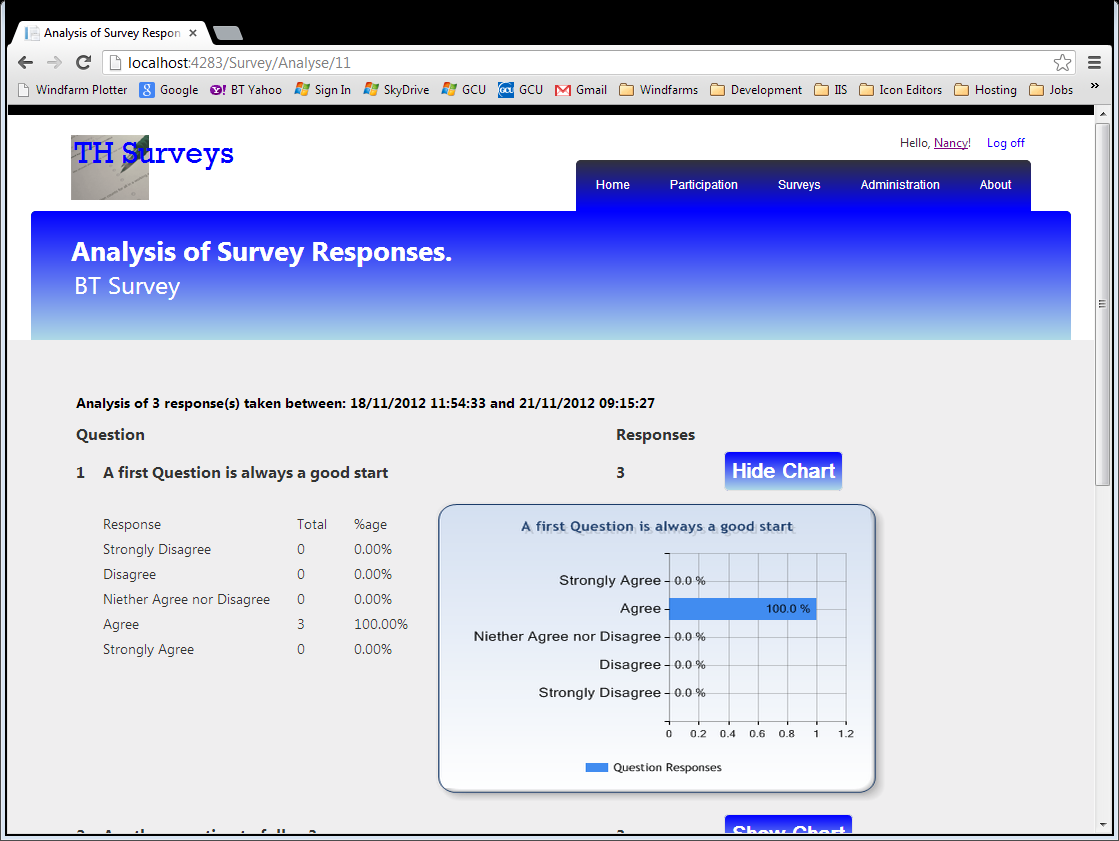


Figure - Analysis of Survey Responses

###### Functionality

All questions for the survey are shown. For each question, all possible responses are shown with the corresponding number of actual responses. Against each question, the total number of respondents is shown.

Clicking **Show Chart** shows the results as a chart and displays it next to the responses.

Clicking **Hide Chart** removes the chart showing the results.

###### Security

This view is secured and restricted to users in the role USER.

###### Links

###### Back to Surveys navigate to the list of surveys for the user.

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Analysis/QuestionList | Controller action method providing the info |
| QuestionList.cshtml | View |
| SurveyAnalysisViewModel | The model supporting the view. |

###### NB.

No partial views are used in this view. This is just to demonstrate another way of coding the Razor views. This results if much more code in the view itself.

The view also makes use of jQuery to control adding the results chart to the view. It is accomplished by using jQuery to add an <img> tag to the DOM, setting its src attribute to the Analysis/GetSurveyChart action method. This causes the browser to retrieve the src, thus executing the action method thus displaying the chart on the view. The action method itself returns a FileResult object, which is the image of the chart

### Approve surveys Page

This page lists the surveys awaiting approval. The Administrator selects a survey to approve by checking the checkbox next to the survey.

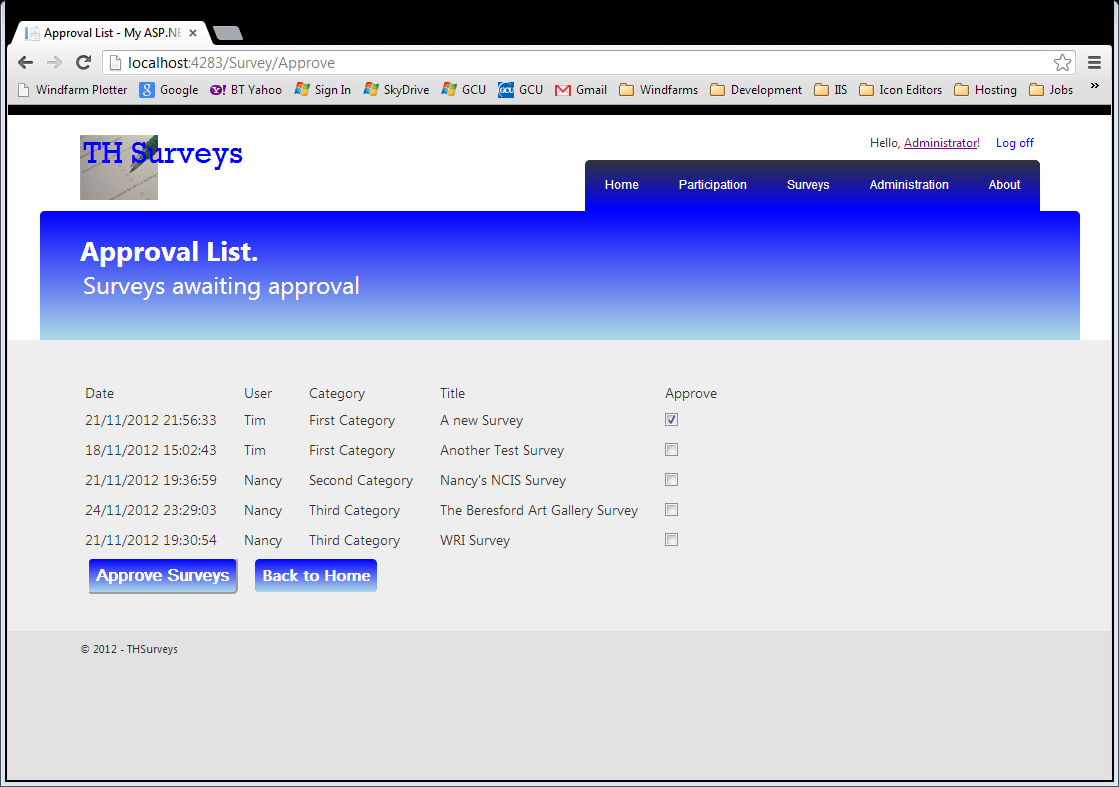


Figure - Approve Survey View

###### Functionality

Selecting a survey for approval is done by clicking the approve checkbox next to the desired survey.

Clicking the **Approve Survey** button changes the status of the selected survey(s) to Live.

###### Security

This view is secured and restricted to users in the role ADMIN.

###### Links

**Approve Surveys** button, upon successful update returns to this view.

**Back to Home** navigates to the Home page of the application.

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Survey/Approve | Controller action method providing the info |
| Approve.cshtml | View |
| ApprovalListViewModel | The model supporting the view. |
| MapSurveyToApprovalListViewModel | Custom Action filter that maps the Surveys to the ApprovalListViewModel. |
| ValidateAntiForgeryToken | Action filter to ensure built in protection against XSRF attacks is checked, on the POST action. |

###### NB.

The ViewModel has been carefully constructed to allow the default model binder to be used. It results in a bit more code in the action method, but this can easily be refactored away from here.

### An Error page

Displays the trapped error should one occur.

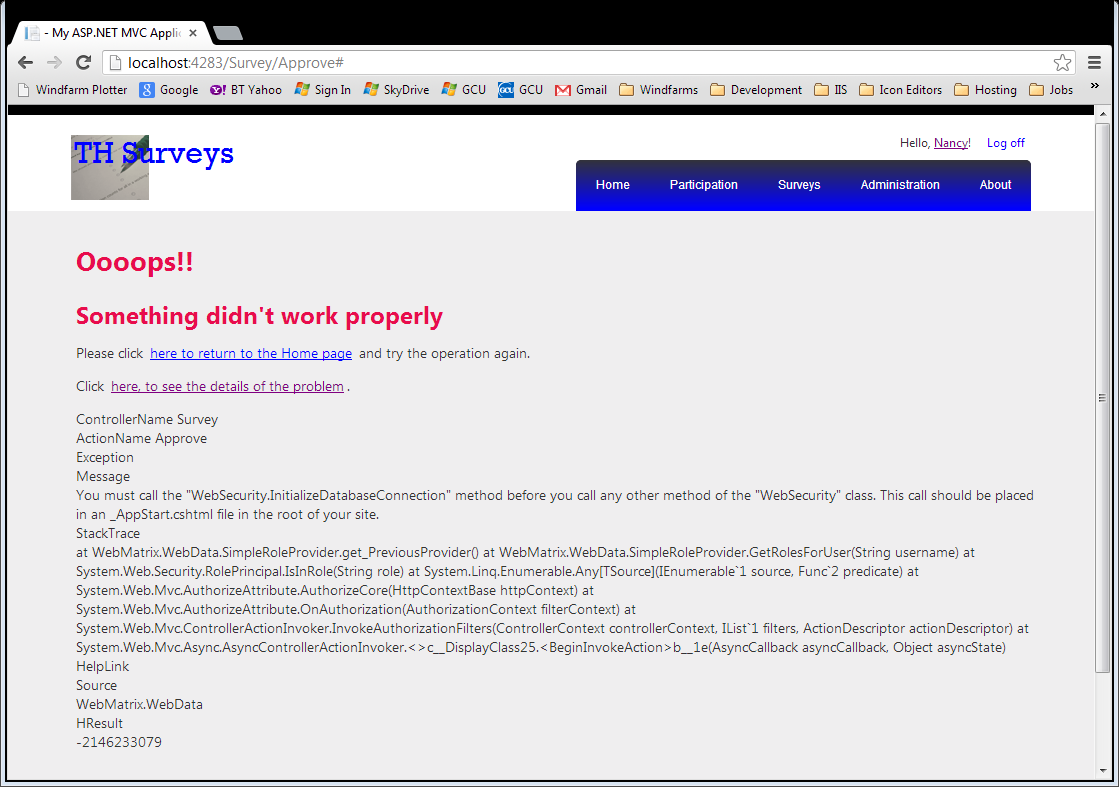


Figure - An Error Page

###### Functionality

Should this error page be shown, clicking on the link to **see the details** shows the error information, which is hidden by default.

Clicking on the **return** link, either returns directly to the Home page, or if the user is logged on, forces a log off and then returns to the Home page.

###### Links

The links are described in the functionality section.

###### MVC Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| Error.cshtml | View for general errors |
| ErrorNotAuthorised.cshtml | View for not authorised errors |
| PageNotFound.cshtml | View for pages not found. This can be rendered by the Home controller, PageNotFound action method, in response to the catchall route defined in the custom routing table. |

###### NB.

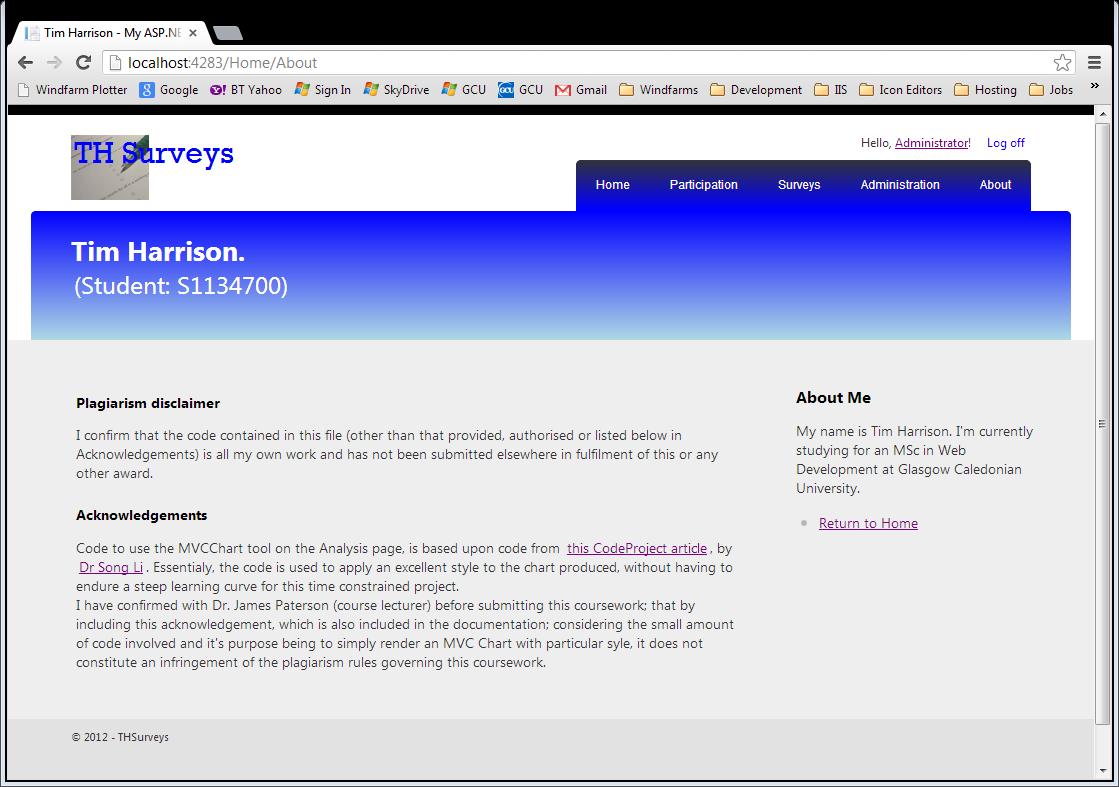
jQuery is used to determine if the user is logged on by checking for the existence of the ‘logoutForm’ DOM element. This check is done this way to successfully recover from exceptions where the membership authentication is the problem. Access anything to do with the authentication will cause a further problem and potential loop. However, even if this is the problem, forcing a logoff still works and clears the authentication problem.

###### Future Development

The code within each of the error views is very similar and should be refactored to a partial view.

### About Page

This page shows the acknowledgements and disclaimer required by the coursework.



# Use of the MVC Framework

## Overview

The MVC framework is extremely customisable by design. Some areas of customisation are more useful than others.

MVC follows a strict cycle to process each request from a browser. It uses a routing table to determine which Controller and action method should handle the request. One of the most useful points of customisation in this process is the definition of this routing table.

Once a controller action is given the responsibility of handling the request, it gathers any information it needs and returns the required view. This tends to be how the controller responds to a GET HttpRequest, and the points of customisation are the use of Custom Action filters (override the OnActionExecuted method) and the use of ViewModels to describe the data for a specific view.

An POST HttpRequest from the browser would make use of either the default or a Custom Model Binder and perhaps the Action filter (override the OnAtionExecuting method this time).

There are many other parts of the MVC Framework where the behaviour can be altered. The following sections discuss those given consideration in the design of this application. The cover the following topics:

Dependency Resolver An implementation of the Inversion of Control or Dependency Injection

Routing a custom routing engine to manipulate the Url to represent a user friendly set of Urls that can be entered directly into the browsers address bar to access application functionality.

Controllers and Actions The part of the framework that handles the communication between the Views (UI) and the Model (business model)

View Models A way of ensuring that only the data required by the view is actually sent and received from the view.

Model Binding The binding of the input received from the request to the view model accessed by the controller action method.

Action Filters A means of providing pre and post processing for the controller action method. This may be to manipulate the data being passed or something quite unrelated like logging requests of validation the request is the correct type.

Html Helper A helper method to allow the Razor view engine render html appropriate for the intended use of data within the view.

The use of other components and their location within the project is also discussed, covering topics such as mapping between view models and business models, validation and security. It is not always appropriate to customise the default behaviour, and customisation should be discouraged when the default behaviour will suffice. However, customisation can lead to clean elegant solutions.

Some approaches within this application are used to demonstrate the different ways to use the framework.

## Dependency Resolver: IoC

The Dependency Resolver is the technique the MVC4 framework offers to apply dependency injection for class dependencies. A custom dependency resolver could be implemented, but this is not a simple task. Far better to make use of a 3rd party framework designed specifically for this task.

Ninject, a NuGet package, is a well proven package intended to provide an IoC to facilitate dependency injection. The MVC4 implementation of Ninject instantiates a custom version of the Dependency Resolver. This is achieved automatically by locating the code in the **App\_Start** folder, and creating a **Bootstrapper**, which is then responsible for launching the application.

Within this solution, some dependencies entirely within the UI require to be bound in addition to dependencies between the Core and Infrastructure projects. Those dependencies within the UI layer are located within the RegisterServices within the app\_start folder. Those dependencies to the Core and Infrastructure projects are located within a special IOCMappings project so that references to both project can be made, without causing circular references in the UI project. This contains a static RegisterServices method which takes in the kernel instance and binds the appropriate modules, adding them to the kernel.

Thus the Ninject Dependency Resolver contains all bindings necessary for the application.

## Routing

Today’s thinking is that the Url is considered part of the applications interface. While most users never actually enter a Url directly, except to access the application, into the address bar, some do.

To keep the application as accessible as possible, its Urls are kept as simple, short and as guessable as possible in accordance with the guideline, the application supports Urls of the structure **Survey/doSomething/anOptionalId**. This means a user has the option of typing any of the following:

/Survey/Top5

/Survey/Choose

/Survey/Take/id of survey

/Surveys

/Survey/List

/Survey/Create

/Survey/Approve

/Survey/AddQuestion/id of survey

/Survey/Analyse/id of survey

A set of custom routes for the application achieves this.

Using a custom routing table decouples the design of the controllers from the url structure, giving the freedom to develop the most appropriate controllers to support the functionality of the application.

Other controller action methods are secured so that typing them directly in the browser address bar will result in a PageNotFound error. The actual additional routes that work for the application do not necessarily follow the above pattern, and should therefore not be guessable.

## Controllers and Actions

The controllers are split to reflect a single responsibility for each, but also taking account of the security aspect.

|  |  |  |
| --- | --- | --- |
| **Controller Name** | **Secured** | **Purpose** |
| Home | Unsecured | Listing surveys and allowing participation |
| Survey | Secured | User maintenance of their surveys, submission for approval, approval of surveys. |
| Question | Secured | Responsible for adding questions to a survey |
| Analysis | Secured | Responsible for providing analysis of the responses to a survey. |
| Account | mixed | Responsible for account registration, logon and logoff. |

Code within the various action methods is kept as light as possible so that its responsibility is, as much as possible, to simply control the interaction between the views and the model behind the application. This involves the abstraction of such things as mapping processes, validating the data received from the view models (UI) and assigning the correct view to render.

This is achieved by the use of Action filters, data mapping classes, model binders both default and custom and the use of Dependency injection for references to the appropriate repositories for accessing the business model.

## View Models

The decision to use ViewModels for each view with the application was taken so that no business model class should be referenced within them. This promotes good separation of concerns between the business model in the Core project, and the UI itself.

This keeps the data structure hidden from the UI and therefore the user. It also reduces to the minimum the amount of data that must be rendered for each view, as there should be nothing included within the view model that is not required by the view. This keeps the UI as light as it possible to support the functionality.

Careful construction of the view model can make it easier to use the default model binder; by designing it so that the default model binder is able to cope with the structure of the model.

## Model Binding

The model binding within the application has been performed using the both the default model binder and a custom model binder. This is to demonstrate both techniques but also it is not always appropriate or necessary to use custom model binding as the default model binder can cope with many situations.

The Approve Surveys uses a more complex view model, but nevertheless uses the default model binding.

However, this is not the best choice here as the validation to check for no surveys being selected for approval is difficult to check for, except within the action method itself since the model state is always valid. This sort of validation should not be done within the action method as this is not one of its responsibilities. It should respond appropriately to the model state being either valid or invalid. The most appropriate place to perform this check would be to use a custom model binder, override the BindModel method and perform the check in there. Adding an error to the model state when no surveys are selected would allow the action method to re-render the view and give appropriate feedback to the user. This would be included in future development to correct and improve the User experience for this view.

On the other hand, the take survey does use the only custom model binder in the application. It is able to respond appropriately when a specific question has not been answered.

## Action Filters

###### Use of Built-In Filters:

TheMVC framework provides various built in action filters used within the application are

**ValidateAntiForgeryToken**: Ensures the anti-forgery token is checked each time.

**OutputCache**: This can be used to improve the performance of the application but should only be used for static data or data that is not particularly volatile, but has not been implemented yet within this application.

**ChildActionOnly**: This marks an action method for execution as a child method and is used to restrict the use of action methods, and prohibit users from accessing them from the browser address bar.

###### Custom Filters

A custom action filter, that overrides the **OnActionExecuting** method, has been written to provide similar security as the **ChildActionOnly** attribute, but for methods that are the target for Ajax only calls from the client layer. It is called the **AjaxActionOnlyAttribute**. It validates the request to be an Ajax request, and returns an Http 404 request if the request is not an Ajax request.

Another custom action filter has been written to validate that the registered user is allowed to access the survey supplied in the request. It also overrides the OnActionExecuting method so that it executes before the action method, and has the opportunity to redirect to a NotAuthorised page if the user is not allowed access to the particular survey.

Custom Action filters are also used for mapping the business models to the view models for specific views. This is used where possible, to reduce the level of coding in the controllers. Overriding the OnActionExecuted() allows the mapping to be done after the action method has completed its processing. For consistency throughout this application, this technique is adopted for mapping date to the view model on the GET action methods.

It allows standard and common mappings to be completed simply by adding an attribute to the relevant controller.

###### Error handling Filters

The standard HandleError attribute is used to provide the error handling for the application. This is in conjunction with the configuration within the **web.config** file to set the default 404, 401 and 402 errors. All controllers are marked with the **HandleErrorAttriibute** at the class level so that it operates for all action methods within them.

This provides a mechanism to improve the robustness of the application and its ability to handle error conditions gracefully, therefore improving the user experience.

## Mapping Classes

Various classes and ViewModels are mapped between each other within the application. A series of mapping classes are provided to do this mapping.

As discussed in the Action Filters section, to keep a consistent approach throughout the application, and attempt to keep the code in the controller action methods as light as possible, following the Single Responsibility principle, mapping tasks are abstracted out of the controller action methods.

For controllers that respond to HttpGet requests, the mapping is generally performed using the AutoMapper package, with the code being placed within an ActionFilterAttribute. These Action filters are placed within the UI project under the Filters folder.

The HttpPost action methods do not use AutoMapper, instead use specific mapping classes, as they often need to use repositories to retrieve data to complete the mapping process. It is not a good idea to perform this type of processing within an Action filter as it can cause problems with the data context, such as the context being disposed by the time the filter requires to access it. These mapping classes are held within the UI Mappings folder, and called from within the action method with one line of code, still keeping the action method code light.

## Html Helper

It is possible to enhance the abilities of the Razor view engine to render the correct html required fo the application. Here, to be able to display the possible answers to a question using a series of radio buttons poses the problem of grouping the radio buttons for each question, otherwise the standard RadioButonFor<> helper method will render such that only one radio button can be selected on the entire form, which is no good in this situation.

A custom Html Helpers is written to allow a group id to be passed into the helper and allow the group of radio buttons to be limited to this for each question, allowing multiple groups of button to exist on the form at once. The custom helper is RadioButtonListFor<> and is used in the Take Survey view.

## Validation

Validation can be performed at many levels within the application. The business model would perform validation to allow for use with different UI layers, and not rely on the UI to perform validation. Validation is possible within the server side of an MVC application, by the use of data annotation attributes on the view models, as well as specific validation performed within model binders. There is the option of performing validation on the client part of the UI, to improve the responsiveness of the application and give users more immediate feedback. This tends to be performed using jQuery (JavaScript) based validation.

Validation on the Business model would be strictly the implementation of business rules and logic. The UI validation would not necessarily be concerned with business logic but instead ensuring the validity and accuracy of the data being returned from the views.

Within the MVC framework, it is possible to define the validation rules using the data annotations on the view models, and this is accessible by both server and client side validation. This application therefore, implements the validation this way and makes use of the client and server side validation.

Although there is not a huge amount of validation required by this application.

The framework uses jQuery-validation and unobtrusive ajax to perform the client side validation, it is switched on in the **web.config** file, by adding the following to the **appsettings** sections

<add key="ClientValidationEnabled" value="true" />

<add key="UnobtrusiveJavaScriptEnabled" value="true" />

# Division of Functionality

With an eye on the user experience of the application, where it needs to be responsive the logic and coding should be performed on the client layer. Where the information required resides on the server layer, communication should be asynchronous to allow the UI to remain responsive while the request is being dealt with. The application makes use of partial views so that only the part of the page that is relevant to the request is updated. The server side is reserved for persistence of data, implementation of the business logic, and more complex processing.

## Client

Where appropriate, use is made of jQuery to keep the view as responsive as possible. It is used when performing validation, manipulating controls directly on the view as well as controlling the asynchronous calls to the server side action methods whenonly parts of the UI need to be refreshed with data.

The input views, such as Creating a Survey, Adding Questions to a Survey, Approving a Survey and actually taking a Survey make use of this. The Survey Analysis, and Choose surveys by Category also make use of asynchronous calls to the server side when requesting an Analysis chart and the surveys for a particular category.

## Server

Functionality is kept on the server side for operations where the data model is involved, along with operations of a more complex nature, such as the analysis of survey results.

The beauty of server side code is that it can respond to full http requests, when the entire page is being refreshed or changed, as well as serving up partial views for a page. The use of partial views is equally possible when responding to Ajax requests, which when called can perform the complex calculation or data model communication, without holding up the UI.

Calls to external resources can be made asynchronously for the server side also, keeping things ticking over while the request is serviced. This part of the server side has not been well used before as the coding model was cumbersome and difficult, but the Task based asynchronous behaviour implemented by .Net 4 means this should become more widely used, even for things like database access.

## Core

The core project is effectively the implementation of the Business Model within the application. It is a separate project to ensure separation of concerns between the UI layer, the MVC project, and the business model.

It exposes the various interfaces through which the UI later communicates with it; often using the dependency resolver implemented using Ninject. The logic of the business model is implemented here, which includes changing status of the survey and the provision of an analysis service for a survey.

In this instance the business model is very simple and is basically a series of POCO classes with a few methods to provide some limited logic to implement. However, far more complex models would still be implemented in this way.

This approach allows different UI layers to interact with this business model and means the UI layer can concern itself purely with UI concerns

## Infrastructure

The Infrastructure project, in an application this size, basically provides the Data Context (database access as EF Code first is used based on the Code Model classes) and the concrete implementations of the various repositories used by the UI. The interfaces for these repositories are made available to the UI in the Core project. It is essentially a Data access layer in this application.

It does, however, promote further separation of concerns and allows components to be interchangeable readily.

The use of Dependency injection along with the Unit Of Work pattern allow the Db Context to be injected into the various repositories, retaining this separation of concerns and facilitating easier unit testing.

# Cross cutting Concerns

## Security:

### Protection and Robustness against attack

All secured action methods should do the necessary checks to ensure the Url is not entered from the address bar in an attempt to circumvent the security. Additionally, all methods that receive POST requests should ensure that the data being received is from a recognised source, namely only the applications views.

The common types of attack fall into the following categories:

###### XSS Cross Site Scripting

The MVC Razor engine automatically guards against this type of attack by ensuring that all html content is encoded by default. Use of the **Html.Raw** helper will disable this encoding, if required. This application has no need to make use of the @Html.Raw helper and therefore derives its protection against XSS type attacks by default.

###### XSRF Cross Site Request Forgery

MVC 4 framework provides an **Html.AntiForgeryToken** helper and a **ValidateAntiForgeryTokenAttribute**, designed to protect against this type of attack. It works by generating a random token, which is placed within the form, using the helper, when it is rendered. The attribute, which marks a POST action method, validates this token and passes the request to the method if the token is valid. This protects because sites are not able to get at the token placed within the form.

This only works on the POST request, so to help protect against this type of attack , HttpGet requests should never update anything. This application follows this principle and also implements the AntiForgeryToken on the form posts.

###### Cookie Stealing

This is where external JavaScript can access cookies to gain information and access sensitive information. This can be disabled by adding the following **<httpCookies httpOnlyCookies=true>** within the System.Web section of the Web.Config file. This can be overridden should the need to access a cookie arise, but this application has no need to use cookies so script access is disabled.

###### Open Redirection

This is where the url contains a redirection url. This should be checked to be local, so that only redirection to view local to this site can be used. The only place within the application where this redirection occurs is at the logon stage. The Account controller methods should be changed to check for this using Url.IsLocal(redirectedUrl). This is currently not implemented, due to time constraints and the amount of testing required to ensure I don’t break the application. However, this would be implemented as future development, as it is not difficult to protect against and therefore the protection offered by the framework should be taken.

### Membership, authentication and Authorisation

The application has a requirement to secure certain operations and views. The MVC Internet application template provides a Simple Membership and authentication mechanism that can be integrated with the data model.

This allows the definition of users that would be allowed access to the secured part of the system. Users can be attached to a role to further restrict access.

This allows use of the **AuthorizedAttribute** to secure those controller classes that need to be. Additionally, the specific methods that need to be restricted to certain roles can have the AuthorizeAttribute added specifying the role that can access it.

Part of the decision process when determining which controllers should be created, is to take account of the security angle, and keep secured and unsecured as much as possible to different controllers.

Creation of Custom Authentication Attributes is neither needed by this application nor encouraged.

* They are difficult and require vast amounts of unit testing. Security coding is a huge task in its own right and many developers have spent eons developing security related code. The MVC framework comes with built-in security including authentication and membership routines. Given the short timescales for developing this application, it would not make sense to attempt to code custom security.
* As a rule, you should never consider custom security engines, save to enhance the default with some specific requirements. DO NOT RE-INVENT THE WHEEL.

## Exception Handling

The standard error handler mechanism is used. Activated from within the web.config file, but adding

<customErrors mode="On" defaultRedirect="~/Error.cshtml">

<error redirect="~/ErrorNotAuthorised" statusCode="401" />

<error redirect="~/PageNotFound" statusCode="404" />

</customErrors>

to the system.web section.

The specific pages are stored in the view/shared folder. Each error page is designed to allow the error information to be viewed and also a link to return to the Home page of the application.

Depending on whether a user is signed on or not, the route to the home page might be by logging off the application.

This approach is adopted as some errors occurred in debugging, where the Razor engine is checking the **User.Identity** contained within the request, but the connection to the web security has broken and the **InvalidOperation** exception occurs. This is clear by logging off the application.

This demonstrates the security problem mentioned in the Exception handler section. Here, the Administration page was selected from the menu, but with the app running for some time, the membership and authentication seems to have stopped. It causes the “Invalid Operation Exception” with the reason stated in the sample. Clicking the link to “return to the home page” will force a logoff (using jQuery to check the form to determine if the user is still logged on, as the link to the membership seems not to be working) and return the user to the home page. This seems to clear the problem with the membership database and allows the app to continue.

## Code Modules and Patterns

### Unit of Work Pattern

Entity Framework (EF), when performing updates to the data context can experience problems when multiple objects are updated within the same **SaveChanges** operation. When information is retrieved using EF, it sets up a change tracker so that it can determine what changes are made and how to apply them to the database. If the source of the updates comes from different read operations, multiple instances of the change tracker are created. The update will then fail because EF cannot determine the changes correctly across all change tracker instances. To avoid this, each read operation must use the same instance of the DB context so that a single change tracker is used.

The Unit Of Work pattern is adopted, in conjunction with Ninject to control the lifetime of the Entity Framework Data Context. All repositories access the DB context through this Unit of Work, which is injected into all repositories. The instance of the DB context is maintained by the IoC container, which ensures the same instance is used for the lifetime of the HttpRequest.

### Inversion of Control (DI)

Use of the Ninject Dependency Injection package is made, specifically the MVC4 version, and used to implement Dependency Injection for the application. Ninject follows the MVC4 implementation by providing the container as a custom Dependency Resolver.

It is used to inject concrete instances of class in the Infrastructure project, allowing the UI to reference the interfaces of abstract classes defined in the Core Project. This maintains the separation of concern between the three main projects.

### Factories

The use of Factories and Abstract factories is made to remove the creation of dependent classes from the main code and avoid using the **new** keyword.

Most were set up as static classes originally which work perfectly, but this did not help the unit testing aspect, so were changed to abstract factories to make it possible to mock the creation of the classes within the unit tests.

Not all factories were changed to abstract ones. Future development would be to refactor this so that all factories used are abstract ones.

# Unit Testing

### Scope of Tests

A major emphasis of this project is to demonstrate the testability of the various components and customisations made within the MVC framework.

The testing in this project however, is not exhaustive, rather an indication of some techniques that can be employed to define unit tests.

The MSTest framework, delivered with VS is used to run the tests. Where appropriate, the Moq framework is used to mock objects and limit the unit testing to the logic within the class being tested.

Unit tests are included for each area discussed in the remainder of this section.

### Structure of Tests

The structure of all unit tests is the same, namely Arrange, Act and Assert.

**Arranging** a unit test can consist of many things, but mainly the need to create an instance of the class being tested, along with the mocking of any dependencies the class may have and creating an instance of the expected input, which may have to be mocked also.

**Act** is the execution of the class method being tested.

**Assert** is the checking of the expected output from the method being tested.

The purpose of each unit test has to be kept in mind when constructing the tests. The purpose of the unit test is to test the logic within the method only, and exclude from this any dependencies that the class has.

The MVC Framework, with its many areas of customisation, lends itself to this type of testing. As with any project, it is worth spending the time creating these tests to minimise the effect of changes within the application.

## Areas of Testing

### Custom Routes

The test on the custom routing table was the one area where it was extensive. The unit tests are held within the Routes folder of the Test project.

With the Url structure and the resulting routing table determined late on in the project, it was necessary to design the route table completely separately from the application. Once tested, it would then be plugged into the application without breaking it, or at least minimising the amount of debugging required to integrate the table. This is akin to a Test Driven Development of the Routing table.

To test the routes both incoming and outgoing urls must be tested. A complete analysis of all controller action methods and the intended url must be identified and specific test designed to check the functionality.

In the test project, the tests are grouped into the Incoming Tests and Outgoing Tests. The Routing table class in the test is where the routes were originally tested, before plugging them into the application itself.

The specific tests are numerous but basically check that a Url routes to its expected destination, that a malformed Url drops through to the catchall and that the appropriate Url is generated when using the controller action. The document, RoutingTable.xlsx shows the intended routing that is being tested within this unit test module.

### Controllers

As a sample of controller action method tests, the Take survey process was chosen as it involves the most customisation and demonstrates how the individual components can be unit tested knowing how they integrate within the application. Other than that, there are too many controllers and action methods to adequately provide units for them all within the time constraints of this coursework.

The types of tests suitable for an action method handling a GET request are:

Check the expected view is rendered

Check that any expected data is correctly contained in the context.

The types of tests suitable for action method handling a POST request are:

The appropriate redirection or view is rendered if the model state is valid

The original view is re-rendered if the mode state is invalid.

The model state can be mocked by adding a model error to the context: **controller.ModelState.AddModelError(“”, “test error”).**

The assertions can be for the correct view and then to check the contents that the expected data and/or error messages exist within the context.

In this test project there are tests for Home controller:

|  |  |
| --- | --- |
| IndexReturnsListOfSurveys() | A list of surveys is returned where the surveys are found. |
| IndexReturnsEmptyViewIfNoData() | The view still returns successfully with an empty list if there is no data to display |
| TakeRedirectsToThankYouWithResultsOnSuccess() | If the data returned is valid the controller should redirect to the ThankYu view and include the results of taking the survey |
| TakeReturnsViewIfModelInvalid() | If the data returned is not valid, the TakeSurvey view is re-rendered with the original data repopulated |
| About() | The default sample test provided by the Test framework |

### Custom Model Binders

The take survey process has the only custom model binder, **TakeSurveyModelBinder**, within the application so the unit tests are centred round this module.

To test the model binder, we must mock the incoming controller context and the binding context, and populate them with the necessary data, either valid or invalid as the test dictates. The HttpRequest needs to be mocked but the controller and binding contexts do not.

We assert the test by checking that the model state is as expected and that the data exists in the view model.

In this project the custom model binder is tested for the following possibilities.

|  |  |
| --- | --- |
| TakeSurveyBinderOKWithValidData() | Passes populated ViewModel with data with the modelstate as Valid when the data is valid |
| TakeSurveySetsModelStateToFalseWithInvalidData() | Passes a modelState error if the data is invalid. |

### Action Filters

The only non-mapping action filter, namely the AjaxActionOnly filter was chosen as the module to test. This was a good example in that it shows the limit of the basic tools available for unit testing.

This module basically checks that the incoming HttpRequest is an ajax originated request. It does this by checking the IsAjaxRequest() method on the HttpRequest.

As it turns out, MS have implemented this method as a Static extension method. Moq cannot deal with either of these class types.

This left the choice of finding a technology that does, which points us to MS’s own Fakes and Shims testing tools. The way to mock or intercept a static or extension method is to employ a shim, which essentially rewrites the code for the object at runtime. It appears simple to use on the face of it, but causes major problems with MVC and causes a severe error “**Operation may destabilize the runtime**”. This is well documented on the internet and basically demonstrates that the tools are, for the moment, incompatible.

Time constraints of the project mean that this test is left not functioning but not replaced. The code, however, is still in the project but the libraries have been removed as they stop any other tests from running.

### Html Helpers

The only Html Helper was also created as part of the Take Survey process. The **RadioButtonListFor** helper method was created to allow multiple groups of radio buttons to be displayed on the view at once. This represents the options for each question. The existing RadioButton helpers leave all radio buttons with the same name and therefore only one would be selectable from the entire set of questions. This helper groups each of the options for each question allowing the respondent to answer each question.

To test this helper, Moq I used to create mock versions of the ViewContext and the ViewData Container. An instance of the viewModel is created and populated with appropriate values. This arranges the test.

The output from the helper is an MVCHtmlString, containing the Html to be rendered in the view. This string can be interrogated to see if it contains particular elements and attributes. This is how the tests are asserted.

For this helper the test involved are contained in the Helpers folder and execute the two tests

|  |  |
| --- | --- |
| RadioButtonListRendersOK() | Passes if the html string contains all the required elements and attributes |
| RadioButtonListRendersChecked() | Passes if the html string contains the ‘checked’ attribute as well. This is set if the supplied checkedValue is the same as the actual value. |

### Mappings

There are various mapping classes to convert the view model classes to the business model classes that tend to be used when processing POST HttpRequests, these are not done with action filters rather with specific classes as we often need to reference the data through the repositories and want to avoid the possibility of problems with these calls. By calling these classes directly from the controller action method, the data context is guaranteed to be available and not disposed of.

There are two mapping classes associated with the Take Survey process: namely **MapTakeSurveyViewModelToSurvey**, called if the viewmodel is successfully bound and the model state is valid, and **ReinstateTakeSurveyviewModel**, which is used to reinstate the static information in the view model should the model state be invalid.

Arranging the tests involves using Moq to mock the dependencies, so that it is simple to override the behaviour of these dependencies. The class being tested can be instantiated directly and the Map method executed.

Asserting the results involves checking the expected data is contained in the output class. In the case of the test where the survey isn’t found, we execute the Map method from within a try…catch block, so that we can catch the null reference exception that should be thrown.

Four tests are included within the Mappings folder and perform the following:

|  |  |
| --- | --- |
| MapTakeSurveyVMToSurveyOK() | This maps to the survey if the data is valid and the survey exists |
| MapTakeSurveyVMToSurveyFailsIfSurveyNotFound() | This fails if the data is valid, but the survey cannot be found |
| ReinstateTakeSurveyViewModelOK() | This reinstates the view model data successfully if the model state is invalid. |
| ReinstateTakeSurveyViewModelFailsIfQuestionNotfound() | This fails to reinstate the view model data when the question cannot be found. |

# 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 must be refactored to eliminate such namespace mismatches.

# Acknowledgements

## Use of the MVC Charting tool

To avoid a steep learning curve and to understand how it works, various sources were investigate on the internet. A specific CodeProject article [Using ASP Net Charting with Image Map in MVC](http://www.codeproject.com/Articles/297677/Using-ASP-Net-Charting-with-Image-Map-in-MVC) proved to be the main source, by [Dr Song Li](http://www.codeproject.com/script/Membership/View.aspx?mid=5002996). The code used in the project is similarly structured but not identical, except for the code that sets the Chart settings, essentially the style of the chart.

Having checked beforehand, with Dr. James Paterson, that this is not plagiarising any code, as it doesn’t constitute a major part of the work, I have chosen to use the style from Dr. Song Li’s article.

1. There will be no restriction placed upon a respondent to ensure they take a particular survey only once. Such a feature is considered as Future Development.

   [↑](#footnote-ref-1)