Cohaesus Projects Ltd

IMPLEMENTATION OF THE ENTITY FRAMEWORK CODE-FIRST APPROACH IN A .NET MVC APPLICATION

By

Andrew James Dick

Technical Report

Submitted to

Matt Meckes

of Cohaesus Projects Ltd

in Partial Fulfilment of the Requirements

for the Technical Report Segment

# Contents

[Contents 2](#_Toc461995979)

[Introduction 1](#_Toc461995980)

[Technology Review 2](#_Toc461995981)

[SQL 2](#_Toc461995982)

[Relationships 2](#_Toc461995983)

[Keys 3](#_Toc461995984)

[Functions 4](#_Toc461995985)

[Stored Procedures 4](#_Toc461995986)

[Schema 5](#_Toc461995987)

[Indexes 5](#_Toc461995988)

[Entity Framework 6](#_Toc461995989)

[History 7](#_Toc461995990)

[Workflows 8](#_Toc461995991)

[Code First Migrations 10](#_Toc461995992)

[TestRail Application 11](#_Toc461995993)

[Database Design 11](#_Toc461995994)

[6.1.1 Application goal 11](#_Toc461995995)

[6.1.2 Database Structure 11](#_Toc461995996)

[Project Setup 13](#_Toc461995997)

[6.2.1 13](#_Toc461995998)

[Seeding the database 16](#_Toc461995999)

[Project Authentication 18](#_Toc461996000)

[Data Models 22](#_Toc461996001)

[Identity 22](#_Toc461996002)

[Projects 24](#_Toc461996003)

[Test Case 25](#_Toc461996004)

[Application View Model 27](#_Toc461996005)

[Controllers 28](#_Toc461996006)

[User Base 29](#_Toc461996007)

[Application Users 29](#_Toc461996008)

[Home 31](#_Toc461996009)

[Projects 32](#_Toc461996010)

[Test Case 34](#_Toc461996011)

[Views 37](#_Toc461996012)

[Layout 37](#_Toc461996013)

[Application User 38](#_Toc461996014)

[Projects 40](#_Toc461996015)

[Test Cases 42](#_Toc461996016)

[Conclusions 43](#_Toc461996017)

[Further Work 44](#_Toc461996018)

[Acknowledgements 45](#_Toc461996019)

[References 46](#_Toc461996020)

[Bibliography 47](#_Toc461996021)

[Glossary 48](#_Toc461996022)

[Appendix 50](#_Toc461996023)

[User Requirements i](#_Toc461996024)

[Sitemap iii](#_Toc461996025)

[Wireframes i](#_Toc461996026)

# Introduction

Applications traditionally depend on database services to handle their data storage. Where database services interact with their data via rows and tables, applications (traditionally written in higher-level programming languages) access theirs via classes, methods, etc. This can (and usually) results in a layer of resistance between the two, resonating especially with developers unfamiliar with the SQL environment. Bloated queries will hamper performance, and any structural changes to the database or application architecture would subsequently increase the level of resistance. Object/Relational Mapper frameworks attempt to counter this by raising the level of abstraction from relational to an entity level. Instead of manually scripting ADO.NET code for data access and retrieval, these frameworks aim to provide an automated platform for developers to access database data via domain-specific objects using LINQ queries.

The primary aim of this project is to investigate the integration of the Entity Framework, an Object/Relational Mapper, in a .NET MVC application. The report will document the construction of a simple .NET application that allows users to create and store projects, populate the projects with test case scenarios, and add users to the project. The evaluation of the application will focus on the Entity Framework’s key aspects, architecture, and its ease of integration with any associated technologies.

# Technology Review

This chapter will discuss the key concepts of the two major technologies that will be utilised by the project application; SQL, and the Entity Framework.

## SQL

Structured Query Language (SQL henceforth) is commonly referred to as a relational database. A more appropriate description would be a special programming language that is used to communicate with a Relational Database Management System (RDMS). SQL has a number of key features that provide multiple advantages over a regular database. Like most other databases, it can retrieve and manipulate data. It can also restrict user access by providing secure authentication via a login – including Windows Authentication - to gain access. It can also perform backups and data restoration, process data through highly-efficient compiled stored procedures, and automate tasks to run on a fixed schedule.

### Relationships

SQL allows us to create relationships between pieces of data. There are four potential relationships that can be administered in an SQL database. The most common form of relationship is usually One-to-Many. Many, in this instance, does not always have to be used to full effect. A user could belong to a single project, many projects, or even not belong to any; the distinction is that they can have, rather than they must. In a visual representation of the one-to-many relationship, the ‘one’ is identified with a key symbol, where the ‘many’ is identified with an infinity symbol.

Another common relationship is the Many-to-Many variant. There are several approaches that can be taken to model this. An additional or repeating column could be added to the project database table to accommodate a second user, or a list of comma separated values could be added in the original actor column. However, both of these methods are seen as bad practice and leads to poor database design. The recommended approach is to remove the author column entirely and add a junction/linking table, and name the table the concatenation of the two parent tables. The joining table creates two One-to-Many relationships, since you cannot directly express a Many-to-Many relationship in relational databases. Project and User would both be assigned the ‘one’ side of the relationship, and ProjectUser would be assigned both ‘many’ counterparts.

Although a One-to-One relationship is possible, it is not very common in practice. Since database tables do not have to connect to other tables to exist (or be utilised), none could also be considered an official relationship.

### Keys

Relationships are usually based on keys. A key is a unique identifier for each row in a database table, and is referred to as the primary key. This key can be referenced elsewhere to establish relationships between other database tables. A primary key that is referenced in another database table is referred to as a foreign key. In reference to the relationships described previously, *one* refers to a primary key. Conversely, *many* references the foreign key.

Using databases keys leads to *referential integrity* (Chapel, 2016). Referential integrity is therefore a data property that ensures that any foreign key referenced in a table exists as a primary key in another.

### Functions

An SQL function exists to provide calculations on data, and only returns either a single scalar value or table data. Every SQL server database comes with its own set of built-in functions, which can be found within the database with the *fun\_* prefix. Each function contain a parameters folder and the format of the output value. The function’s input parameters are then added within a set of parenthesis. All parameters begin with an *@* symbol, and have their data type defined afterwards via *AS, e.g. @FullDate AS DATETIME.*Additional input parameters can be added optionally using a comma separated list within the parenthesis. A *returns* statement is then added outside of the parenthesis alongside its requested data-type, and an *AS* statement will then define what operations the function will perform. Although not required, a *BEGIN* and *END* block is an efficient, neat way to encapsulate the operational logic.

### Stored Procedures

A stored procedure is simply a group of SQL statements grouped together under a single heading. Without them, code would need to be rewritten for each data query. With the stored procedure, the collection of statements can be created and then executed within another query using the *EXECUTE* statement. Typically, a Stored Procedure changes data in the underlying tables, but it is able to return a value (it can return a 0 or a 1 to indicate success or failure, respectively).

### Schema

A database schema can be thought of as the table design architecture for a database, similar to a blueprint (Welling and Thomson, 2003). It is a collection of database objects associated with one particular database schema name. This allows a logical grouping of tables, procedures, and views together in a database, leading to a higher level of organisation and increased code readability. The schema itself does not contain any data, but instead displays the column names, as well as any associated primary and foreign keys. If a table is created and no schema is specified, SQL will automatically add the default **dbo**to your object. You can also give permissions to a schema, so that users only have the ability to view schemas they are assigned to.

### Indexes

Indexes exist in SQL to allow for faster searching of specific columns of tables in a database. Creating an index that contains regularly referenced columns allows the processor to calculate the position of the requested data at a faster rate than searching through the entire table, leading to increased system performance. However, if most of the rows in a table are regularly processed, sequential remains the preferred method of searching. Indexing is not so much of a problem on smaller tables, and therefore will be technically considered out of scope for the purpose of this paper.

## Entity Framework

The Entity Framework is an open-source ADO.NET framework provided by Microsoft, and has become its core data access platform for the construction of .NET applications. It provides developers with tools to automate the manipulation, retrieval, and storage of data in a database; and allows them to structure their code based on their respective business model, rather than being confined to the structure of the database itself. Microsoft defines its Entity Framework as:

*“...an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects, eliminating the need for most* *of the data-access plumbing code that developers usually need to write.”*

As an object-relational mapper, the Entity Framework circumvents a potential impedance mismatch between relational database data and domain class objects. It achieves this by creating data access classes for the database. Objects can then interact with (or be the basis of) these classes, based on configured object-relational mappings (O’Neill, 2008). These data access classes are commonly referred to as the Entity Data Model, and free the user from writing code that directly accesses the database data via ADO.NET. This effective separation of concerns makes the application more robust, since it will be easier to support and maintain over a longer period of time.

### History

Originally released in 2008, the Entity Framework was included with the .NET framework 3.5 Service Pack 1 and Visual Studio 2008 Service Pack 1. Although it was released as EFv1, it was also commonly referred to as EFv3.5, correlating with the .NET framework it was released on. The framework provided basic OR/M support using the Database First approach (see Workflows, below) in the EF designer.

Entity Framework v4.0 introduced new key features such as lazy loading, whereby related object data is not loaded until it is specifically requested (usually via the *virtual* navigation property), and Plain Old CLR/C# Object (POCO) Support. POCOs are simple entities of the domain that allow you to define your object model idiomatically, without having to have your objects inherit from Entity Framework's Entity Object (Lerman and Miller, 2011). This essentially provides a greater degree of freedom when designing and implementing classes, since they possess fewer requirements in order to work correctly. Version 4.0 also saw the release of an alternate workflow - Model first - allowing developers to construct their model through the use of a designer tool. The framework's third workflow was released in v4.1, Code First. This was further utilised in v4.3 when Code First Migrations were added to the repertoire, allowing developers to incrementally change a database created using the Code First workflow as it evolved. The EF v4.x framework also included its independent release as a package on NuGet, and adopted the [http://semver.org](http://semver.org/) standard for semantic versioning.

Entity Framework v5.0's arguably defining feature was that it was released as open-source for Visual Studio 2010 onwards. This allowed developers to contribute to the development of the framework by addressing bug fixes, or implementing features they felt would benefit the framework. Although the framework was open source, the product was still shipped with Microsoft licenses and keys for those that were only interested in using it. EF v5.0 also provided a number of small changes to the EF Designer, such as multiple diagrams per model. The EF5 NuGet package also came installed by default on any new ASP.NET or MVC projects created in Visual Studio 2012.

On its latest major release, EF 6 hosts an extensive list of available features for models created with either the Code First or the EF Designer. The release is designed to build applications that utilise .NET 4.0 and 4.5, and must be created in Visual Studio 2010 (by download) or later. The framework is now completely separate from Microsoft’s .NET framework (an entity, if you will), and the runtime can be installed via NuGet to allow shipping out-of-band between releases of Visual Studio. Although no new tools were added from v5, the majority of features were adapted to work with models created in either the code-first or the Entity Framework designer.

### Workflows

The Entity Framework facilitates multiple approaches to access a relational database from a .NET application, by supporting four main development workflows, and uses two main considerations to determine the most appropriate approach.The first consideration is whether the application will connect to either a pre-existing populated database, or a newly-created one. The other consideration depends on the developer’s preference on generating the database model. It will either be generated using a designer tool, or by writing code. These two considerations lead us to four potential options:

#### Model first | New Database

The model is created in the designer using boxes and lines inside a designer tool. The database is then created from the model, and the classes are then auto-generated based on the boxes and lines drawn in the designer.

#### Database First | Existing Database

Database first involves reverse engineering a boxes and line model using the designer. Classes are then auto-generated from the model. You can then tweak the mapping in the shape of the classes in the design surface. Classes are then auto-generated from the model, based on the boxes and lines drawn in the designer.

#### Code First | New Database

The code first approach allows us to define the model using code. The model is made up of domain classes that allow you to interact with the application. Optionally, you can supply additional mapping and configuration code to further specify the model. The database is then created from the model. If the model changes at any point, you can use code first migrations to evolve the database (see Migrations, below).

#### Code First | Existing Database

Similar to the code first approach, where you still define the model using code, but this time it is mapped to an existing database. Tools are available to reverse engineer the model

### Migrations

Once a mapping has been created between a relational database and an object-orientated application, modifying the application model would cause future updates to the database to fail, since the context has changed after the database has been created. We can rectify this mismatch by enabling code first migrations. Migrations calculate any differences between the current database context and the application model, and display the changes in the Migrations folder before the user pushes their changes and updates the database.

When migrations are enabled, a Migrations folder is created along with a Configuration class file. The configuration file is designed for developers to configure migrations within their database context, and allow for additional methods to be created, e.g. seeding the database with dummy data, rolling an application back to a previous migration.

Migrations are executed via the Package Manager Console in Visual Studio. The add-migration command calculates any differences between the database context and the application model, and update-database pushes the changes to the database.

# TestRail Application

*Divided into numbered and headed sections. These sections separate the different main ideas in a logical order*

## Database Design

### 6.1.1 Application goal

One of the most important questions to consider when structuring a database, is to determine the exact use for your database. It is simply not enough to answer 'the database stores necessary product and other information'. We should instead consider the goals of the application it is tied to? For the purposes of this paper, the goal of the database is:

"The application will provide a platform to allow a team of developers to generate test scripts for web development projects. Developers should be able to create individual test cases for each feature, functionality or requirement of a project. This will include the case's priority, any preconditions that must be satisfied before the test can take place, the steps to emulate the functionality, the state of the case (e.g. whether it is passing or failing), and any comments relating to it's overall status.The test cases can then be added to a Test Pass, which will allow the user to record the state of each test case at a given point in time. The cases can then be assigned to developers and any milestones associated with the project."

### 6.1.2 Database Structure

Since no user data previously exists, a new database will be created to store the site's data. With the application goal in mind, we can plan the database's architecture by breaking key features down into tables. A User table should store each of the developer's personal details, and should be restricted only to details associated with their personal account. A Projects table will be created to store all projects that are entered into the database. A TestCase table will store all of the attributes tied to each test case.

With the tables in place, we can then focus on the individual table columns. When designed correctly, the collective columns should cover the entire functionality of the application: The User table should store the user’s personal details, such as their name and email address. A Projects table will store the name, unique code, and description of each project that is added to the database. The Test Case table will store it's associated project, the case title, a short description of the expected functionality, the current state of the test case, and any comments relating to the test case.

**6.1.3 Relationships**

Since a user can be assigned to many projects, and a project should be able to have multiple users, a many-to-many relationship will be established between the two tables. Many test cases will be associated with a single project, resulting in a one-to-many relationship between the two.

**6.1.4 Creating the Database**

Since we are adopting the code-first workflow approach, we will now generate the database by creating classes based on the database design. As this is a new project built in Visual Studio 2015, the Entity Framework NuGet package comes pre-installed.

**6.1.5 Web.config Connection String**

To connect the project to our database, we will need to edit the web.config file. We will amend our DefaultConnection in the connectionStrings element. Since we are using our local machine during development, we can create a default connection that will use our localdb to store our database. To do so, we will remove the default connectionStrings element, and create our own. We will call this connection ‘ApplicationDbContext' by adding it to the name property. For the connectionstring property, we need to add three properties: the data source; the database that will be created, and any integrated security. Since we will be using our localdb, we can set "data source = .". We will name our database TestRailMVC in line with our application, and set our integrated security to Security Support Provider Interface (SSPI), which will allow us to utilise Windows Authentication.

## Project Setup

6.2.1 [Installing Microsoft SQL Server](https://www.microsoft.com/en-in/download/details.aspx?id=42299)

Since the project will be hosted locally during development, we need to configure SQL server accordingly. For this project, we will be installing SQL 2014 Server Express, and SQL Server Management Studio 2014 express. As previously described, the database engine will create relational databases for online transaction processing or online analytical processing data. This includes creating tables for storing data, and databases objects such as indexes, views, and stored procedures for viewing, managing, and securing data.

The database engine is the core service for storing, processing, and securing data. SQL Server Browser listens for incoming requests for Microsoft SQL Server resources and provides information about SQL Server instances installed on the computer. This is turned off in SQL server installation be default. Collation refers to a set of rules that determine how data is sorted and compared. Character data is sorted using rules that define the correct character sequence, with options for specifying case sensitivity, accent marks, kana character types, and character width. This defaults to Latin1\_General\_CI\_AS, and does not need to be changed. For authentication, SQL Server defaults to using Windows accounts as authentication for the server. We will instead change that to Mixed Mode, which allows both Windows authentication AND SQL Server authentication. The Server Administrator (sa) accounts have unrestricted access to the Database Engine, and will be selected for our purposes. To confirm the installation worked as intended, open SQL Server configuration Manager, select SQL Server Services, and ensure the instance -  SQL Server (SQLEXPRESS) - appears in the right hand pane.

We will now install the SQL Server Management Studio. We will perform a new installation of SQL Server 2014 to install a new instance of SQL server on the machine. By default, all available features are checked, which we will stick with. With both installed, we can then use SQL Management Studio to connect to SQL Server Database Engine.

6.2.2 Project Setup

We will be using Microsoft Visual Studio 2015 Community to create the project. Since no users will initially exist in the database, the project will be built using the code-first approach, connecting to a new database that will be populated over time.

When creating a new project, we will select a new ASP.NET Web Application, and name both the application and the solution name 'TestRailMVC'. Since we will be applying the Model View Controller architecture to the project, we will select the MVC ASP.NET 4.5.2 template. The MVC template will allow us to create an ASP.NET application using the Model View Controller architecture, and will allow us to enable fast, test-driven development for creating applications that use the latest standards.

The project will use individual user accounts for authentication. This will allow users to use existing credentials from another provider, such as Facebook, Google+, etc. (see Authentication, below).

6.2.3 Version Control

The project will use Git as a method of version control during development. During the project set up, we can select the checkbox to create a new git repository with the repo. This will initialise an empty repository for the project, including a .gitattributes and default .gitignore file. We will add a README.md file to explain the inner workings of the project to any other developers that join the project, and replace the default .gitignore file with one sourced from <https://www.gitignore.io/api/visualstudio>. Since this is a demonstration application, it will be signed with the MIT license, which is a short and simple permissive license with conditions only requiring preservation of copyright and license notices. Work will be tagged on release.

## Seeding the database

During development we will seed the project database with dummy data. We could accomplish this with the SQL insert script. However, entity framework allows us to automate this process by utilising code-first migrations. We can do this by enabling migrations in the project by using the Enable-Migrations command in the Package Manager Console. That will output the following data:

Package Manager Console Host Version 3.4.4.1321

PM> Enable-Migrations

Checking if the context targets an existing database...

Code First Migrations enabled for project TestRailMVC.

With migrations enabled in the project, we can now add rows of dummy data to each of our tables. The Enable-Migrations command creates a configuration.cs file in a newly-created Migrations folder (it will create the folder if one does not currently exist). Since the scope of the seed migration is restricted to a single push to the database, automatic migrations is set to false. We will then add the add 'using TestRailMVC.Models;' to our configuration file to load our models.

Returning to the Package Manager console, we will now execute two commands:

Add-MigrationInitial

Update-Database

Add-Migration Initial generates the code used to create our database, and Update-Database will populate the database with our dummy data.

PM> Add-Migration Initial

Scaffolding migration 'Initial'.

The Designer Code for this migration file includes a snapshot of your current Code First model. This snapshot is used to calculate the changes to your model when you scaffold the next migration. If you make additional changes to your model that you want to include in this migration, then you can re-scaffold it by running 'Add-Migration Initial' again.

PM> Update-Database

Specify the '-Verbose' flag to view the SQL statements being applied to the target database.

Applying explicit migrations: [201608180737172\_Initial].

Applying explicit migration: 201608180737172\_Initial.

Running Seed method.

PM>

## Project Authentication

6.4 Authentication

As we are building this project in Visual Studio 2015 (2013 update 3 is the minimum requirement), and each user of our application (Cohaesus staff members) has a Google account, we will take advantage of using Google as an OAuth 2.0 3rd-party method of authentication. Although Visual Studio also allows us to enable two-factor authentication for this approach, it will be considered out of scope for the purposes of this project. When the project is created, the Authentication method will be set to 'Individual User Accounts'. In order to utilise the third-party authentication, we need to enable SSL in our project. We can do so by opening the project properties and setting the Enable SSL Boolean option to true. Doing so will generate an SSL URL (in our case it is [https://localhost:44376](https://localhost:44376/)), which we can then add as the Project URL in the Web tab of our project properties. Finally, to enhance security we will require that all requests are completed using HTTPS. If we now build the project, IIS Express will generate a self-signed SSL certificate to prevent any SSL warnings in the browser.

To link our project to Google, we create a new TestRailMVC project in the Google Developer Console. Inside the newly-created project, we will create an OAuth client ID for our web application. We add our previously generated SSL URL to the Authorised JavaScript origins field, and we will set our Authorised Redirect URLs to the same address, but we will also append '/signin-google' to it. The former attribute designates where the authentication will take place, where the latter defines where the user will be redirected to after authenticating. This will generate a unique client id and secret, which we can uncomment (3rd party authentications are commented out by default) and add to the app.UseGoogleAuthentication() method, located in the Startup.Auth class file. When a user now authenticates with Google, they will be redirected to the Register page of the project where they can register their account. Once the account is registered, the entry is added to the AspNetUsers table of the project database.

We will also set the same properties to display in the the ExternalLoginConfirmationViewModel class inside the Models/AccountViewModels.cs file. When the user successfully authenticates with Google, they will be redirected to the external login confirmation view, where they can enter any additional details to complete their application profile.



Now that both models have been amended, we now update the respective controller. The file we are amending is Controllers\AccountController.cs.



Finally, with the controller updated, we will create fields on the register page view to allow users to add their name to their profile, and display their name as their username instead of the default email address. The file we will be updating is Views\Account\ExternalLoginConfirmation.cshtml:



With all of the files amended, we can now successfully add an Authentication Migration and update the database. Building the project will now allow users to click the Google button as a method of authentication, and once authenticated a new row will be created for them in the AspNetUsers table, containing their additional information (forename and surname).

## Data Models

Data models are constructed to outline application entities and the relationships between them. We will now create a class file for each entity within the project: User, Project, and TestCase. Each of these will be added to the Models folder as a separate class file, and combined with the project’s pre-defined classes (AccountViewModels, IdentityModels, and ManageViewModels) they collectively form the project’s Data Model.

### Identity

The identity model is composed of two classes; an ApplicationUser that houses our custom user properties and its relationship with our Project model, and an ApplicationDbContext that ties our entities to a specific database context specified in our web.config’s connection string.

Where other classes will be created from scratch, Entity Framework comes with its own predefined User class, [Identity User](https://msdn.microsoft.com/en-us/library/microsoft.aspnet.identity.entityframework.identityuser(v=vs.108).aspx). The class comes with properties traditionally associated with an application user, such as Login details, roles, claims, etc. Since our Identity Model comes with a pre-defined ApplicationUser class that inherits from IdentityUser, we can simply add any additional properties to our ApplicationUser class. This will allow us to utilise the fully integrated functionality of the IdentityUser, and then extend the class with our custom properties, i.e. forename and surname. It also means we do not need to include an ID, since one will be already be generated from the ID property within IdentityUser. Since many projects can be associated with a user, we will also add a virtual navigation list property to our Projects class file.



Our ApplicationDbContext class will inherit Entity Framework’s IdentityDbContext from the System.Data.Entity namespace. From here we can set our Project and TestCase entities with the DbSet<> type. DbSet will then map the <> property to the table in our database. For example, DbSet<Project> Projects will map any instantiations of our Project class to the Projects table in our database. For that reason, it is considered good practice to name our database tables plurally, and the class file singularly. There is no need to include our IdentityModel class file here, as the Entity Framework has already mapped this to the pre-generated AspNetUsers table in our database.



### Projects

The Project class file will follow a similar structure to User. String fields will be created for the project’s title, code, and description. A virtual navigation property will be added to access all relevant ApplicationUsers and TestCases tied to the project, respectively. Since we are not inheriting from a base class, an ID property is required. By including the System.ComponentModel.DataAnnotations namespace in our class file, we can generate an integer value that automatically increments for each row in the table. Assigning the [Key] property to ID informs the Entity Framework this is our primary key value, though it searches for properties named or ending with Id by default.



### Test Case

Our final entity will be the TestCase class. The class will define the general properties associated with a test case scenario: A unique identifier, title, priority, any pre-conditions required before testing, steps to reproduce the test case, status, and any comments related to the result. A virtual navigation property will be added to TestCase to track the project it is bound to; Since only one project can be associated with any given Test Case, a list type is not required and we can simply define the property as having the Project type.



Status and Priority are both a set of pre-determined values that will be displayed to the user via a drop-down list. We can define these values by including the System.Web.Mvc namespace in our class file and adding enums. To reference an enum, we simply declare the enum in our property type definition, e.g. public Priority Priority { get; set; }. These values were originally defined in each of the views that they appeared, and is considered poor practice. By defining them in our entity, we instead create a single point of reference for any fields that wish to access the values within our application.



### Application View Model

Since we have added the forename and surname as required fields in our ApplicationUser class in IdentityModel, we will need to update our ApplicationViewModel to accommodate the additional fields. We can do this by updating the RegisterViewModel and the ExternalLoginConfirmationView-Model (see 6.4 Authentication). RegisterViewModel will handle users that register their details in the application itself, where ExternalLoginConfirmation-ViewModel will allow users that authenticate with Google to add the custom properties to their profile.



## Controllers

All of our project’s controllers are stored within the Controllers folder, and contain ‘Controller' as a suffix in the file name. We will create custom controllers for each of our data model entities, which will be built using the ‘*MVC5 Controller with Views, using Entity Framework*’ option. This option requires us to specify a controller name for each entity; a model class file, which will assign an entity we wish to execute CRUD operations on; and a database context class to determine how the controller will talk to the respective table in the database. Building the controller will also generate views based on the specified model.

### User Base

Since each of our controllers require an instantiation of our ApplicationDbContext, and will likely also implement functionality that requires the current logged in user, we can create a custom controller that will not generate views, but will instead instantiate our db context and bind the current logged in user to our CurrentUser class (of ApplicationUser type). By making the UserBaseController public and abstract, we ensure that the class can be accessed by our other controllers, on the basis that it is inherited rather than instantiated. UserBaseController will continue to inherit from the Controller base class, but we will amend each of our other controllers to now inherit from the UserBaseController. Each controller modified in this manner will have access to our db variable, and can access the current user via CurrentUser, e.g. CurrentUser.Forename.



### Application Users

The ApplicationUsers controller will use the ApplicationUser class as our model, and will update the AspNetUsers table in our database. When a user accesses the ApplicationUser/Index URL to add another user to a project, the controller will return a list of all users from our Users database table, and will generate a ViewData value “ProjectId” from the id input parameter. We create this value to pass the project ID to a hidden field in the User/Index view, which allows us to access it in our POST method. The Index POST method will then accept two input parameters; ProjectIdentifier, which will source the hidden value passed in from the GET method, and UserIdentifier which contains the id value of the selected user. If the model state is valid, the controller will then find both parameters, add the selected user to the project, save the changes to the database, and then redirect the user back to the Project Details page. If the state is invalid it will simply redirect the user to the ApplicationUser Create view.



To prevent users from gaining access to the detail or edit pages of other users, we will restrict the access by adding a simple conditional statement of the respective ApplicationUsers GET method. If the requested id does not match the id of the current logged in user, a 404 error will be generated. Otherwise, it will display the details page of the current user.



### Home

The Home Controller is automatically generated in an MVC application, and is used to access the homepage and other views that are created on a default mvc application. Unused methods were stripped from the controller, and a redirect was added to the Project Dashboard, so that authenticated users were immediately redirected to their project dashboard after logging in. Adding the [Authorize] attribute redirects users to the login page if they are not logged in, and the project dashboard if they are.



### Projects

The [Authorize] attribute was added to our entire ProjectsController class, ensuring that only authenticated users will have access to any of the methods contained within the class.  In addition to this, a GetUserProject method was created. This method can be applied in instances where we only want users that are associated with the project to perform specific actions, e.g accessing the project details page or deleting the project. The method will either return the project (confirming the user belongs to it), null, or an exception if multiple id's are found.



Our Project/Details/id GET method displays all of the information associated with the project, including the project details (code, title, description), as well a list of all test cases and users bound to the project. We will invoke our GetUserProject method to determine whether the user belongs to the project, displaying the details page if the return is not null, and a 404 if it is (or an exception if multiple are found).



Since users added to the project are displayed via a partial view on the details page, we will use this method to pass the “ProjectId” ViewData attribute to the user list. This will allow us to retrieve the value later in our RemoveUser method, whenever a user attempts to remove another user from the project. The method, similar to our ApplicationUsers POST method, will accept two parameters: one identifying the current project (the ViewData attribute mentioned previously) and the id of the user that has been selected for removal. The method will search the Current User’s list of projects for the ProjectIdentifier, and will locate the UserIdentifier from our Users table in the database. It will then remove the user from the project, save the changes, and refresh the project details page for the user. The removed user should no longer appear in the Users partial view list.



When a user creates a project, we want to automatically add the user to the project. To accomplish this we utilise the project object input parameter and the CurrentUser functionality from our UserBase controller. The project is added as a new list item to our CurrentUser.Projects property. The project is then added to the database Projects table, the database changes are saved, and the user is redirected to the project dashboard. The newly created project should now appear on the user’s project dashboard.



### Test Case

Similar to the project controller, we will create a method that will determine whether a User is associated with a test case. The method will return the test case if the user is associated with it, by evaluating each test case that is tied to a project the user belongs to against the test case id parameter.



Similar to automatically adding a user to a new project, we want to automatically add a new test case to the project from which it was created from. Again we will pass a hidden ViewData attribute, ProjectId, which will pass the project’s id to a hidden field in the Test Case’s Create view. When the user invoked the Create POST method, the controller will identify the project from the ProjectIdentifier, find the project in the database, and add our newly created test case to the project object. The controller will then save the database changes, and return the user to the project details page the test case was created from. The test case should now show in the Test Cases list on the project details page.



Finally, when deleting a test case from a project, we want to remove the test case form the database and then redirect the user back to the project details page.



## Views

Creating a controller for each of the entity data models will in turn scaffold a number of .cshtml views for each model; Create, Delete, Details, Edit, and Index. Each of the views is returned when the GET method is invoked from the respective controller. The views provide an interface for our users to interact with our TestRail database. The application consists of three primary views: ApplicationUsers, Projects, and TestCases. When the appropriate GET methods are requested, these views are injected into our shared view, \_Layout.

### Layout

The layout view forms the general structure of each page of the application. Since this is a straightforward application, each page will adopt the same shared layout, though multiple layouts can occur. The @RenderBody() section will inject each GET method as it is called.



The sidebar’s @RenderSections are also defined here, though their required option is set to false so they will only render if the section is defined in the view that is called.



### Application User

Our ApplicationUser view generates a list of all users registered in our Users database table. The view is accessed when a user invokes the ApplicationUser/Index GET method, by clicking the ‘Add a User’ button from a project details page. The view displays the Forename, Surname, and Email column headers, and then generate a loop to display each user registered on the system. For each user in the system (which translates to each item in our ApplicationUser model), we will display the user’s name and email address. Each item will also contain two hidden fields; one which stores the user’s id as UserIdentifier, and the ProjectIdentifier, which is the ViewData ProjectID attribute that is passed from the ApplicationUser controller’s Index GET method (see 6.6.1). Each item is wrapped in a form element, and an ‘Add to Project’ submit button is used to trigger the Index POST method.



Since a list of users will be displayed on the Project Details page, we will add a partial view that will be rendered when a user visits a Project Details page. The list will display each user that is associated with the project’s username and a delete button (styled as a link). In a similar fashion to the ApplicationUser list, each user (item) will be wrapped in a form, and the delete button will trigger the RemoveUser method in our Projects controller (6.6.4). Two hidden fields also store the id of the project and the id of the user to be removed.



### Projects

The Project Dashboard is generated by the Project Index view. The view displays a list of all projects that the user is associated with via a simple foreach loop. The project title is converted from a @HTML.DisplayFor() to @HTML.ActionLink() to allow the user to click the title to progress through to the respective project detail page. Users will also be able to access the project’s Edit and Delete methods from this view.



Here we will also define content for the Shared layout’s SidebarProjectDashboard section; a simple button that allows users to create a new project from the sidebar.



The Project Details view contains two partial views and sidebar sections. The first partial view renders a list of all test cases associated with the project. The first sidebar section adds two buttons to the sidebar; one to allow users to create new tests cases, and the other to add users to the project.



The second partial view is rendered inside the second sidebar section. SidebarProjectUsers lists all users that belong to the project, and also provides functionality for removing them.



### Test Cases

The test case views remain for the most part unchanged from those built by the Test Cases controller. The only addition is the \_ProjectTestCases partial view, which is called from the Project Dashboard to display a list of all test cases tied to the project. The foreach loop has been stripped to only display critical information in the list view: Name, Priority, Status, and links to allow the user to edit or delete the test case. Similar to the Project Dashboard project list, an ActionLink is used on the test case title to redirect users to the respective test case details page.



# Conclusions

*A short, logical summing up of the theme(s) developed in the main text*

# Further Work

Further work would see the implementation of roles to vary the access privileges available to users. All users currently possess the ability to create and delete a project or test case, and add or remove a member from the project (note that users can only perform these actions on projects which they themselves are assigned to). The introduction of roles would see the users split into three distinct categories: Admin, which would retain all previously mentioned privileges; Member, that can create and edit projects and test cases but not delete them, and can add users but not remove them; and Watchers, who would be added with complete read-only access rights to any projects they have been associated with.

To streamline the workflow, users would be able to quickly change the status of a test case on the Project Details page, instead of Test Case Details. Test case comments would also be stored within the test case (rather than updating a single comment field), along with a timestamp and a collection of the user's details that commented on the case. A tags section would allow the user base to group related tickets, and add relevant information to the test case i.e. browser information. A mobile-first overhaul of the design of the application, including support for all latest browsers. Unit tests.

# Acknowledgements

I would like to express great appreciation to Philip Beaman for his continual advice, direction and feedback throughout the project.

I would also like to extend thanks to Matt Meckes and Jamal Osman for their valuable opinions, and to Barrett Simms for his insight on the project foundations.

Finally, special thanks to my partner, Catriona, for her continual patience and support over the duration of the project.

# References

O'Neil, E.J., 2008, June. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (pp. 1351-1356). ACM.

Mike Chapel. 2016. What is Referential Integrity? [ONLINE] Available at:<http://databases.about.com/cs/administration/g/refintegrity.html> . [Accessed 26 July 2016]]

Welling, L. and Thomson, L., 2003. *PHP and MySQL Web development*. Sams Publishing.

Lerman, J. and Miller, R., 2011. *Programming entity framework: code first*. "O'Reilly Media, Inc.".

# Bibliography

*Other published sources of material, including websites, not referred to in the text but useful for background or further reading.*

# Glossary

**.NET Framework -**A programming infrastructure created by Microsoft for building, deploying, and running applications and services that use .NET technologies, such as desktop applications and Web services. The .NET framework contains three major parts: the Common Language Runtime, the Framework class library, and ASP.NET.

**ADO.NET -** A set of computer software components that programmers can use to access data and data services from the database. It is part of the base class library that is included with the Microsoft .NET Framework.

**ASP.NET** **-** An open source server-side web application framework designed for web development to produce dynamic web pages.

**CRUD -**Create, Read, Update, and Delete. The four basic functions of persistent storage.

**Entity Data Model** **-** A model that describes entities and the relationships between them.

**HTTPS -** Similar to HTTP scheme, aside from it's scheme token. HTTPS informs the browser to use an added encryption layer of SSL on requests to protect the traffic of information.

**Language Integrated Query (LINQ) -** A Microsoft programming model and methodology that essentially adds formal query capabilities into Microsoft .NET-based programming languages. LINQ offers a compact, expressive, and intelligible syntax for manipulating data.

**Model View Controller (MVC) -** Software design pattern used to promote code reusability, and implement separation of concerns.

**NuGet** - A free and open-source package manager designed for the Microsoft development platform (formerly known as NuPack).

**Object-relational mapping (OR/M) -** A programming technique for converting data between incompatible type systems in object-orientated programming languages. This creates, in effect, a "virtualobject database" that can be used from within the programming language.

**OAuth -**  An open standard for authorization, allowing sites and apps the ability to allow users to log in with third-party accounts (Such as Facebook, Google, LinkedIn, etc) without exposing their password.

**Plain Old CLR Object (POCO) -** Also known as Plain Old C# Object, and Plain Old Class Object. An object that does not derive from some special base class, nor do they return any special types for their properties.

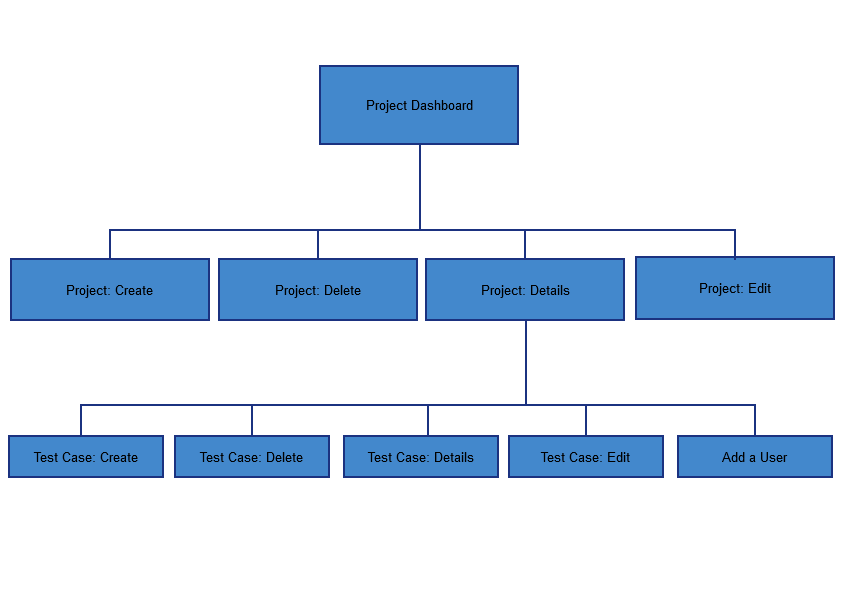
**Secure Sockets Layer (SSL) -** The industry-standard in security technology for establishing an encrypted link between a web server and a browser, ensuring that all data passed between the browser and the server remains private.

# Appendix

## User Requirements

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **As a…** | **I want to…** | **So that…** | **Acceptance Criteria** | **Category** |
| 1 | User | Register a new account | User can log into the application and access the project dashboard | User can log into the application and access the project dashboard | Account |
| 2 | Dev | Provide a 'Register a new user' link | Users appear in the ASPNetUsers database table | Users appear in the ASPNetUsers database table | Account |
| 3 | Dev | Provide third-party authentication | Users appear in the ASPNetUsers database table | Users appear in the ASPNetUsers database table | Account |
| 4 | User | Add my name to the account | Forename and surname appear in the account profile details | Forename and surname appear in the account profile details | Account |
| 5 | User | Enter my email address and password on the login screen | Username and Log out options replace Login on the main navigation | Username and Log out options replace Login on the main navigation | Authentication |
| 6 | User | Log into the app with third-party credentials | Externally via Google account | Externally via Google account | Authentication |
| 7 | Dev | Prevent users from seeing projects they are not associated with | Only users that are authenticated and added to the project can view the project and any child pages | Only users that are authenticated and added to the project can view the project and any child pages | Authentication |
| 8 | User | Be able to click on the main logo anywhere on the site | Clicking on the TestRail logo will return user to project/index | Clicking on the TestRail logo will return user to project/index | Navigation |
| 9 | User | View my account details | User can access their respective user/details page | User can access their respective user/details page | Navigation |
| 10 | Dev | Prevent users from accessing other user's details | Users attempting to access another user/details page will be met with a 404 | Users attempting to access another user/details page will be met with a 404 | Navigation |
| 11 | User | Be able to log out from anywhere on the site | Log out link appears in main navigation, reverts to Log In when the user has successfully logged out | Log out link appears in main navigation, reverts to Log In when the user has successfully logged out | Navigation |
| 12 | User | View a personalised project dashboard when I log in | User is redirect to projects/index after successfully authenticating | User is redirect to projects/index after successfully authenticating | Dashboard |
| 13 | User | Be able to create a new project from the project dashboard | Project is added to the Projects database table | Project is added to the Projects database table | Dashboard |
| 14 | User | Be automatically added to a newly-created project | Project appears in the user's project dashboard after creation. | Project appears in the user's project dashboard after creation. | Dashboard |
| 15 | User | Be redirected to my project dashboard after creating a project | User is redirected to projects/index from project/create | User is redirected to projects/index from project/create | Dashboard |
| 16 | User | Be able to access a project's details from the project dashboard | Edit the project details | Edit the project details | Dashboard |
| 17 | User | Delete a project and all associated test cases from the dashboard | Delete the project | Delete the project | Dashboard |
| 18 | User | Be able to click on a project title | Clicking on a project title redirects the user to project/details/id | Clicking on a project title redirects the user to project/details/id | Project |
| 19 | User | Be able to see the project code and title on the project details page | Project code and title appear in the sub-header of the project details page | Project code and title appear in the sub-header of the project details page | Project |
| 20 | User | Be able to see a description of the project on the project details page | Description appears in a jumbotron below the sub-header | Description appears in a jumbotron below the sub-header | Project |
| 21 | User | See all users that are added to the project | Users added to the project display in a list on the sidebar | Users added to the project display in a list on the sidebar | Users |
| 22 | User | Add users to the project | User clicks 'Add a User' button, redirected to users/index, selects a user from the list of application users, redirected back to project/details/id, new user appears in the Users sidebar list | User clicks 'Add a User' button, redirected to users/index, selects a user from the list of application users, redirected back to project/details/id, new user appears in the Users sidebar list | Users |
| 23 | User | Remove users from the project | User disappears from the Users sidebar list | User disappears from the Users sidebar list | Users |
| 24 | User | Be able to see all of the test cases that have been created for the project | All test cases with the project's id as a foreign key will be displayed in a list below the jumbotron | All test cases with the project's id as a foreign key will be displayed in a list below the jumbotron | Test Case |
| 25 | User | Create a new test case for the project | User clicks 'Add a Test Case' button, redirected to testcase/create, redirected back to project/details/id on POST, test case appears in test cases list | User clicks 'Add a Test Case' button, redirected to testcase/create, redirected back to project/details/id on POST, test case appears in test cases list | Test Case |
| 26 | User | Be able to edit the test case | User clicks on the test case Edit link, redirected to testcase/edit/id, redirected back to project/details/id on POST, updated information displays in testcase/details/id | User clicks on the test case Edit link, redirected to testcase/edit/id, redirected back to project/details/id on POST, updated information displays in testcase/details/id | Test Case |
| 27 | User | Be able to delete the test case | User clicks the test case Delete link, redirected to testcase/delete/id, redirected back to project/details/id on POST, test case no longer appears in the test case list | User clicks the test case Delete link, redirected to testcase/delete/id, redirected back to project/details/id on POST, test case no longer appears in the test case list | Test Case |
| 28 | User | Click on the test case title | User is redirected to testcase/details/id | User is redirected to testcase/details/id | Test Case |
| 29 | User | Return to the project details page | Each child page of Project/Details/id contains a 'Return to Project' link at the bottom of the page | Each child page of Project/Details/id contains a 'Return to Project' link at the bottom of the page | Test Case |

## Sitemap



## Wireframes

