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

# Abstract

*A summary of the whole report including important features, results and conclusions*

# Contents

[Introduction 1](#_Toc461894124)

[Literature Review 2](#_Toc461894125)

[SQL 2](#_Toc461894126)

[5.1.1 Relationships 2](#_Toc461894127)

[Entity Framework 10](#_Toc461894128)

[History 11](#_Toc461894129)

[5.2.3 Workflows 12](#_Toc461894130)

[5.2.4 Code Migrations 14](#_Toc461894131)

[TestRail Application 15](#_Toc461894132)

[Database Design 15](#_Toc461894133)

[6.1.1 Application goal 15](#_Toc461894134)

[6.1.2 Database Structure 15](#_Toc461894135)

[Project Setup 17](#_Toc461894136)

[6.2.1 Installing Microsoft SQL Server 17](#_Toc461894137)

[Seeding the database 20](#_Toc461894138)

[Project Authentication 22](#_Toc461894139)

[Data Models 27](#_Toc461894140)

[Controllers 36](#_Toc461894141)

[Views 45](#_Toc461894142)

[Conclusions 48](#_Toc461894143)

[Further Work 49](#_Toc461894144)

[Acknowledgements 50](#_Toc461894145)

[References 51](#_Toc461894146)

[Bibliography 52](#_Toc461894147)

[Glossary 53](#_Toc461894148)

# Introduction

Applications traditionally depend on database services to handle their data storage. Where database services work on their data via rows and tables, applications (traditionally written in higher-level programming languages) construct, manipulate, and retrieve their data 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 will increase the level of resistance. Object/Relational Mapper frameworks attempt to reduce this resistance 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 allow them to add other users to the projects. The evaluation of the application will focus on the Entity Framework’s key aspects, architecture, and its ease of integration with any associated technologies.

# Literature Review

## SQL

Structured Query Language (SQL henceforth) is commonly referred to as a 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 to the data by providing secure authentication via a login that demands the user provide a username and password to gain access. SQL can also process data, allowing backups and data restoration to take place. It also process data through highly-efficient compiled stored procedures. It can also create automated tasks using tools that will automatically run tasks on a fixed schedule.

### 5.1.1 Relationships

SQL allows us to create connections between pieces of data. These connections are more commonly known as relationships. There are four potential relationships that can be administered in an SQL database: One-to-Many; Many-to-Many; One-To-One, and None. The most common form of relationship is usually One-to-Many. A single actor can have starred in many films, so we can quantify this by saying and one actor is related to many films. Many, in this instance, does not always have to be used to full effect. An actor could have worked on a single film, many films, or even no films. The distinction is that they can have starred in many, 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. Elaborating on the previous example, an actor can have acted in multiple films, but in this instance a film will usually have multiple actors. There are several different approaches that can be taken to model this. An additional or repeating column could be added to the film database table to accommodate a second actor (e.g. actor2), 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. The table is usually named the concatenation of the two parent tables, i.e. our actor and film tables would produce a joining table called *actorfilm*. The joining table creates two One-to-Many relationships, since you cannot directly express a Many-to-Many relationship in relational databases. *A*ctor and title would both be assigned the ‘one’ side of the relationship, and actor film would be assigned both ‘many’ counterparts.

Although a One-to-One relationship is possible, it is not very common in practice. If one row in one table is pointing to a one row and only one row in another table, you might as well combine them to a single row in both places. Finally, some people consider none to be an official relationship in relational databases. Since database tables do not have to connect to other tables to exist (or be utilised), some consider none to be an official relationship, although that is a topic for debate. For the purposes of this paper, we will treat it as a genuine relationship.

5.1.2 Keys

Relationships are usually based on keys, where a key is a unique identifier for each row in a database table. If we stored every actor in the Actor database table, there is a reasonable chance that some of them may share the same name. For that reason, it becomes difficult to target the specific actor’s row based on their name alone. You could also store and reference other columns, such as age and address. However, that quickly becomes cumbersome and would be considered poor database design. Instead, we can assign a unique identifier to the row. This is done by adding a new column, usually following the naming syntax of [name]ID. Furthering our actor/film example, an actorID column would be added to the actor table. Since the actorID column is a unique identifier for each row in the actor table, we refer to it as the primary key.

This ID can be referenced elsewhere to create relationships between other database tables. Whenever a primary key is referenced in another database table, we refer to it as a foreign key. In the relationships we described above, when we talk about *one* we are referring to a primary key. Conversely, whenever we say *many* we are referencing the foreign key.

Using keys in databases in this manner leads to referential integrity. For example, if we establish a One (Actor)-To-Many (Film) relationship with a database, and then try to add a film with an actor that does not appear in the actor table, an error will be thrown. Referential integrity is therefore a data property that ensures that any foreign key referenced in a table exists as a primary key in another (Chapel, 2016).

5.2.3 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, where the schema name is referred to as the schema owner, or the owner of the related group of objects. This allows us to logically group tables, procedures, and views together in our 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. For example, all employee-related objects could be contained within an employee schema, where anything visitor-related would be grouped under visitor. 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 the schemas they have access to.

5.2.4 Functions, Stored Procedures, and Triggers

Functions, stored procedures, and triggers are all database objects containing code that are executed as a single unit. In SQL server, each of these objects are usually written in T-SQL, but can theoretically exist in any .NET language. All three of these objects share some common features, and resemble functionality that other platforms may refer to as either a method, a sub-routine, or a module. For instance, any of the objects can target a specific database via a USE statement at the beginning of a new query. Although the USE statement is not technically required for the query to run, it is considered good practice if the object is scoped within a single database. However, as a rule the *CREATE* statement must be the first declaration in a batch of statements. We can circumvent this by adding a GO statement after the USE, which begins a new batch of statements and eliminates the highlighted problem.

USE RandomDatabase

GO

CREATE …

Similar to creating a table, each object can be generated using the *CREATE* statement. Similarly, we can amend or delete each of the objects by replacing the CREATE statement with ALTER and DROP, respectively. When naming a custom function, stored procedure or trigger, it is considered good practice to add a simple prefix to differentiate it from the inbuilt system alternatives e.g: *fn (function); try (trigger),* and sp (Stored Procedure).

Alongside their common functionality, each of the objects process their own unique traits and methods of execution. A stored procedure is simply a group of SQL statements grouped together under a single heading. Without them, you would need to either re-write your code every time you wanted to query a set of data. With the stored procedure, the collection of statements can be created and then executed within another query using the *EXECUTE* statement. Typically the point of a Stored Procedure is to change data in the underlying tables, but it is allowed to return a value if it wants to (it commonly returns a 0 or a 1 to indicate success or failure, respectively).

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 of these functions contain a parameters folder, informing the user what is required for the function to work, 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 it’s requested data-type, and an *AS* statement will then define what operations the function will perform. Although not necessarily required. a *BEGIN* and *END* block is an efficient, neat way to encapsulate the operational logic. It contains the most important instruction of a function’s logic — the one that returns the final answer — and is defined via a *RETURN* statement.

A trigger is a special type of stored procedure that you can attach to various events which happen in your database. Unlike a function, triggers never return data. For this reason, we do not need to provide a *returns* statement; a trigger always runs as a reaction to another event. Triggers can be enabled and disabled using the ENABLE / DISABLE statements, and can either target a specific trigger or all triggers in the database. It is also possible to set the order in which the triggers fire. There are three primary types of triggers: Data Manipulation Language; Data Definition Language, and Logon. DML triggers work in conjunction with the INSERT, UPDATE, and DELETE events. These events are associated and can be attached to tables or views in your database, and can either be an AFTER or INSTEAD OF trigger type.  If we follow this with an AFTER statement, the trigger will execute immediately after the event was set for. If we instead designate the trigger as INSTEAD OF, the trigger will execute instead of whatever is originally specified. DML Triggers can also be used for data validation. If the designated criteria is not met, a DML trigger can raise an error via a RAISERROR to report the problem. DDL triggers refer to anything that modifies the objects in the database, corresponding with the three main keywords in SQL server. CREATE, ALTER, and DROP. You can create DDL triggers that fires only on events within a single database, or even across an entire server*.* Logon triggers fire stored procedures in response to a LOGON event. This event is raised when a user session is established with an instance of SQL Server. Logon triggers fire after the authentication phase of logging in finishes, but before the user session is actually established.

5.1.5 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.

5.1.6 Joins

Joins in SQL server are used to retrieve data from two or more related tables. In general tables are related to each other the using foreign key constraints explained previously. SQL server has three primary types of joins: Inner, Outer, and Cross. Inner joins will only return records that exactly match the query parameters, and any non-matching records will be eliminated from the result. Outer joins are composed of three further sub-categories. A Left Outer join will return records that exactly match the parameters of the query, but will also include any non-matching records from the left table. A Right Outer join will do the same, but will return all records from the right table instead of the left. As expected, the Full Outer join will return all data, including any non-matching records from any specified table. A cross join is slightly different from the rest, in that it does not include an ON statement. It instead returns the Cartesian product of the two tables involved in the join. For example, if the Actors table had 10 rows and the Films table had 4, a Cross Join between the two tables would return 40 records.

5.1.7 Views

## Entity Framework

### Introduction

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* 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 EntityObject (Lerman and Miller, 2011). Essentially, this provides a greater degree of freedom when designing and implementing your classes, since they possess fewer requirements in order to work correctly. In addition to core features, v4.0 also saw the release of an alternate workflow, Model first, allowing developers to construct their model through the use of a designer tool. Version 4.1 also saw the introduction of the framework's third workflow, 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 EF designer.

### 5.2.3 Workflows

There are several different approaches you can take to use the Entity Framework to access a relational database from a .NET application. Entity framework supports four main development workflows, and uses two main considerations to determine the most appropriate approach.

The first consideration is usually outwith the developer’s control, as it depends on whether you will be connecting to either a pre-existing database that is already pre-populated with tables and data, or building it from scratch. It is outwith our control mainly due to the fact that it will depend on whether or not you are connecting to the client's established user base. The other consideration lies in the developers hands, and depends on their 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 for you 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.

#### 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.

### 5.2.4 Code Migrations

# 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 redirectered to th

publicclassExternalLoginConfirmationViewModel

{

     [Required]

     [Display(Name = "Email")]

     public string Email { get; set; }

     [Required]

     [Display(Name = "Forename")]

     public string Forename { get; set; }

     [Required]

     [Display(Name = "Surname")]

     public string Surname { get; set; }

}

Now that both models have been amended, we now update the respective controller. The file we are amending is Controllers\AccountController.cs. Since we want the user's username to display their forename and surname instead of the default email address, we will amend the UserName property in our user object in addition to adding the newly-created variables.

publicasyncTask<ActionResult> ExternalLoginConfirmation(ExternalLoginConfirmationViewModel model, string returnUrl)

{

     [...]

     var user = new ApplicationUser

     {

          UserName = model.Email,

          Email = model.Email,

          Forename = model.Forename,

          Surname = model.Surname

     };

     [...]

}

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:

@using (Html.BeginForm("ExternalLoginConfirmation", "Account", new { ReturnUrl = ViewBag.ReturnUrl }, FormMethod.Post, new { @class = "form-horizontal", role = "form" }))

{

    [...]

  <divclass="form-group">

     @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })

     <divclass="col-md-10">

          @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })

          @Html.ValidationMessageFor(m => m.Email, "", new { @class = "text-danger" })

     </div>

     </div>

     <divclass="form-group">

       @Html.LabelFor(m => m.Forename, new { @class = "col-md-2 control-label" })

       <divclass="col-md-10">

          @Html.TextBoxFor(m => m.Forename, new { @class = "form-control" })

          @Html.ValidationMessageFor(m => m.Forename)

       </div>

   </div>

   <divclass="form-group">

        @Html.LabelFor(m => m.Surname, new { @class = "col-md-2 control-label" })

        <divclass="col-md-10">

           @Html.TextBoxFor(m => m.Surname, new { @class = "form-control" })

           @Html.ValidationMessageFor(m => m.Surname)

        </div>

   </div>

   [...]

}

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.

**6.5.1 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.

public class ApplicationUser : IdentityUser  
{  
  [Display(Name = "Forename")]  
  [DataType(DataType.Text)]  
  [Required]  
  public string Forename { get; set; }  
  
  [Display(Name = "Surname")]  
  [DataType(DataType.Text)]  
  [Required]  
  public string Surname { get; set; }  
  
  // Many to Many relationship with Projects  
  public virtual List<Project> Projects { get; set; }  
  
  public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)  
  {  
      // Note the authenticationType must match the one defined in CookieAuthenticationOptions.AuthenticationType

      var userIdentity = await manager.CreateIdentityAsync(this, DefaultAuthenticationTypes.ApplicationCookie);

      // Add custom user claims here  
      return userIdentity;  
  }

}

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.

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>

{

  public DbSet<Project> Projects { get; set; }  
  public DbSet<TestCase> TestCases { get; set;}  
  
  public ApplicationDbContext()  
      : base("ApplicationDbContext", throwIfV1Schema: false)  
  {  
  }  
  
  public static ApplicationDbContext Create()  
  {  
      return new ApplicationDbContext();  
  }  
}

**6.5.2 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.

  public class Project

  {  
    [Key]  
    [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]  
    public int Id { get; set; }  
  
    [Display(Name = "Name")]  
    [DataType(DataType.Text)]  
    [Required]  
    public string Name { get; set; }  
  
    [Display(Name = "Project Code")]  
    [DataType(DataType.Text)]  
    [Required]  
    public string Code { get; set; }     
  
    [Display(Name = "Description")]  
    [DataType(DataType.MultilineText)]  
    public string Description { get; set; }  
  
    // Many to Many relationship with Users  
    public virtual List<ApplicationUser> Users { get; set; }  
  
    // One to Many relationship with TestCases  
    public virtual List<TestCase> TestCases { get; set; }

  }

**6.5.3 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.

  public class TestCase  
  {  
    [Key]  
    [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]  
    [Required]  
    public int Id { get; set; }  
  
    [Display(Name = "Title")]  
    [DataType(DataType.Text)]  
    [Required]  
    public string Title { get; set; }  
  
    [Display(Name = "Priority")]  
    public Priority Priority { get; set; }  
  
    [Display(Name = "Preconditions")]  
    [DataType(DataType.MultilineText)]  
    public string Precondition { get; set; }  
  
    [Display(Name = "Steps")]  
    [DataType(DataType.MultilineText)]  
    public string Step { get; set; }  
  
    [Display(Name = "Status")]  
    public Status Status { get; set; }  
  
    [Display(Name = "Comment")]  
    [DataType(DataType.MultilineText)]  
    public string Comment { get; set; }  
  
    // One to many relationship with Project  
    public virtual Project Project { get; set; }  
  }  
   
  public enum Status  
  {   
    Pass = 1,   
    Fail = 2,   
    Blocked = 3,  
    Invalid = 4  
  }  
  
  public enum Priority  
  {  
    High = 1,  
    Normal = 2,  
    Low = 3

  }

**6.5.4 ApplicationViewModel**

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 RegisterViewModel and ExternalLoginConfirmationViewModel; RegisterViewModel will handle users that register their details in the application itself, where ExternalLoginConfirmationViewModel will allow users that authenticate with Google to add the custom properties to their profile.

public class RegisterViewModel  
{  
  [Required]  
  [EmailAddress]  
  [Display(Name = "Email")]  
  public string Email { get; set; }  
  
  [Required]  
  [DataType(DataType.Text)]  
  [Display(Name = "Forename")]  
  public string Forename { get; set; }  
  
  [Required]  
  [DataType(DataType.Text)]  
  [Display(Name = "Surname")]  
  public string Surname { get; set; }  
  
  [Required]  
  [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]  
  [DataType(DataType.Password)]  
  [Display(Name = "Password")]  
  public string Password { get; set; }  
  
  [DataType(DataType.Password)]  
  [Display(Name = "Confirm password")]  
  [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]  
  public string ConfirmPassword { get; set; }  
}  
  
public class ExternalLoginConfirmationViewModel  
{  
  [Required]  
  [Display(Name = "Email")]  
  public string Email { get; set; }  
  
  [Required]  
  [Display(Name = "Forename")]  
  public string Forename { get; set; }  
  
  [Required]  
  [Display(Name = "Surname")]  
  public string Surname { get; set; }

}

## Controllers

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.

**6.5.1 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.

public class ApplicationUser : IdentityUser  
{  
  [Display(Name = "Forename")]  
  [DataType(DataType.Text)]  
  [Required]  
  public string Forename { get; set; }  
  
  [Display(Name = "Surname")]  
  [DataType(DataType.Text)]  
  [Required]  
  public string Surname { get; set; }  
  
  // Many to Many relationship with Projects  
  public virtual List<Project> Projects { get; set; }  
  
  public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)  
  {  
      // Note the authenticationType must match the one defined in CookieAuthenticationOptions.AuthenticationType

      var userIdentity = await manager.CreateIdentityAsync(this, DefaultAuthenticationTypes.ApplicationCookie);

      // Add custom user claims here  
      return userIdentity;  
  }

}

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.

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>

{

  public DbSet<Project> Projects { get; set; }  
  public DbSet<TestCase> TestCases { get; set;}  
  
  public ApplicationDbContext()  
      : base("ApplicationDbContext", throwIfV1Schema: false)  
  {  
  }  
  
  public static ApplicationDbContext Create()  
  {  
      return new ApplicationDbContext();  
  }  
}

**6.5.2 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.

  public class Project

  {  
    [Key]  
    [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]  
    public int Id { get; set; }  
  
    [Display(Name = "Name")]  
    [DataType(DataType.Text)]  
    [Required]  
    public string Name { get; set; }  
  
    [Display(Name = "Project Code")]  
    [DataType(DataType.Text)]  
    [Required]  
    public string Code { get; set; }     
  
    [Display(Name = "Description")]  
    [DataType(DataType.MultilineText)]  
    public string Description { get; set; }  
  
    // Many to Many relationship with Users  
    public virtual List<ApplicationUser> Users { get; set; }  
  
    // One to Many relationship with TestCases  
    public virtual List<TestCase> TestCases { get; set; }

  }

**6.5.3 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.

  public class TestCase  
  {  
    [Key]  
    [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]  
    [Required]  
    public int Id { get; set; }  
  
    [Display(Name = "Title")]  
    [DataType(DataType.Text)]  
    [Required]  
    public string Title { get; set; }  
  
    [Display(Name = "Priority")]  
    public Priority Priority { get; set; }  
  
    [Display(Name = "Preconditions")]  
    [DataType(DataType.MultilineText)]  
    public string Precondition { get; set; }  
  
    [Display(Name = "Steps")]  
    [DataType(DataType.MultilineText)]  
    public string Step { get; set; }  
  
    [Display(Name = "Status")]  
    public Status Status { get; set; }  
  
    [Display(Name = "Comment")]  
    [DataType(DataType.MultilineText)]  
    public string Comment { get; set; }  
  
    // One to many relationship with Project  
    public virtual Project Project { get; set; }  
  }  
   
  public enum Status  
  {   
    Pass = 1,   
    Fail = 2,   
    Blocked = 3,  
    Invalid = 4  
  }  
  
  public enum Priority  
  {  
    High = 1,  
    Normal = 2,  
    Low = 3

  }

**6.5.4 ApplicationViewModel**

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 RegisterViewModel and ExternalLoginConfirmationViewModel; RegisterViewModel will handle users that register their details in the application itself, where ExternalLoginConfirmationViewModel will allow users that authenticate with Google to add the custom properties to their profile.

public class RegisterViewModel  
{  
  [Required]  
  [EmailAddress]  
  [Display(Name = "Email")]  
  public string Email { get; set; }  
  
  [Required]  
  [DataType(DataType.Text)]  
  [Display(Name = "Forename")]  
  public string Forename { get; set; }  
  
  [Required]  
  [DataType(DataType.Text)]  
  [Display(Name = "Surname")]  
  public string Surname { get; set; }  
  
  [Required]  
  [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]  
  [DataType(DataType.Password)]  
  [Display(Name = "Password")]  
  public string Password { get; set; }  
  
  [DataType(DataType.Password)]  
  [Display(Name = "Confirm password")]  
  [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]  
  public string ConfirmPassword { get; set; }  
}  
  
public class ExternalLoginConfirmationViewModel  
{  
  [Required]  
  [Display(Name = "Email")]  
  public string Email { get; set; }  
  
  [Required]  
  [Display(Name = "Forename")]  
  public string Forename { get; set; }  
  
  [Required]  
  [Display(Name = "Surname")]  
  public string Surname { get; set; }

}

## 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 from the respective controller. The views provide an interface for our users to interact with our TestRail database. For the purpose of this report, the application contains three primary views: ApplicationUsers, Projects, and TestCases.

**6.7.1 ApplicationUser View**

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.

@foreach (var item in Model)

     {

          using (Html.BeginForm())

          {

          @Html.AntiForgeryToken()

          <tr>

               <td>

                    @Html.DisplayFor(modelItem => item.Forename)

               </td>

               <td>

                    @Html.DisplayFor(modelItem => item.Surname)

               </td>

               <td>

                    @Html.DisplayFor(modelItem => item.Email)

               </td>

               <td>

                    <input type="submit" value="Add to Project" class="btn btn-default" />

               </td>

               <td>

                    @Html.Hidden("UserIdentifier", [item.Id](http://item.Id))

               </td>

               <td>

                    @Html.Hidden("ProjectIdentifier", ViewData["ProjectId"])

               </td>

          </tr>

     }

}

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.

# 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 (EDM)** **-** 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.