

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

Introduction	1
Technology Review	2
SQL	2
Relationships.....	2
Keys	3
Functions.....	4
Stored Procedures	4
Schema.....	4
Indexes	5
Entity Framework.....	5
History	7
Workflows	8
Migrations.....	10
TestRail Application	11
Database Design	11
Application goal.....	11
Database Structure	11
Relationships.....	12
Project Setup.....	13
Database Connection	13
Database Seed	14
Project Authentication	15
Version Control.....	16
Data Models	17
Identity	17
Project	19
Test Case.....	20
Account View	21
Controllers	23
User Base	23

Application Users	24
Home.....	26
Projects.....	27
Test Cases	30
Views.....	32
Layout	32
Application User	33
Projects.....	35
Test Cases	39
Conclusions	40
Further Work.....	41
Acknowledgements	42
References	43
Bibliography.....	44
Online Tutorials.....	44
YouTube Tutorials	44
Glossary	45
Appendix	i
Wireframes	i
User Requirements	ii
Sitemap.....	iv

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 through alternate means, such as classes and methods. This can (and usually) results in a layer of resistance between the two, resonating especially with developers unfamiliar with ADO.NET and/or an SQL environment. Bloated queries will hamper performance, and any structural changes to the database or application architecture subsequently increases 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 aim of this project is to evaluate the integration of the Entity Framework 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 each 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; Structured Query Language, 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 data sublanguage that is used to communicate with relational databases within Relational Database Management Systems (Melton and Simon, 1993). 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. SQL employs aspects of both Data Definition and Data Manipulation Languages (DDL and DML, respectively). It utilises the Data Definition syntax for creating and destroying database objects, and Data Manipulation for the retrieval and manipulation of the subsequent data (Owens, 2006).

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 user column. However, both of these methods are seen as bad practice and leads to poor database design. The recommended approach is to remove the user 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

Relational database relationships are based on keys. A key is a unique identifier for each row in a database table (Elias, 1985), and is often referred to as the primary key. McGraw-Hill (2005) elaborates that the primary key is also recognised as a *super-key*, where no two distinct rows in the database table may share the same attribute value. This key can be referenced elsewhere to establish relationships between other database tables. A primary key that migrates to another database table is referred to as a foreign key. In reference to the one-to-many relationship described previously, *one* refers to a primary key. Conversely, *many* references the foreign key. Using database keys in this manner leads to *referential integrity* (Chapel, 2016), ensuring 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 also 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 the 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 as 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. It also allows them to structure their code based on their respective business model, rather than being confined to the structure of the database itself. Microsoft (2016) 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.”

The Entity Framework adopts SQL to create an ADO.NET adaptation, Entity SQL. Entity SQL is a Data Manipulation Language that allows for the manipulation of data within the Entity Data Model via LINQ queries (Blakeley et al., 2007). 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 formulate the Entity Data Model, and free the user from writing code that directly accesses the database data via ADO.NET. Blakely et al. (2007) also stipulates that by raising the level of abstraction between the database system and a .NET application from a relational to an entity level, Entity SQL (and the framework as a whole) aims to eliminate the impedance mismatch that can occur between the two.

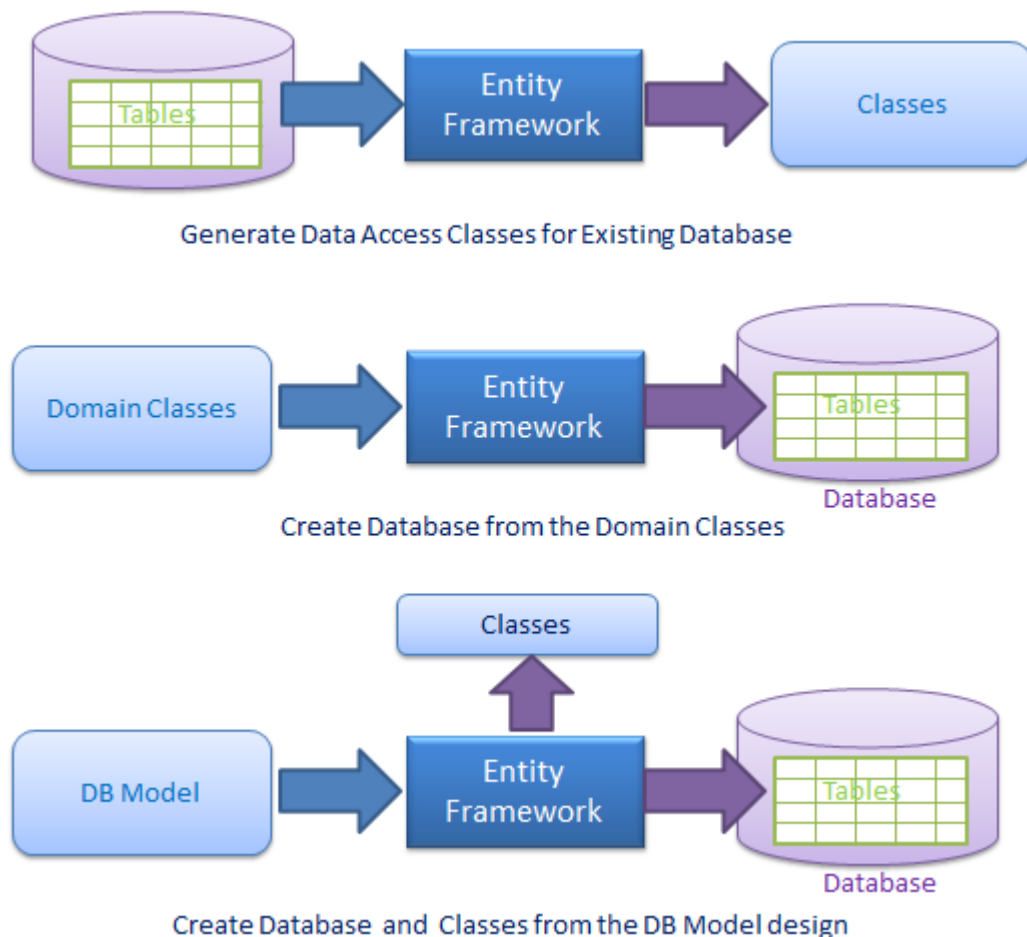


Fig. 1.2.1: Entity Framework Overview (Source: Entity Framework Tutorial. 2016. What is Entity Framework? <http://www.entityframeworktutorial.net/what-is-entityframework.aspx>)

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 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 Entity Framework designer.

Entity Framework v4.0 introduced key features such as lazy loading, where related object data is not loaded until it is specifically requested (usually via the *virtual* property), and Plain Old CLR 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). 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 released the framework's third workflow, Code First, which was further utilised in v4.3 when Code First Migrations were released. The EF v4.x framework also included its independent release as a package on NuGet, and adopted the <http://semver.org> standard for semantic versioning.

Entity Framework v5.0's defining feature was that it was released as open-source for Visual Studio 2010 onwards. Developers were now able 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. Entity Framework 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, and will be the version that is integrated within the project application. 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, and the runtime is installed via NuGet to allow shipping out-of-band between releases of Visual Studio. Although no new tools were added from the previous version, 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 it is instead mapped to an existing database.

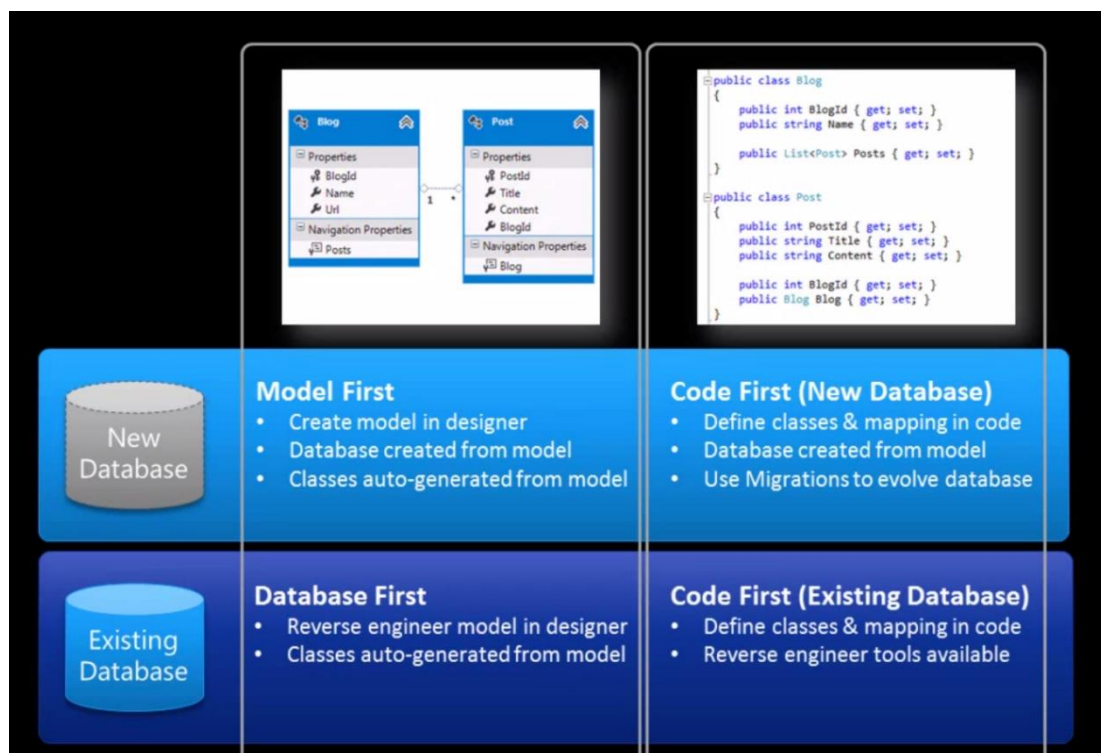


Fig. 1.2.2: Entity Framework Workflows (Source: Microsoft. 2016. MSDN Entity Framework Development Workflows. <https://msdn.microsoft.com/en-gb/data/jj590134>)

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. This mismatch is rectified 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 allows for additional methods to be created, e.g. seeding the database with dummy data, or 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

Database Design

Application goal

The application will provide a platform to allow a team of developers to generate test scripts for web development projects. Users 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."

Database Structure

Since no application data previously exists, a new database is generated to store the information. With the application goal in mind, the database's architecture is mapped 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.

Relationships

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

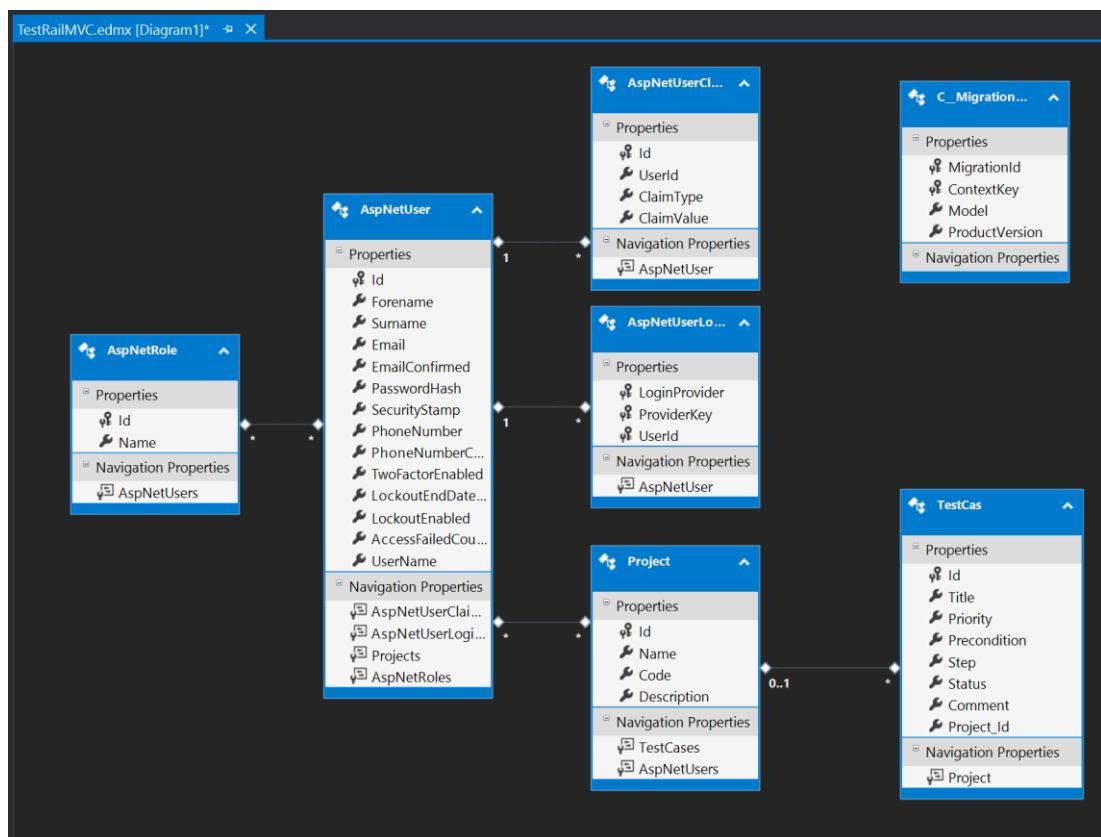


Fig. 2.1.1: TestRail Database Relationships (Source: Dick, Andrew J. 2016. GitHub.

https://github.com/AndrewJDick/TestRailMVC/blob/master/docs/Screenshots/TestRail_Designer-Relationships.PNG)

Project Setup

Microsoft Visual Studio 2015 Community is used to construct the project. A new ASP.NET Web Application project is created, and both the application and the solution name are named 'TestRailMVC'. The Model View Controller architecture is added to the project via the MVC ASP.NET 4.5.2 template. 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.

Database Connection

The project's web.config file allows the application to connect to a specified database, by amending the DefaultConnection in the connectionStrings element. The default connection will use localdb to store the database. The connection is named 'ApplicationDbContext' by adding it to the name property. The connection string requires four additional properties: the data source; the name of the database, integrated security, and the database provider. Since localdb is used, the data source is set to '.'. The database is named 'TestRailMVC' in line with the application, and the integrated security is set to Security Support Provider Interface (SSPI), which allows the use of Windows Authentication to access the database. Since the application connects to the MySQL client, System.Data.SqlClient is defined as the Provider Name.

```
<connectionStrings>

  <add name="ApplicationDbContext" connectionString="data source=.; database =
  TestRailMVC; integrated security = SSPI;" providerName="System.Data.SqlClient" />

</connectionStrings>
```

Fig. 2.2.1: Database Connection String (Source: Dick, Andrew J. 2016. GitHub.
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Web.config>

Database Seed

The project database will be seeded with dummy data during development. Though this could be accomplished with the SQL insert script, the Entity Framework automates this process through code-first migrations. This is achieved by enabling migrations in the project by using the *Enable-Migrations* command in the Package Manager Console. The *Enable-Migrations* command creates a *Configuration.cs* file in a newly-created *Migrations* folder. The *TestRailMVC.Models* namespace is added to the configuration file to allow it access to the project data models.

Two commands are then executed in the Package Manager console: *Add-Migration Seed*, and *Update-Database*. *Add-Migration Seed* generates the code used to create the database, where *Update-Database* will push the changes and populate the database with the dummy data. The migration file includes a snapshot of the current Code First model. This snapshot is used to calculate the changes to the model when the next migration is scaffolded. Before updating the database, any additional changes to the model can be included in the migration by running the *Add-Migration Seed* command again (where *Seed* is the name of the migration).

Populating the database is accomplished in three stages within a *Seed* method: Generating dummy data, assigning relationships, and adding the data to the respective tables in the database. Dummy data is generated by creating new instances of a class defined in the data models. Relationships are then established by adding new list items to respective class properties. Finally, the dummy data is then pushed to our database context.

```

protected override void Seed(TestRailMVC.Models.ApplicationDbContext context)
{
    // Dummy data
    var u1 = new ApplicationUser()
    {
        Id = "1",
        Forename = "Andrew",
        Surname = "Dick",
        Email = "andrew.dick@cohaesus.co.uk",
        UserName = "andrew.dick@cohaesus.co.uk"
    };

    var p1 = new Project()
    {
        Id = 1,
        Name = "Ultimate Banner Builder",
        Code = "285MM134",
        Description = "Provides a boilerplate for generating HTML banners."
    };

    // Allocations
    u3.Projects = new List<Project>() { p1 };

    // Seed the database
    context.Users.AddOrUpdate(u1);
    context.Projects.AddOrUpdate(p1);
}

```

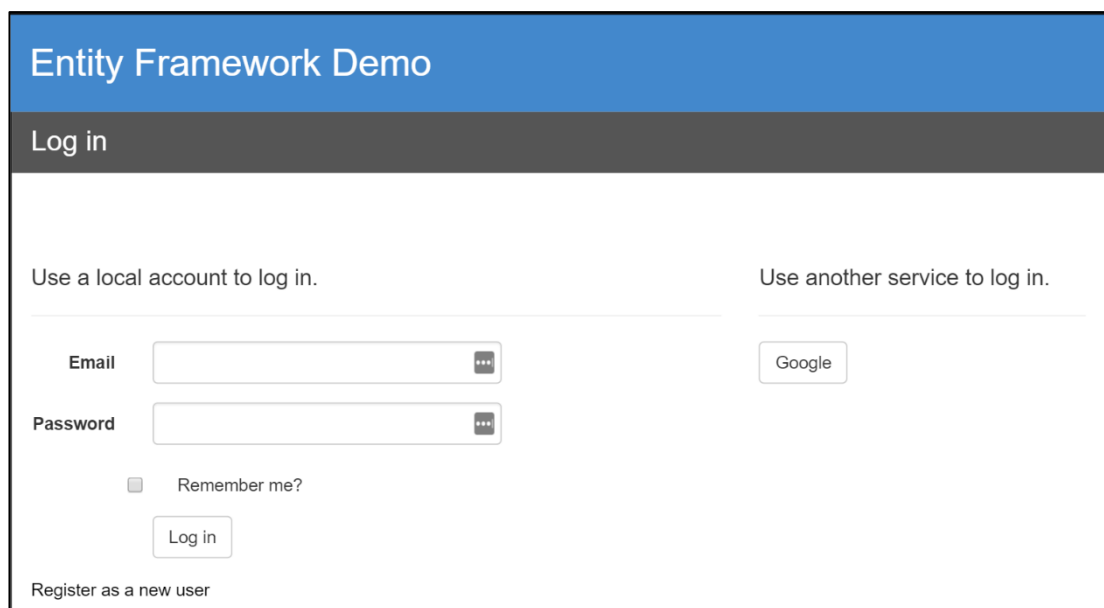
Fig. 2.2.2: Database Seed (Source: Dick, Andrew J. 2016. GitHub.
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Migrations/Configuration.cs>)

Project Authentication

In addition to the default login functionality provided by the application, the project will also implement an OAuth 2.0 3rd-party authentication method. In order to utilise third-party authentications, SSL needs to be enabled in the project. This is accomplished by opening the project properties and setting the Enable SSL Boolean option to true. Doing so will generate an SSL URL, which is then added as the Project URL in the Web tab of the project properties. All requests are completed using HTTPS. To prevent any SSL warnings in the browser, IIS Express generates a self-signed SSL certificate when the project is built.

To link the project to Google, a new TestRailMVC project is created in the Google Developer Console. Inside the newly-created project, an OAuth client ID is created for our web application. The SSL URL is added to the Authorised JavaScript origins field. The Authorised Redirect URLs is set to the same address, but 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 authentication. This generates a unique client id and secret, which is added to the app.UseGoogleAuthentication() method in the Startup.Auth project class file. When a user now authenticates with Google, they will be redirected to the project Register page where they can register their account. Once registered, the entry is added to the AspNetUsers table of the project 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.



Entity Framework Demo

Log in

Use a local account to log in.

Use another service to log in.

Email

Password

☐ Remember me?

Log in

Register as a new user

Fig. 2.2.3: TestRail Login Screen (Source: Dick, Andrew J. 2016. GitHub.

https://github.com/AndrewJDick/TestRailMVC/blob/master/docs/Screenshots/TestRail_ApplicationUser-Login.png)

Version Control

The project will use Git as version control during development. A new git repository is initialised during setup. The .gitignore file is replaced with a custom file tailored for Visual Studio projects, sourced from <https://www.gitignore.io/api/visualstudio>. The application is then signed with the MIT license.

Data Models

Data models are designed to outline application entities and the relationships between them. A class file is created for each entity within the project: User, Project, and TestCase. Each of these are added to the Models folder, and combined with the project's pre-defined models – AccountView, Identity, and ManageView - they collectively form the project's Data Model.

Identity

The identity model is composed of two classes; an ApplicationUser that defines all user properties, and an ApplicationDbContext that binds the entities to a specific database context specified in the web.config's connection string.

Where other classes are created specifically for the application, Entity Framework ships with a pre-defined user class, IdentityUser. The class contains properties that are traditionally tied to an application user: login credentials; roles; claims, etc. Since the ApplicationUser class inherits from IdentityUser, any additional user properties can be defined with the ApplicationUser class. As the ID is inherited from IdentityUser, there is no need to define an ID property in ApplicationUser. Many projects can be associated with a user, so a virtual navigation list property is also added to the 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; }
}

```

Fig. 2.3.1: Application User (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Models/IdentityModels.cs>)

ApplicationDbContext inherits Entity Framework's IdentityDbContext from the System.Data.Entity namespace. The Project and TestCase entity declarations are set with the DbSet<> type, which maps the property to the database table. For example, DbSet<Project> Projects will map any instantiations of our Project class to the Projects table in the database. The ApplicationUser entity is automatically mapped to the AspNetUsers table in the 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();
    }
}

```

Fig. 2.3.2: Database Context (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Models/IdentityModels.cs>)

Project

The Project class file contains fields for the each project's title, code, description, and a virtual navigation property to define the relationships with ApplicationUsers and TestCases. Since Project does not inherit from a base class, an ID property is required. By including the System.ComponentModel.DataAnnotations namespace, an incrementing integer value is generated automatically for each table row's unique identifier. 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; }
}
```

Fig. 2.3.3: Project Model (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Models/Project.cs>)

Test Case

The TestCase 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 is added to TestCase to track the project it is bound to. Only one project can be associated with any given Test Case, a list type is therefore not required and is instead defined with the Project type.

```
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; }
}
```

Fig. 2.3.4: Test Case Model (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Models/TestCase.cs>)

The Status and Priority fields both contain pre-determined values that are displayed to the user using a drop-down menu. This is accomplished with enums, and requires the

System.Web.Mvc namespace to be included in the class file. To reference an enum, we simply declare the enum in our property type definition. These values were originally defined in each view that they appeared, although this 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 enum Status
{
    Pass = 1,
    Fail = 2,
    Blocked = 3,
    Invalid = 4
}

public enum Priority
{
    High = 1,
    Normal = 2,
    Low = 3,
}
```

Fig. 2.3.5: Test Case Enums (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Models/TestCase.cs>)

Account View

Since Forename and Surname are required fields in the ApplicationUser class, the Account View Model is updated to accommodate the additional fields. This is accomplished by updating the Register and ExternalLoginConfirmationViewModel; RegisterViewModel will handle users that register their details within the application, where ExternalLoginConfirmationViewModel allows users that authenticate with Google to add the custom properties to their account.


```

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; }
}

```

Fig. 2.3.6: Account Model: Register (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Models/AccountViewModels.cs>)

```

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; }
}

```

Fig. 2.3.7: Account Model: External Login (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Models/AccountViewModels.cs>)

Controllers

Custom controllers are created for each data model entity using the ‘*MVC5 Controller with Views, using Entity Framework*’ template. Each controller requires a name, a model class file that assigns an entity to execute CRUD operations, and a database context class to determine how the controller will communicate with the respective table in the database. All project controllers are stored within the Controllers folder, and use ‘Controller’ as a suffix in the file name. Building the controller also generates views based on the specified model.

User Base

Each controller requires an instantiation of the ApplicationDbContext, and will likely implement functionality that requires the current logged in user. A custom controller is created to instantiate the db context and bind the current logged in user to the CurrentUser class. By making the UserBase Controller public and abstract, the class can be accessed by other controllers, on the condition that it is inherited rather than instantiated. UserBase Controller continues to inherit from the Controller base class, where each of the other controllers now inherit from the UserBase Controller.

```
public abstract class UserBaseController : Controller
{
    public ApplicationDbContext db = new ApplicationDbContext();

    public ApplicationUser CurrentUser
    {
        get
        {
            return db.Users.Find(User.Identity.GetUserId());
        }
    }
}
```

Fig. 2.4.1: User Base Controller (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/UserBaseController.cs>)

Application Users

The `ApplicationUsers` controller uses the `ApplicationUser` class as its model, and updates the database's `AspNetUsers` table. When a user accesses the `ApplicationUser/Index` view (e.g. to add another user to a project), the controller returns a list of all users from the `Users` database table, and generates a `ViewData` value “`ProjectId`” from the `id` input parameter. This value passes the ID of the project that the user was on before accessing the `Index` view, to a hidden field in the `ApplicationUser/Index` view, which allows the respective `POST` method to retrieve the value. The `POST` method accepts two input parameters; `ProjectIdentifier`, which sources the hidden value passed from the `GET` method, and `UserIdentifier`, that contains the `id` value of the selected user. Assuming a valid model state, the controller locates both parameters, adds the selected user to the project, saves the changes to the database, then returns the user to the `Project Details` page. If the state were invalid, it would simply redirect the user to the `ApplicationUser Create` view.

```
public class ApplicationUsersController : UserBaseController
{
    // GET: ApplicationUsers
    public ActionResult Index(int id)
    {
        int projectId = id;

        // Passes the Project Id to a hidden field in the Add a User (Index) view
        ViewData["ProjectId"] = projectId;

        return View(db.Users.ToList());
    }
}
```

Fig. 2.4.2: User List (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/ApplicationUsersController.cs>)

```

// POST: ApplicationUsers
[HttpPost]
public ActionResult Index(string UserIdentifier, int ProjectIdentifier)
{
    if (ModelState.IsValid)
    {
        // Find the first project in the database's Projects table whose primary
        // key matches the project identifier
        var project = db.Projects.First((p) => p.Id == ProjectIdentifier);

        // Locate the user identifier in the database and add it to the project
        project.Users.Add(db.Users.First((u) => u.Id == UserIdentifier));

        db.SaveChanges();

        // Redirect to Project Details page
        return RedirectToAction("Details", "Projects", new { id =
        ProjectIdentifier });
    }

    return View("Create");
}

```

Fig. 2.4.3: User List: Add User (Source: Dick, Andrew J. 2016. GitHub.

<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/ApplicationUsersController.cs>)

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

```

// GET: ApplicationUsers/Details/5
public ActionResult Details(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    if (CurrentUser.Id != id)
    {
        return HttpNotFound();
    }

    return View(CurrentUser);
}

```

```
// GET: ApplicationUsers/Edit/5
public ActionResult Edit(string id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    // Logged in user can only view their own personal details.
    if (CurrentUser.Id != id)
    {
        return HttpNotFound();
    }

    return View(CurrentUser);
}
```

Fig. 2.4.4: Restricted User Access (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/ApplicationUsersController.cs>)

Home

An MVC application automatically generates a Home Controller, and is used to access the application homepage. Since various methods redirect the user to the homepage after authentication, a simple redirect is added to the Index method. Combined with adding the [Authorize] attribute to the controller, the controller will therefore redirects users to the login page if they are not logged in, and the project dashboard if they are.

```
[RequireHttps]
[Authorize]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return RedirectToAction("Index", "Projects");
    }
}
```

Fig. 2.4.5: Homepage (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/HomeController.cs>)

Projects

The [Authorize] attribute is also added to the entire Projects controller, ensuring only authenticated users have access to any encapsulated methods. A GetUserProject method is also created. This method can be applied in instances where only users associated with the project can perform specific actions, e.g. accessing a project details page or deleting a project. The method will either return the project (confirming the user belongs to it), null, or an exception if multiple ids are returned.

```
public class AuthenticatedBaseController : ControllerBase
{
    protected UserManager<ApplicationUser> UserManager { get; set; }
}

[Authorize]
public class ProjectsController : AuthenticatedBaseController
{
    public Project GetUserProject(int? id)
    {
        // Either returns the project (confirming the user belongs to the it), null,
        // or an exception if multiple id's are found.
        return CurrentUser.Projects.SingleOrDefault(p => p.Id == id);
    }
}
```

Fig. 2.4.6: GetUserProject Method (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/ProjectsController.cs>)

The Project Index (i.e. the Project Dashboard) GET method simply returns a list of all projects the user is currently associated with, by utilising the CurrentUser class from the UserBase controller. Since the id parameter can be null in this instance, no exception will be thrown if a user does not belong to any projects.

```
// GET: Projects
public ActionResult Index(int? id)
{
    // Return all Projects that the User has been added to
    return View(CurrentUser.Projects);
}
```

Fig. 2.4.7: Project Dashboard (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/ProjectsController.cs>)

The Project/Details/id GET method returns all information associated with a project, including the project details (code, title, description), and a list of all test cases and users bound to the project. The GetUserProject method is invoked to determine whether the user belongs to the project.

```
// GET: Projects/Details/5
public ActionResult Details(int? id)
{
    // Passes the Project Id to a hidden field in the Project User partial view
    ViewData["ProjectId"] = id;

    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    if (GetUserProject(id) == null)
    {
        return HttpNotFound();
    }

    return View(GetUserProject(id));
}
```

Fig. 2.4.8: Project Details (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/ProjectsController.cs>)

Since users added to the project are displayed through a partial view on the project details page, the Project/Details/id GET method is used to pass the “ProjectId” ViewData attribute to the user list. This allows its retrieval later in the RemoveUser method whenever a user attempts to remove another from the project. The method, similar to our Application Users POST method, accepts two parameters: one

identifying the current project (the ViewData attribute mentioned previously), and the id of the user selected for removal. The method searches the Current User's list of projects for the Project Identifier, then locates the User Identifier from our Users table in the database. It then removes the user from the project, saves the changes to the database, and refreshes the project details page. The removed user will no longer appear in the Users partial view list.

```
// POST: RemoveUser
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult RemoveUser(int ProjectIdentifier, string UserIdentifier)
{
    Project project = CurrentUser.Projects.FirstOrDefault(p => p.Id ==
        ProjectIdentifier);

    ApplicationUser user = db.Users.Find(UserIdentifier);

    project.Users.Remove(user);

    db.SaveChanges();

    return RedirectToAction("Details", "Projects", new { id = ProjectIdentifier });
}
```

Fig. 2.4.9: Remove User (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/ProjectsController.cs>)

Users should automatically be added to a project whenever they create one, and is accomplished by utilising the project input parameter and the CurrentUser functionality from the UserBase controller. The project is added as a new list item to the 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 now appears on the user's project dashboard.


```
// POST: Projects/Create
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "Id,Name,Code,Description,Users")] Project
project)
{
    if (ModelState.IsValid)
    {
        // Automatically add the logged in user to the created project
        CurrentUser.Projects = new List<Project>() { project };

        db.Projects.Add(project);

        db.SaveChanges();

        return RedirectToAction("Index");
    }

    return View(project);
}
```

Fig. 2.4.10: Create a Project (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/ProjectsController.cs>)

Test Cases

Similar to the project controller, a method is created that will determine whether a user is associated with a test case. The method returns the test case if the user is belongs to the parent project, by evaluating each test case that is tied to a project the user belongs to against the test case id parameter.

```
public class TestCasesController : ControllerBase
{
    public TestCase UserAssociatedTestCase(int? id)
    {
        return CurrentUser.Projects.SelectMany(p => p.TestCases).SingleOrDefault(tc =>
            tc.Id == id);
    }
}
```

Fig. 2.4.11: User Associated Test Case (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/TestCasesController.cs>)

A new test case should be automatically added to the project it was created from. A hidden ViewData attribute, `ProjectId`, will pass the project's id to a hidden field in the Test Case Create view. When the user invokes the Create POST method, the controller will identify the project from `ProjectIdentifier`, locate the project in the database, and add our newly created test case to the project. 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 will now show in the Test Cases list on the project details page.

```
// GET: TestCases/Edit/5
public ActionResult Edit(int? id)
{
    var testcase = UserAssociatedTestCase(id);

    var projectId = testcase.Project.Id;

    ViewData["ProjectId"] = projectId;

    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    if (UserAssociatedTestCase(id) == null)
    {
        return HttpNotFound();
    }

    return View(UserAssociatedTestCase(id));
}

// POST: TestCases/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "Id,Title,Priority,Precondition,Step,Status,Comment")] TestCase testCase, int ProjectIdentifier)
{
    if (ModelState.IsValid)
    {
        db.Entry(testCase).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Details", "Projects", new { id = ProjectIdentifier });
    }

    return View(testCase.Project.Id);
}
```

Fig. 2.4.12: Editing Test Cases (Source: Dick, Andrew J. 2016. GitHub <https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/TestCasesController.cs>)

Finally, when deleting a test case from a project, the test case is removed from the database, then the user is redirected back to the project details page.

```
// POST: TestCases/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    TestCase testCase = db.TestCases.Find(id);

    var projectId = testCase.Project.Id;

    db.TestCases.Remove(testCase);

    db.SaveChanges();

    return RedirectToAction("Details", "Projects", new { id = projectId });
}
```

Fig. 2.4.13: Deleting Test Cases (Source: Dick, Andrew J. 2016. GitHub <https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Controllers/TestCasesController.cs>)

Views

Creating a controller for each entity will in turn scaffold a number of .cshtml views for each model; Create, Delete, Details, Edit, and Index. Each view is returned when the GET method is invoked from the respective controller. The views provide an interface for our users to interact with the application database. For the purpose of this report, the application contains three primary views: Application Users, Projects, and Test Cases.

Layout

The layout view forms the general structure of each page in the application. Since this is a relatively small 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.

```

<div class="row wrapper page__dashboard">

  @RenderBody()

</div>

```

Fig. 2.5.1: Rendering the Body (Source: Dick, Andrew J. 2016. GitHub https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Views/Shared/_Layout.cshtml)

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.

```

<aside class="col-xs-12 col-md-3">
  <div>
    @RenderSection("SidebarProjectDashboard", required: false)
  </div>

  <div>
    @RenderSection("SidebarProjectDetails", required: false)
  </div>

  <div>
    @RenderSection("SidebarProjectUsers", required: false)
  </div>
</aside>

```

Fig. 2.5.2: Sidebar Sections (Source: Dick, Andrew J. 2016. GitHub https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Views/Shared/_Layout.cshtml)

Application User

The ApplicationUser view generates a list of all users registered in the 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, then generates a loop to display each 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. 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)
            </td>

            <td>
                @Html.Hidden("ProjectIdentifier", ViewData["ProjectId"])
            </td>
        </tr>
    }
}

```

Fig. 2.5.3: List of Users (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Views/ApplicationUsers/Index.cshtml>)

Forename	Surname	Email	
Test	User1	testuser1@cohaesus.co.uk	Add to Project
Test	User2	testuser2@cohaesus.co.uk	Add to Project
Test	User3	testuser3@cohaesus.co.uk	Add to Project

Fig. 2.5.4: Add a User View (Source: Dick, Andrew J. 2016. GitHub
https://github.com/AndrewJDick/TestRailMVC/blob/master/docs/Screenshots/TestRail_ApplicationUser-Index.png)

Since a list of users will be displayed on the Project Details page, a partial view will be rendered when a user visits a Project Details page. The partial 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 will be wrapped in a form, and the delete button will trigger the RemoveUser method in our Projects controller. Two hidden fields also store the id of the project and the id of the user to be removed.

```
@model IEnumerable<TestRailMVC.Models.ApplicationUser>

<table class="table">

    @foreach (var item in Model)
    {
        using (Html.BeginForm("RemoveUser", "Projects"))
        {
            @Html.AntiForgeryToken()

            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.UserName)
                </td>

                <td>
                    <input type="submit" value="Remove" class="sidebar__btn--link"/>
                </td>

                @Html.Hidden("UserIdentifier", item.Id)
                @Html.Hidden("ProjectIdentifier", ViewData["ProjectId"])
            </tr>
        }
    }
</table>
```

Fig. 2.5.5: List of Users Partial View (Source: Dick, Andrew J. 2016. GitHub https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Views/ApplicationUsers/_ProjectUsersPartial.cshtml)

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 through a foreach loop. The project title is converted to an *@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.

```

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink(item.Name, "Details", new { id = item.Id })
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Code)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Description)
        </td>

        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.Id }, new { @class =
                "red" })
        </td>
    </tr>
}

```

Fig. 2.5.6: Project Index (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Views/Projects/Index.cshtml>)

Entity Framework Demo			
Project Dashboard			Create a project
Name	Project Code	Description	
Test Project 1	123TP021	A random project for test purposes	Edit Delete
Test Project 2	123TP024	Another project to display on the Project Dashboard	Edit Delete
Test Project 3	123TP025	A dummy project built using React	Edit Delete
Test Project 4	123TP026	Demonstration of the Test Case Create View	Edit Delete
Test Project 5	123TP028	A handlebars-based project built using Backbone.js	Edit Delete

Fig. 2.5.7: Project Dashboard View (Source: Dick, Andrew J. 2016. GitHub
https://github.com/AndrewJDick/TestRailMVC/blob/master/docs/Screenshots/TestRail_Project-Index.png)

The Shared layout's SidebarProjectDashboard section is also defined here; a simple button that allows users to create a new project from the sidebar.

```

<!-- Sidebar Content -->
@section SidebarProjectDashboard {

    <button class="sidebar__button" onclick="location.href=@Url.Action("Create",
    "Projects")"> Create a project </button>

}

```

Fig. 2.5.8: Project Dashboard Sidebar (Source: Dick, Andrew J. 2016. GitHub <https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Views/Projects/Index.cshtml>)

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.

```

<div>
    <h3>Test Cases</h3>

    @Html.Partial("~/Views/TestCases/_ProjectTestCasesPartial.cshtml",
    ViewData.Model.TestCases)
</div>

<p>
    @Html.ActionLink("Return to Project Dashboard", "Index")
</p>

<!-- Sidebar Content -->
@section SidebarProjectDashboard {

    <button class="sidebar__button" onclick="location.href=@Url.Action("Create",
    "TestCases", new { id = Model.Id })"> Create a TestCase </button>

    <button class="sidebar__button" onclick="location.href=@Url.Action("Index",
    "ApplicationUsers", new { id = Model.Id })"> Add a User </button>

}

```

Fig. 2.5.9: Project Details View (Source: Dick, Andrew J. 2016. GitHub <https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Views/Projects/Details.cshtml>)

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.


```

@section SidebarProjectUsers {

    <h2 class="sidebar_header">Users</h2>

    @Html.Partial("~/Views/ApplicationUsers/_ProjectUsersPartial.cshtml",
        ViewData.Model.Users)
}

```

Fig. 2.5.10: Project Details Users (Source: Dick, Andrew J. 2016. GitHub
<https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Views/Projects/Details.cshtml>)

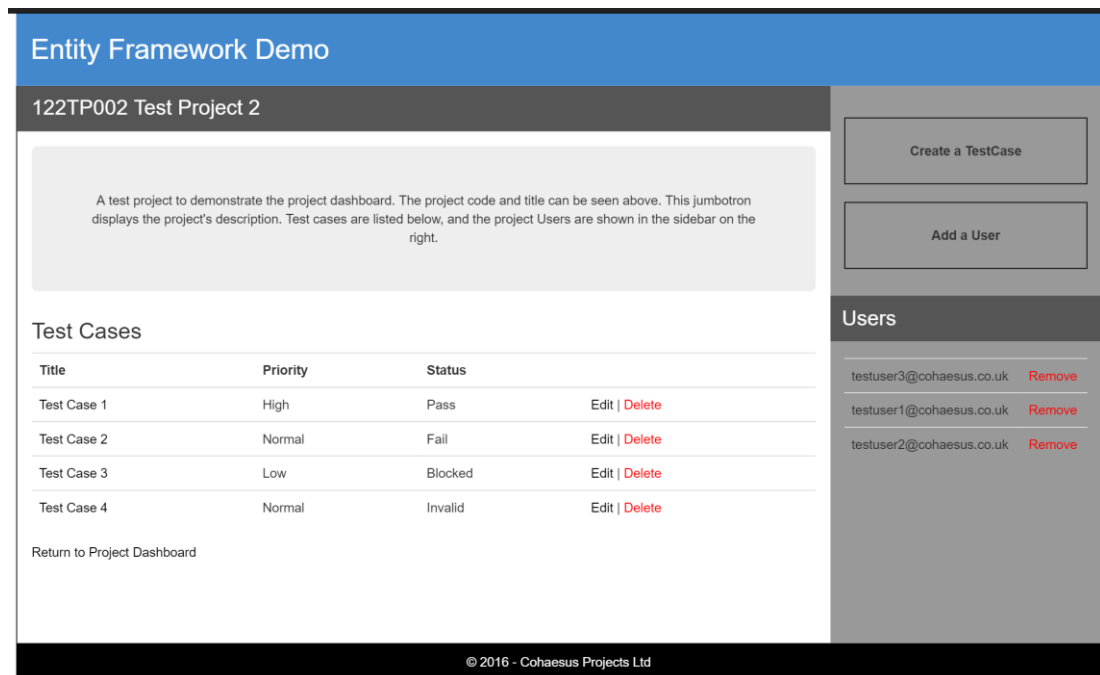


Fig. 2.5.11: Project Details View (Source: Dick, Andrew J. 2016. GitHub
https://github.com/AndrewJDick/TestRailMVC/blob/master/docs/Screenshots/TestRail_Project-Details.png)

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.

```
@foreach (var item in Model)
{
    <tr>
        <td>
            <!-- linkText, actionName, controllerName, routeValues, htmlAttributes -->
            @Html.ActionLink(item.Title, "Details", "TestCases", new { id = item.Id },
                null)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Priority)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Status)
        </td>

        <td>
            @Html.ActionLink("Edit", "Edit", "TestCases", new { id = item.Id }, null)|
            @Html.ActionLink("Delete", "Delete", "TestCases", new { id = item.Id },
                new { @class = "red" })
        </td>
    </tr>
}
```

Fig. 2.5.12: Project Test Case Partial View (Source: Dick, Andrew J. 2016. GitHub https://github.com/AndrewJDick/TestRailMVC/blob/master/TestRailMVC/Views/TestCases/_ProjectTestCasesPartial.cshtml)

Conclusions

This report evaluated and outlined the key features associated with the ADO.NET Entity Framework and its integration within a simple .NET MVC application. Applications until recently have traditionally adopted a Data Driven Design approach; where application software is designed on business logic data. Developers are therefore restricted to building the application based on the targeted database schema. Object/Relational mappers such as Microsoft's Entity Framework instead apply Domain Driven Design, by raising an application's level of abstraction from a relational to an entity level via an Entity Data Model.

In doing so, it eliminates the need for developers to implement data-access code by allowing them to utilise LINQ queries to perform operations on domain-specific objects. Doing so leads to increased performance, less code - and theoretically a lower skill cap - to accomplish complex data-relational tasks between an application and its data storage.

Further Work

Further work would see the implementation of roles to vary the user privileges. 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: Admins, who would retain all previously mentioned privileges; Members, who 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 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.

Finally, a mobile-first overhaul of the design of the application, including support for all latest browsers, would enhance the application's aesthetic, and the implementation of unit tests would help to create a more robust project infrastructure.

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 patience and support over the duration of the project.

References

Adya, A., Blakeley, J.A., Melnik, S. and Muralidhar, S., 2007. Anatomy of the ADO .NET Entity Framework

Awad, E.M., 1985. *Systems analysis and design*. McGraw-Hill Professional

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

Melton, J. and Simon, A.R., 1993. Understanding the new SQL: a complete guide.

O'Neil, E.J., 2008, June. Object/relational mapping 2008: hibernate and the entity data model (edm). ACM.

Owens, M., 2006. The definitive guide to SQLite

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

Microsoft. 2016. Entity Framework | The ASP.NET Site. [ONLINE] Available at: <http://www.asp.net/entity-framework>. [Accessed 21 July 2016]

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

Bibliography

Application Source Code

<https://github.com/AndrewJDick/TestRailMVC>

MSDN Entity Framework Documentation

<https://msdn.microsoft.com/en-gb/data/ee712907>

ASP.NET Entity Framework

<http://www.asp.net/entity-framework>

Online Tutorials

Entity Framework 6.0 Tutorial

<http://www.entityframeworktutorial.net/entityframework6/introduction.aspx>

Pluralsight Entity Framework Code First Migrations

<https://www.pluralsight.com/courses/efmigrations>

Lynda SQL Essential Training

<https://www.lynda.com/SQL-tutorials/SQL-Essential-Training/139988-2.html>

YouTube Tutorials

Entity Framework Tutorial

https://www.youtube.com/playlist?list=PL6n9fhu94yhUPBSX-E2aJCnCR3-_6zBZx

ASP.NET MVC Tutorial

<https://www.youtube.com/playlist?list=PL6n9fhu94yhVm6S8I2xd6nYz2ZORd7X2v>

Glossary

.NET Framework - A programming infrastructure created by Microsoft for building, deploying, and running applications and services that use .NET technologies. 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 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 its 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 "virtual object database" that can be used from within the programming language.

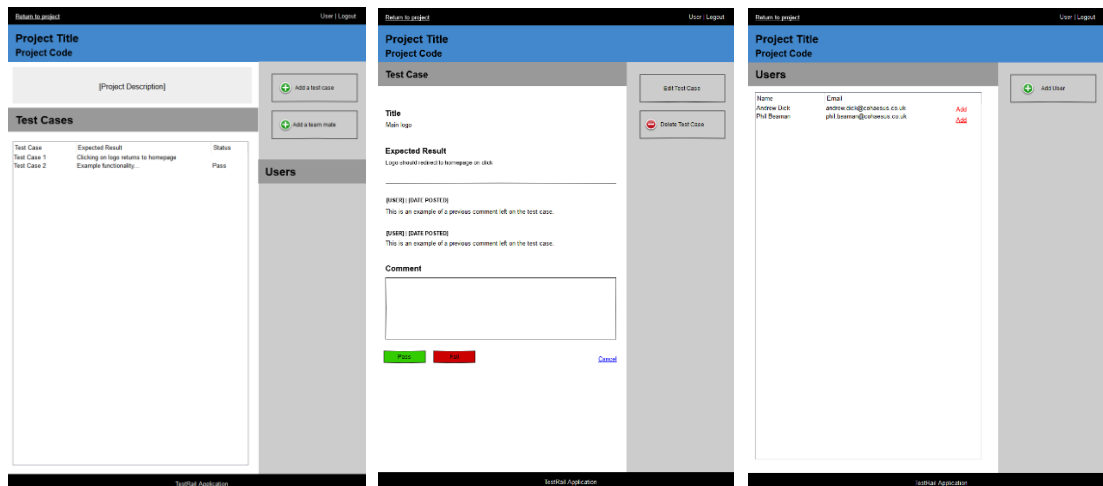
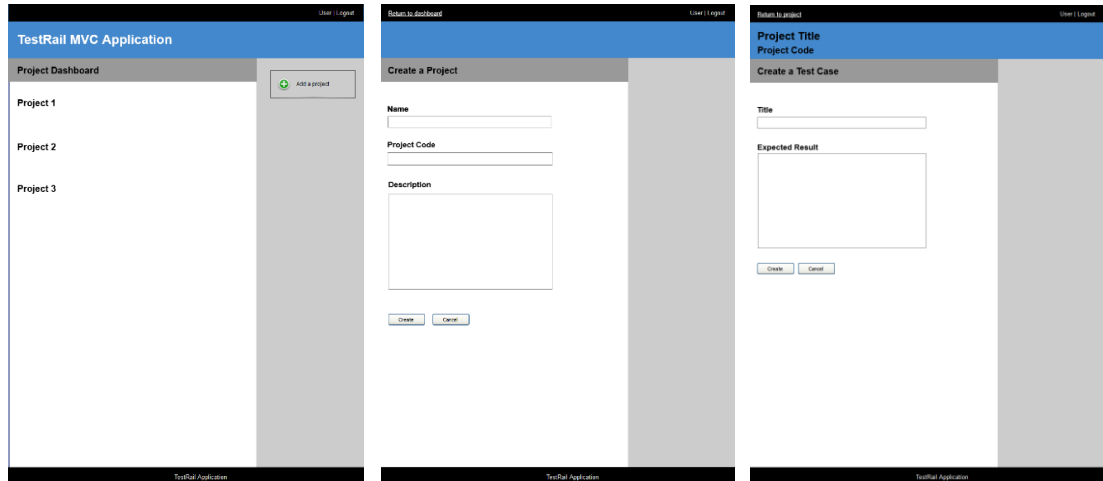
OAuth - An open standard for authorization, allowing sites and apps the ability to allow users to log in with third-party accounts (Such as Facebook, Google, LinkedIn, etc.) without exposing their password.

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

Wireframes



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

