**ENTERPRISE FRAMEWORKS**

**CAR RENTAL WEBSITE**

**CAVAN GAELS**

**Brian Burns x10205284**

**Sean O’Shea x10206230**

**Torsby Attipoe x11106409**

1. Overview
2. Application
3. Background research and investigations
4. Project Plan
5. Software development methodology employed
6. Requirement Analysis

* Functional Requirements
* Non Functional Requirement

1. Use cases
2. Architecture/Design approach
3. Models (Class Models / Data Models etc.)
4. Implementation of particular OOP constructs
5. Design patterns and architectural patterns implemented in the application
6. How cross-cutting concerns have been handled
7. Configuration of the application
8. Security
9. Scalability of the application
10. Testing Approach
11. Other relevant features of the application, e.g.,

* use of client-side processing,
* use of Ajax,
* use of web services,
* use of a workflow engine etc.
* use of an ORM tool
* use of dependency injection / IoC containers

1. References

**OVERVIEW:**

Enterprise Framework project to develop a Car Rental System.

Development environment used is the .NET framework

MicroSoft integrated development environment Visual Studio

Architectural pattern used is MVC4

API is REST

**APPLICATION:**

**Car Rental Website**

The purpose of the application is the provision of a web based Car Rental System . The application allows the user to book car hire on line for the rental of cars for varying periods from different locations throughout the country. The application also allows owners of fleets of cars to register their details, upload fleets of cars available for hire, change cars available for hire. The information on Cars, Bookings, Suppliers are all stored in a Relational Database which has been deployed to the cloud. The information stored can be edited, deleted, displayed and searched.

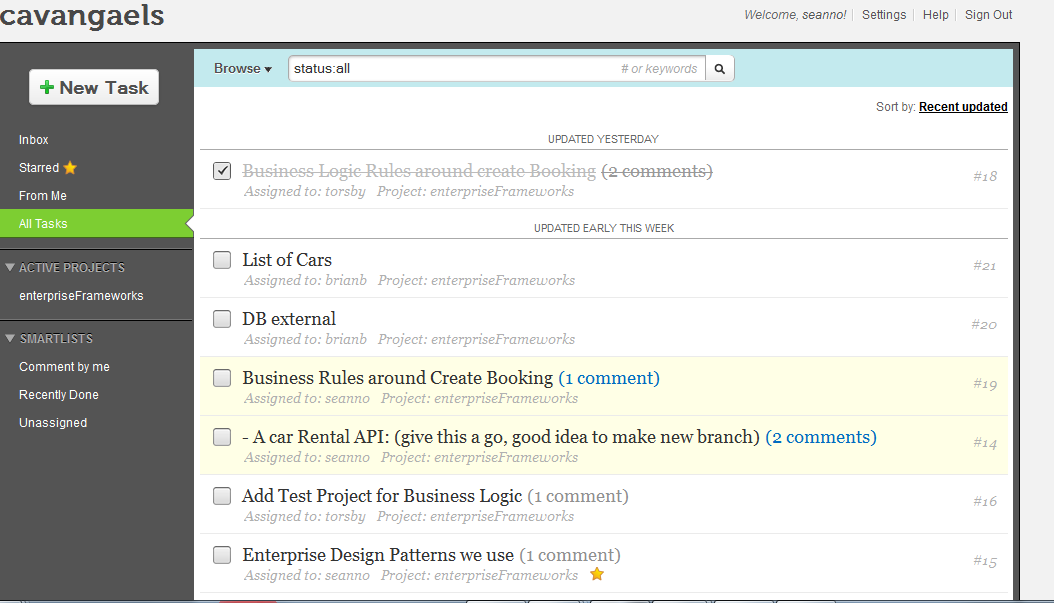
**BACKGROUND RESEARCH & INVESTIGATIONS**

Pluralsight, the .net website and w3schools were used as part of our research to get insight on the C# and MVC framework.

**PROJECT PLAN**

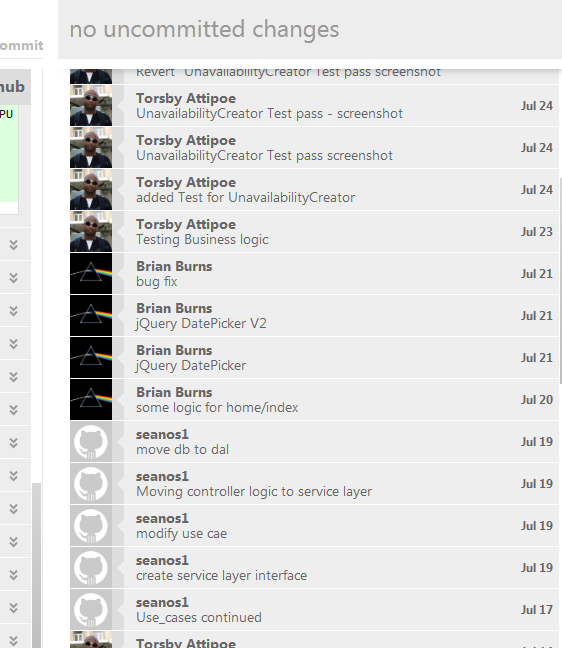
**TaskAnt**

Whilst doing the project we used a web based Repository called TaskAnt. This is a team based tasks web tool which we used to assign, track and search tasks allocated to various members of the team.



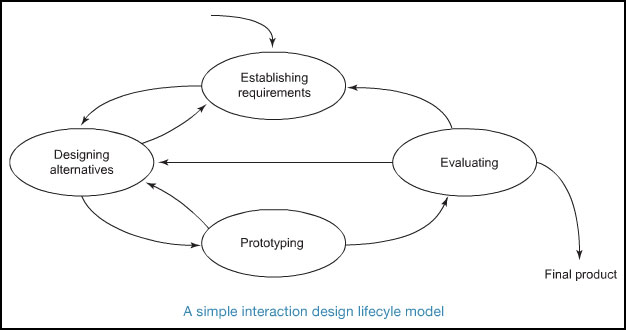
**Git Hub**

We also used **GitHub** which is a web-based hosting service for software development projects that use the Git revision control system. This recorded all the commits and changes to our projects carried out by different members of our Team and allowed us all up to date information on the project as it progressed.



**SOFTWARE DEVELOPMENT METHODOLOGY**

Simple Interactive Design Model recommended by Rogers et All, 2011 was implement to help with the development methodology and process.

****

Simple Interactive Design Model (p. 332, Rogers et All, 2011),

**REQUIREMENTS ANALYSIS**

Functional Requirement

* **User can search and book car**
* **Booking must be in future,**
* **Booking end date must be after start date**
* **Booking must have car\_id, customer\_id**
* **Replace a car with a different car**
* **Booking should calculate total cost (daily rate \* number of days)**
* **Booking should calculate total cost (daily rate, number of days)**
* **Users can register details**
* **Users can log in and log out**

Non Functional Requirement

**Non-Functional Requirements**

The application’s non functional requirements will include:

* Web application is to be programmed in C#, and built on ASP.NET MVC Architecture
* The web application must be fully tested
* Scalable
* To be responsive
* Easy to use

**USE CASES**

# Use Case 1: User logs on

* User asked to input pickup location
* The user is presented with a drop down box of a list of locations
* The user chooses the location that they require to pick up the car
* User fills out the following details:
  + Pick up date & time
  + Drop off date & time
  + Car Class
  + Email address (optional)
* User Enters Submit Button
* The user is given a report of the number of cars that are available

# Use Case 2: A list of cars available is given from the cheapest to the dearest

* The cars are listed from the cheapest to the dearest.
* The cars are described by the Make, Model and the Number of passengers it can hold & the luggage space.
* The user then selects the car that they want to hire by Select Button
* The user is then given a screen with the details of the car, the location, the pickup and drop off times and the total rental cost.
* The user is asked to confirm by clicking a Button called Confirm
* The user is then re directed to a new page to register their details and confirm the booking

# Use Case 3: User Completes a form giving their Details and are Requested to enter their Payment Details

The customer is requested to fill in a form giving their first name, last name, age, email address

They are then requested to fill in their payment details. Their credit card or debit card.

The user is then required to press a button to confirm the booking

A page is returned confirming the booking

An order confirming the booking is also sent to the car supplier by email

# Use Case 4: Car Suppliers Log in & Report

* Supplier enters login button
* Supplier enters id and password
* The supplier is then returned a page with supplier details. Giving their name address and the number of current cars on the database
* This supplier will include an option to list from the cars available from that supplier & the dates & times the cars are hired from
* The supplier then selects the required options it wants from the report, either a full list of cars or just a list of the confirmed bookings.
* The supplier is then returned a list showing the Registration of the Car, The Car description (name, make, model & colour) & current bookings in the car.
* The supplier also has option to amend car listing.

# Use Case 5: Supplier amends car details by either entering a new csv file of amending an individual car record.

* From car suppliers page while logged in the supplier is given the option to amend the car listing by clicking in a button.
* The supplier is then presented with a drop down listing with options to either
  + Remove a car either temporarily or permanently
  + Replace a car with a different car
  + Add a new car
  + Replace the full list of cars with a new list by updating with a new csv file.

If the supplier decides to remove the car either temporarily or permanently they can change the status of the car to not available or change the status to available

If a new additional car becomes available the supplier can add a record for that by entering the Reg of the car, the name, make model & colour.

* object*Export As* drop-down field
* The user then selects the *Generate Report* button
* The is re-directed to a new page which displays the generated CSV file

**ARCHITECTURE / DESIGN APPROACH**

The architectural design approach used is the layer approach. We have used the layer approach ie Model

View

Controller

Business Logic Layer

The Data Access Layer

Three principal layers

1. Presentation Layer - provision of services, display of information in html, handling of user requests, mouse clicks, keyboard hits, http requests, batch API.
2. Domain Layer -Logic that is the real point of the system
3. Data Source -Communication with databases, messaging systems, transaction managers,

**MODELS ( CLASS MODELS / DATA MODELS)**

**Domain Model**

We have used a domain model in our project**.** A domain model is an object model of a domain that incorporates both behaviour and data. A domain model creates a web of interconnected objects (in our case ie Booking Business Object, Car Business Object, Customer Business Object, Supplier Business Object. We provide an interface over the domain model (Service Layer) to offer a contract of methods for the controller to call.

An example of code from the Car Business Object

**BusinessLogic\CarBObj.cs**

namespace BusinessLogic

{

public class CarBObj

{

private SupplierBObj \_supplier = null;

private Decimal \_dailyRate = 50;

private int \_carId;

public CarBObj(SupplierBObj supplier)

{

\_supplier = supplier;

}

public void setId(int id)

{

\_carId = id;

}

public int getId()

{

return \_carId;

}

**ISSUES DURING PROJECT**

1. Impedence MisMatch

====================

Microsoft SQL Server did not like foreign key relationships that may cause cycles or multiple cascade paths. [\*Stackoverflow\*](http://stackoverflow.com/questions/851625/foreign-key-constraint-may-cause-cycles-or-multiple-cascade-paths)

[\*MSDN\*](http://msdn.microsoft.com/en-us/library/ms186973%28v=sql.105%29.aspx)

As an example we had tables Bookings, Cars both linking to Supplier by FK SupplierId.

Ideally both models should contain a reference to the supplier without a problem but SQL Server played it ultra conservative and complained about multiple cascade paths.

Cascades are rules to carry out if a user deltes a key to which foreign keys point (e.g. ON DELETE of an entity). The issue happens if there is a triangular relationship between Booking to Supplier to Car and back to Booking. What we found was the relationship must not cycle like this.

The solution is to create clear cascade rules for ONDELETE or to remove the offending foreign relationship. For the moment, the simple solution was to remove the link between supplier and booking (not ideal).

**2. Database in Azure**

**Firewalls block the required port for AzureSql**

**3. Encryption Issue**

**We attempted to encrypt the password to access the database but this caused configuration errors and as a result we had to un-encrypt again. This leaves the password in the code.**

**Attempt to encrypt**

**Web.config file**

<connectionStrings configProtectionProvider="RsaProtectedConfigurationProvider">

<EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"

xmlns="http://www.w3.org/2001/04/xmlenc#">

<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

<EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">

<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1\_5" />

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

<KeyName>Rsa Key</KeyName>

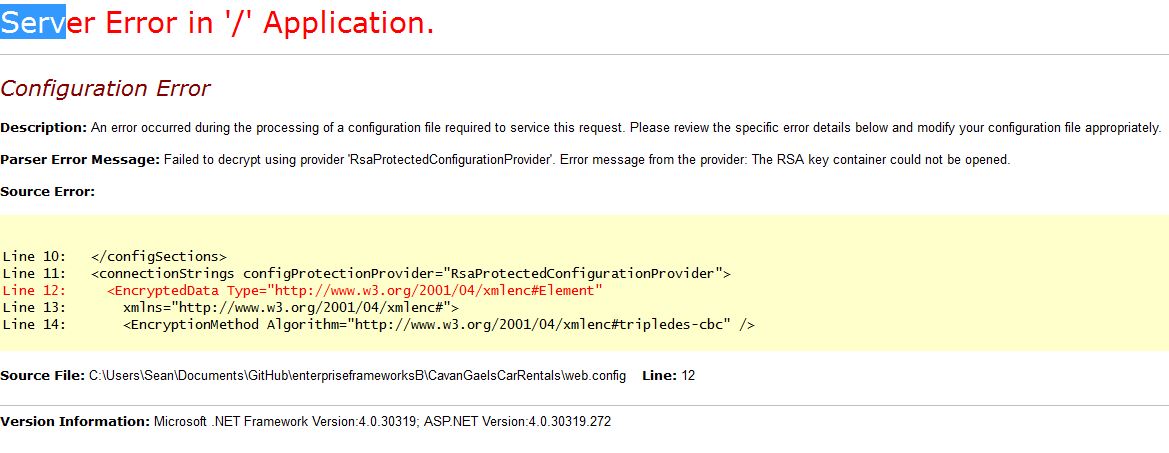
</KeyInfo>

<CipherData>

<CipherValue>WI4PTl7VUltIYXgYtaqH+8nZbSes0JRohrQhgBEq3uyY5z4bz0RlbBvtC+IQeELQnbBkj/dMMQaB6ly2Nfon5tkZAIi0eyDhRZ8gOTSG0avwfZBYuUQbDVk9hu1uZW6PfkGgUC/rhrlyHTQyIomRaNUrqaKPau94Iy4F+0xpCrU=</CipherValue>

</CipherData>

**Error report**

****

**IMPLEMENTATION OF PARTICULAR OOP CONSTRUCTS**

Dependencies between the business logic objects such as “Booking”, “Car”, “Unavailable date” were introduced through object composition. These objects were created with the required business rules. Their private properties were accessed through public methods that enforced the rules.

The Service Layer instantiates the required business objects and lets them play out the business rules defined in the tests. The service layer calls methods such as Booking.Valid which returns true or false and Booking.create which returns an unavailable period which matches the rules.

**DESIGN PATTERNS AND ARCHITECTURAL PATTERNS IMPLEMENTED**

The application started with an MVC layer and it became necessary to redefine the M in MVC. Originally the models were Active Record objects that mapped directly to the database. We had a large set of business rules that would be best implemented using domain logic. The domain logic was set up as an independent class library which could be used throughout multiple applications. With the models moved to data access layer and business objects for the domain model the controller was part of the presentation layer and required presentation models to match.

We created ViewModels which transported data to the views which provided the exact data the presentation layer needed. The object relational models were then used in the DAL layer.

The controller code was considered part of the presentation layer and as much as possible logic was moved to a service layer that sat on the logic layer. This acted like a transaction script for each controller action.

Repetitive database queries caused us to create a repository layer that sat on the DAL layer. Mostly we did not require data transfer objects between ViewModels and logic or between logic and DAL because the ViewModels themselves acted as DTOs.

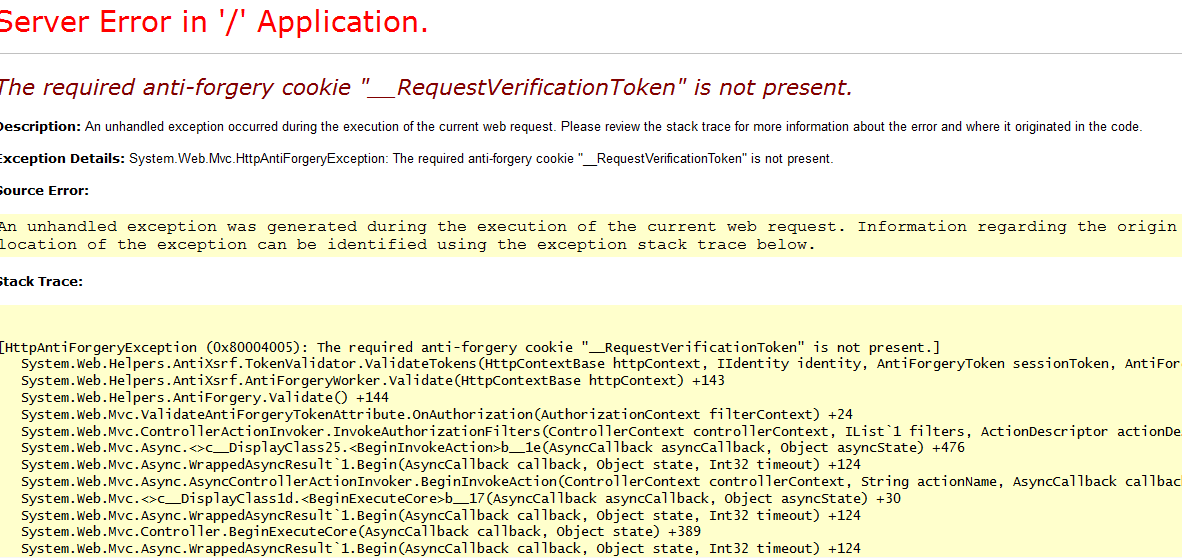
Linq Entity Framework hides the movement of data between the plain old clr objects and the database and as such acts like a datamapper.

The unit of work is similarly hidden in the entity framework because changes to objects are monitored. For example we have one case where a create booking action creates both a booking and an unavailability, these are both added to entity framework before calling SaveChanges.

**HANDLING OF CROSS-CUTTING COPNCERNS**

Cross Site Request Forgery (SCRF) happens when a malicious user gets one of your website users to submit a form post that they didn’t intend to submit, but the browser allows the request to work because it will submit the proper authentication cookie. The form we use when the user logs in creates a cookie which stays around for the duration of the browser session at the very least. If the user logs on to the site and then goes to an irreputable site or opens an email from a malicious person we could end up with a CSRF attack. To reduce the possibility of an attack we have used an anti forgery token. An anti forgery token is trying to ensure that when the user submits a form to your site that the form they are posting came from your server and not from some malicious person. It does this by embedding some cryptographic numbers in a hidden form input and also in a cookie and making sure they compare correctly. It means if someone uses the cryptographic code in the form they won’t be able to set a cookie in the user’s browser.

Using the “VailidateAntiForgeryToken” attribute in the create booking form we get an error as below.



We sort that error out by putting a cookie in the html page which verifies that form is submitted from clients machine.

**CONFIGURATION OF THE APPLICATION**

The foundation of the application is based on the configuration design paradigm and building the web application using .Net naming and coding conventions. The assisted in facilitating the development as there was less configuration to be done.

An example is exporting HTML and CSS into C#. This was made easy because Visual Studio C# and MVC4 has an inbuilt HTML template, so it was just a matter of replacing it with what was being imported.

**SECURITY**

The application contains the basic account registration and login features of an MVC4 application.

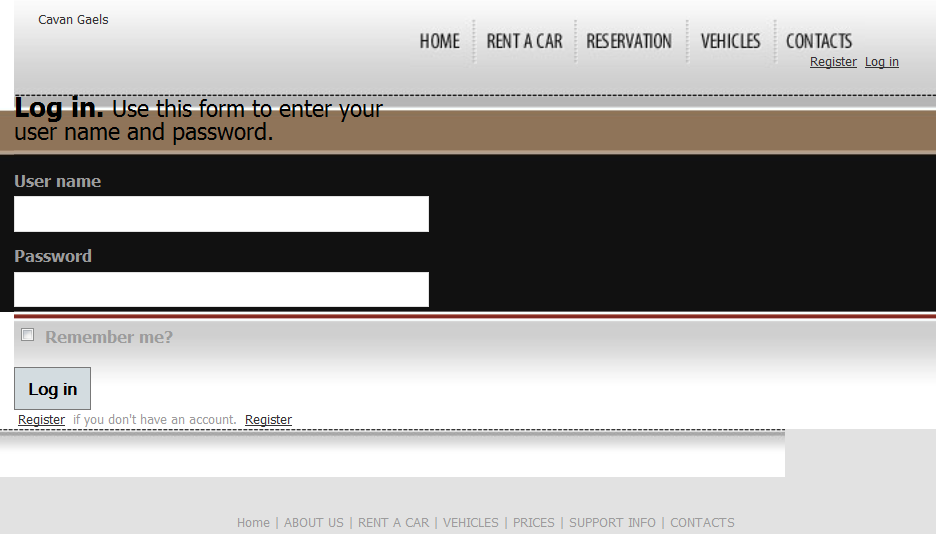
Two types of users exist, Customers & Suppliers.

Customers must login or register before they can hire a car. They supply username and email address.

Suppliers must register initially with the special password given to them by us. They then register themselves giving user name, email and password which is verified twice

They when they are logged in can update their list of cars as a CSV file. Only logged in suppliers can do this.

Logged in suppliers can also use CRUD, Create, Read, Update Destroy.

****

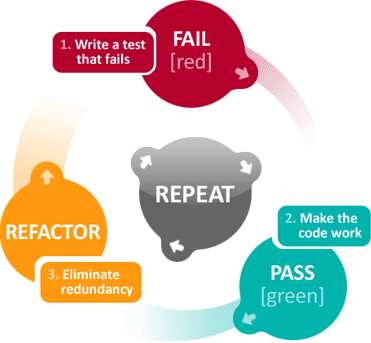
**SCALABILITY OF THE APPLICATION**

This application uses a SQL database which stores variable information. As the database grows there is the need to scale up as a viable solution for a certain potentially large classes. The .NET MVC helps resolve this problem. The .NET MVC 4 is a framework for building scalable, standard-based web application using well established design patterns and the power of the ASP.NET and the .NET framework. MVC4 has Async libraries available which means that tasks can run on the server Asynchronously at the same time

**TESTING APPROACH**

****

*Test-driven development cycle (TDD from a Wikipedia source )*



The TDD Mantra: "red, green, refactor, repeat"

### Add a test

In this test-driven development, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented based on feature's specification and requirements. This was accomplished through [use cases](http://en.wikipedia.org/wiki/Use_case) and [user stories](http://en.wikipedia.org/wiki/User_story) that cover the requirements and exception conditions.

### Run all tests and see if the new one fails

This validates that the [test harness](http://en.wikipedia.org/wiki/Test_harness) is working correctly and that the new test does not mistakenly pass without requiring any new code. The new test should also fail for the expected reason.

### Write some code

The next step is to write some code that will cause the test to pass.

### Run the automated tests and see them succeed

If all test cases now pass, then code meets all the tested requirements.

### Refactor code

Code is now up as necessary with the confident that [code refactoring](http://en.wikipedia.org/wiki/Code_refactoring) is not damaging any existing functionality. The concept of removing duplication is an important aspect of any software design.

### Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality.

A unit test driven development approach was followed as described below.

**Functional Testing**

**Test were performed for ensure system was working fine and producing the desired results. For instance, it was required that a test for Booking for the Business Logic should do the following:**

* Booking must be in future,
* Booking end date must be after start date
* Booking must have car\_id, customer\_id
* Replace a car with a different car
* Booking should calculate total cost (daily rate \* number of days)
* Booking should calculate total cost (daily rate, number of days)

**This is the test to check if the booking end date is after the start date**

**UnitTest1.cs**

[TestMethod]

[ExpectedException(typeof(ArgumentException))]

public void booking\_end\_date\_Should\_be\_after\_start\_date()

{

DateTime pickupDate = new DateTime(2012, 12, 31);

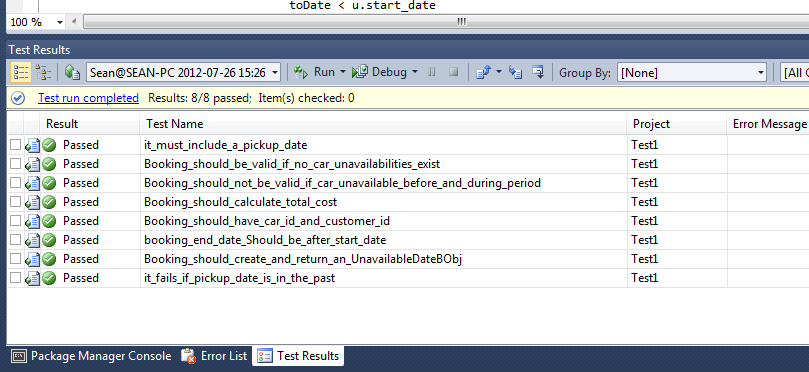
var booking = BookingBObj.newBooking(new CarBObj(new SupplierBObj()));

booking.setBookingRange(pickupDate, -5);

Assert.Fail("Message Constructor should have thrown an exception");

}

**Test Results on all tests 26thJuly2012**

****

**Non Functional Testing**

**Load Testing**

We tested the application to ensure that the more than 1 user could use the system at any one time could use the system and access the data base. This was found to work fine in our limited trial.

**Compatibility Test**

We discovered whilst testing that an error occurred while testing

**OTHER RELEVANT FEATURES OF THE APPLICATION**

**Use of Ajax**

**Ajax is used in when the customer is querying the home page for cars available for the location between start date and end date. An ajax request returns the total number of cars available in a partial**

**Use of Web Services**

**API**

We have created an API to give a list of the total cars available. This is included in an API folder in our project.

**Use of a workflow engine**

Workflow engines were no match to writing behavioural tests for business rules, in some ways we viewed this test-driven development as our workview engine

**Use of an ORM tool**

Linq Entity Framework

**Use of Dependency Injection**

Interfaces were implemented to try to abstract out dependencies between presentation layer and logic. It is expected in future to implement dependency injection of service logic layer into the controller to allow for easy testing of the controller. That way a simple test service logic class will stub out the service layer.

Due to constraints we just got as far as implementing the interface.

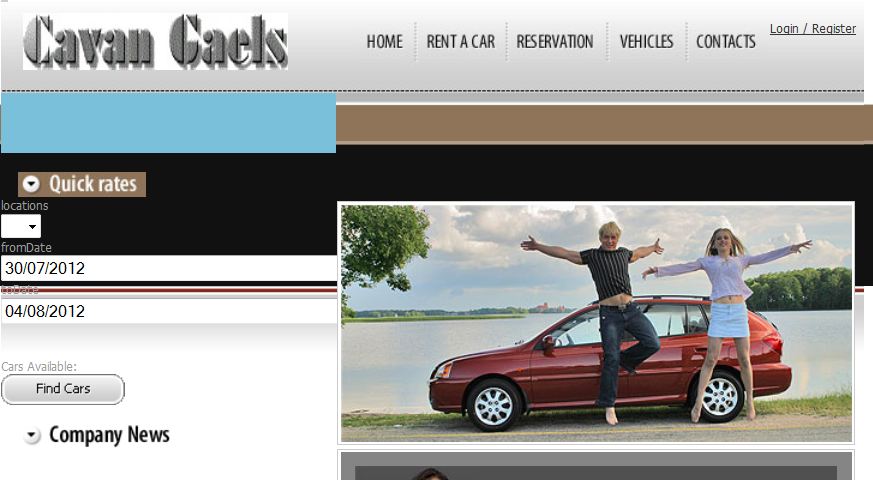
**DATABASE:**

The application is using the SQL Azure package as the database to store the user information.

The data can be created, deleted, edited and retrieved.

**/Permission/SearchIndex**

A dropdown box allows the database to be searched by the cars available for hire in locations..

****

**SEARCHING THE DATABASE:**

The database can be searched by the available cars for hire from a supplier location.

The BookingController sets the search parameters for the SearchIndex View.

**BookingController.cs code to set search parameters:-**

namespace CavanGaelsCarRentals.Controllers

{

public class BookingController : Controller

{

private IServiceLayer logic = new ServiceLayer();

//

// POST: /Booking/

[HttpPost]

public ActionResult Index(LocationsUI requestedTimePlace)

{

BookingUI carList = new BookingUI();

carList = logic.ListAvailableCars(requestedTimePlace);

return View(carList);

}

**Code for dropdown box in the HomeIndex.cshtm View:-**

Database view in visual studio showing user records

The database is stored in the App\_Data folder and is called

**References**

[1] Rogers, Y., Sharp, H., Preece, J. (2011)., *Interaction Design: beyond human-computer interaction*, 3rd Edition. Wiley Publishing, Inc, USA

[2] <http://www.agileapps.co.uk/methodology/continuous.html>

Test-driven Development (TDD)