ASSIGNMENT CIS2055 - WEB APPLICATIONS ARCHITECTURE AND SYSTEMS DEVELOPMENT

ANDREA NAUDI (29698M)

BACHELOR OF SCIENCE IN INFORMATION TECHNOLOGY (SOFTWARE DEVELOPMENT)

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Andrea Naudi	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

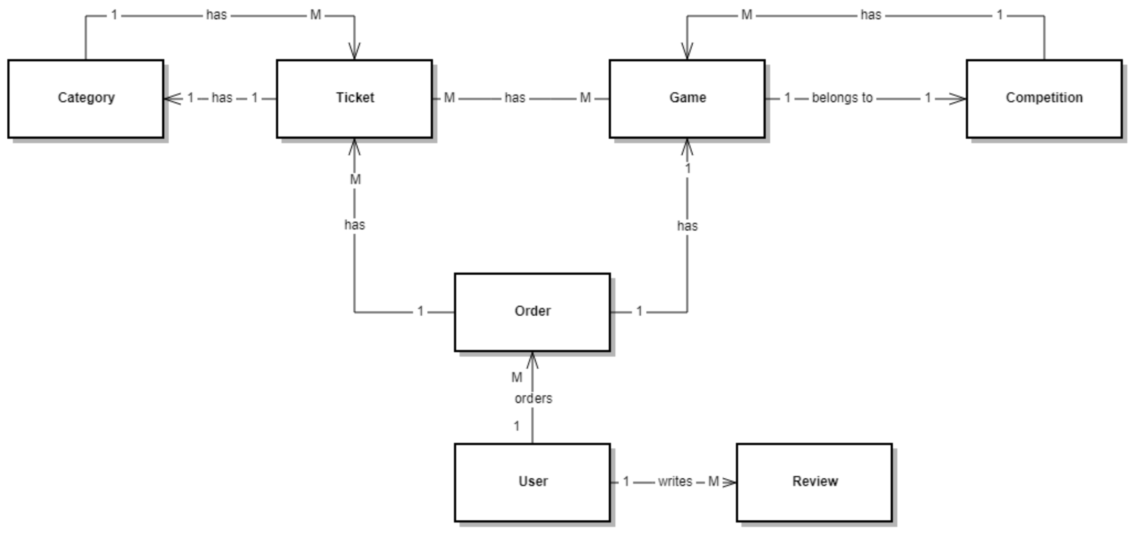
CIS2055	Web Applications Architecture and Systems development Assignment
Course Code	Title of work submitted

21/05/2018
Date

Task Definition

The web application designed is a ***Season Ticket*** management system. Apart from regular tickets, sports teams around the world offer season tickets to their fans. A season ticket may be used by people other than the owner. Sometimes, season tickets are also categorized; meaning that tickets in a higher category get more privileges such as better seating and dinners before the game. This application allows an administrator to offer multiple season tickets online (trusted to him/her by season ticket holders). Users can then log in, choose, and purchase these tickets for their desired match. Users can also review and give feedback on their experience by using the Reviews section. In short, this system allows the administrator to be the direct link between the public and season ticket holders.

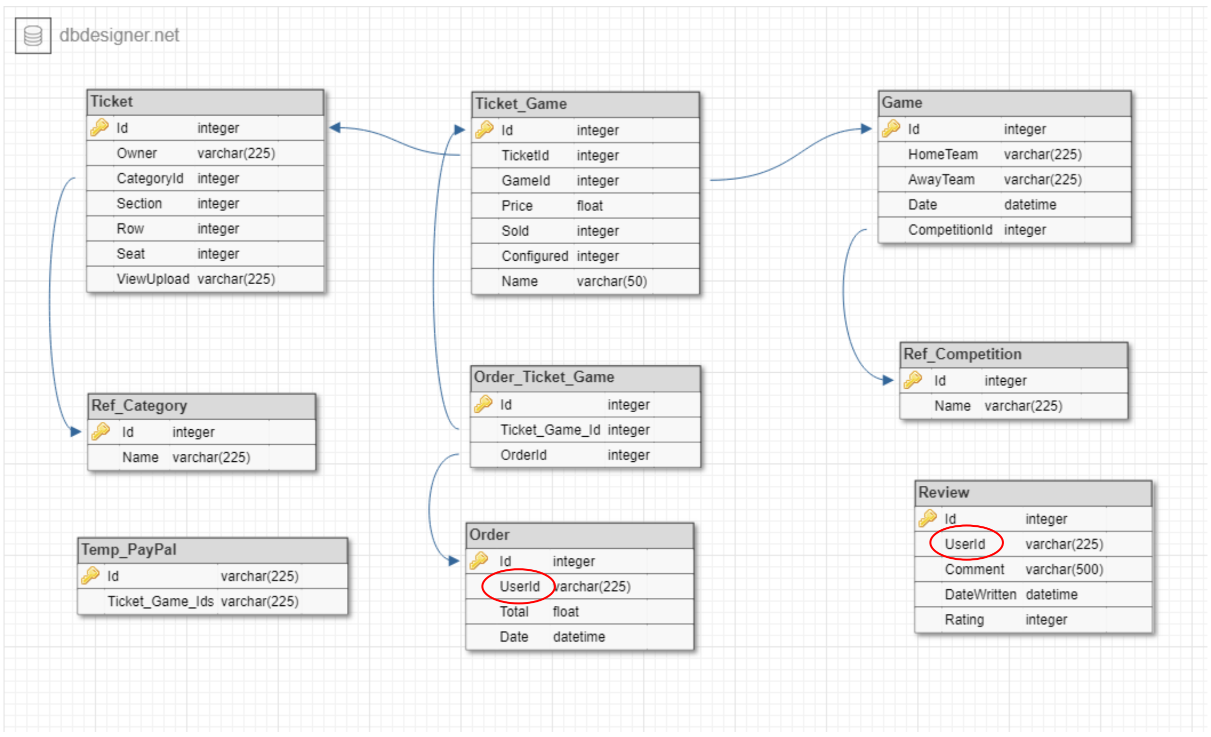
Class diagram



The class diagram above shows that:

- A ticket has 1 category.
- A game belongs to 1 competition.
- A category has many tickets.
- A competition has many games.
- A ticket can be used for many games, and a game has many tickets.
- A user can place many orders which consist of many tickets belonging to the same game.
- A user can write/read many reviews.

Schema



Ticket

Id	integer
Owner	varchar(225)
CategoryId	integer
Section	integer
Row	integer
Seat	integer
ViewUpload	varchar(225)

Ticket_Game

Id	integer
TicketId	integer
GameId	integer
Price	float
Sold	integer
Configured	integer
Name	varchar(50)

Game

Id	integer
HomeTeam	varchar(225)
AwayTeam	varchar(225)
Date	datetime
CompetitionId	integer

Ref_Category

Id	integer
Name	varchar(225)

Order_Ticket_Game

Id	integer
Ticket_Game_Id	integer
OrderId	integer

Ref_Competition

Id	integer
Name	varchar(225)

Temp_PayPal

Id	varchar(225)
Ticket_Game_Ids	varchar(225)

Order

Id	integer
UserId	varchar(225)
Total	float
Date	datetime

Review

Id	integer
UserId	varchar(225)
Comment	varchar(500)
DateWritten	datetime
Rating	integer

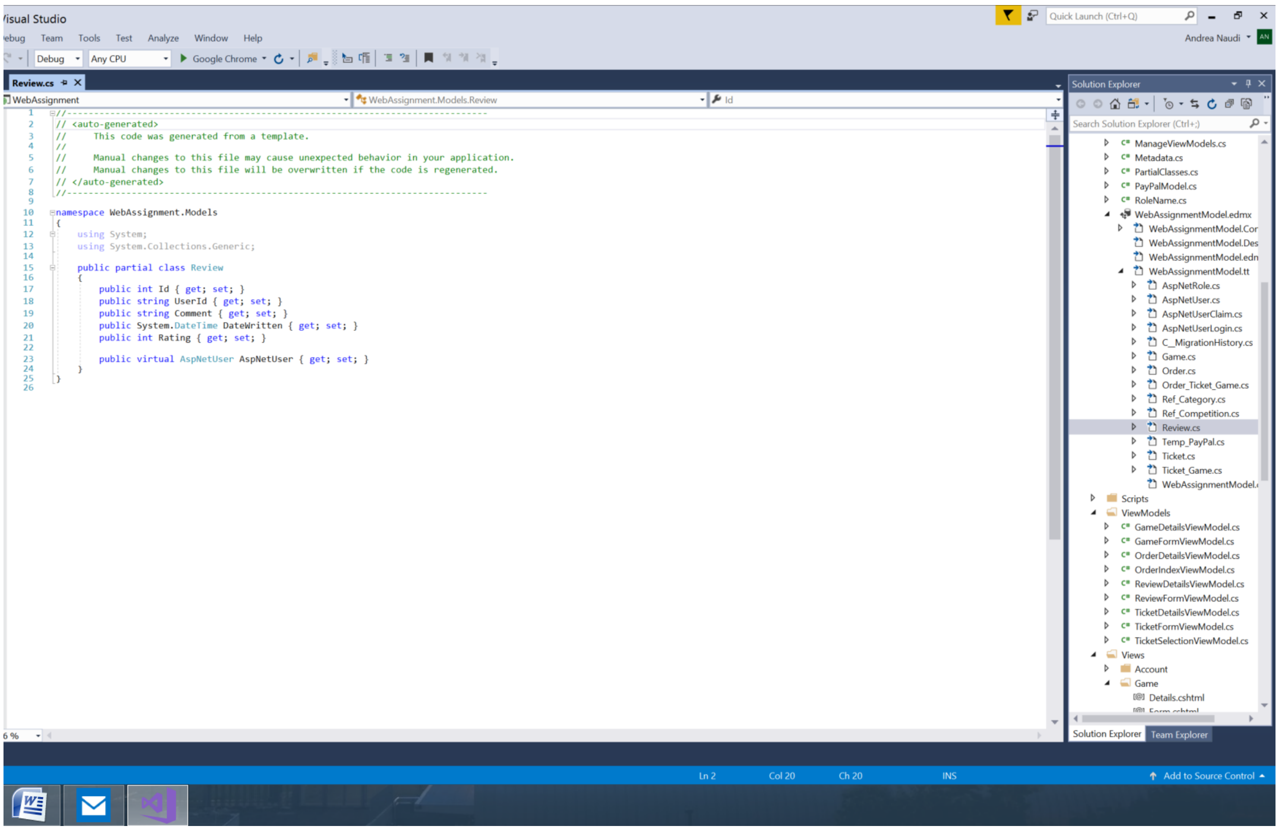
The image above represents the actual schema of the database. It shows the links between entities through foreign keys and tables. Note that the AspNet Identity tables are not represented because they are created by the framework itself. Entities would be linked to the AspNetUsers table with a foreign key 'UserId', an example being the Review table. The Temp_PayPal table will be explained further on. Linking tables are used for many to many relationships. Such table contains a foreign key to the tables it is linking. One to many (example Game - Ref_Competition, One competition has many games) relationships built using a foreign key in one of the tables.

Models

With a DB first approach the models are generated from the database, meaning that the models being created are:

- Game
- Order
- Ticket
- Ticket_Game
- Order_Ticket_Game
- Review
- Ref_Competition
- Ref_Category
- Temp_PayPal
- All the Identity models created by Asp.Net MVC.

The Review model as generated by Asp can be seen below:



```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated from a template.
//
//     Manual changes to this file may cause unexpected behavior in your application.
//     Manual changes to this file will be overwritten if the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace WebAssignment.Models
{
    using System;
    using System.Collections.Generic;

    public partial class Review
    {
        public int Id { get; set; }
        public string UserId { get; set; }
        public string Comment { get; set; }
        public System.DateTime DateWritten { get; set; }
        public int Rating { get; set; }

        public virtual AspNetUser AspNetUser { get; set; }
    }
}
```

Controllers

Controllers being used are:

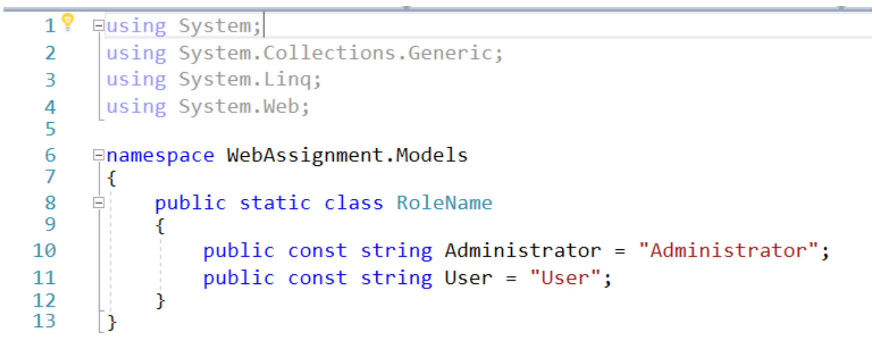
- TicketController - CRUD operations on tickets. Also contains a function which enables the admin to update prices.
- GameController - CRUD operations on games. Also contains a function which enables the admin to update prices.
- OrderController - CR operations on orders. Manages game and ticket selection, payment processing after a redirect from PayPal, and allow the admin to mark tickets as 'configured' (a process which needs to be done by the admin from the team's website).
- PayPalController - handles all the actions related to PayPal such as Notify, Cancel and Redirect.
- ReviewController - handles CRUD operations on reviews. Note that reviews are only removable/editable by the user who wrote it and by an admin.

Create actions (Ticket, Game, Review) : A new empty View Model object is created and passed to a view. The form creates a request to the Save action with the updated View Model object. A new entity object is created and the fields of the View Model are copied into the entity object which is then saved to the database.

Edit actions (Ticket, Game, Review) : The entity is retrieved from the database (according to the Id parameter). A View Model object is created, and the fields are copied from the entity object. The View Model object is passed to the form and the new data is sent to the Save action. In the Save action the old entity is loaded from the database (by Id of View Model object) and its data is updated to match the View Model object.

To create an order, the New() action loads all the games into the 'GameSelection'. On selecting a game, the gameId is passed to the TicketSelection action. The available tickets to this game are then loaded into another the 'TicketSelection' view. From this view, the selected tickets are posted to the Form() action. This action validates these Ids and loads the tickets into the 'Form' view. In this view, the user must enter appropriate details and continue to PayPal. Upon redirect from PayPal, an Order entity is created together with the appropriate linking tables. These are then stored in the database.

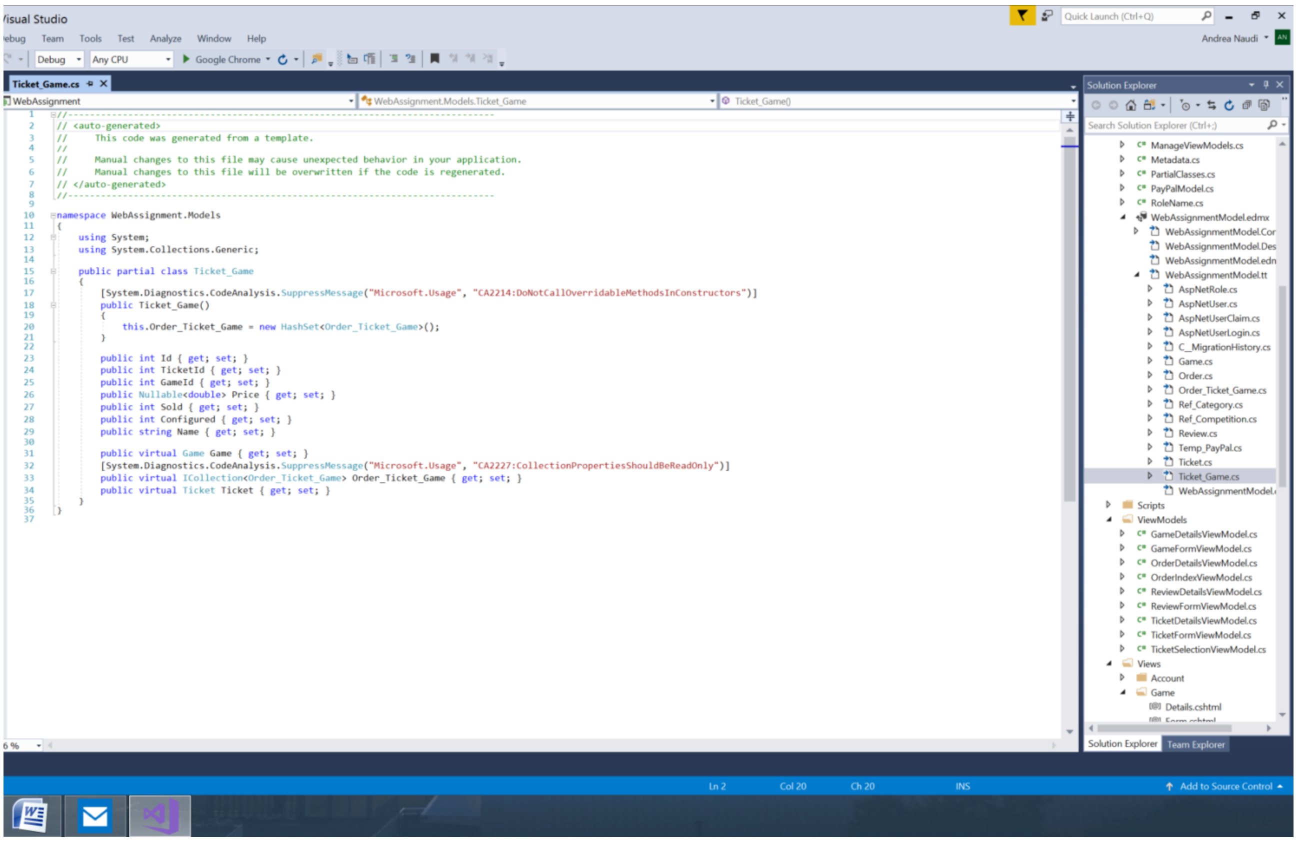
All actions in the controllers are given a custom name that makes sense (with respect to the action to be performed). If some actions can be only performed by the admin, the controller or particular action is decorated with a data annotation '[Authorize(Role="Administrator")]. In order to avoid magic strings, I made use of a class RoleName which has a property string for every role. Therefore I would use [Authorize(Role=RoleName.Administrator)]. Actions are also decorated with [HttpPost] or [HttpGet], depending on the type of request needed.



```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace WebAssignment.Models
{
    public static class RoleName
    {
        public const string Administrator = "Administrator";
        public const string User = "User";
    }
}
```

Linking Tables

In order to assign a ticket to many games or vice versa, the admin selects these games from a multiselect picker in the Ticket form (or Game form if vice-versa). A record is placed into the Ticket_Game table, which also stores information such as the price of the particular Ticket_Game (the same ticket can have a different price for different games), an indication of whether the ticket is sold for the particular game, and the name of the person using that ticket (must be entered after selecting tickets to order). Similarly, an order is linked to Ticket_Game instances through the Order_Ticket_Game table.



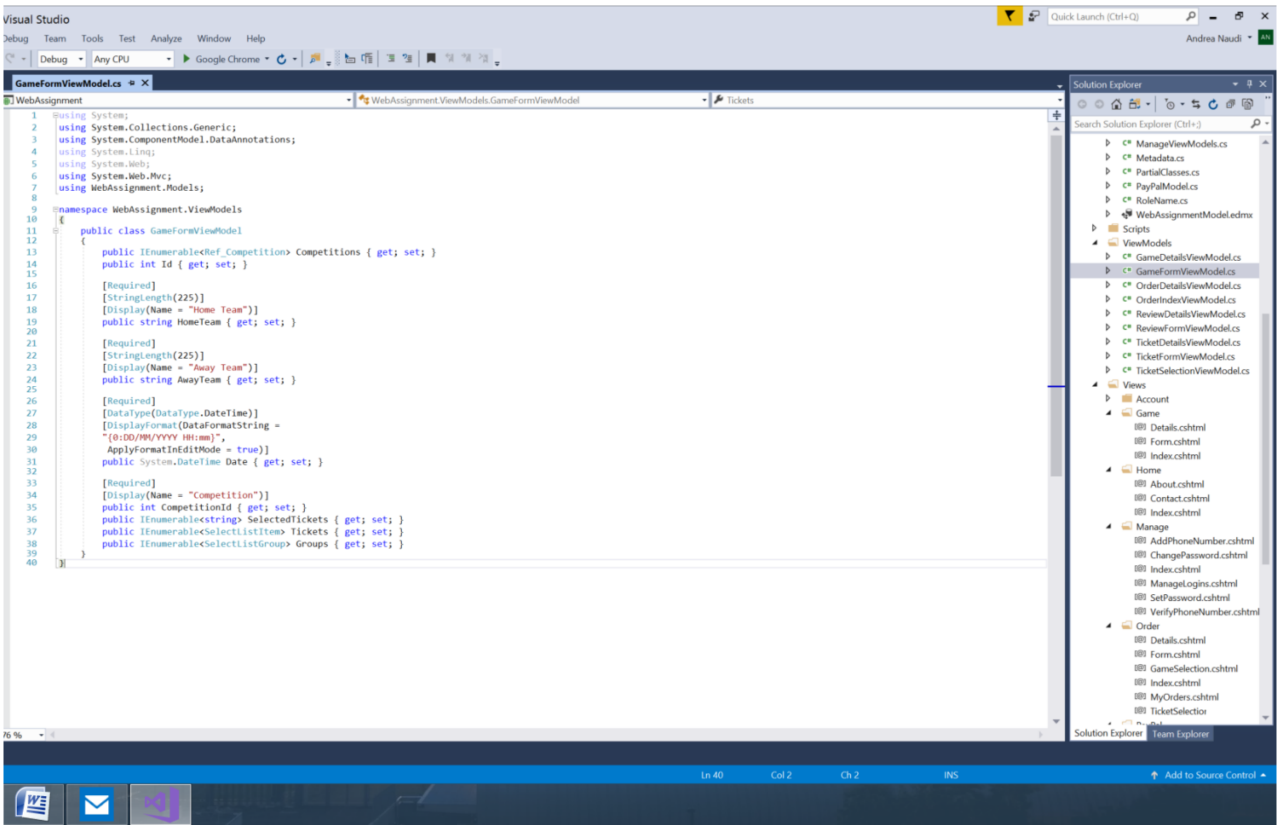
```csharp
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated from a template.
//
//     Manual changes to this file may cause unexpected behavior in your application.
//     Manual changes to this file will be overwritten if the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace WebAssignment.Models
{
    using System;
    using System.Collections.Generic;
    
    public partial class Ticket_Game
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Ticket_Game()
        {
            this.Order_Ticket_Game = new HashSet<Order_Ticket_Game>();
        }
    
        public int Id { get; set; }
        public int TicketId { get; set; }
        public int GameId { get; set; }
        public Nullable<double> Price { get; set; }
        public int Sold { get; set; }
        public int Configured { get; set; }
        public string Name { get; set; }
    
        public virtual Game Game { get; set; }
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Order_Ticket_Game> Order_Ticket_Game { get; set; }
        public virtual Ticket Ticket { get; set; }
    }
}
```

View Models

This web app makes use of various view models such as:

- GameFormViewModel
- GameDetailsViewModel
- Similar for other entities (tickets, orders and reviews)

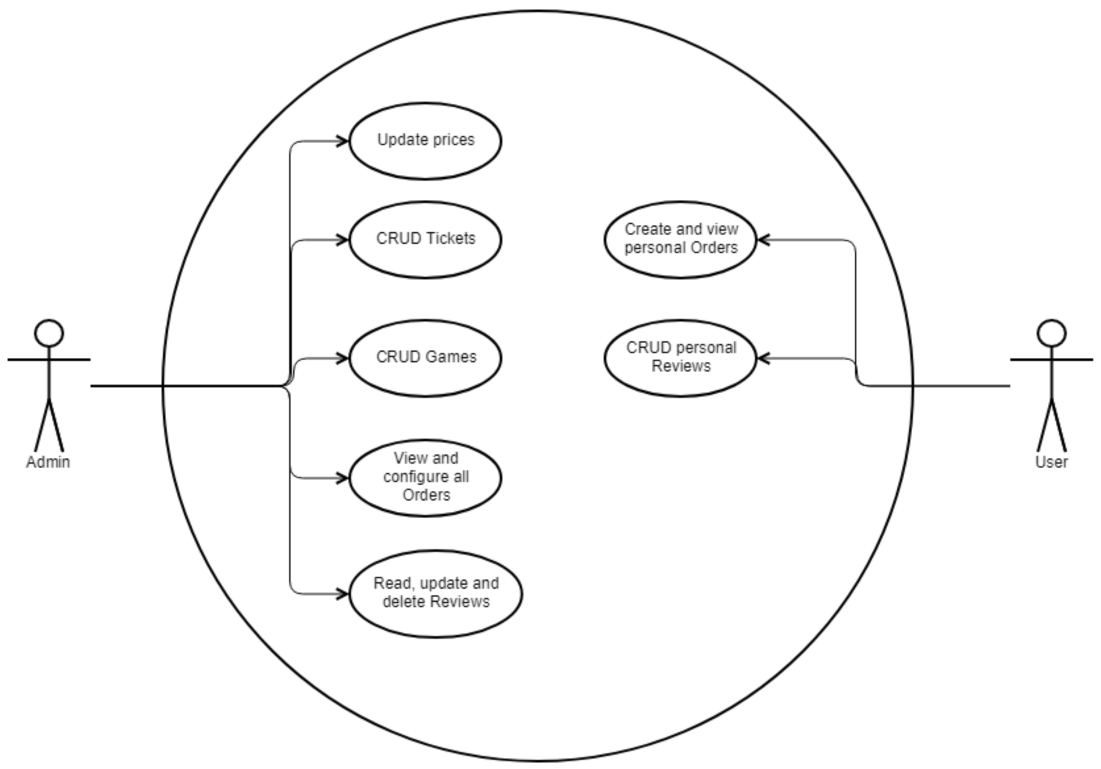
An GameFormViewModel is created when the admin selects 'new game' or 'edit game'. This is then passed to the Form view, its data is posted on submit and a Game object is created or modified to be stored in the database. The GameFormViewModel contains properties which are then mapped to the Game entity's ones. In order to validate the data inputted by the user, data annotations are used. These are used to specify certain rules on the particular field, such as 'required'. Prior to saving, these are checked and if one of these rules is broken the form is returned to the user. Frontend validation is also present by jQuery. On the contrary GameDetailsViewModel does not contain data annotations, since this is only being used to show data to the user. The view model acts as an extra layer between the user and the database. The other view models work in an identical manner. The GameFormViewModel can be seen below:



```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using WebAssignment.Models;

namespace WebAssignment.ViewModels
{
    public class GameFormViewModel
    {
        public IEnumerable<Ref_Competition> Competitions { get; set; }
        public int Id { get; set; }

        [Required]
        [StringLength(225)]
        [Display(Name = "Home Team")]
        public string HomeTeam { get; set; }

        [Required]
        [StringLength(225)]
        [Display(Name = "Away Team")]
        public string AwayTeam { get; set; }

        [Required]
        [DataType(DataType.DateTime)]
        [DisplayFormat(DataFormatString =
        "{0:DD/MM/YYYY HH:mm}",
         ApplyFormatInEditMode = true)]
        public System.DateTime Date { get; set; }

        [Required]
        [Display(Name = "Competition")]
        public int CompetitionId { get; set; }
        public IEnumerable<string> SelectedTickets { get; set; }
        public IEnumerable<SelectListItem> Tickets { get; set; }
        public IEnumerable<SelectListGroup> Groups { get; set; }
    }
}
```

Roles

The roles used are two simple ones, the Administrator and the User. The Use Case diagram below explains the permissions given to each role:


Update prices
CRUD Tickets
CRUD Games
View and configure all Orders
Read, update and delete Reviews
Create and view personal Orders
CRUD personal Reviews
Admin
User

An administrator can perform CRUD operations on Games and Tickets, and update prices. Initially prices are set to defaults, according to category. The admin can also view and configure orders. Finally, reading, updating and deleting reviews are also possible. By default an account is given the 'user' role. The admin role may be added via the database.

The users can create and view their Orders and perform CRUD on reviews. Update and delete are only possible if the review was written by the user in question.

Different Layouts

The layout of the application varies depending on the user logged. The only difference being the available options in the navigation bar and its lists. For example, a user does not have the 'Ticket' option available as permission is not granted. This option is available to the admin, who can create a new ticket or manage existing ones. The 'Games' option is identical to the 'Tickets'. The 'Orders' section is available to both roles however the options of both differ. The admin can jump straight to the index page and the user can see 'My Orders' and 'New Order'. The other options in the navigation bar are identical for both roles.

PayPal 3rd party API

Reference: http://logcorner.com/asp-net-mvc-paypal-integration/

On clicking 'Checkout with PayPal' the controller creates a PayPal model with the necessary details which is then automatically submitted to PayPal (Sandbox in this case). Since no data is saved until redirect from PayPal, a Temp_PayPal object is created to store the Ticket_Game object Ids while the user is in the PayPal website. After payment and redirect, the order is saved in the database. Credentials of a sandbox account to test this feature may be found in the ***paypal credentials.txt***.

Frontend external widgets

- Bootstrap Multiselect picker with live search: https://silviomoreto.github.io/bootstrap-select/examples/
- Bootstrap datepicker: https://github.com/Eonasdan/bootstrap-datetimepicker

Both these tools are integrated with the app through Javascript and Bootstrap. Various Different keys/options can be used to enable different settings to these pickers. The biggest challenge found with the multiselect picker was populating it with options and saving to the database. To populate it, a List of type SelectItem is used. A foreach loop is then used to put each item in the picker. Furthermore a list of type String is used for the picker to save the selected choices. On the backend, this string of values (ids in my case) is iterated and data is stored in the database accordingly.

Security Aspects

Aspects used include:

- RequireHttpsAttribute: ensures that the site is only accessible from a secure url
- AuthorizeAttribute: ensures that all the pages on the site require the user to be logged in the system to be viewable.
- [Authorize(Role = @@@)] in various controllers to ensure that an action is only accessible from users with the particular role.
- Anti-forgery token: to prevent Cross-Site Request Forgery (CSRF) attacks
- Checks on the backend to prevent hacks from DOM changes, an example being in the OrderController Form Action() line 115 [Ref 1]
- [HttpPost] and [HttpGet]
- 2 factor authentication and email verification.

Cross-browser Testing

The application was developed/tested using Google Chrome and Firefox. Both seem to be functioning well. The different functions and pages were executed/visited from both browsers.

Requirement Coverage

- Registration and external registration (Facebook).
- Email verification on register.
- Optional phone two factor authentication.
- Automatic assignment of 'User' role. Admin role done from database.
- CRUD operations on entities.
- Linking of two main entities, Tickets and Games.
- Reviews may only be deleted/edited by and admin or the user who created it. This concept does not make much sense but was done for this specific requirement; as the Review is the only entity accessible by both roles apart from the Orders.
- View Models are being used.
- Forms and correct components for the data to be input like numbers and dates. Validation across both server and client.
- Two layouts are being used.
- Custom action names in controllers allow elegant urls.
- 3rd party api - PayPal.
- Use of Authorize annotation.
- Use of HttpGet and HttpPost annotations.

Challenges Faced

The biggest challenge I faced during this assignment was connecting my application to the PayPal website. For some reason all documentation I found was not totally correct as they passed data to PayPal via the web.config file. Eventually I tried doing this from the controller and everything worked well. Furthermore, the 2 frontend widgets gave me some trouble when it came to connect them. Customising them with their different keys was not too difficult once integrated.

Another challenge I faced was the phone two factor authentication. Apparently the Maltese number they give you does not have SMS functionality, so I had to opt for a foreign number instead. For some unknown reason, sometimes Twilio takes long to send a message or doesn't send one at all. In these rare cases one is required to send a new request for a new code. Facebook login also gave me some trouble due to an update to a package and dependencies. Logging in worked well once the dependent package was downgraded.