Contents

[Technical Solution 2](#_Toc509176295)

[Overview 2](#_Toc509176296)

[Server 3](#_Toc509176297)

[Client 5](#_Toc509176298)

[Generic UI 5](#_Toc509176299)

[Customization 6](#_Toc509176300)

[Database 6](#_Toc509176301)

[Entity Framework 6](#_Toc509176302)

[Manual Database Mapping 7](#_Toc509176303)

[Database Customization 8](#_Toc509176304)

[Initializer 9](#_Toc509176305)

[Model 11](#_Toc509176306)

[Objects 11](#_Toc509176307)

[Properties 14](#_Toc509176308)

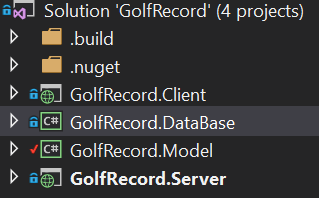
[Services 19](#_Toc509176309)

[Programing Techniques 21](#_Toc509176310)

[Specifics 21](#_Toc509176311)

# Technical Solution

## Overview



The screen shot above shows the layout of my solution. Since I am using Naked Objects to aid me in the project I have been given a template which consists of 4 projects; client, database, model, server. Each of these projects will be discussed below. The use of the template allows for separation of concerns, this means I can separate the code into their distinct objectives for example keeping all the code that affects the database separate from all the code that affects the client. For future this means that I could create a completely new client for my project without having to edit the rest. This is shown by the diagram below of the architecture for how my project would be deployed in future. At that this instance the project is all being run on one machine including the server and the client but for the future a separate machine would be used to run the server.

Runs on a Server machine

Database

Model

Client

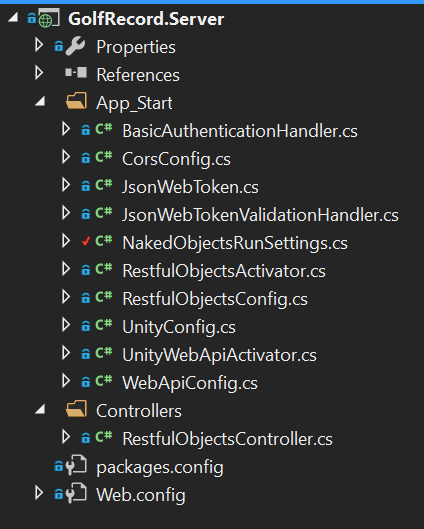
Server Project

The diagram shows the client is separate to the content that is run on the server machine this is because the client is generic in the web browser therefore can be run on a computer or a smartphone which was a request from the customer for the project. The dashed line shows how the RESTful API is used to communicate between the client and the server. The other arrows show the dependency of each project, i.e. the Database is dependent upon the model.

The use of naked objects also gives me various other code and files to create a more advanced project given the time constraints. This allows focus on the functionality and the business code rather than spending a larger proportion of my time on the set up and defining of the interface, which although is important would take a considerable length of time.

## Server

The server takes all the logic from the model and the database and uses the logic to create a RESTful API for the client to communicate with the server using the HTTP request, response techniques. I have not created the server project, but it has been supplied to me by Naked Objects. As mentioned in the Overview the server would eventually become a separate machine that the clients phone or laptop would access.



I have made configurations to the server within the NakedObjectsRunSettings class. The screen shot of the code is all the changes that I have personally made:

private static Type[] Types

{

get

{

return new Type[] { manually enter subclasses since the framework cant discover the objects.( put down the bottom)

typeof(MatchStableFord),

typeof(MatchStrokePlay),

Here I had to declare all the object types that the user can discover so the server knows what to search for when creating the menu.

typeof(MatchPlay),

typeof(FourPlayerHoleScore),

typeof(TwoPlayerHoleScore),

typeof(ClubManager),

typeof(Player)

};

}

}

private static Type[] Services

{

get

{

The services are also defined here so that the server doesn’t confuse the services as objects to be added to the menu. The services will be mentioned at a later stage within the model in more depth.

return new Type[] {

typeof(GolferServices),

typeof(MatchServices),

typeof(CourseServices),

typeof(HoleServices)

};

}

}

Here I am defining the main menu that is visible at the homepage. The reason for using the services since the homepage isn’t an object these are the methods that we can navigate with and retrieve or create domain objects.

public static IMenu[] MainMenus(IMenuFactory factory)

{

return new IMenu[] {

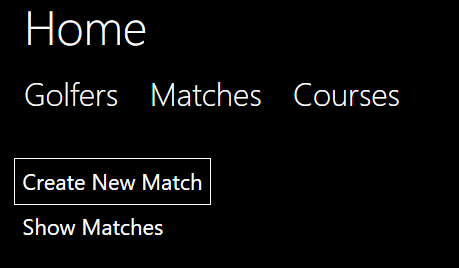
factory.NewMenu<GolferServices>(true, "Golfers"),

factory.NewMenu<MatchServices>(true,"Matches"),

factory.NewMenu<CourseServices>(true,"Courses")

};

}



This screen shot is the product of the code above. I have expanded the Matches showing all the methods in the MatchServices.

public static IAuthorizationConfiguration AuthorizationConfig()

{

var config = new AuthorizationConfiguration<DefaultAuthorizer>();

// config.AddNamespaceAuthorizer<MyAppAuthorizer>("MyApp");

// config.AddNamespaceAuthorizer<MyCluster1Authorizer>("MyApp.MyCluster1");

config.AddTypeAuthorizer<MatchStrokePlay, StrokePlayAuthoriser>();

config.AddTypeAuthorizer<MatchStableFord, StableFordAuthoriser>();

config.AddTypeAuthorizer<MatchPlay, MatchPlayAuthoriser>();

config.AddTypeAuthorizer<Player, PlayerAuthoriser>();

config.AddTypeAuthorizer<ClubManager, ClubManagerAuthoriser>();

config.AddTypeAuthorizer<Hole, HoleAuthorier>();

config.AddTypeAuthorizer<Group, GroupAuthoriser>();

config.AddTypeAuthorizer<Match, MatchAuthoriser>();

return config;

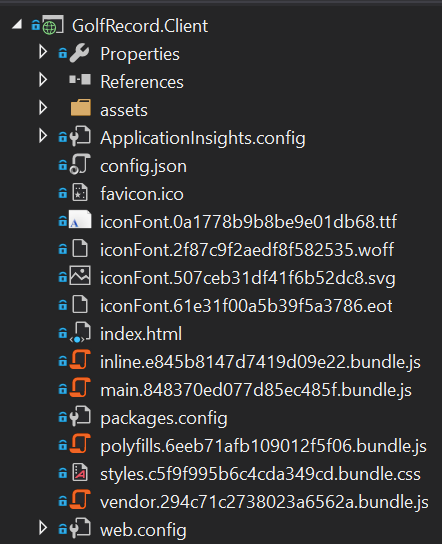
}

Finally, I have had to declare all the Authorizers so that the server doesn’t think they are objects than our accessed by the user. Authorizers will be further discussed in the model.

}

}

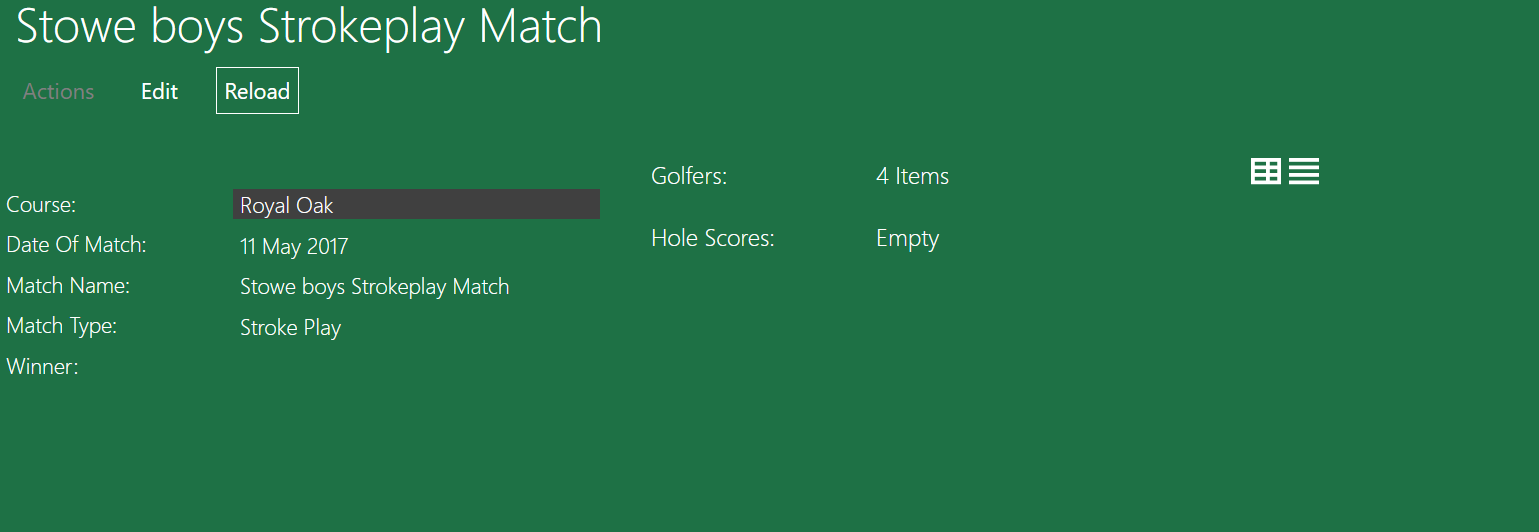
## Client



### Generic UI

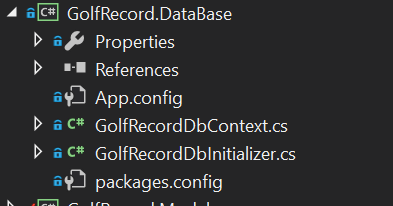
The client is a generic user interface that has been supplied by Naked Objects. There are thousands of lines of code which vary in language. For example some of the classes are written in HTML like the index.html, other classes are written in css. One of the benifical factors of using the NakedObjects generic UI is when executing an action that takes in an object, the user may drag and drop an object in. Here is a screen shot of the user dragging an object into a parameter.

### Customization

I have made a few changes to code all of which are in the config.jason file here I have told the solution I am using authentication and an authorizer this is the demo version of auth0. I will talk about auth0 at a later stage in the Specifics section. In addition, in the config.jason file I have chosen to change a few colors of objects in order to help the user distinguish what they are looking at. An example of this is for the match object. 

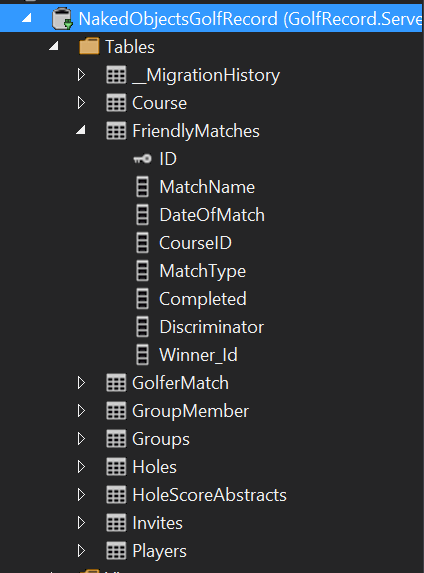
As you can see the Match is green, but the course is black this helps the user to know that from match they can travel to the course by clicking that link which now stands out.

## Database



### Entity Framework

The database is working off an object relational mapping framework made by Microsoft called entity framework. This synergizes well with my project as it automatically sets up the database through the code I have written. where each object is a separate table and an Icollection is a 1 to many relationships. The list of tables is shown below.



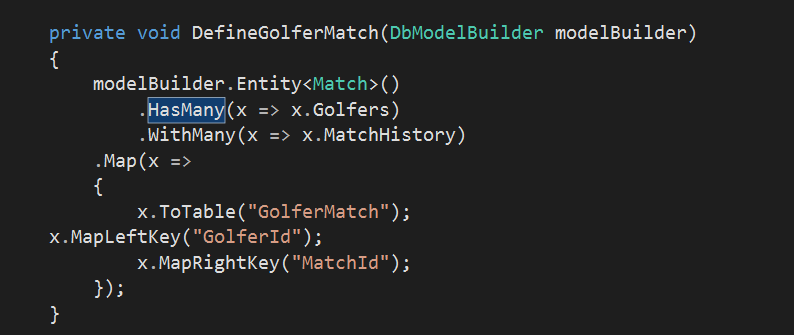
Here I have expanded the Matches table to show all the columns. The key next to the ID represents the primary key. I have customized the table here to called it “Friendly Matches” How I have done this will be mentioned Database Customization but the reason I have done this is because in future this system could be used for completive matches with separate rules.

There are a few tables which I have manually mapped which will be talked about in manual mapping section below. The use of Entity Framework means that I can use SQL reports to obtain certain information this is useful for analyzing the data collected and for a user to check the data for their past matches.

### Manual Database Mapping

When there is a problem where the entity framework hasn’t quite managed to map the database correctly I have had to manually do it. This is done by creating a function in the DBContext class which defines a new table with a primary key using modelbuilder this function is then called in a separate function called OnModelCreater

I have had to manually map a table twice; one is a many to many relationship f where many Golfers have many match histories, but a match can belong to many golfers. To solve this, I have defined a link table called Golfer match which can be seen in the screenshot above. This new table is a link table used to turn the many to many into two one to many relationships. In the new table golfer match it uses a compound foreign key as the primary key which are GolferID and MatchId. The code below shows how I have done this. The code is written within the DBcontext and is being called using a method from the modelbuilder function which can be seen below the screen shot.



Here I am stating that the many to many relationship that currently exists. Is this a linq statement?

Here I am defining the new tables name and what the compound foreign primary key is.

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

Only the bottom two are creating a new table. The other 3 methods are for customization which will be mentioned in Database Customization.

DefineGolfer(modelBuilder.Entity<Golfer>());

DefineMatch(modelBuilder.Entity<Match>());

DefineCourse(modelBuilder.Entity<Course>());

DefineGolferMatch(modelBuilder);

DefineGroupMember(modelBuilder);

}

The second example of manual mapping is a one to itself relationship. This is for the GolferFriends. Where a Golfer may have many golfers as friends. Once again to solve this problem I have created a new table called GroupMember which uses a new object that has been created in the model called Groups. A group has a golfer who is the owner and a collection of Golfers who belong to the group.

private void DefineGroupMember(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<Golfer>()

Group is an entirely new object that I created from with the Model which can be seen In the Model. I created the new object so that I can have two different primary keys and therefor it can be mapped correctly.

.HasMany(x => x.Groups)

.WithMany(x => x.Members)

.Map(x =>

{

x.ToTable("GroupMember");

x.MapLeftKey("Id");

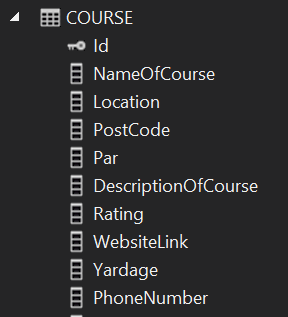
x.MapRightKey("GolferId");

});

}

### Database Customization

In addition to manual mapping I have also been able to customize the database where I feel a better name could be used. An example of this can be seen below.



Here is the table for course. I have renamed a few coloumns in the table so that they read better to the user. DescriptionOfCourse is actually CourseDescription this does change the property when the code is running its only for the database.

private void DefineCourse(EntityTypeConfiguration<Course> courseconfiguration )

{

courseconfiguration.ToTable("Course");

courseconfiguration.HasKey(c => c.Id);

courseconfiguration.Property(c => c.Address).HasColumnName("PostalAddress");

courseconfiguration.Property(c => c.CourseDescription).HasColumnName("DescriptionOfCourse");

courseconfiguration.Property(c => c.CourseName).HasColumnName("NameOfCourse");

}

### Initializer

For creating example data used for test runs I am using an Initializer. It is a separate class within the Database project. The class consists or a group of methods that I have created returning an object which are saved to the Context. These methods are then accessed above with sample data being passed in for each parameter required. There are various other techniques which could be used instead for example SQL Inserts but I chose to use an Initializer as it reads better with its syntactic sugar of “=”. This means that there are already golfers and matches inside the program before I run it so to do a test run I don’t need to create 4 separate golfers then add the 4 golfers to the match and then add 18 scores for each golfer. This becomes increasing important for when I’m doing my test. To add an example data, I need to create a help method which returns that object. This method is then called which each value declared and then saved to the database. Changes currently made while the project is running doesn’t get saved but for future when the application goes live this would be changed so that people matches get saved.

The code below shows an example of a method within the DBInitializer;

I have set the Initializer up so that if the user doesn’t enter a match when calling the method is automatically set the type to strokeplay since this is the most common.

private Match AddNewMatch(string name, DateTime date, int courseID, MatchType matchType = MatchType.StrokePlay)

{// work for each match type

Here I am using a switch case statement to ensure the match is one of the three types and therefore doesn’t break.

Match m = null;

switch (matchType)

{

case MatchType.StrokePlay:

m = new MatchStrokePlay() { MatchName = name, DateOfMatch = date, CourseID = courseID, MatchType = matchType };

break;

case MatchType.MatchPlay:

m = new MatchPlay() { MatchName = name, DateOfMatch = date, CourseID = courseID, MatchType = matchType };

break;

case MatchType.StableFord:

m = new MatchStableFord() { MatchName = name, DateOfMatch = date, CourseID = courseID, MatchType = matchType };

break;

default:

break;

}

Context.Matches.Add(m);

Context.SaveChanges();

return (m);

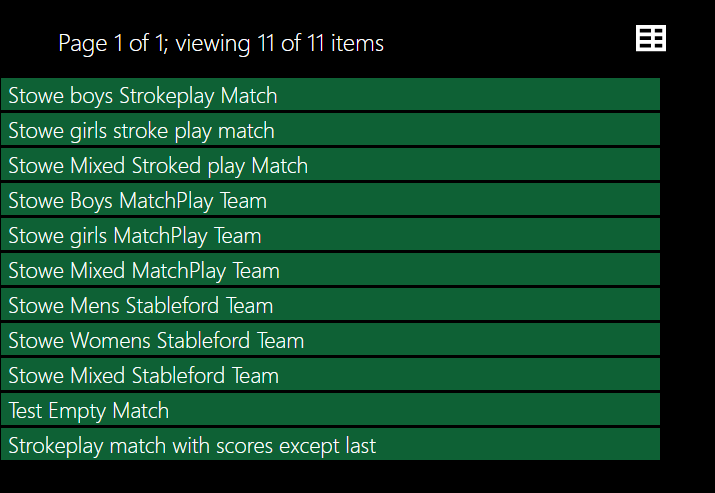
}

An example of how this method would be called is shown below. Once again at the end of the method I am saving the changes to the context. It doesn’t need to be at the end of each call but I have done it to be safe.

var s9 = AddNewMatch("Stowe Mixed Stableford Team", date1, 4, MatchType.StableFord);

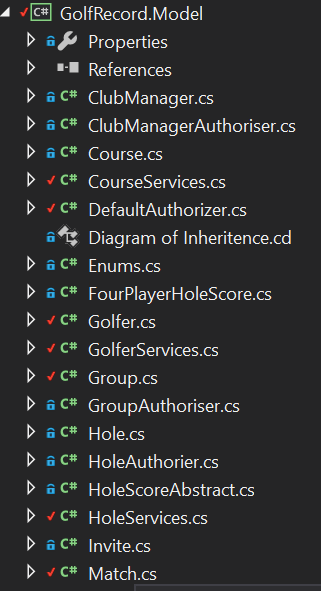
context.SaveChanges();

The screen shot below shows all the example matches I have created for testing. There is; an all-male, all-female and mixed gender match for each match type. Some of these matches have scores already added in but some don’t.



## Model

The model is the most important part of my code as it is where I have written all the business logic and functional application. The model is made up of 3 types of classes. Each of the types of classes will be discussed below. Within the Model I have defined all of my objects, their services and there authorizers where necessary. A lot of my program evolves around the authorization since it is live multiuser application and therefore many players or managers will have access to different properties and actions. These will also be discussed below. The screen shot below shows the classes within the Model project.



### Objects

An object is a computing construct used to represent a either a physical object or a more abstract object for example a holescore. An object consists of properties and actions. The code for an Object can be seen below.

public class Golfer

At the top I have declared an Object Container this is so that I can access certain actions that are provided by Naked Objects where an object is not provided.

{

public IDomainObjectContainer Container { set; protected get; }

public GolferServices GolferConfig { set; protected get; }

public CourseServices CourseConfig { set; protected get; }

public MatchServices MatchConfig { set; protected get; }

Here I am importing the methods from other classes called services.

[NakedObjectsIgnore]

public virtual int Id { get; set; }

[Title][MemberOrder(1)]

public virtual string FullName { get; set; }

[MemberOrder(2)]

public virtual int Handicap { get; set; }

Each of these variables are properties that belong to the object. Some of the properties have words in [] these are attributes and will be discussed down below under attributes. A screen shot showing how the interface portrays these properties is under the code. Properties will be discussed below under

[Optionally][MemberOrder(4)]

public virtual string Mobile { get; set; }

[Optionally]

public virtual Gender Gender { get; set; }

public virtual Title Position { get; set; }

public virtual string Username { get; set; }

#region Friends (collection)

private ICollection<Golfer> \_Friends = new List<Golfer>();

public virtual ICollection<Golfer> Friends

{

get

An Icollection is Naked Objects version of a group of objects all under the same object type. Here is a group of Golfers named friends. Each Icollection has a method to add a value to the object this is the method below

{

return \_Friends;

}

set

{

\_Friends = value;

}

}

public void AddFriend(Golfer golfer)

{

Friends.Add(golfer);

}

[PageSize(3)]

public IQueryable<Golfer> AutoComplete0AddFriend([MinLength(2)] string matching)

{

return GolferConfig.AllGolfers().Where(g => g.FullName.Contains(matching));

}

#endregion

This shows a complementary method. Complementary methods will be discussed below under Complementary methods.

#region MatchHistory (collection)

private ICollection<Match> \_MatchHistory = new List<Match>();

public virtual ICollection<Match> MatchHistory

{

get

{

return \_MatchHistory;

}

set

{

\_MatchHistory = value;

}

}

public void AddMatchHistory(Match match)

{

MatchHistory.Add(match);

}

public IQueryable<Match> AutoComplete0AddMatchHistory([MinLength(2)] string matching)

{

return MatchConfig.ShowMatches().Where(m => m.MatchName.Contains(matching));

}

#endregion

#region Groups

public Group CreateNewGroup()

{

var group = Container.NewTransientInstance<Group>();

group.GroupOwner = GolferConfig.Me();

return group;

}

private ICollection<Group> \_Groups = new List<Group>();

public virtual ICollection<Group> Groups

{

get

{

return \_Groups;

}

set

{

\_Groups = value;

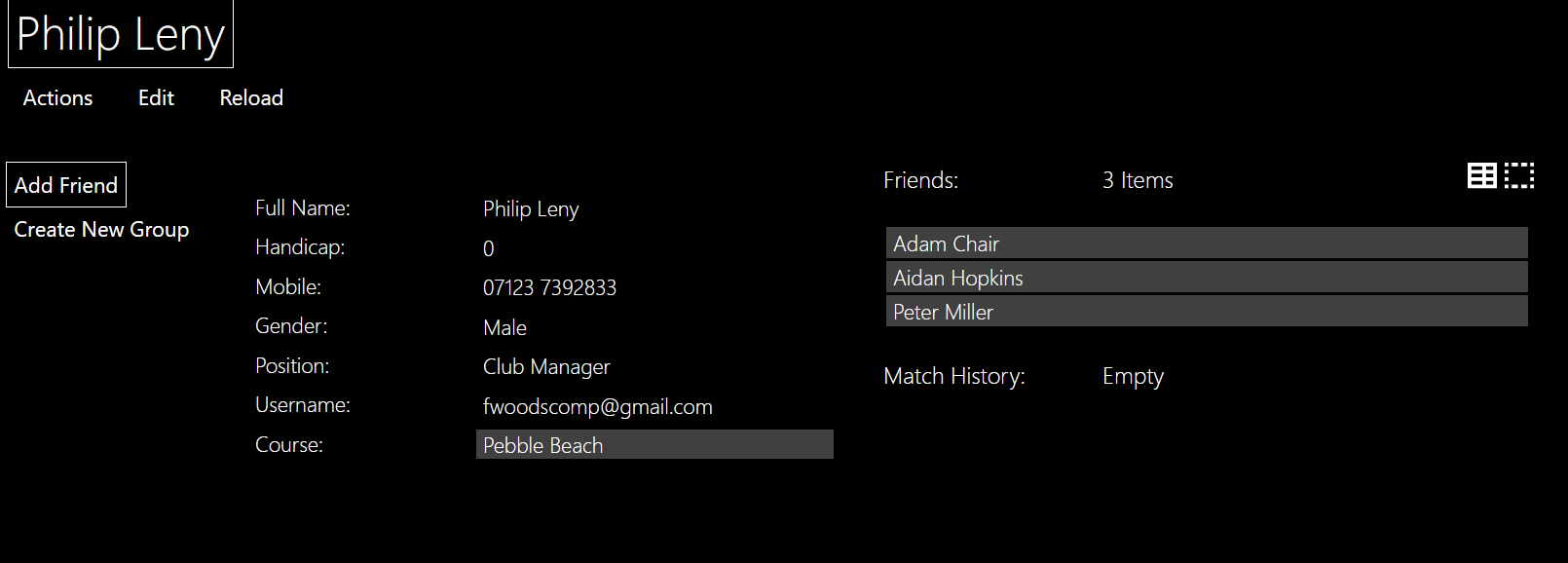
}

}

#endregion

}

}

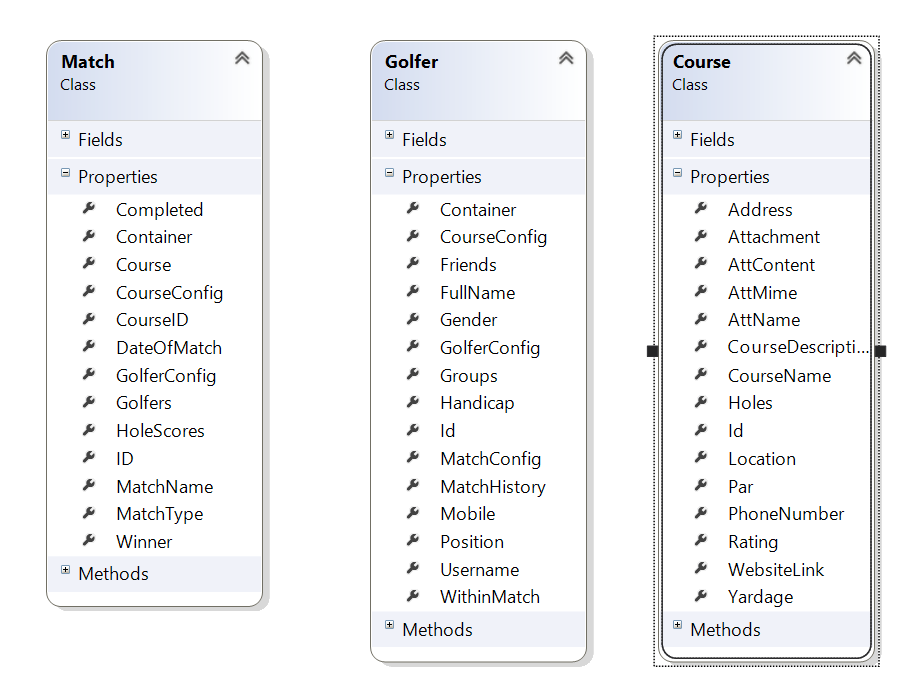


Here are the Icollections I have exapanded the collection of friends.

Since the propetie Course is decalered as the object Course is can be used as a link for the user to access the course.

### Properties

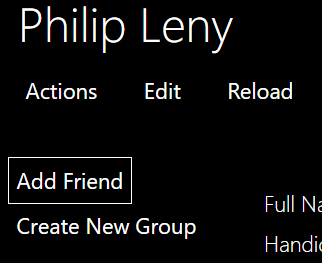
As mentioned above properties are the values assigned to an object. Some properties are hidden using the [NakedObjectsIgnore] attribute. Attributes will be mentioned below. With Inheritance If an object inherits from another object for example; player inherits from golfer this means that the properties declared in the super class, Golfer are accessible from player but not the other way around.



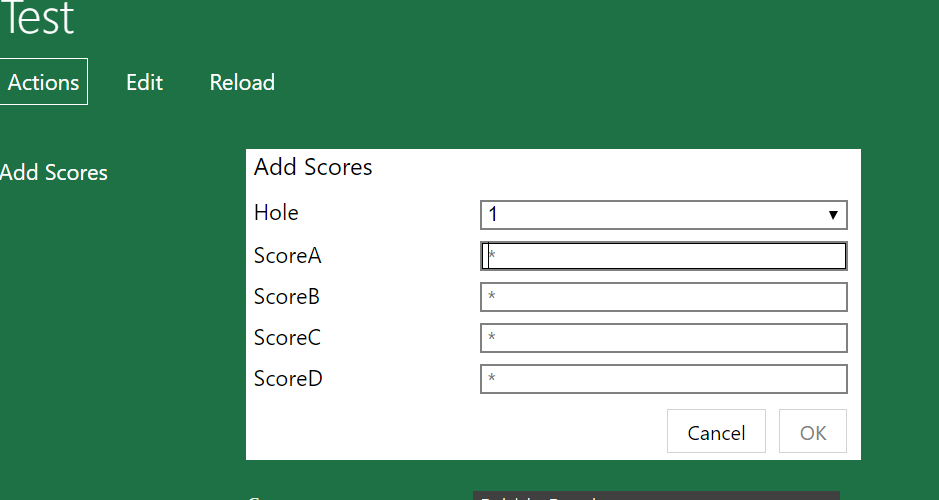
The 4 properties, “Attachment, AttContent,AttMime, AttName:” are used to portrayal photo.

This photo shows the properties of the 3 main objects. As you can see some objects have a lot of properties therefore it is important to use authorization and attributes. Both of which are discussed below.

#### Actions

Actions are methods defined within an object. They are accessed by the user from within the object by pressing the tab in the top left. By pressing this tab, the list of actions accessible the user is displayed. An example of this is seen below.

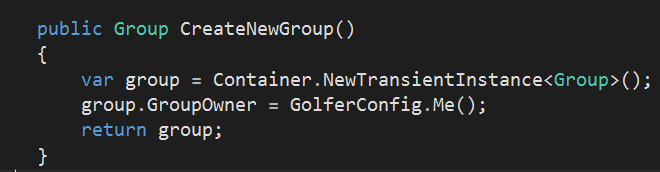
To prevent a user from seeing an action you can either use authorization or hide complementary method. Some actions take in parameters. Which using attributes can be optional otherwise Naked Objects will set to necessary by default. If a method requires another object as a parameter the Naked Objects will allow provide a drag and drop interface but the use of complementary methods will provide a user friendly technique of adding the object. All of these techniques are further discussed below. The screen shot below shows the user interface for entering the parameters for a action.



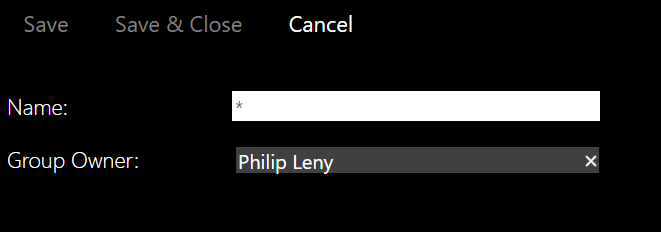
The asterixis within the boxes indicate that parameter is not optional. The Ok button is faded and cannot be accessed therefore preventing errors. The Hole parameter is a drop down of the number of holes which automatically increments each time the method is called using complementary methods.

Finally if a parameter is constant you can automatically provide the parameter by default. Since an object can either be transient or persisted. A value can be specified while the object is transient before it has been persisted therefore the value can be specified before the user can access it. (I think this is true but would like to check)

The code for this is shown below:

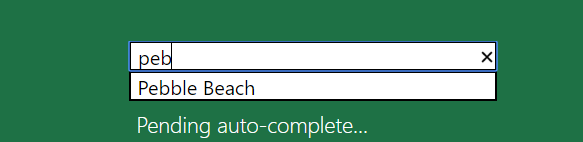


Since the variable, “group” is returned after the property group owner is specified therefore it is still in the transient stage. This is portrayed to the user in the following screen shot:



#### Complementary Methods

A complementary Method is a method which complements another method. An example of this is autocomplete. To make my program more user friendly I have added autocomplete to prevent the user having to type in the full name or search through pages of golfers. An example of this is seen below.



A complementary method is declared by using <ComplementaryMethod>0<methodtocomplement> An example of this is:

public IQueryable<Golfer> AutoComplete0AddFriend([MinLength(2)] string matching)

Naked Object specifies a select group of complementary methods. The use of 0 specifies the difference between the two methods. This means that all the complementary methods are hidden from the user.

Two more examples of helper methods are in the match class. These two methods are called Choices and Default they synchronize well together. Choices specifies the possible options. Default is a method for a automatic parameter.

The code showing these two complementary methods and the result interface.

public IList<Hole> Choices0AddScores()

{

This provides a drop down menu of all the possible holes

if (HoleScores.Count == 0)

{

return Course.Holes.ToList();

}

else

{

// return Course.Holes.ToList();

return (from h in Course.Holes

from s in HoleScores

where h.Id != s.HoleId //a querry across two sources.

select h).ToList();

}

This sets the value for hole to the last value + 1

}

public Hole Default0AddScores()

{

int nextHole = 1;

if (HoleScores.Count > 0)

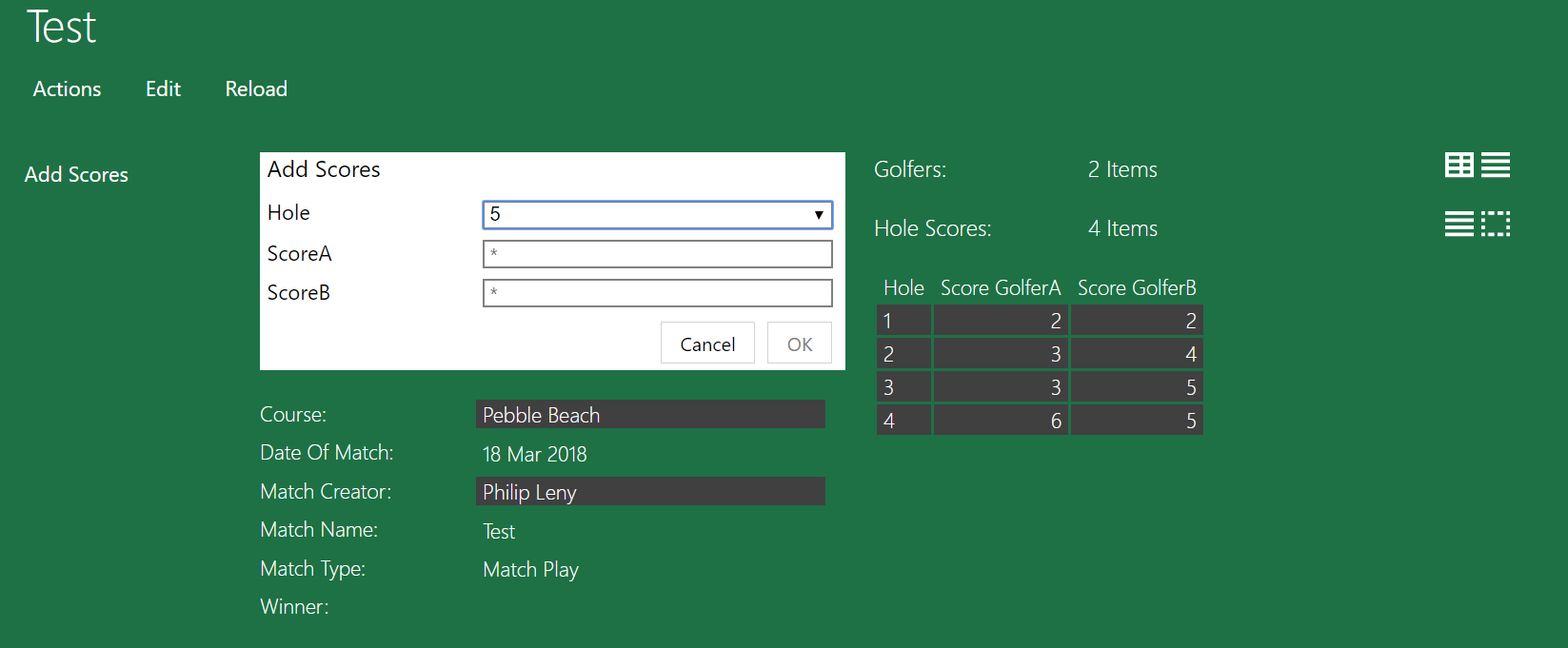
{

nextHole = HoleScores.Max(hs => hs.Hole.HoleNumber) + 1;

}

return Course.Holes.First(h => h.HoleNumber == nextHole);

}



As you can see there is 4 scores already added and the method has automatically set itself to 5. This is due to the default complementary method.

The choices method resulted in a drop down just like an enum would. This is very helpful since some courses require 9 holes and some 18 therefore 2 sets of enums would be required.

#### Attributes

Attributes are parameters used for the interface portrayed to the user. This can range from ordering properties, hiding properties or declaring a property as optional. There are many different attributes. Below is an example of: MemberOrder, Title, NakedObjectsIgnore, Pagesize and finally Optionally. These are all attributes that I have included in my program. The screen shot below shows most of these attributes. You can have more than one attribute per property. Attributes can also be applied to methods.

##### MemberOrder

MemberOrder is an attribute used to portray the properties to the user in a certain order.

As you can see from the screen shot the first property is full name the second is handicap and they are the most important mobile is next.

##### Title

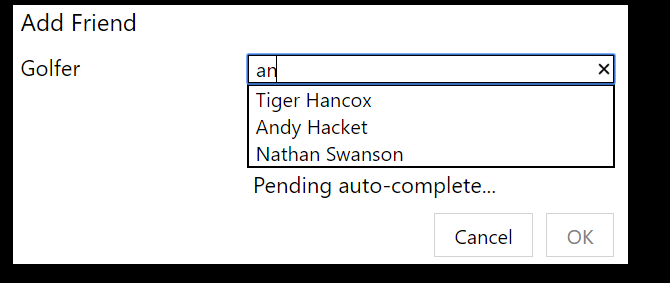
Title declares the title of a certain object. An example of this is the title of Golfer being the Golfer’s name. This can be seen in the screen shot as the title.

##### NakedObjectsIgnore

NakedObjectsIgnore is used to prevent the user from seeing a certain property an example of this is preventing the user from seeing the ID of a golfer. This is because the Id of the golfer provide no value to the golfer but is a necessary value for the program.

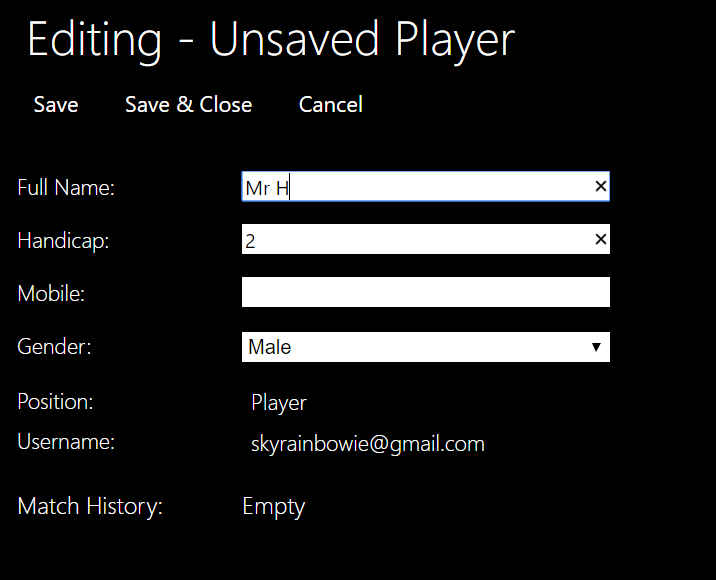
##### PageSize

PageSize is used in a complementary method of autocomplete. When the user in entering the name of the object they are searching for it will provide a list of 3 possible objects. This is not portrayed in the screen shot above but can be seen in the one below.



##### Optionally

This declares a property as optional therefore doesn’t need to filled in. An example of this is a golfer entering their mobile. Some users may not want to share such personal information to all the users therefore I have made it optional. When creating an object using a action or service method an optional parameter can be seen as having no astrix.



As you can see the Save and Save & Close actions are available to be used meaning I can create this golfer without entering a mobile.

### Services

Each of the main objects also have a services class this contains the helper methods that are used to access an object. When click the main header you are given a drop down of options. These options are the methods from within the services. Service methods are held outside the main class since they are methods that don’t require an action to access it. For example if you need to create a golfer but you don’t have a golfer to access it.

The code for a service is shown below:

namespace GolfRecord.Model

{

public class GolferServices

{

#region Injected Services

//An implementation of this interface is injected automatically by the framework

public IDomainObjectContainer Container { set; protected get; }

#endregion

Once again I am using the Container to access methods since I have no object.

public IQueryable<Golfer> AllGolfers()

{

return Container.Instances<Golfer>();

These methods return a type IQueryable this means if there is a lot of the that object it returns in page by page rather than all in one table like a ICollection

}

public IQueryable<ClubManager> AllManagers()

{

return Container.Instances<ClubManager>();

}

public Golfer Me()

{

var username = Container.Principal.Identity.Name;

var user = AllGolfers().Where(g => g.Username.ToUpper().Contains(username.ToUpper())).SingleOrDefault();

if (user == null)

{

user = Container.NewTransientInstance<Player>();

user.Username = Container.Principal.Identity.Name;

user.Position = Title.Player;

return user;

}

else

{

return user;

}

}

[NakedObjectsIgnore]

public bool IsPlayer()

{

if (Me() != null)

{

return true;

}

else

{

return false;

}

}

public IQueryable<Golfer> FindGolferByName(string name)

{

return AllGolfers().Where(c => c.FullName.ToUpper().Contains(name.ToUpper()));

}

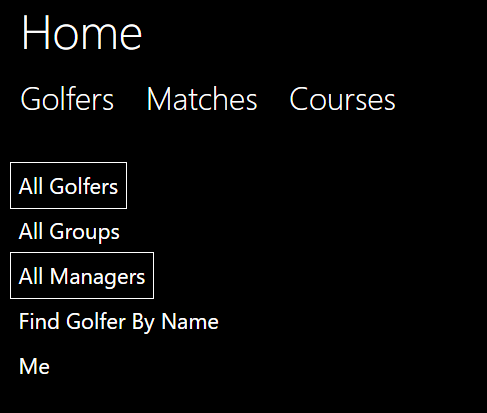
public IQueryable<Group> AllGroups()

{

return Container.Instances<Group>();

}

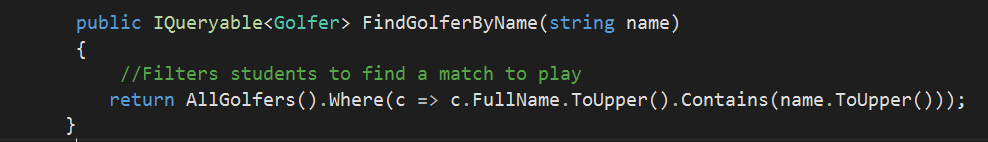
These are portrayed in the user interface as:



As you can see they don’t require an object to access the methods.

### Programing Techniques

#### LINQ

Throughout multiple stages of my programing I have used C# multi paradigm ability to add “LINQ” statements to my code whether deemed useful an example of this is within the class GolferConfig.

The use of functional programing here is useful as it is easy to read, and you know exactly the result that you should obtain, therefore its very reliable. Finally, it reduces the amount of statements of code significantly as function program requires no sequential statements. This can be shown in the line of code above without a Linq statement this could have taken multiple lines of code.

### Specifics

#### Algorithms

Within golf there are some very intricate rules for calculating scores and finding the winner. The most confusing for me was the stableford. The rules are mention in the design section. To come up with an algorithm I decided to split the problem up into separate functions and then work from there. This technique worked well.

Inherits from match.

public class MatchStableFord : Match

{

public int[] TotalScores;

public void AddScores(Hole hole, int ScoreA, int ScoreB, int ScoreC, int ScoreD)

{

var hs = Container.NewTransientInstance<FourPlayerHoleScore>();

Container.Persist(ref hs);

The user inputs the score. I have had to persist hs so that the user enters the score and then the algorithm can work.

hs.ScoreGolferA = ScoreA;

hs.ScoreGolferB = ScoreB;

hs.ScoreGolferC = ScoreC;

hs.ScoreGolferD = ScoreD;

hs.Hole = hole;

HoleScores.Add(hs);

Here I am calling the functions and returning them as an array.

int[] StrokeIndexs = StrokeIndexEffect(hole);

int[] GenderEffectOfGolfer = GenderEffect(hole);

int[] Scores = { ScoreA, ScoreB, ScoreC, ScoreD };

TotalScoreCalculated(hole, Scores, hs, StrokeIndexs, GenderEffectOfGolfer);

Container.Persist(ref hs);

HoleScores.Add(hs);

if (hole.HoleNumber == Course.Holes.Count)

{

Winner = Golfers.ElementAt(FindWinner());

for (int i = 0; i < 4; i++)

{

Golfers.ElementAt(i).MatchHistory.Add(this);

}

This Iteration will automatically end the match and calulate the winner once the score for the final hole has been added. It will also add the match to match history

}

}

private int[] GenderEffect(Hole hole)

{

int[] ParsForEachG = new int[4];

for (int i = 0; i < 4; i++)

{

if (Golfers.ElementAt(i).Gender == Enums.Gender.Female)

{

ParsForEachG[i] = 1;

}

Since Gender plays an effect in calculating the score per golfer I need to access whether they are male or female and return a value based on that.

else

{

ParsForEachG[i] = 2;

}

}

return ParsForEachG;

}

private int[] StrokeIndexEffect(Hole hole)

This method returns the effect per each golfer based on gender and the stoke index of the hole.

{

int[] Difficulties = new int[4];

for (int i = 0; i < 4; i++)

{

if (Golfers.ElementAt(i).Gender == Enums.Gender.Female)

{

Difficulties[i] = 19 - hole.RedStrokeIndex;

}

else

{

Difficulties[i] = 19 - hole.StrokeIndex;

}

}

int[] Handicaps = new int[4];

for (int i = 0; i < 4; i++)

{

Handicaps[i] = Golfers.ElementAt(i).Handicap - Difficulties[i];

}

return Handicaps;

}

private int ModifiedPar(FourPlayerHoleScore hs, int handi, int intitialPar)

{

int FinalPar = 0;

Using all the values from the previous methods calulates the final effect of the par which will act on the score.

if (handi >= 1)

{

if (handi >= 18 & intitialPar == 2)

{

FinalPar = hs.Hole.Par + 2;

}

else if (handi >= 1 & handi < 18 & intitialPar == 2)

{

FinalPar = hs.Hole.Par + 1;

}

else if (handi >= 18 & intitialPar == 1)

{

FinalPar = hs.Hole.RedPar + 2;

}

else if (handi >= 1 & handi < 18 & intitialPar == 1)

{

FinalPar = hs.Hole.RedPar + 1;

}

}

return FinalPar;

}

private int FindScore(int Score, int Par)

{

int TotalScore = 0;

Using the par obtained from the other methods it calculates the final score.

if (Score - Par == 1)

{

TotalScore += 1;

}

else if (Score - Par == 0)

{

TotalScore += 2;

}

else if (Score - Par < 0)

{

TotalScore += ((Score- Par) - 2)\* (-1);

}

else

{

TotalScore += 0;

}

return TotalScore;

}

[NakedObjectsIgnore]

public void TotalScoreCalculated(Hole hole, int[] Scores, FourPlayerHoleScore hs, int[] handicaps, int[] ParsForEachG)

{

int[] FinalPar = new int[4];

int[] TotalScores = new int[4];

for (int i = 0; i < 4; i++)

{

FinalPar[i] = ModifiedPar(hs, handicaps[i], ParsForEachG[i]);

}

for (int i = 0; i < 4; i++)

{

TotalScores[i] += FindScore(Scores[i], FinalPar[i]);

}

}

[NakedObjectsIgnore]

Calls the other methods to obtain the final score for each Golfer

public int FindWinner()

{

int Gwin = 0;

for (int i = 0; i < 4; i++)

{

if (TotalScores.Min() == TotalScores[i])

{

Gwin = i;

Calculates which golfer has the lowest score and therefore is the winner.

}

}

return Gwin;

}

#### Authorization & Authentication

Authorization and authentication are the most important aspects of the project since the project is meant to be a multi user application if there was no authorization the project wouldn’t know who was playing and then you could change anyone’s user information. To obtain the golfer Identity I am using Auth0 which is very used as a login system. I have not created Auth0 I am just using it. Auth0 requires an email to login which then obtained in the Me() method is GolferServices. If the Email has not already been associated with a golfer then this acts an account creation and you need to create a golfer and fill out your details otherwise this is how you access you profile so you can see your invites and other properties.

Each object has an authorizer which contains two function. IsVisable() and IsEditable(). This means that a user might be able to see something like another golfers name but they can’t change that. Which allows for the capability of multi user.

The code for a authorizer is shown below:

public class ClubManagerAuthoriser : ITypeAuthorizer<ClubManager>

{

public bool IsEditable(IPrincipal principal, ClubManager manager, string memberName)

{

if (manager.Username == principal.Identity.Name)

{

return true;

Only you can edit your profile.

}

else

{

return true;

}

}

public bool IsVisible(IPrincipal principal, ClubManager manager, string memberName)

Prevents the user from seeing the AddMatchHistory method so that they cant fake their history

{

if (memberName == "AddMatchHistory")

{

return false;

}

else if (((manager.Friends.Count == 0) & (memberName == "Friends"))

| ((manager.Groups.Count == 0) & (memberName == "Groups")))

{

return false;

}

else if (manager.Username == principal.Identity.Name)

{

return true;

}

else

{

if ((memberName == "Mobile") | (memberName == "AddFriend") | (memberName == "AddCourseToFavourites"))

{

Prevents over users adding friends for you, a course to your favorites or your mobile.

return false;

}

else

{

return true;

}

}

}

}

}

As mentioned in the server all the authorizers had to be declared so the server doesn’t think they are separate classes.

The testing of my authorization will be shown in the Testing Section.