Contents

[Technical Solution 2](#_Toc508572452)

[Overview 2](#_Toc508572453)

[Server 3](#_Toc508572454)

[Client 5](#_Toc508572455)

[Generic UI 5](#_Toc508572456)

[Customization 6](#_Toc508572457)

[Database 6](#_Toc508572458)

[Entity Framework 6](#_Toc508572459)

[Manual Database Mapping 7](#_Toc508572460)

[Database Customization 8](#_Toc508572461)

[Initializer 9](#_Toc508572462)

[Model 11](#_Toc508572463)

[Objects 11](#_Toc508572464)

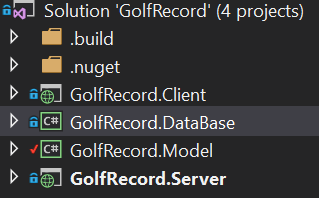
[Services 12](#_Toc508572465)

[Programing Techniques 12](#_Toc508572466)

[Specifics 13](#_Toc508572467)

# 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

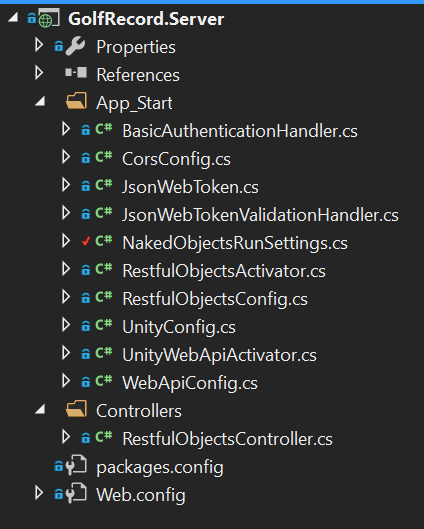
Server Project

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 major need\* 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[] {

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[] {

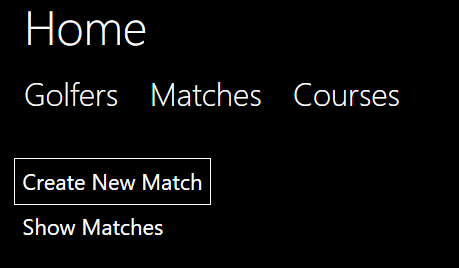
factory.NewMenu<GolferServices>(true, "Golfers"),

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;

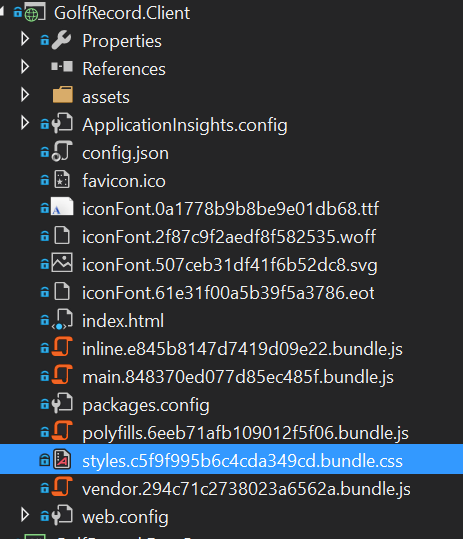
}

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.

}

}

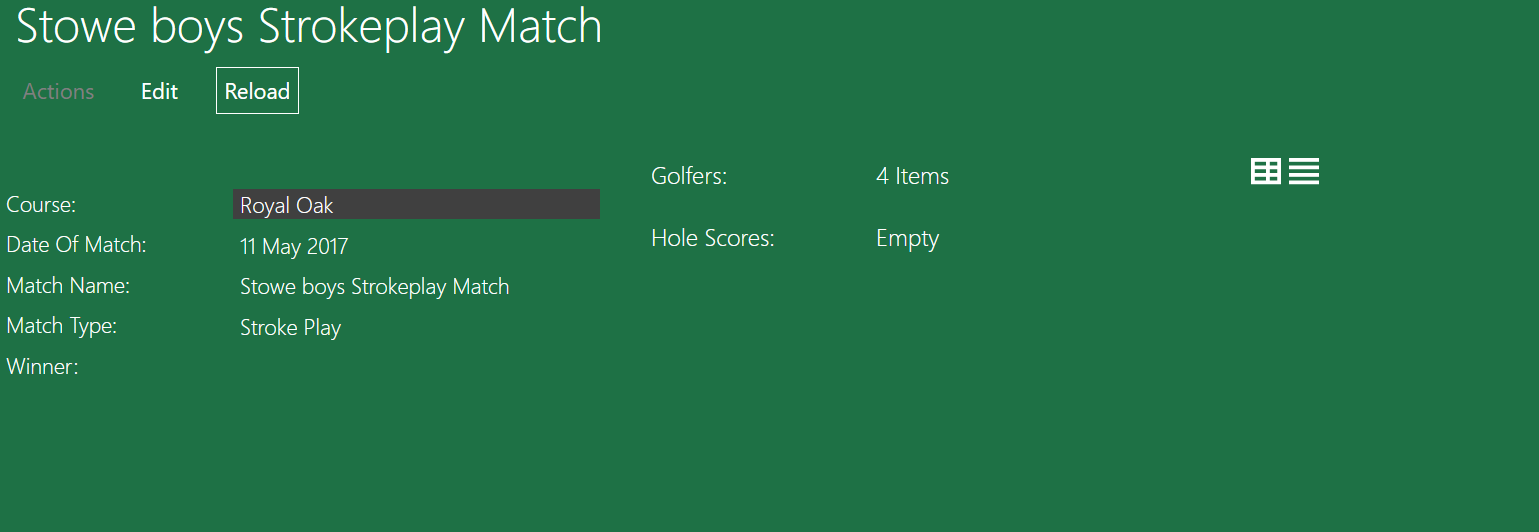
## Client



### Generic UI

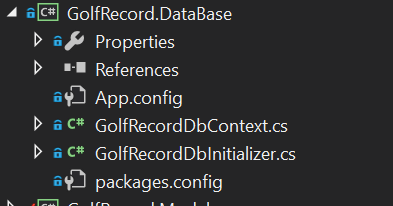
The client is a generic user interface that has been supplied by Naked Objects. There are thousands of lines of code which vary in language and therefore I am using the code given by NakedObjects allowing me to focus on the business side. Since I am using a client server architecture at a later stage I would be able to create an entirely new user interface. 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. 

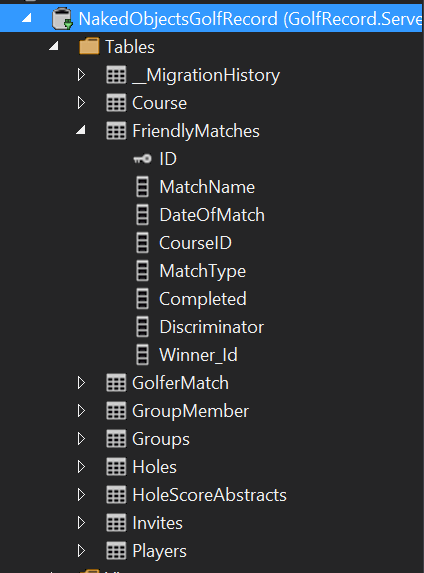
As you can see the Match is green but the course is black this helps the user to know that from match they can travel to the course by clicking that link which now stands out.

## 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.



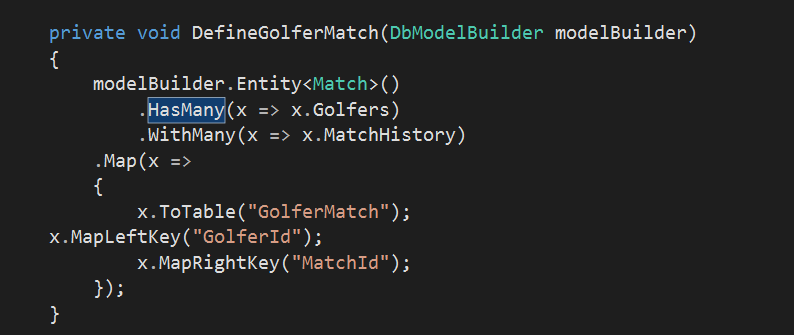
Here I have expanded the Matches table to show all the columns. The key next to the ID represents the primary key. I have customized the table here to called it “Friendly Matches” How I have done this will be mentioned Database Customization but the reason I have done this is because in future this system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

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");

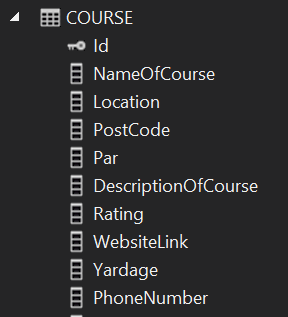
x.MapRightKey("GolferId");

});

}

### 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);

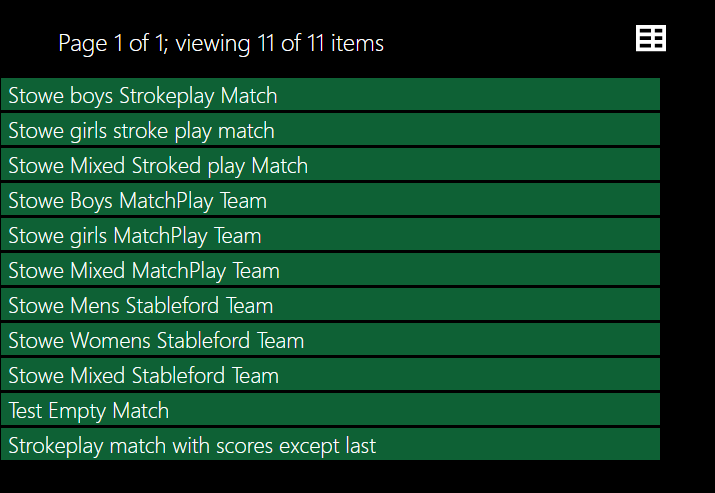
}

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.

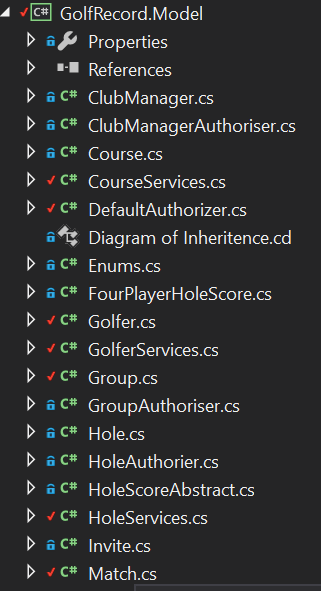
var s9 = AddNewMatch("Stowe Mixed Stableford Team", date1, 4, MatchType.StableFord);

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



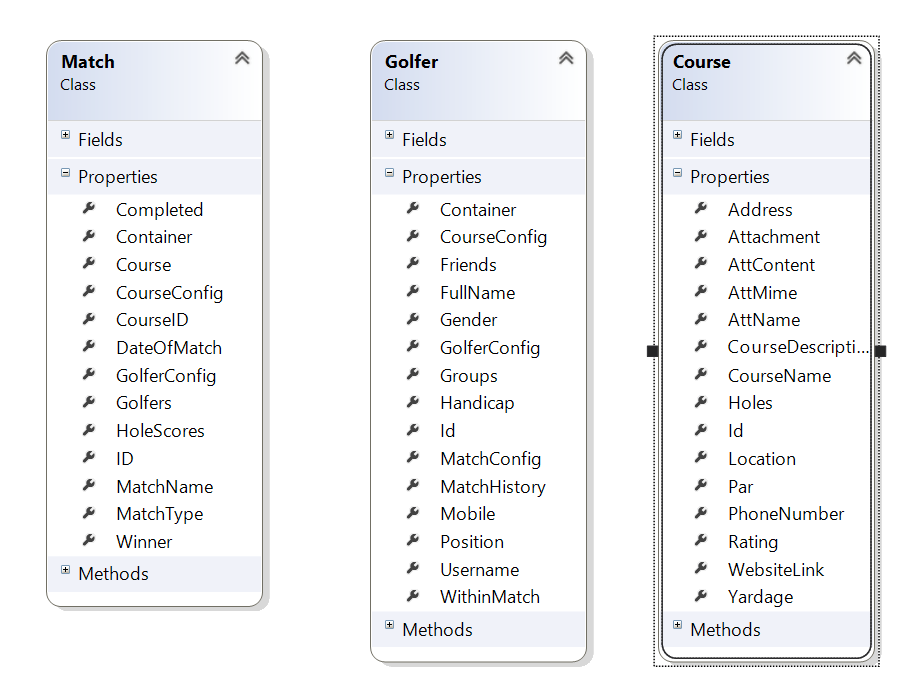
The Model is where I have written most of my code as it is the most important part. The Model contains all the objects, their services, their authorizer and other classes.

### Objects

The objects are what the user access. Each object contains properties, actions and attributes each of these will be discussed below

#### Properties

The properties of an object is the values assigned to it for example a Golfers name. Some properties are hidden to different users based on their authorization.



This photo shows the properties of the 3 main objects.

#### Actions

Actions can be accessed from within the object.

#### Complementary Methods

To be filled in

#### Attributes

To be Filled in

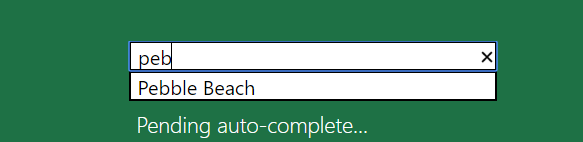
### Services

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.

### Programing Techniques

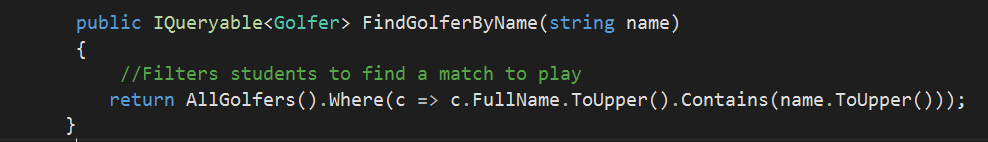
#### User-friendliness

In order to make my program more user friendly to some of the actions 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.



I have also added enums which result in a drop down which not only is aesthetically pleasing is also helpful for the user.

#### 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.

I’m really not sure how to write up my algorithm since Im not going through it step by step.

Will sort out at clinic.

#### 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.