Design

Here I will perform a general review and explanation of how and why my program works in the way it does. It will be broken into two main sections; the Web API section and the Discord+Twitch Bot section.  
And the comparison section, where I will show the key differences between the 2 versions of the project; there are 2 versions of the project, as after finishing the first version I realised that I had an incredible amount of duplicate code and that the code base as a whole was incredibly ineffective. But mainly I wanted the ability to handle multiple currencies in one instance of the bot and api.  
  
The majority of objectives have been met, however some weren’t done in the most effective manner. But I have tried my best to make code as efficient and effective as possible.

**Requestor –** A user that has performed a web request to the web api.  
Code – Indicates the highlighted text represents a small snippet of code

Web-API

The web-api is necessary to allow for bots, other than our bot, to manipulate and view currency data. It will also allow us to create a web site, where currency creator can configure their currency and the viewing of leader boards, for viewers to see their ranking inside of the currency in terms of balance and watch time.  
Without the web-api we wouldn’t be able to serve data to these web pages and it would also prevent the introduction of custom bots.  
It would’ve been possible to merge the web-api and the bot into one program, but my experience told me that this would’ve caused long hangs on both the bot and the web-api whenever the other was performing a larger operation. So by splitting them it will prevent said hangs.

The webapi composes of 2 key sections:

**The database emulation and interaction**, which composes of a set of objects designed to replicate the data stored in the database’s tables; along with the functions required to read and manipulate data in said tables. Which will allow for far simpler manipulation of the database, along with removing the need for duplicate code whenever we need to perform read/manipulate actions.

And the **web request handling**, which composes of two handler objects, one for Post requests and one for Get requests, which will take in a set of headers and/or parameters. Which will correspond to a set action; this usually entails the reading of 1 or more of the objects discussed above and/or the manipulation and saving of said objects.

Due to the web handling managing the manipulation of data in the database, the database interaction was created first, so I will start there.

Database Emulation

The emulation of the database into a set of objects allows for far more elegant integration with the data. And will also allow for us to place the sql functions inside of the objects in 2 different types; functions that fetch data will be static so they can be used without an instance of the object, And data manipulation functions, that will only be available inside of an object instance as data in the objects will be used in the sql commands. Also, the functioning of the sql actions will allow for far easier changing/fixing of sql commands. As they will only have to be modified in a handful of places, rather than in many places throughout the code.

Base Object

public class BaseObject {

//All objects will have an ID value

public int ID;

//All objects will need to be convertable into json format for transmission

public Newtonsoft.Json.Linq.JToken ToJson()

{

return Newtonsoft.Json.Linq.JToken.FromObject(this);

}

}

Due to all objects sharing an ID variable and a **ToJson** Function, we created a BaseObject class. Which all database objects will inherit from.  
The **ToJson** function will allow us to convert the object into a json, allowing for simpler transmission.

Example Object Variables

In this example; from the **Bot** object; 2 main types of data is specified. Firstly, data directly related to this object, ie **AccessToken**, **RefreshToken**, **BotName**, etc.  
Secondly other **Database Objects**, i.e. **Currency** and **OwnerLogin**. Which point specifically to an entry in another table, the connection between the **Objects** us created by using a **foreign key** in the **objects** entry in the database.  
The **key** allows us to then use the **FromID** function for that **Database Object** in order to fetch the **Object**, which can then be assigned to the associated variable.

//Define variables to replicate the Bot table

public Currency Currency;

public string AccessToken, RefreshToken, BotName;

public DateTime TokenRefreshDateTime;

public Login OwnerLogin;

public bool IsSuperBot=false;

Functions that contain static do not require an instance of the object to exist to be called. So will be used for functions that fetch an **Object**. Whereas functions that don’t contain static WILLrequire an instance of an object, hence will be used for saving and updating of data, as the data that will be saved/updated will be read from the instance of the object.

SQL Functions Structure

All SQL functions begin by creating a set of parameters List<OleDbParameter> Params = new List<OleDbParameter> { } this list will then be able to have instance of **OleDbParameter**. Which are created by stating the string that represents its position in the sql command, followed by the value that will be used in that position.   
new OleDbParameter("StringInCommand",Value)

**OleDbParameter**’s are extremely useful, as they allow us to keep data and the commands separate, which greatly reduces the risk of unwanted actions via SQL injection.  
Once added into the list, it can be passed included with an sql command.  
To perform an sql command, we will either **Execute** or **ExecuteReader**, the prior will not return any data, but the latter will return a List of string arrays, which will represent each row and then each cell inside the row. To perform either we must include the **SQL Command** and optionally any **Parameters** that we create above. For example  
Init.SQLi.ExecuteReader(@"SELECT \* FROM Bots WHERE (((Bots.LoginID)=@LoginID));", Paramaters); will return a List<String[]> which contains all columns from the Bots table, where the LoginID matches the LoginID in the table. This List<String[]> Can then be converted into a **Database Object**.

Example Object Functions

The following functions are examples of sql commands that follow a very similar structure across the different objects.

public static Bot FromJson(Newtonsoft.Json.Linq.JToken Json)

//Convert a json into a Bot object

{

return Json.ToObject<Bot>();

}

**FromJSON** allows us to convert a given Json into an object. The Json will typically be identical to an object on the requestor’s machine.  
This object can then be modified and/or have a given sql command performed on it.

public static Bot FromID(int ID,bool WithSecretData=false)

//All Single item From functions follow a similar structure

{

List<OleDbParameter> Params = new List<OleDbParameter> { new OleDbParameter("ID",ID) };

//Create a set of paramaters for the SQL query

List<String[]> RData = Init.SQLi.ExecuteReader(@"SELECT Bots.BotID, Bots.CurrencyID, Bots.AccessToken, Bots.TokenRefreshDateTime, Bots.RefreshToken, Bots.LoginID, Bots.IsSuperBot, Bots.BotName

FROM Bots

WHERE (((Bots.BotID)=@ID));

", Params);

//Select table data from the table, where the BotsID matches the ID paramater

if (RData.Count == 0) { return null; }

//Check we have at least 1 item in the returned sql results

Bot Bot = new Bot();//Create a new bot object

Bot.ID = int.Parse(RData[0][0]);//Set the bots variables using the sql results

if (RData[0][1] != "") { Bot.Currency = Currency.FromID(int.Parse(RData[0][1])); }

if (WithSecretData)//Only add this information if WithSecretData is set to true

{

Bot.AccessToken = RData[0][2];

Bot.TokenRefreshDateTime = DateTime.Parse(RData[0][3]);

Bot.RefreshToken = RData[0][4];

}

Bot.BotName = RData[0][7];

Bot.IsSuperBot = RData[0][6] == "True";

Bot.OwnerLogin = Login.FromID(int.Parse(RData[0][5]));

return Bot;//Return the bot

}

**FromID** takes an **ID**, and will then attempt to fetch data from the corresponding table where the **ID** matches.   
This data will then be placed into an instance of the object.   
This allows us to load data into an object so the data can be interacted with in a simpler manner.

The **WithSecretData** parameter allows us to indicate if secret data should be loaded into the object.  
This data ranges from hashed passwords too users emails.   
This parameter is set to true, when the calling function deems the requestor to be authorised to view this data, or if it is necessary for the act of authentication;  
 In the case of signing in the FromID function is called so that the hashed password can then be compared to the provided password.

**FromForeignKey(Login)** takes a **ForeignID(LoginID)** which allows us to get a list of all of this object, that is associated with the given **ForeignID**.  
In this case, it allows us to get all bots associated with a given login. Allowing us to provide a user with a set of all bots that he owns.  
Similar functions allow us to get all currencies of a given login or all bots of a given currency.

public static List<Bot> FromLogin(int LoginID, bool WithSecretData = false)

//All List item from functions follow a similar structure too the single item functions

{

List<OleDbParameter> Params = new List<OleDbParameter> {

new OleDbParameter("LoginID",LoginID) };

List<String[]> RData = Init.SQLi.ExecuteReader(@"SELECT Bots.BotID, Bots.CurrencyID, Bots.AccessToken, Bots.TokenRefreshDateTime, Bots.RefreshToken, Bots.LoginID, Bots.IsSuperBot, Bots.BotName

FROM Bots

WHERE (((Bots.LoginID)=@LoginID));

", Params);

List<Bot> Bots = new List<Bot> { };

//By not returning null and instead returning an empty list, we remove the necesity to check for a null object, in place of an empty list

foreach (String[] Item in RData)

//Instead of only creating a single object, we loop through all items in the sql results

{

Bot Bot = new Bot();

Bot.ID = int.Parse(Item[0]);

if (Item[1] != "") { Bot.Currency = Currency.FromID(int.Parse(Item[1])); }

if (WithSecretData)

{

Bot.AccessToken = Item[2];

Bot.TokenRefreshDateTime = DateTime.Parse(Item[3]);

Bot.RefreshToken = Item[4];

Bot.IsSuperBot = Item[6] == "True";

}

Bot.BotName = Item[7];

Bots.Add(Bot);//And we add each object into our list of objects

}

return Bots;//return the list of objects

}

Once again the **WithSecretData** parameter allows us to indicate if secret data should be loaded.

public void Delete()

{

if (FromID(this.ID) != null)

//Check if the Bot appears in the database

{

List<OleDbParameter> Params = new List<OleDbParameter> { new OleDbParameter("ID", this.ID) };

Init.SQLi.Execute(@"DELETE FROM Bots

WHERE (((Bots.BotID)=@ID));

", Params);

//Delete entry where the BotID matches

}

}

The **Delete** function, as the name suggests, takes the **ID** of the object and will remove it and any associated entities from the database and is meant to **only** be run by authenticated requestors.

The **Save** function, checks if any unique identifiers are present in the database already. If not it will compose an insert statement, by adding extra parameters and to the value statements in order to include all optional data; this only happens in some **save** functions; The insert command is then executed, to add the object into the database.

public bool Save()

{

//Check if DiscordID or TwitchID is already in the database

if (FromTwitchDiscord(this.DiscordID,this.TwitchID,this.Currency.ID) == null)

{

List<OleDbParameter> Params = new List<OleDbParameter> {

new OleDbParameter("Balance",this.Balance),

new OleDbParameter("CurrencyID",this.Currency.ID)

};//Set the sql paramaters

string PostStatment = "",PreStatment="";

//If DiscorID isnt null, we add it to our params and value statments

if (DiscordID != null) {

Params.Add(new OleDbParameter("DiscordID", DiscordID));

PreStatment += "DiscordID"; PostStatment += "@DiscordID"; }

//If TwitchID isnt null, we add it to our params and value statments

if (TwitchID != null)

{

//If we have already added to our statments we will need a comma to seperate the values

if (PostStatment != "") { PreStatment += ","; PostStatment += ","; }

Params.Add(new OleDbParameter("TwitchID", TwitchID));

PreStatment += "TwitchID"; PostStatment += "@TwitchID";

}

Init.SQLi.Execute(@"INSERT INTO Viewer (Balance, CurrencyID, " + PreStatment+ @") VALUES (@Balance, @CurrencyID, " + PostStatment+@")", Params);

//insert the viewer into the table

return true;

}

return false;

}

A Boolean is used in order to indicate if a conflicting unique identifier was found.

Some **save** functions don’t perform the unique identifier check, as they don’t have any unique data.  
However these objects are usually limited by how many a login/currency can have associated with them; so this is checked by the calling function.

public bool UpdateToken()

{

if (FromID(this.ID)!=null)

{

this.AccessToken = Networking.TokenSystem.CreateToken(64);

this.LastLoginDateTime = DateTime.Now;

List<OleDbParameter> Params = new List<OleDbParameter> {

new OleDbParameter("AccessToken",Init.ScryptEncoder.Encode(this.AccessToken)),

new OleDbParameter("LastLoginDateTime",this.LastLoginDateTime.ToString()),

new OleDbParameter("ID",this.ID)

};

Init.SQLi.Execute(@"UPDATE Logins SET Logins.AccessToken = @AccessToken, Logins.LastLoginDateTime = @LastLoginDateTime

WHERE(((Logins.LoginID) = @ID));

", Params);

return true;

}

return false;

}

The **Update** function will check if the current objects **ID** is present in the database. If it is, it will take the data inside the object, that is modifiable, and will update the values inside of the database.