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  
**OwlCoinV2** – The original version of the project.  
**OwlCoinV3 –** The new version of the project. **OwC –** Abbreviation of **OwlCoin**

# 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.

### Using the SQL Object

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 Table WHERE (((Table.Property)=@ParamterString));", 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**.

### SQL Object

The SQL object has to key functions; it simplifies the execution of sql commands and formats read data into a simpler and more elegant form. The key benefit of having the execution of sql commands in one function, is that if an issue is found with said code, it only need be adjusted in 1 place, as opposed to in many places throughout the code base.

private OleDbConnection Conn;//Stores The Active Database Coneection

private string DBase = "";//Stores The Databases File Path

public SQL(string DataBase)

{

DBase = DataBase;

RestartConn();

}

private void RestartConn()

{

if (Conn != null) {

/\* If connection is open, close it\*/

if (Conn.State == System.Data.ConnectionState.Open) { Conn.Close(); } }

Conn = new OleDbConnection("Provider = Microsoft.ACE.OLEDB.12.0; Data Source = " + DBase + ".accdb");

Conn.Open();// Open a new database connection

}

Upon Program start, we create an instance of the **SQL object** and point it to our database file. The **RestartConn** function is ran, which checks if the database connection is already open, if it is we close it. Then we create an instance of the database connection and point it too the desired database file. This connection instance will be used by the **Execute** and **ExecuteReader** functions.

The **ExecuteReader** and **Execute** functions are very similar, the key difference is that **Execute** does not perform a readfrom the database, so instead of OleDbDataReader Results = Command.ExecuteReader(); we simply perform Command.ExecuteNonQuery(); and then return true or false, based on the out come of the execution; ie success or failure.

public List<String[]> ExecuteReader(String sCommand, List<OleDbParameter> ParamCollection = null)

{

// Create the command, using the opened connection and the sql string

OleDbCommand Command = new OleDbCommand(sCommand, Conn);

if (ParamCollection != null) {

for (int i = 0; i < ParamCollection.Count; i++)

{Command.Parameters.Add(ParamCollection[i]); } } // Add the paramaters

// Execute the reader and store the result

OleDbDataReader Results = Command.ExecuteReader();

// Create a list of String[] too store the rows and collumns of the results

List<String[]> LResults = new List<string[]> { };

while (Results.Read()) // Keep reading untill all is read

{

string[] Data = new string[Results.FieldCount]; // Create a temporary String[]

// Place each collumn in the row into the array

for (int i = 0; i < Results.FieldCount; i++)

{ Data[i] = Results.GetValue(i).ToString(); }

LResults.Add(Data); // Add the row to the list

}

Results.Close(); // Terminate read and pass the formatted results back

return LResults;

}

The **ExecuteReader** function works by first creating a new instance of the **OleDbCommand** object and providing it with our current database connection along with the provided SQL command string.  
We then iterate through every **OleDbParamater** in the **ParamCollection** adding each parameter to the command object as we go.  
We now execute the reader and read each cell in the returned data set. As we read each row we iterate through the cells in the row, adding them to a string array. Once all cells of the row are read, we add the string array into our list of string arrays. Once all rows are read, we return the list of strings.

### 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

}

FromIDtakes 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 DiscordID 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.

### Object specialties

Due to the function of some objects, they function slightly differently in some functions and may have additional functions built into them.

//Load the configuration files into the bot object

public void LoadConfigs(bool WithLogin = false)

{

//Only load the login config if WithLogin is true

if (WithLogin) {

LoginConfig = FileManager.ReadFile("./Data/CurrencyConfigs/" + ID + "/Login.config.json");

}

CommandConfig = FileManager.ReadFile("./Data/CurrencyConfigs/" + ID + "/Command.config.json");

}

For example, the **Currency** object contains the **LoadConfigs** which will load the configuration files from storage. The inclusion of **WithLogin** allows us to withhold the **Login** config file, which contains OAuth tokens for the bot intergrations.

Hence this file is only accessible from **SuperBots** and by the API itself when a requestor attempts to authorise us to use one of our integrations for their twitch/discord.

//Create a directory for the configuration files

System.IO.Directory.CreateDirectory("./Data/CurrencyConfigs/" + C.ID);

//Copy the example config files into the directory

System.IO.File.Copy("./Data/DefaultConfigs/Command.config.json",

"./Data/CurrencyConfigs/" + C.ID+ "/Command.config.json");

System.IO.File.Copy("./Data/DefaultConfigs/Login.config.json",

"./Data/CurrencyConfigs/" + C.ID + "/Login.config.json");

The **Currency** object’s **Save** function also has code to create a new directory to store its configuration files. Along with code to copy the Default configurations files into the new directory. By having these default files, it reduces the amount of work needed to configure the bot by the currencies creator. Along with removing the need to handle empty/absent configuration files.

The **Viewer** object has a special **Increment** function, which will increase the **WatchTime** and **Balance** of the **Viewers** determined by the provided lists of **Twitch** and **Discord** ids.

//Increment the balance and watchtime by the given amount for all accounts with the given ids

public static bool Increment(List<string> DiscordIDs = null, List<string> TwitchIDs=null,int BalanceIncrementBy=0,int WatchTimeIncrementBy=0)

{

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

new OleDbParameter("BalanceIncrement", BalanceIncrementBy),

new OleDbParameter("WatchTimeIncrement",WatchTimeIncrementBy)

};

string WhereStatement = "";

i = 0;

foreach(string DID in DiscordIDs)//Cycle through every ID in the discord id set

{

//Add a paramater containing the discord id to the set

Params.Add(new OleDbParameter("DiscordID" + i, DID));

//and an OR between each statement

if (WhereStatement != "") { WhereStatement += " OR "; }

//Add on the conditional statement

WhereStatement += "Viewer.DiscordID=@DiscordID" + i;

i++;

}

i = 0;

foreach (string TID in TwitchIDs)//Does the same as above just for twitch ids

{

Params.Add(new OleDbParameter("TwitchID" + i, TID));

if (WhereStatement != "") { WhereStatement += " OR "; }

WhereStatement += "Viewer.TwitchID=@TwitchID" + i;

i++;

}

Init.SQLi.Execute(@"UPDATE Viewer SET Viewer.Balance = Viewer.Balance + @BalanceIncrement, Viewer.WatchTime = Viewer.WatchTime + @WatchTimeIncrement

WHERE (((Viewer.DontReward)=False) AND (" + WhereStatement+@"));

", Params);

//Increment all matching ids balances and watchtime by the given amount

return true;

}

To perform this we create a **WhereStatement** which will store the composite conditional statements which require the **Viewers** **Discord** or **Twitch** ID to match before we increment their values.

For the incrementation we make use of SQL’s ability to perform simple mathematical operations and so we just add their current **Balance** to the **BalanceIncrement** in order to determine the new **Balance.**

## FileManager Functions

The **ReadFile** function will check for the existence of a file at the given **FilePath**, if a file found, the contents of the file will be read into a string. This file is then attempted to be parsed into Json format. In the circumstance the string is not able to be parsed into Json we return null; This prevents requestors from receiving an invalid Json and they will instead receive nothing, which should be far easier to deal with.

public static Newtonsoft.Json.Linq.JToken ReadFile(string FilePath)

{

if (System.IO.File.Exists(FilePath))//Check if the file exists

{

string Raw = System.IO.File.ReadAllText(FilePath);//Read the file}

//Try to convert the file contents to json form and pass it back

try { return Newtonsoft.Json.Linq.JToken.Parse(Raw);

catch { return null; }//If it cant be converted return null

}

return null;

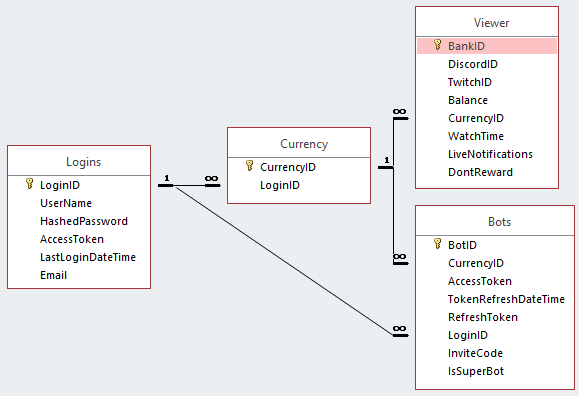
}

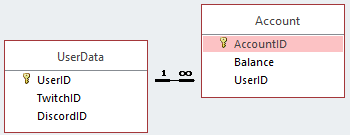
As will be described in the **Checks** section of the **WebAPI**, we check any received Json that it conforms to our requirements, in order to prevent this code from being necessary. But this code is included just in case.

The **WriteFile** function simply works to abbreviate the System.IO.File.WriteAllText too just FileManager.WriteFile. Bar that, it serves no other purpose.

## Database and Configuration file structure

### Database Structure

The database follows a normalised form, which limits repeats and associates data directly with its unique key(s).

However there are some extra steps that could’ve been taken to further remove duplicates. Which would’ve involved moving the **DiscordID** and **TwitchID** into a **UserData** table and then linking to said table using its **UserID**; which was how I structured the **OwlCoinV2** code.   
However this was not an option for **OwlCoinV3**, as it is necessary to allow each individual currency to manage what **Twitch** and **Discord** Ids are used for each **Viewer.** As otherwise it would be necessary to have a master program running to manage the association of the IDs. And it would allow for a **Bot** of any **Currency** to reasign the **IDs** of users in others, hence removing users access to their **Viewer** and isntead giving it to someone else.

### Configuration File Structure

**OwlCoinV3** is on the Right and a snippet of **OwlCoinV2** is on the Left.

The configuration files for the **Currencies** follows the **Json** format, which I decided would be better  
to use than **XML** as I have greater experience with **Json** and **XMLs** bulkyness made it a poor choice as it would make the files hard to read as a human and also far larger, therefore slower to send than the same **JSON.**

For security perposes the configuration is split into 2 separate files, the **Command** and **Login** config. The prior contains all the settings customizable by the **Currencies** owner and the latter which contains OAuth tokens and other private information; hence the **Login** config is only ever sent to **SuperBots**.

#### Responses and Parameters

**Responses** contain the message that is to be returned if a given event occurs, these usually contain **Parameters** which look like “<@ParameterName>” and should get replaced in a **Message Parser** by the corresponding data. For example **<@Amount>** in the **Balance** response would be replaced by the users current **Balance** in the **Currency.**

#### Example Configuration Segments

Many parts of the configuration files follows the same handful of layouts, hence I will layout theses layouts and explain why I structured them in that manner.

"Emotes": [  
 {  
 "Name": "ParamaterName",  
 "Twitch": "TwitchEmoteString",  
 "Discord": "<:DiscordEmoteStringAndID>"  
 },  
 Next Emote  
]

The **Emote** segment contains an array of **Emote** objects, which contains a **Name** when used in a **Response** in this manner “<@Name>” it will get replaced with the **Twitch** or **Discord** emote string based on the destination of the message.

The **Command** segment defines what will trigger the **command** to occur, along with the **responses** that will be returned.  
The **TwitchEnbaled** and **DiscordEnabled** **Booleans** indicate if this **command** is useable when the command originates from **Twitch** or **Discord**.   
**RequireLive** further restricts the **command** to only be useable when the streamer is currently streaming.  
**Commands** is a list of all possible **strings** that should be used to trigger the **command**, and should always be stored in lower case.  
**Responses** stores the **parameterised responses** that will be returned upon a certain outcome of the command.  
Further values may be stored in this segment, such as **MinimumPayment** for the **Pay** command and a list of **Items** in the case of the **Fish** command.  
Several special cases occur for the **Command** object.   
Firstly **FallbackMessage** which does not have any **Commands** and instead may be triggered if no other **Command** is triggered.  
Secondly **ErrorResponses** which simply stores generic **responses** for issues such as invalid inputs, api issues, etc.  
Finally **SimpleResponses** which merges the **Commands** and **Responses** into one set. Which uses the **Key** to identify the **string** that will trigger the **Response**; the **Value**; this is used for non api interactive commands, such as returning a link.

“CommandName”:{  
 “TwitchEnabled”:true,  
 “DiscordEnabled”:false,  
 “RequireLive”:true,  
 “Commands”:[  
 “CommandVariant1”,  
 “CommandVariant2” ],  
 “Responses”:{  
 “Success”:”<@SenderUser> Success”,  
 “Failed”:”<@SenderUser> Failure” }  
}

## Web Request Handling

The web server allows for managed access to the data stored inside of the database from anywhere on the www. Which mainly enables people to sign up to our website and then create and manage their currency; along with the ability to serve useful information to currency users, such as leader boards and help pages.  
It would’ve been possible to use a pre-existing sql server to manage the database, however this would not allow for checking of authorization and data in the manor I desired. Nor would it allow for more complex functions such as performing spin-off requests which are necessary when the currency owner provides OAuth authorization codes for our integrations.

The **Web Request** code is separated into 4 key sections, the **Event Handlers**, **Method Handlers, Initialiser** and the **Objects**.  
The **Event Handlers** job is to take the incoming **request**, put the received information into a **Standardised Request Object**, obtain the **Response Object** from the **Method Handlers** and then return the contents of the **Response Object** to the requestor.  
The **Method Handler** is specific for the request method (**GET** typically for fetching OR **POST** typically for modifying) and will perform a set of actions based on the content of the **Request**, this ranges from reading an entry based on an **ID** in the **Request Header**, to incrementing the **Balance** of all **IDs** given in the **JSON Request Data**.  
The **Method Handler** frequently makes use of methods inside the **Checks** namespace, which contains functions for checking the validity of received information, in order to determine if the **Request** is safe before performing any **Database Interactions.**

### Response Object

public class ResponseObject//This object stores the data that will be returned to the requestor

{

public Newtonsoft.Json.Linq.JToken Data;

//This will store the json, for the data that will be returned to the requestor

public int Code;

//These are used in place of a code and error message in the response, to seperate errors from the backend data handling and errors with the networking

public string Message;

public Newtonsoft.Json.Linq.JToken ToJson()

//Allows us to convert this object to json form, for transmission

{

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

}

}

The **Response Objects** job is to hold the data that will be returned to the **Requestor**. This includes the **Code** and **Message** which functions to identify what sort of error has occurred; if any; and a short description of said error. **Code** has the value of 200 when the **Request** was successful.

The **Data** **JSON** stores the **JSON** version of the **Database Object(s)** that is to be returned to the **Requestor**. This can be empty and usually is when and error has occurred.

The **ToJson** function; as the name suggests; converts the **Object** into **JSON** format, which will allow for it to be transmitted to the **Requestor**.

### Standardised Request Object

public class StandardisedRequestObject

{

/\* This Object places useful and frequently used data in an easy to access set of variables inside of the object

\* This will allow for shorter code, and by placing it in an object, the data can be kept together in a very elegant manner.\*/

public string URL,Method;

public string[] URLSegments;

public Dictionary<string, string> URLParamaters,StateParamaters;

public System.Collections.Specialized.NameValueCollection Headers;

public ResponseObject ResponseObject;//By keeping the response object and request data here, we wont need to pass it seperatly to functions

public Newtonsoft.Json.Linq.JToken RequestData;

public HttpListenerContext Context;//We store the original data for circumstances where the data is not stored seperatly in this object

public StandardisedRequestObject(HttpListenerContext Context,ResponseObject ResponseObject) // When creating the object we will require the ListenerContext and the ResponseObject that are being used

{

Headers = Context.Request.Headers;//Set the objects data

URL = Context.Request.RawUrl.ToLower();

Method = Context.Request.HttpMethod.ToLower();

URLSegments = URL.Split("/".ToCharArray());

URLParamaters = GetParamaters(Context.Request.RawUrl);

if (Method == "post")//If the method is post, read the posted data into json format and store it

{

string StreamString = new System.IO.StreamReader(Context.Request.InputStream).ReadToEnd();

if (StreamString != "") { RequestData = Newtonsoft.Json.Linq.JToken.Parse(StreamString); }

}

this.Context = Context;//Set the objects object references

this.ResponseObject = ResponseObject;

}  
}

The **Standardised Request Object** stores the **ListenerContext** in a layout that allows for simpler and easier access to frequently used data.

To do this, the initialiser for the **Object** takes values from the **ListenerContext** and places them in their respective variables. These actions sometimes includes and additional action in order to format the data into an even more useful form.  
For example, the **URLSegments** variable is assigned a string array of the split **URL.** Which will allow us to easily view the contents of a url segment.

The **RequestData** stores any **JSON** data sent with the request, hence we only attempt to read this data for **POST** requests, as this is the only time data should be sent in this form.

More complex functions may be performed to further improve data usefulness.

Dictionary<string, string> GetParamaters(string URL)

{

Dictionary<string, string> Params =

new Dictionary<string, string> { };

if (URL.Contains("?"))

//Only attempt if the url does contain a ?

{

string[] ParamSet = URL.Split("?".ToCharArray())[1].Split("&".ToCharArray());

//split the parameter string into its individual variables

foreach (string Param in ParamSet)

//Go through each variable and add the key and value into the dictionary

{

string[] SplitParam = Param.Split("=".ToCharArray());

if (SplitParam.Length == 2)

{

Params.Add(SplitParam[0].ToLower(), SplitParam[1]);

}

}

}

return Params;

}

For example the **GETParameters** function, will extract any **URL Parameters** and store them in a Dictionary. Which will allow us to quickly and easily read data from these parameters in the future.

To expand on this functionality, a similar **GetStateParams** function was necessary to extract sub parameters stored in the **State** url parameter. Which typically appears during integration authorisation.  
*It is very similar to GetParameters; the main difference being the split character; so it is not shown here.*

### Initialiser

public static int Size=0;

static HttpListener Listener;

public static void Start()

{

Listener = new HttpListener();

// Initalise the Listener and configure it

Listener.Prefixes.Add("http://+:"+Backend.Init.APIConfig["Port"]+"/");

Listener.Start();

Listener.BeginGetContext(HandleRequest, null);

//When we recive a request send to the the HandleRequest procdeure

if (Listener.IsListening) { Console.WriteLine("Web API is now running!"); } // Report that the listener is running

}

The **Initialiser** contains code for the setup of the **Web Server** and the initial handling of a **Request.**  
It composes of 3 functions, **Start**, **HandleRequest** and **RequestThread**.

The job of the **Start** function is to perform the setup of the **WebServer** and point the **WebServer** to the **HandleRequest** function.

static void HandleRequest(IAsyncResult Request)

{

new Thread(() => RequestThread(Listener.EndGetContext(Request))).Start();

//Create a thread of RequestThread, in order to prevent delay in handling new requests

Listener.BeginGetContext(HandleRequest, null); // Restart listener

}

The **HandleRequest** function will create a Thread for the **RequestThread** function to run in; which prevents delays handling new requests; and then repoints the **WebServer** too the **HandleRequest** function, allowing for a new **Request** to be received.

static void RequestThread(HttpListenerContext Context)

{

string Event = Context.Request.RemoteEndPoint + " Visited " + Context.Request.RawUrl + " Using " + Context.Request.HttpMethod;

Console.WriteLine(Event);

Size=(Size+1)%100;

if (Size == 0) { Console.Clear(); }

HttpListenerResponse Resp = Context.Response;

// Create the Listener Response and set response parameters

Resp.StatusCode = 200;

Resp.ContentType = "application/json";

ResponseObject ResponseObject = new ResponseObject();

// Create a reponse object and assign default values

ResponseObject.Code = 400; ResponseObject.Message = "Non-Specific Bad Request";

try

{

// Create a StandardisedRequestObject and provide it to the Get or Post function based on the method used by the request

StandardisedRequestObject Req = new StandardisedRequestObject(Context, ResponseObject);

if (Req.Method == "get") { Get.Handle(Req); }

if (Req.Method == "post") { Post.Handle(Req); }

}

catch (Exception E)

{ Console.WriteLine(E); ResponseObject.Code = 500; ResponseObject.Message = "Internal Server Error"; }

// If an unhandled error occurs set fallback values

byte[] ByteResponseData = Encoding.UTF8.GetBytes(ResponseObject.ToJson().ToString());

// Convert the response object into its json equivalent and then into its byte values

try

{

// Send the byte response data to the requestor

Resp.OutputStream.Write(ByteResponseData, 0, ByteResponseData.Length);

Resp.OutputStream.Close();

}

catch { Console.WriteLine("Unable to send response too " + Context.Request.RemoteEndPoint); } // If we cant send the response report the error to console

}