

**Software Engineering – SET09102**

**Coursework Report**

40216858

Messaging System

1. **Specification**

The aim of this coursework was to create an application that was able to process messages and output them in JSON file format. As each message is input, the system detects and validates whether it is an Email, SMS, or Tweet. It was then required carry out specific tasks, depending on what the message type is. This includes setting the maximum message size, picking up any “mentions” *(e.g. @johnsmith),* “hashtags” *(e.g. #example)*, sanitizing messages by filtering out URLs to a quarantine list, and converting “textspeak” abbreviations to their full meaning *(e.g. LOL<Laughing out loud>)*.

1. **Design & Implementation**

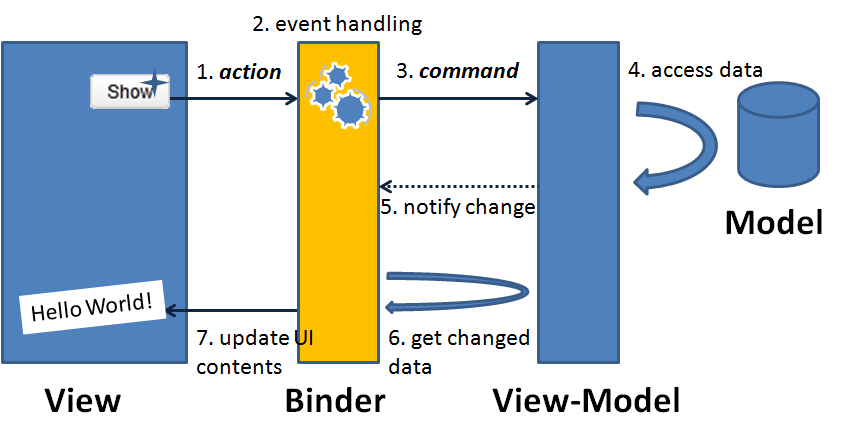
The system was designed using C# and a WPF user-interface. The entire design was based on a MVVM (Model-View-ViewModel) architectural pattern ***(Fig 1)***. This meant separating the user-interface into Model (xaml) classes, and the business logic (data model) into generic C# classes. I was then able to incorporate the View Model classes which are responsible for most of the “heavy-lifting”, handling most of the logic which will be carried out between the two.

Figure - MVVM Example

The MVVM pattern was used as it allows for easier maintenance of the system and provides a better overall file structure, separating the application logic and the UI effectively. By splitting up the workload into coding functionality and design, it was easier to manage as the system was gradually developed.

There are five views in this system, one “*MainWindow*” which encompasses the entire application, and four “*UserControl*” windows – one for each of the tasks that were outlined in the initial specification **(Fig 2)**.

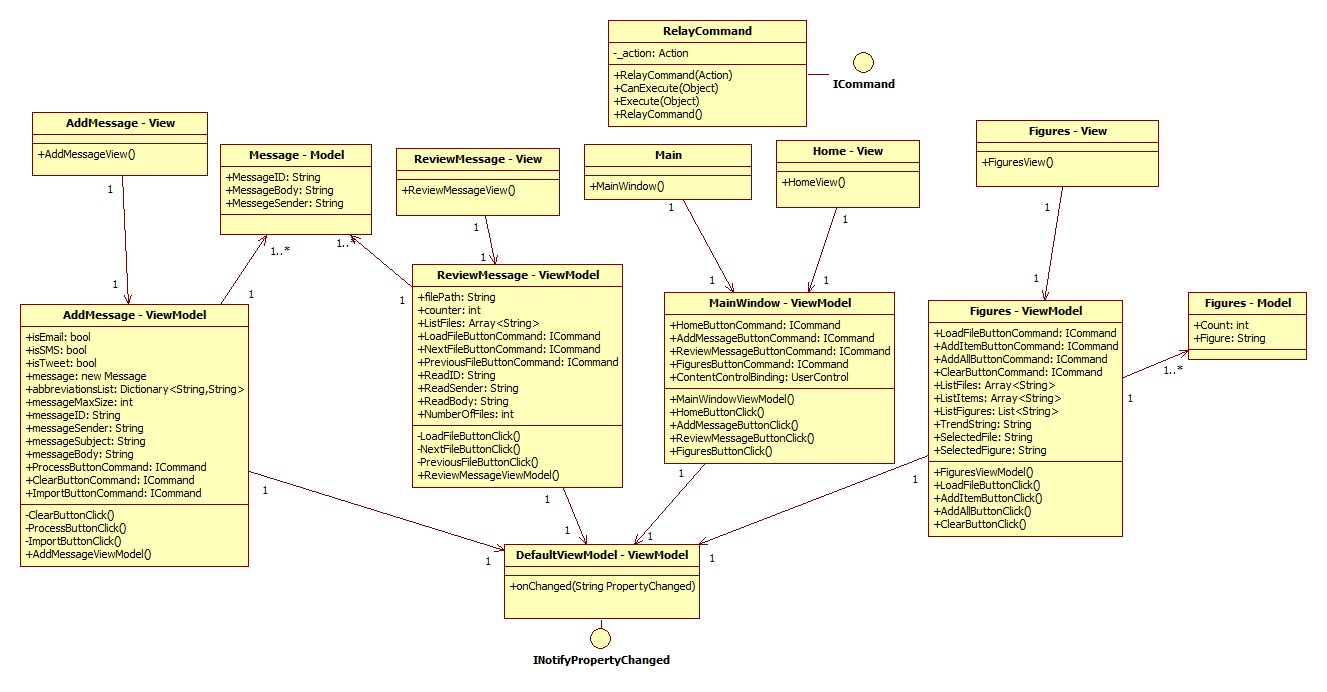


Figure -UML Class Diagram

* 1. **Main View**

When opening the application, the user will be presented with the MainWindow View. This shows a Menu on the left pane, with buttons which link to each of the four UserControl Views. The right-hand pane contains a Content View which is a container for each UserControl, initially showing the Home View, which contains a Login section **(Fig 3)**.

The login prevents the user from accessing any of the other views in the system before validating their credentials. Again, as this is for demonstration, it will accept any username and password combination at current, but if the application is developed further in the future then it could easily be linked with a user database.

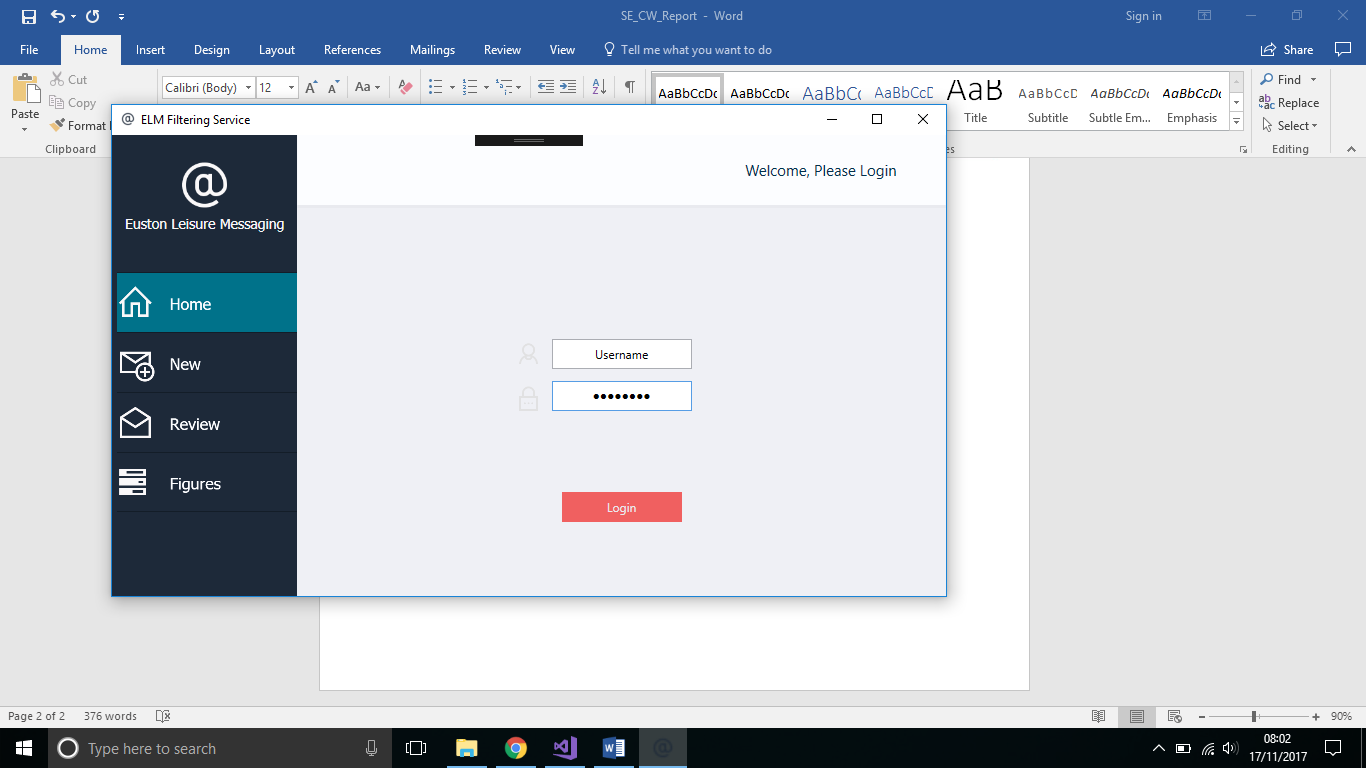


Figure – MainWindow & Home View

* 1. **Add Message**

The AddMessage View is arguably the most important part of the system. It allows the user to input messages of different types and carries out the tasks set out in the specification – sanitizing emails of URLs and filtering through keywords in SMS & Tweets.

The View is split up in to various sections, the first of which is the “Sender” textbox and button **(Fig 4)**. This is where the user inputs the address/username/number from which the message was sent. Before the main message can be input, this sender textbox must be filled out and verified. Different techniques are used to verify that one of the correct formats is being used, including Regular Expressions (Regex) **(Listing 1)**.

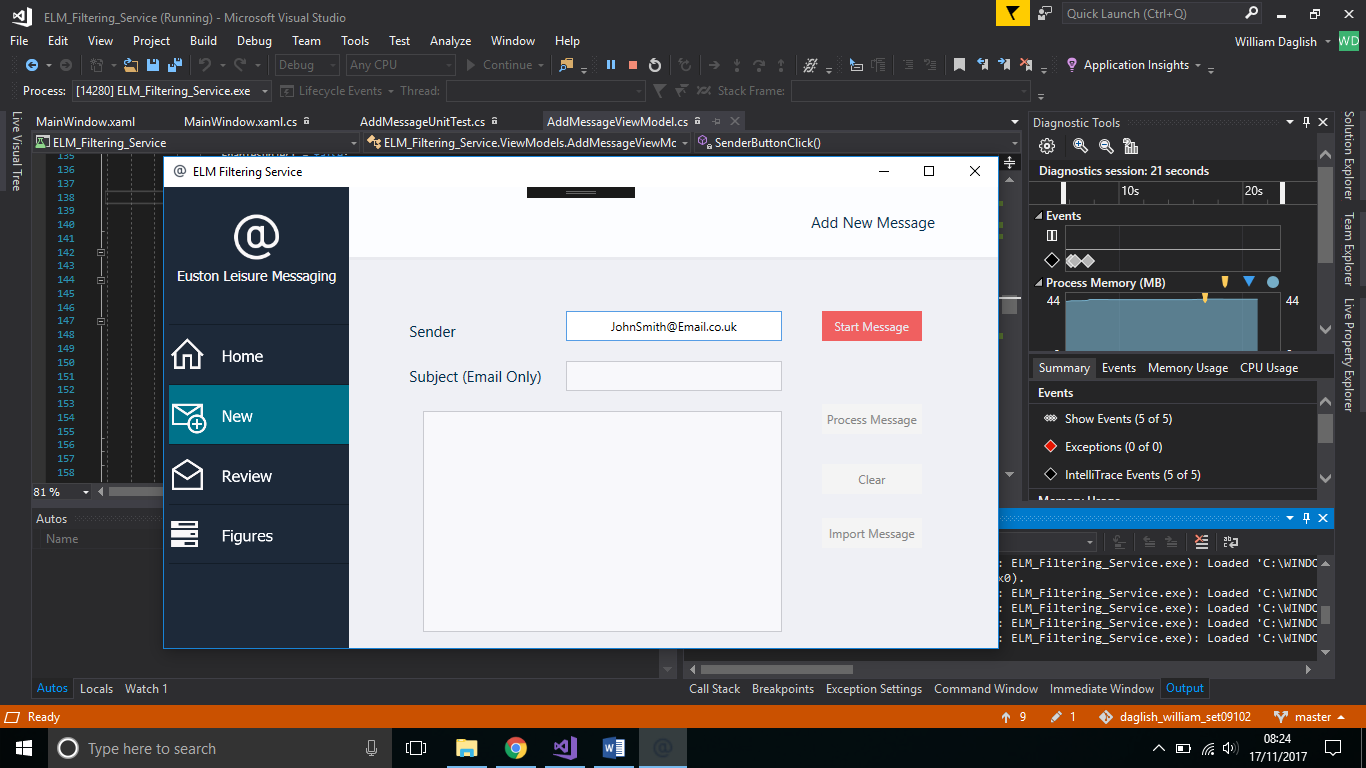


Figure - Sender Textbox



Listing 1 - Regex to check for Phone Number

Once a sender has been filled out, the user then presses the Start Message button. This button has a command binding which calls the method in the AddMessageViewModel class. This then verifies the validity of the Sender and the user can then input the message. Each type of message will have a different maximum character count, and will be given an ID number – consisting of a character (S/E/T) depending on the message type, and a 9-digit number which is generated by a random function

After completing a message, the “Process” button can be pressed which also has a command binding. This causes the processing method to be called, which is where the logic takes place that carries out the other tasks in the specification.

If the message being sent is an SMS or Tweet, then any textspeak abbreviations are included with an expanded meaning next to them. This is done with the use of a Dictionary which contains two sets of strings – a Key and a Value. For example the Key “LOL” would have a value of “Laughing out loud”. The application simply iterates through each word in the message string and attempts to find a match for any Key in the dictionary. If a match is found then the Value is inserted after the match **(Listing 2)**.



Listing 2 – Finding textspeak

If the message is a Tweet, the process function also iterates through each word to find hashtags and mentions **(Listing 3)**. On the other hand, if the message is an Email then the function attempts to find URLs instead. If any of these items are found, then they are stored to a log so that they can be reviewed later.



Listing 3 – Finding hashtags & mentions in tweets

The final part of the “Process” function is the serialization of the message into a JSON file **(Listing 4)**. By making use of the Json.NET Newtonsoft package, the message object is easily formatted into different lines and stored so that it can be reviewed later.



Listing 4 – Serializing a message to a JSON file

* 1. **Review Message**

The next part of the application is the Review Message section. To make it user friendly, messages can be easily selected from a List and loaded, or can be cycled through one at a time by using two “backward” and “forward” buttons **(Fig 5)**.

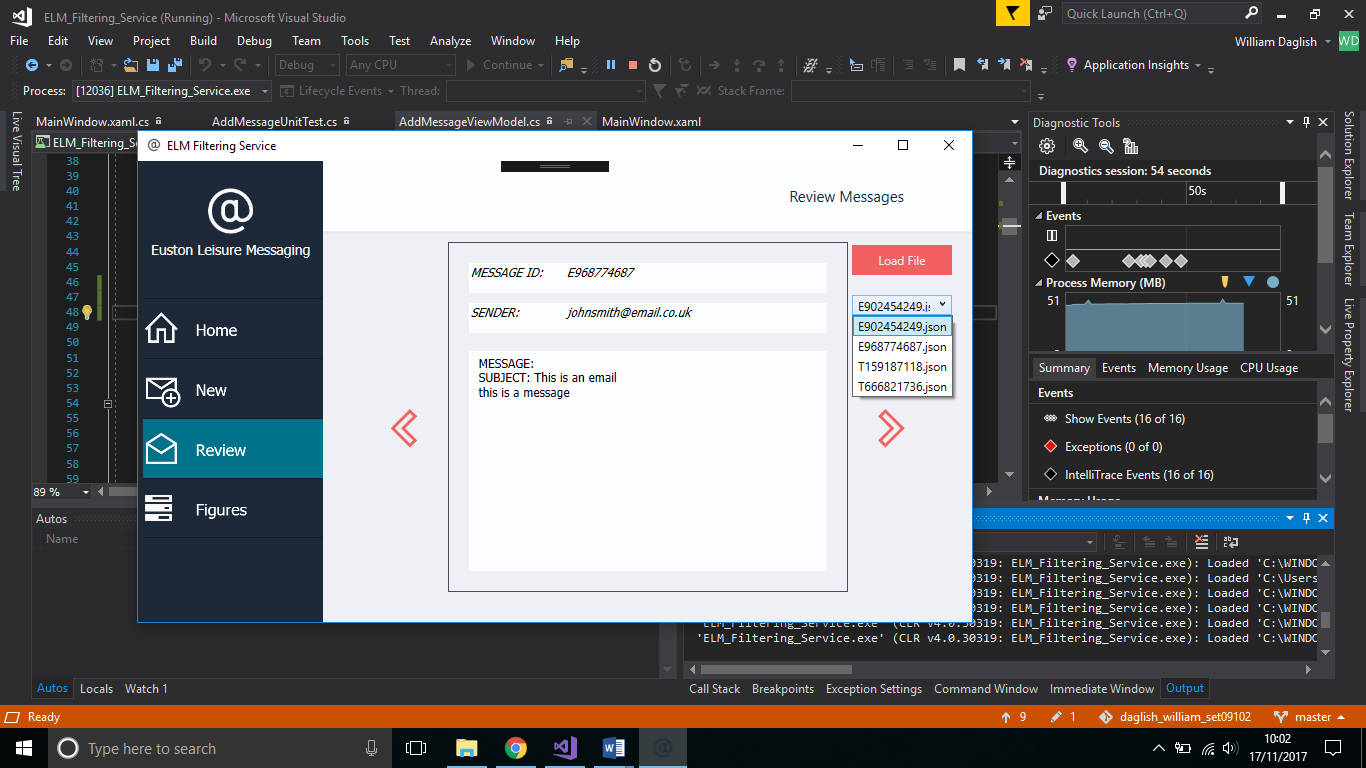


Figure – Reviewing a Message

As each message is loaded, it is deserialized by the “LoadFile” function. Each part of the message is loaded into a list of strings, and then displayed in “textblocks” within the View with the use of bindings.

**2.4 Figures View**

The final part of the application is the ability to view various figures that are extracted from different messages. These figures relate to specific strings that are detected from each different message, such as Hashtags, Mentions, URL quarantines, and Incident Reports (SIR).

As explained in section 2.2, the system iterates through each word in the message as it tries to find strings like Hashtags. Each is added to a List and once the function is complete, the list is stored to a txt file – with the filename format of “TwitterFigs\_dd\_mm\_yy” (other formats include SIR, and Quarantines).

In the Figure View, a user can select which file they wish to look at from the drop-down list. They then choose to “Load File” and can then select individual figures to load (Specific hashtags etc.) or just load in all the figures from that file. These are then displayed on screen, with the count of occurrences and a “bar-chart” format to make it easier to read **(Fig 6).**

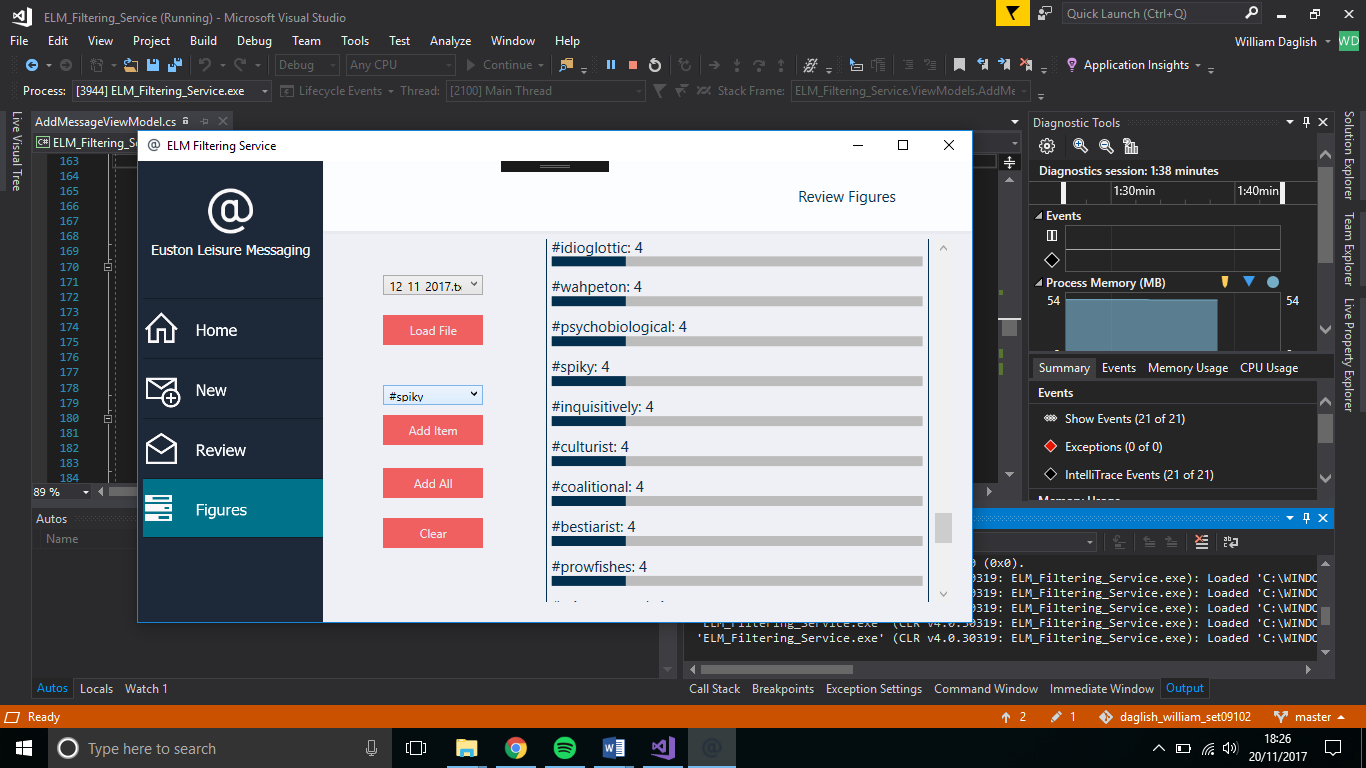


Figure – Viewing some Twitter Hashtags

1. **Testing**

Throughout various stages of development, I was required to test specific methods and functionality of my application. For example, after trying to implement a method to check for a Phone Number in the “Sender” textbox, I would need to ensure that it successfully completed this task, and had restrictions in place so it could not be completed by a different string format.

To keep track of my testing throughout the duration of this project, I kept a spreadsheet which contained my functionality testing list. This included Test Cases, test data, expected output, and if a test case was currently working as required or not. By doing this, I was able to ensure that any functionality which was not working as required, could later be addressed and fixed.

My focus for test cases was based upon the functionality described in the coursework specification. Along with ensuring these functions were working, I added test cases which tried to “break” the application. For example, entering an email address with no domain, or a twitter handle with more than 15 characters. The reasoning for this was to ensure that if a user made an error, they would be alerted by the application with the use of a “MessageBox”. Once all functionality was working, I was happy that the testing stage of the project was complete **(Fig 7)**.

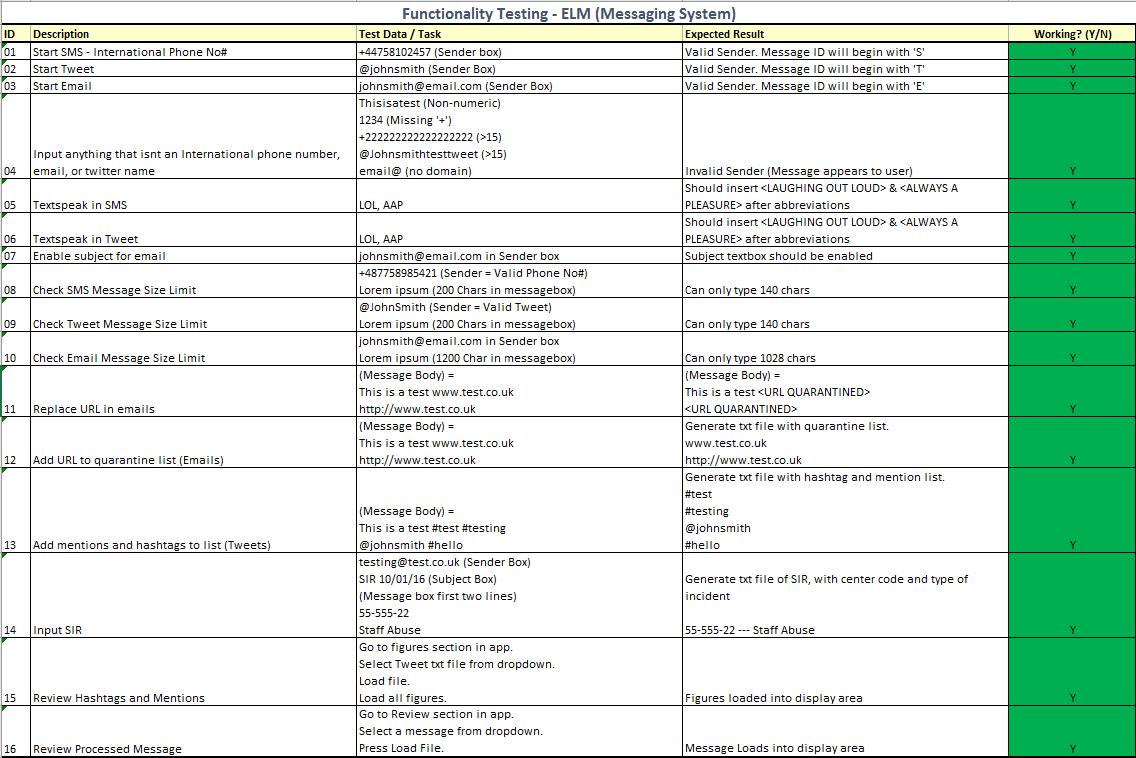


Figure – Record of tests carried out

1. **Version Control**

To keep track of any changes made in the project, I made use of version control, specifically I used GitHub – a Git based method. This meant that frequent updates and changes could be uploaded with a short description of any work carried out for that “commit” **(Fig 8)**.

These commits were recorded and allowed me to backtrack any code I had at any previous stage in the development of the application. This is particularly useful if anything such as data loss or corruption occurred.

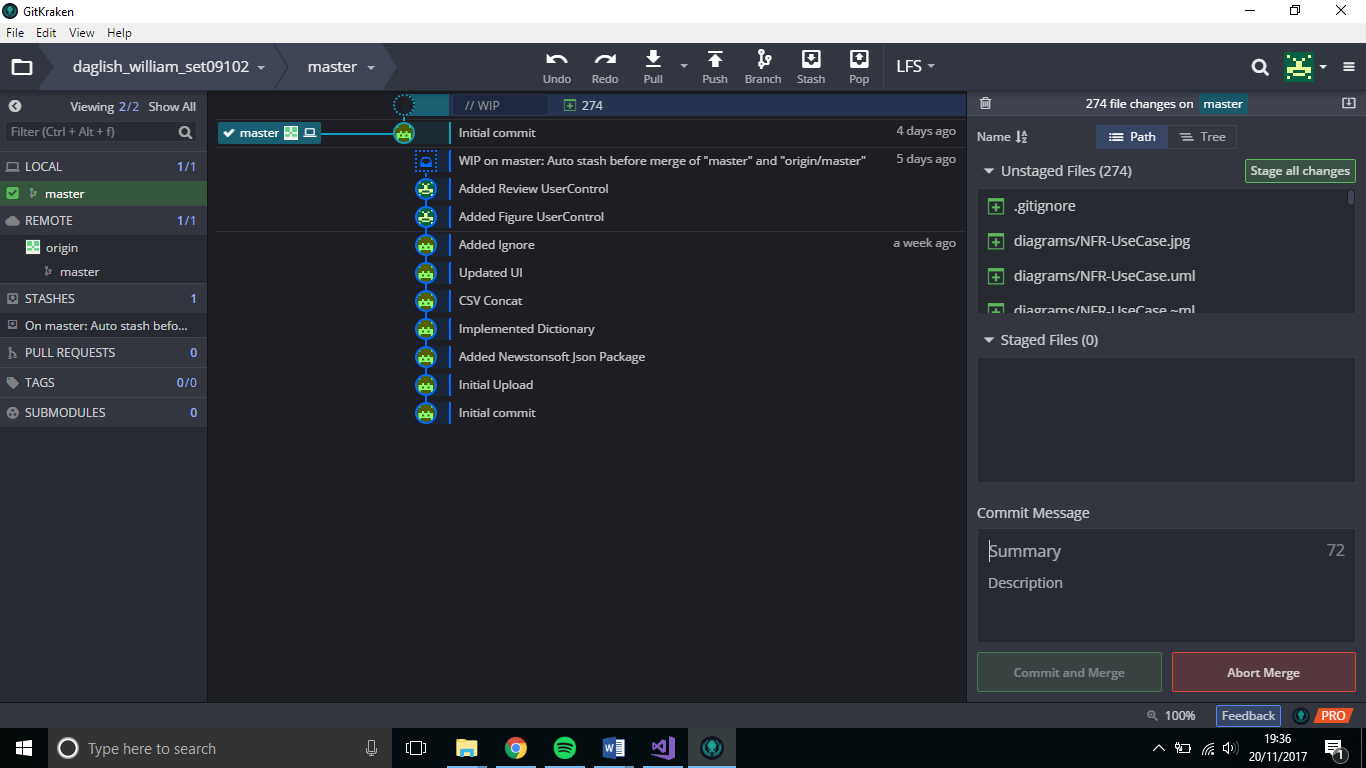


Figure – Using GitKraken software to upload Commits to GitHub Repository

Another use for version control systems such as Git, is the ability to allow teams to work together. For example, the repository can be broken down into various sections. A main “trunk” which contains the fully completed code, which can be released to users, and various “branches”. The branches would contain development work that is being carried out by various parts of the team.

By adopting this method of version control, you can ensure that the work carried out by each team does not conflict with others. Each section of the repository would also have its own policies. For example, this can include things such as a Unit Testing branch, or a Releasable Trunk.

1. **Evolution Strategy**

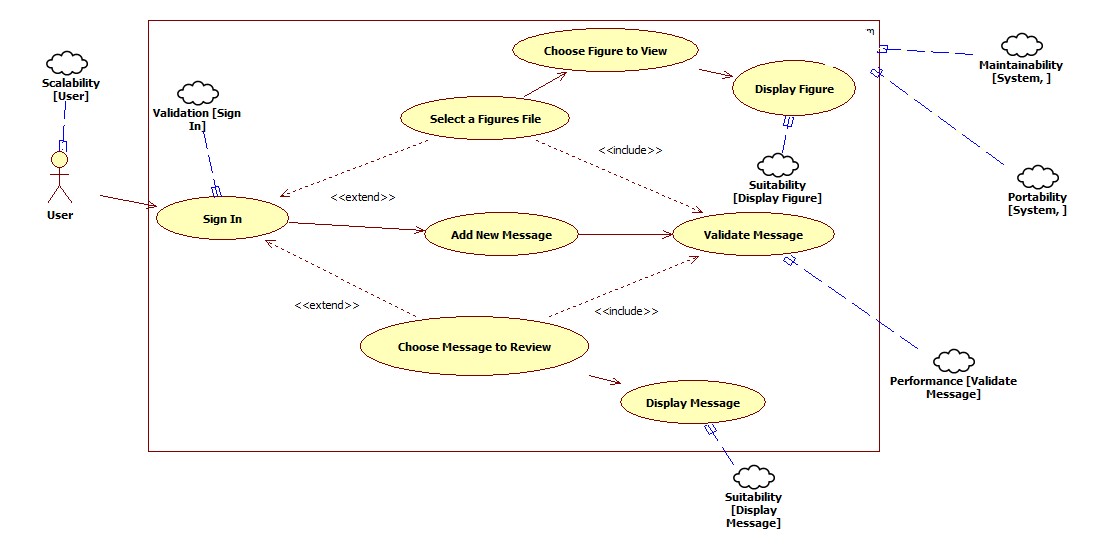
As is the case with any software system, evolution (updates, maintenance etc.) is inevitable. During the planning process of this application, I was able to decide on the best approach for the software design.

As I explained in the start of section 2, the MVVM design meant keeping the GUI and the actual business logic separate. By doing this it means that if at any point in the future I decide to update the GUI, or change from WPF to a different system for rendering the user interface, I can do so with ease. For example, I can still use the same methods I do in this system, but in a Windows Forms system.

The actual design of the GUI is relatively straight forward. My current system could be completely overhauled in terms of look and feel, and the current bindings could be linked with the newly designed interface items (textboxes, buttons etc.).

Whilst keeping the interface views and the business logic separate makes the updating of any GUI features easy, the same can be said for updating any of the functionality. The ViewModels (C# Code) is a lot easier to follow and less bulky, with no references to any WPF features (E.g. Textbox.Text == “…..”). A change such as increasing the maximum Tweet size from 140 to 280 characters (which Twitter are planning), would be extremely easy – simply changing one integer in the C# code, and making no changes to the interface.

# **Use Case Diagrams with NFRs**



# **UML Class Diagram**