# **SOFTWARE REQUIREMENTS SPECIFICATION (SRS)**

FOR

Napier Bank Message Filtering Service (NBMFS) – Software Engineering Module Number SET09102

Version 1.0  
Created by: Frankie Pollock  
Created for: Napier Bank (NB)

# **DOCUMENT CHANGE HISTORY**

|  |  |  |
| --- | --- | --- |
| Version Number | Date | Description |
| 1.0 | 4th of November 2022 | First Draft |
| 1.1 | 24th of November 2022 | Final Draft |

# ***PREFACE:***

*This report contains the Software Requirements Specification (SRS) of a messaging sorting system for Napier Bank (NB). The purpose of this project is to plan, develop and test a piece of software that will take in messages in different ways. Organise the messages by SMS, Tweet, or Email with verification features and more.*

Contents

[**SOFTWARE REQUIREMENTS SPECIFICATION (SRS)** 1](#_Toc120208216)

[**DOCUMENT CHANGE HISTORY** 1](#_Toc120208217)

[***PREFACE:*** 1](#_Toc120208218)

[**1.** **INTRODUCTION** 3](#_Toc120208220)

[***1.1.*** ***Purpose*** 3](#_Toc120208221)

[***1.2.*** ***Scope*** 3](#_Toc120208222)

[***1.3.*** ***Glossary Of Terms*** 3](#_Toc120208223)

[**2.** **Requirements** 4](#_Toc120208224)

[***2.1.*** ***Functional Requirements*** 4](#_Toc120208225)

[***2.2.*** ***Non-Functional Requirements*** 5](#_Toc120208226)

[**3.** **System Design** 6](#_Toc120208227)

[***3.1.*** ***Use Case Scenario*** 6](#_Toc120208228)

[**4.** **Testing Documentation** 8](#_Toc120208229)

[***4.1.*** **Test Plan** 8](#_Toc120208230)

[***4.2.*** **Testing Methods** 8](#_Toc120208231)

[***4.3.*** **Test Cases** 8](#_Toc120208232)

[***4.4.*** **Test Outputs** 9](#_Toc120208233)

[**5.** **Version Control** 12](#_Toc120208234)

[**6.** **System Evolution** 12](#_Toc120208235)

[**6.1.** **Predicted Evolution** 13](#_Toc120208236)

[**6.2.** **Predicted Maintenance and Cost** 13](#_Toc120208237)

## **INTRODUCTION**

This section will go over the purpose of this document, what is expected from the end result and the glossary of technical terms.

### ***Purpose***

This Software Requirement Specification (SRS) holds the software requirements for the Napier Bank Message (NBM). There will be walkthroughs of design, such as UI or Code. The requirements that must be met in order to fully comply with the client needs and finish the project successfully.

### ***Scope***

The Napier Bank Message Filtering System is to be developed for the Napier National Bank. It will take in messages from different ways such as input from user or reading in from an external source. The filtering system will then determine which type of message has been inputted into the system automatically. The types of messages are SMS Text, Email Messages and Tweets. Each will have their own smaller filtering systems as well. For SMS Text, if there are any textspeak abbreviations they must have explanations (i.e., LOL <Laugh Out Loud>). Emails will have two types of classification, Standard and Significant Incident Reports (SIR). Tweets will also have textspeak abbreviation explanations like SMS, they will also include hashtags that generate a trending list.

### ***Glossary Of Terms***

|  |  |
| --- | --- |
| **Term** | **Meaning** |
| **Software Requirements Specification (SRS)** | **A document that describes the software and how it is expected to perform.** |
| **Textspeak Abbreviations** | **Shortened message language.** |
| **Graphical User Interface (GUI)** | **The menu in which a user interacts with the system.** |
| **Functional Requirements** | **The requirements needed for the system to function as needed.** |
| **Non-Functional Requirements (NFR)** | **Requirements that have constraints and determine how the system will act.** |
| **Constraints** | **Something that will prevent the system from progressing or slow it down.** |
| **Use Case** | **A scenario of the full system being developed.** |
| **Class Diagram** | **An overview of the classes in the system, including relationships, values, and methods** |
| **Transaction Processing Speed** | **The time in which the system takes to process data.** |
| **Response Time** | **The time in which the system responds to client interactions with the system itself.** |
| **Class Diagram** | **A diagram showing the classes of a system, their attributes, and relationships between them.** |
| **Unit Testing** | **Testing each operation from code one at a time to make sure it works as intended, only after one operation works completely can you move on to the next** |
| **Acceptance Testing** | **Testing that the system meets the functional requirements** |

## **Requirements**

The requirements are what is expected from the software to meet the criteria and goal for the clients’ specifications. This section will go over functional and non-functional requirements, product requirements and organisational requirements.

### ***Functional Requirements***

1. The system must be able to accept input from the user in order for the system to be useable.
2. The system must allow for ASCII characters for text so that the format is defined.
3. The system must be able to automatically detect the input message type from the sender for filtering.
4. The messages must be validated for the system length checks.
5. Emails that have URLs must have them replaced with text and send the URLs to a separate list for safety.
6. Emails that have SIR information must have its sort code and nature of incident sent to a separate list.
7. Tweets that contain hashtags will have the hashtags sent to a “Trending” list.
8. The list of hashtags must be viewable.
9. Tweets with other “@whoever” mentions will be send to a mentions list.
10. Emails must have a sender and subject.
11. The system must be able to assign unique message IDs.
12. The system must be able to identify textspeak abbreviations and expand them.
13. The system must be able to export inputs to a serialised JSON File.
14. The system must be able to identify email type from the header.
15. The system must be able to take in inputs from an external text file and display each message one-by-one.

### ***Non-Functional Requirements***

1. The system should correctly respond to the user interactions.
2. The System response time should be quick.
3. The robustness should be tested with minimal failure events.
4. The system should be available to use whenever the user boots it up and wherever.
5. The ease of use should be simplified with help frames where needed.
6. Conformance to good coding standards.
7. Good data integrity.
8. Have the system finished by the 25November 2022.
9. The system has been maintained and tested throughout the creation of the prototype.
10. The storage requirements should be able to handle all the data from the system.
11. Processed user transaction speed should be quick.
12. Security

Non-functional requirements (NFRs) are extremely vital in the end result of the prototype. They make the system being developed actually worthwhile and more than just a functional piece of software, but a complete package. NFRs also have the ability to expand and generate even more core Functional Requirements that are needed to make sure everything is being met to the best of the developer’s abilities. NFRs can be categorized in the following:

**Product Requirements**   
These requirements focus on the way the system will behave such as the speed at which the system is able to execute, how reliable it is and security for protecting data.

**Organizational Requirements**These requirements are what contain the company policies when developing the product, such as creating the environment to allow clients of the product to use in, in the case of NBMFS this will be the bank employees having access granted to them for using the system.

**External Requirements**  
External requirements relate to things such as governmental policies, regulations, legislation, and safety. For the case of NBMFS, the data should be safe and kept secure to comply with data protection.

## **System Design**

### ***Use Case Scenario***

**Diagram

Description automatically generated**

Figure 1: The diagram above depicts the use case of the Napier Bank Messaging Filtration System. Starting from the system start-up then inputting the message, viewing the message, and optionally looking at the Mentions, Trending or SIR list. It also includes some NFRs such as transaction processing time and response time.

**Class Diagram**

A class diagram is used to make a rough draft of what the system is going to look like, what attributes it will contain, what operations it will perform, what classes it will hold, how they relate to each other, the cardinality, and the data type of the attributes.

On the following page, I have the NBMFS class diagram that I had created for help when it comes to creating the prototype and coding the system.

Diagram, schematic

Description automatically generated

## **Testing Documentation**

This section will detail the testing that was done including the strategy, types of testing, identifying test cases, plan. Testing is essential in the development of software, and even before anything has been coded. It will provide a level of quality control by finding anomalies and issues with the system and providing fixes. It also ensures that the system is in accordance with the functional and non-functional requirements.

### **Test Plan**

For the purposes of testing the NBMFS was developed with test driven development system model in mind. For this, I had created tests before any code was even written derived from the functional requirements. For example, the sender of the message can only be in a certain format, so for this I create tests and if they pass the code can be implemented in my main classes. If any tests fail, the code must be addressed and fixed first and re-tested before implementation. This type of development is very beneficial since we know what the system needs to achieve with the FRs and the tests can be comprehensive, it leads to tests that cover the overall system and will allow me to create system documentation in a more streamlined and focussed manner.

Tests are developed with the main objective of finding errors and rectifying them, these include expected errors and unexpected errors. Finding these will require action and documenting any changes made.

There are two types of testing being done, black and white box testing. Black box testing means you have no access to the codebase and testing it blind to make sure its functional from a user standpoint, with minimal issues or bugs that may arise. This will be done for this project via acceptance testing.

White box testing if when you test the components of a system with access to the codebase, this is a great way to ensure that the code is efficient in what it needs to do, works as intended and if any issues do arise that would cause system error, they can be handled and treated. Unit testing will be done to achieve white box testing.

Acceptance testing is when you test the system meets the non-functional and functional requirements of the system. Unit testing is when you test each individual method on it’s own to make sure it’s working as intended, with correct validation and a range of test data to ensure nothing unexpected can cause error.

**Test Items:**The main classes and methods of NBMFS and the methods that they contain. Making sure that validation works at intended, anything that has processing is processed correctly and identify any errors that may arise. The more errors found and rectified in the early stages of development; the less manpower will be needed for future bug fixing.

**Test Deliverables:**Testing results, documentation of errors encountered, and fixes applied. A working prototype derived from the test cases.

### **Testing Methods**

For the purposes of NBMFS, the testing will be done via unit tests in Visual Studio code.

These tests will be done buy testing each attribute per class, asserting that the result that is being given is the result that is to be expected. Only after the tests have passed where they need to be passed, can the next method be created and tested.

I created tests using normal, unexpected, and boundary test data. For example, in order for a Twitter handle to be valid, it needs to be more than 5 characters and less than 16 characters. This generates tests using names that are 5 characters long, 1 character long, 15 characters long, 16 characters long, and 10 characters long.

Alpha testing of UI elements is also something that I have done, ensuring that the UI works as intended is key in being ready for rollout to users, otherwise without a functioning UI the system would be useless and redundant.

### **Test Cases**

For the sake of testing the NBMFS, unit testing will look for results that will assert that two

values are indeed equal, and the test has passed, or that there is an exception where

an exception is expected.

### **Test Outputs**

The following table contains the results of the unit testing done for NBMFS. There are inputs, the expected outcome of the test, the actual outcome, what the issue is if failed, and the resolution made to correct the mistake.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TEST DATA** | **TEST INPUT** | **EXPECTED** | **ACTUAL** | **ERROR** | **FIX** |
| **smsSender** | **+447754346897** | **Pass** | **Pass** |  |  |
| **smsSender** | **+4477543468976** | **Fail** | **Fail** |  |  |
| **smsSender** | **+2908743** | **Pass** | **Fail** | **The phone number is a valid number but was not accepted.** | **Changed the sender validation to allow for length >=7 instead of the original >=12** |
| **smsSender** | **+5947005178356567** | **Fail** | **Fail** |  |  |
| **smsSender** | **+290874** | **Fail** | **Fail** |  |  |
| **smsSender** |  | **Fail** | **Fail** |  |  |
| **smsSender** | **+adsvgfdfddgcf** | **Fail** | **Fail** |  |  |
| **twitterSender** | **@JohnSmith** | **Pass** | **Pass** |  |  |
| **twitterSender** | **@JohnJacobsSmith** | **Pass** | **Pass** |  |  |
| **twitterSender** | **@JohnS** | **Pass** | **Pass** |  |  |
| **twitterSender** | **@JohnJacobsSmithCarter** | **Fail** | **Fail** |  |  |
| **twitterSender** | **@Jsm** | **Fail** | **Fail** |  |  |
| **twitterSender** |  | **Fail** | **Fail** |  |  |
| **twitterSender** | **@J@Smith** | **Fail** | **Pass** | **Invalid twitter handle was allowed** | **Changed the regex validation to only allow 1 @ from the star, no more** |
| **emailSender** | **johnJacobsmith@gmail.com** | **Pass** | **Pass** |  |  |
| **emailSender** | **john.Jacob\_smi12@gmail.com** | **Pass** | **Pass** |  |  |
| **emailSender** | **johnJacobssmithcarternoelsteve@gmail.com** | **Pass** | **Pass** |  |  |
| **emailSender** | **johnJ@gmail.com** | **Pass** | **Pass** |  |  |
| **emailSender** | **johnJacobssmithcarternoelsteven@gmail.com** | **Fail** | **Fail** |  |  |
| **emailSender** | **jo@gmail.com** | **Fail** | **Fail** |  |  |
| **emailSender** |  | **Fail** | **Fail** |  |  |
| **emailSender** | **johnJacobssmith@carternoelsteve@gmail.com** | **Fail** | **Fail** |  |  |
| **emailSubject** |  | **Fail** | **Fail** |  |  |
| **emailSubject** | **Hello You** | **Pass** | **Pass** |  |  |
| **emailSubject** | **SIR 12/12/12** | **Pass** | **Pass** |  |  |
| **emailSubject** | **123 567 911 345 789 1** | **Fail** | **Fail** |  |  |
| **emailSubject** | **H** | **Pass** | **Pass** |  |  |
| **MsgSMS** | **MsgSMS = 70 random characters** | **Pass** | **Pass** |  |  |
| **MsgSMS** |  | **Fail** | **Fail** |  |  |
| **MsgSMS** | **MsgSMS = 140 random characters** | **Pass** | **Pass** |  |  |
| **MsgSMS** | **MsgSMS = 141 random characters** | **Fail** | **Fail** |  |  |
| **MsgTweet** | **MsgTweet = 70 random characters** | **Pass** | **Pass** |  |  |
| **MsgTweet** |  | **Fail** | **Fail** |  |  |
| **MsgTweet** | **MsgTweet = 140 random characters** | **Pass** | **Pass** |  |  |
| **MsgTweet** | **MsgTweet = 141 random characters** | **Fail** | **Fail** |  |  |
| **MsgEmail** | **MsgEmail = 514 random characters** | **Pass** | **Pass** |  |  |
| **MsgEmail** | **MsgEmail = 1028 random characters** | **Pass** | **Pass** |  |  |
| **MsgEmail** | **H** | **Pass** | **Pass** |  |  |
| **MsgEmail** |  | **Fail** | **Fail** |  |  |
| **MsgEmail** | **MsgEmail = 1029 random characters** | **Fail** | **Fail** |  |  |
| **MsgEmail** | **Hi click here** [**https://www.google**](https://www.google) | **Hi click here <URL QUARENTINED>** | **Pass** |  |  |
| **MsgEmail** | **Hi click here** [**http://www.google**](http://www.google) | **Hi click here** [**http://www.google**](http://www.google) | **Fail** | **Http links aren’t quarantined** | **Added Http to the validation of URL to be quarantined** |
| **AbbvrText** | **LOL what is up?** | **LOL <Laughing out loud> what is up?** | **Pass** |  |  |
| **AbbvrText** | **LOL what is up LOL?** | **LOL <Laughing out loud> what is up LOL?** | **Fail** | **The second LOL was not expanded** | **Had to iterate through the message completely until at the end of the message, expanding all abbreviations** |
| **AbbvrText** | **LOL ROTFL** | **LOL<Laughing out loud> ROTFL<Rolling on the floor lauhing>** | **Pass** |  |  |

|  |  |  |
| --- | --- | --- |
| **UI Testing** | **Action Result** | **Explanation** |
| **Click on the Button to upload the text file** | **Opens the window to select the file, if none is selected then you get an error message.** |  |
| **Uploading a non-txt file** | **You get asked to upload the correct file type** |  |
| **SIR list displayed** | **The SIR list is populated if SIR information has been processed** |  |
| **Mentions List displayed** | **The Mentions list is populated if Mentions have been processed** |  |
| **Trending List displayed** | **The Trending list is populated if Trending has been processed** |  |
|  |  |  |
|  |  |  |

Throughout development, anything that would cause issue with the system that could crash or cause it to break has had exception errors put into place and with the user getting a message on what has gone wrong. This makes the system more fluid and will allow the user to identify any mistakes made on their own part.

## **Version Control**

When it comes to the version control of the Message Filtering System it has been developed and will continue to be developed in an agile approach. Agile approaches to development is simple, when you start a project, you divide up tasks in a group with daily meetings going over progress made and what is next. Client interaction and feedback will also be a big part in this, and with the constant feedback from all aspects of this development any changes can be addressed and planned quickly.

Version control has 3 types: **Local**, **Centralised,** and **Distributed** version control.

**Local Version Control** is simply a local version of what is being worked on, stored only on the file’s contents keeping track of any changes made and reconstructing the file with the new changes.

**Centralised Version Control** is when a version is stored and accessed by a team of people. The Version that each person works on is the version retrieved from the access point, work done on their own will still be local.

**Distributed Version Control** is a combining of the previous two. Usually being stored on a server, a team would each have access to the project files and download it to their system. Changes made would be synced back to the server and stored. It requires a level of communication with the team, so no one is conflicting with any others work. Or if there are any conflicts they can be resolved easily.

**Distributed Version Control** is what I would recommend for use with future development, and the tool to so is GITHUB. Below I will go over what it is, how it’s used and a scenario that could play out.

using GIT will be the plan. GIT is going to be a powerful that will allow multiple developers to work on iterations of the software and work both independently and collaboratively. GIT will allow the developers to work from a repository containing all versions of the system.

After the team would be formed and work divvied up between each team member, they would need to make sure that conflict resolution is always being addressed. A simple way of this could be using the Optimistic Locking, implementing a system that would prevent multiple people from working on the same aspect of a system.

Using the below example for adding a Log in system and admin level to view quarantined URLs, this is how future version control of the system could look like, using agile development (SCRUM in this example):

1. Initial Meeting, assigning teams to work on the Log in system, one for having a protected database of Employee Details for the log in system to match against, another to make admin level and view URLs.
   1. Teams branch off and start working on what they need to. Using GIT, they will do the following:
      1. Creating a branch from the master version, they will begin to work on their own/teams’ section of the system. Since each team has something to focus on, conflicts will be minimal and easily fixable if any arises.
      2. Each team will have their portion of the system they worked on synced to the branch and checked for conflicts. Anything that needs resolved will be, if not the development cycle can continue.
      3. After everything has been finished and all systems have been merged. Testing the system to get rid of bugs or conflicts will commence.
      4. After all of this has been done, the new version can be deployed and merged with the main branch.
2. The sprint begins, taking anywhere from 2-4 weeks in 24-hour increments.
3. At the beginning of the day, there is a meeting with the development team and the scrum master. They ask what has been done so far, what’s going to be done for the next meeting and what’s preventing anyone from doing what they need to do.
4. Sprint review, getting the PO to sign off the delivery of the version.
5. Sprint retrospective, the team assess what is good and what can be improved for next sprint.

GIT will allow for the system to be developed in a well irritated fashion with collaboration from multiple developers. It’s ability to create branches and merging them together leads to cost effective way to have individual versions of the system that can be brought together as one and worked on as a team.

## **System Evolution**

System Evolution in the modern world of software is inevitable in order for a system to stay relevant and useable. With this, there has to be a plan for the Message Filtering system to expand and become something larger and well maintained. This can be done in the form of software updates or new requirements that have emerged.

Firstly, there will be corrective maintenance. With the application being released to the userbase, there will issues and bugs that will undoubtedly crop up and need addressed, these could have been missed during the testing stage and need to be addressed afterwards during the evolution. After corrective, there is adaptive maintenance. Adapting the system so that it can work in a range of system environments other than just the system environment it was developed on. Then, after these have been addressed, System Evolution can and will take place.

### **Predicted Evolution**

Considering that the software at its current state functions as a simple messaging filtering system, there are many possibilities for the evolution of the product such as:

* The ability to send messages from within the app (responding to messages or sending new messages)
* Administration level for admin features, such as prohibited messages requiring a certain level of authority to view like quarantined URLs or SIR reports.
* Database storage for storing messages or reports.
* Login system linked to an employee database for secure login and viewings of messages, alongside an encrypted password protection level.
* Add the ability for the user to delete messages instead of saving them all.

Each of these proposals will go through a cycle of evolution. The change request will be made, the source code will be analysed and understood, the code will be modified and tested and lastly the modified system will be delivered and taken from there. As this is an agile development environment, it will be a smooth transition continuing the evolution of the system as the iteration cycle is similar in the way it is handled.

### **Predicted Maintenance and Cost**

When it comes to the maintenance of the software, this is key in making sure it works well and is kept stable over time, especially when the software is changing and being iterated upon. Certain aspects of the Message Filtering system will in the future cause issues and be costly.

For instance, storing of the messages will need a more appropriate storing system, otherwise the messages will be continuously stored locally on the machine and overtime start taking more and more of the local drive storage, leading to either losing valuable data or needing to purchase more storage which can be very costly when purchasing storage for multiple systems. An obvious solution is an external database to store all the messages in the branches and keep them in one secure place. However, this will be costly, considering the fact that the database will need to be kept secure and regularly backed up and have its own maintenance.

Another part of the system that could have issues is the importing of messages and reading them one by one on the screen at a time. Now, with the input file not having a limit on how many messages could be stored in it at a time, it could very quickly become bloated and filled with too much for a single reading session. This would waste the time of the employee having to get through all the messages in one session rather than being able to read some and continue later, which is a waste of company hours and money.

Maintenance brings in many things to consider, such as how expensive certain aspects of the system are to run and maintain. For NBMFS there isn’t anything too complex or deep to be that expensive to maintain, but for future updates and improvements that could quickly change. Costs are always something to take into consideration for maintenance, lifetime costs, first year of maintenance costs and costs when there is heavy maintenance being done for upgrades to the system. Then there is then the costs of updating the GUI to accommodate any changes which again will add to the cost of maintenance. It is a costly part of software but essential if it is to be used over a long period of time.

After years of evolution, the cost of maintenance could realistically get too high, and it could be time to make one of two decisions. Reengineer/Refactoring the system or get rid of it. Getting rid of it is an extreme option and would only be used if the system has completely become redundant or replaced with something better. Reengineering/Refactoring the system can help save costs by making sure the codebase is up to date and cause less issues for the future of maintenance.