Cardiff University – School of Computer Science and Informatics

Final Project Report:
"Freed" News Reader with Kill File

Written by Hani Chaaban (C1158840)

Supervised by Professor Ralph Martin

Moderated by Professor Konstantinos Papangelis

Total word count excl. title page, contents, references and acknowledgements: approx. 24600 words.

Contents

Abstract

This report will outline the background research, architecture design, implementation and testing of a News Reader application which features a “kill file” – a file which allows users to exclude/ignore articles of news which contain specific words. The application will also synchronise user data (such as which articles have already been read) to a Cloud service. I have chosen to develop this application in C# for Android using the Xamarin tools [1] (formerly known as Mono).

The report will outline how I created this application from a research, design and implementation perspective. The logic and methodology I used (along with my reasoning for choosing them) is another component of this report’s focus. To conclude I will summarise and reflect on the project’s progress as a whole in order to outline how I might continue to develop the project in future, and what I might have done differently if I were to begin anew.

1 - Introduction

1.1 - Goals of project / Scope

This section will outline the goals of the project, along with a summary of my chosen scope, which will be expanded on later in the project.

1.1.2 – Goals of project

The project's goal is to create a well featured and usable news reader application for Android devices using Xamarin. The app has a unique feature in using the "Kill File" to filter articles which contain user-specified words. This allows the user to avoid topics which they are not interested in, providing a greater degree of control in their news aggregation.

News reader apps make up some of the most popular downloads on app stores of all kinds. Sometimes even being bundled onto new devices, ready for use as soon as a consumer purchases it. Comparing the most popular apps, many of them share the same basic qualities and features. The innovations they have developed generally fall into the gesture or visual interaction category, rather than application features. This untapped potential is what motivated me to select this project, and also drove my decision to use Xamarin as the foundation technology – as it is my intent to continue development in future and expand to multiple platforms. As such it made sense to develop with a shared, reusable codebase in mind; one that could be leveraged for any type of device.

The application and scope I am aiming to develop will present a variety of challenges. For example, some of the basic challenges include:

- The collection of data from an RSS feed
- Displaying data to the user
- Designing a functional UI for navigation in the app

The complex challenges include:

- Designing an effective architecture for the feature implementations
- Designing logic suitable for cross platform reuse
- Implementing and designing various algorithms in an efficient manner
- Synchronising data with a cloud service using a third party authentication flow

Further challenges will be discussed as the report progresses.

1.1.3 – Scope

Regarding my scope, I have chosen to summarise it as the development of a news reader application with a "Kill File" and Cloud Synchronisation capabilities for the Android platform. This scope does not include the release of the app on the Play Store, or any specific additional features, but it does include the requirements for the app to be robust, usable, and relatively appealing to use.

Additionally, I have devised a specification for my application which supplements my project's scope definition in more detail. This specification can be found in Section 3.

1.2 - Audience

My target audience description is an English speaking, technically-apt person with a desire to consume current-affairs content (in topics of their choosing) while on-the-go.

This is based on the following thought process.

Given that the app is (eventually) destined for a release on the Play store, the type of application I am developing has a very wide target audience, encompassing persons who are interested in news of any kind. This type of audience has no particular subset with regard to age or gender.

Next, I will only be developing the application in my own native language (English) for practical reasons. This allows me to reasonably assume an additional characteristic of my target audience- that they are English speakers. This would usually refer to the US, based on the Google Play Store statistics for regional app downloads, which states that the United States represents the largest number of downloads from the Play store. [2] For the purposes of this project however, I will use the UK as the regional target audience, since the test users available to me are all based in the UK, and thus it would be difficult to specifically target a US audience.

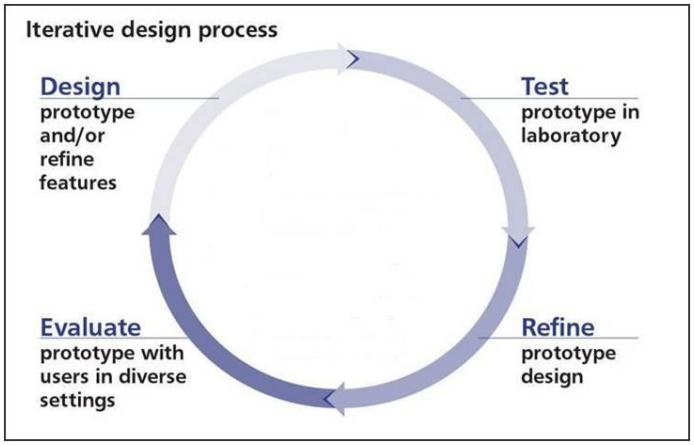
Additionally, the nature of the application being a smartphone app allows me to assume a relatively technically-literate audience, who are familiar with basic touchscreen paradigms such as swiping and general app navigation.

Lastly, the app being a news reader implies an audience with some level of desire to stay on top of current events via their mobile device, and thus a desire to consume that content promptly as it arrives and in any location. These important factors can be used to drive a great deal of my design and feature decisions, as I will describe in later sections.

1.3 – Approach, Development Cycle & Methodology

As the sole developer on this project, I will be working using an iterative methodology. I will begin by outlining a plan and designing an architecture, and while implementing my project I will strive to follow coding best practices where possible, including commenting code, using source control and following the "SOLID" programming principles to the best of my ability. I will also test my code functionally and practically using methods such as Automated Unit Testing and user feedback sessions.

Developing the requirements using an iterative process will mean refining and tweaking features as the project progresses. This is a methodology I am most familiar with, and will enable me to implement changes to functionally "complete" features as I continue to receive feedback from my testers. The iteration cycle can be roughly summarised as "Design, Test, Refine, Evaluate," as shown in the graphic below:



1.4 - Assumptions / Constraints

My primary constraint is mentioned in the goals of my project- that this application will serve as a functional prototype of an app which (with further development) could be released on any platform's app store, but will not necessarily be released on the store upon the project's completion. As such, my focus will be on the identification and design of features, interactions and visuals which might best appeal to my audience, while implementing the core functional requirements of the project in a robust and reusable fashion, rather than meeting all of the specific rules and guidelines of a Store-Released app. This will be further discussed in the Stakeholders section of my report.

I have also assumed that both the project's features are non-specific regarding their implementation mechanism. For example, the requirement to synchronise data with a cloud service could be achieved using Google Drive or Microsoft Azure APIs, and functionally remain identical. Therefore, I have assumed that all such "similarly-implemented" requirements are equally acceptable for the scope of my project, and that implementing any one of them will suffice.

For practical reasons, I am only able to test using my personal device, and have found emulators to produce very unreliable and inconsistent performance results (details of which are in the Testing section of my report). As such all non-UI testing will be performed on my own device: A Xiaomi Mi4 with Android 5.0.2 (CyanogenMod 11). This constraint also implies that the application in its submitted state will be used on hardware with specifications and Android revision which equals or exceeds that of my own device.

Ideally and realistically this testing would be further expanded to a variety of devices in order to establish a minimum requirement for user's devices before release. That being said, the previous constraint does not invalidate the need to optimize my code, and throughout development I will maintain a focus on minimizing the app's usage footprint, especially to prevent UI thread lockups/freezes (as described in the later sections, particularly Testing – Section 7).

More specific, implementation dependant assumptions and constraints may be defined if they arise during development, and will be outlined in the Research, Design and Implementation sections as appropriate.

1.5 – Summary of outcomes

The outcomes of this project (in addition to this report) will be the following:

- An Android app with a functional user interface which runs on Android devices from API level 19 (KitKat) and upwards.
- The app will aggregate RSS data from user-inputted feeds and display them for reading.
- The user will be able to synchronise state, settings and feed data to a Cloud service.
- The user will be able to filter out specific words of their choosing for omission from view.

Additional goals I have set myself include:

- The creation of a modern user interface, which is aesthetically pleasing to the user.
- The ability to locally cache data for use offline when a connection is not available.
- The creation of the app using Xamarin cross platform app development.

More specific features and requirements will be outlined after further background research, and will be documented in the Specification section.

2 – Background Research

This section outlines my background research for the project, which includes the context of my development on Xamarin and Android, market research of similar apps including their strengths and weaknesses, a summary of my stakeholders, methods and tools I might be able to use, and any theoretical problems or development challenges which required research.

2.1 – Android & Xamarin Development

Before beginning, I will outline a short summary of Xamarin and Android, and why I chose these combination of technologies for my project.

2.1.1 – Android

Android is a Linux Kernel based Operating System which targets touch-interactive systems, particularly mobile and tablet devices. Its source code is publicly available from Google [5] to encourage contribution and use from the development community. It was initially developed by a third party (Android Incorporated) and later acquired by Google and unveiled in 2007[6], experiencing a rapid growth to become the most popular Mobile OS globally by 2011 [7].

Applications on mobile platforms such as Android generally benefit from a wealth of additional inputs compared to traditional PC applications due to the ubiquity of sensory inputs; these range from the obvious (touchscreen, GPS, etc.) to the more subtle (light sensor, accelerometer etc.), but all provide a wealth of new opportunities to developers. This doubtlessly contributed to the rapid adoption and huge market for these types of "apps".

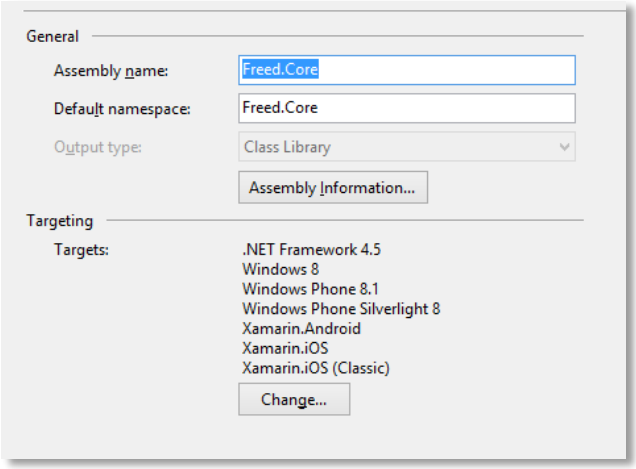
Applications developed for Android OS are usually developed in Java, and as of late using the Android Studio development environment (which is officially developed and maintained by Google) seems to have become the de-facto standard means of creating application for the platform. Despite this, a variety of other options exist, which seems to be due to Android's openness and popularity. Xamarin is one example of this, and there are various others, such as Apache's Cordova [8] toolkit.

These factors, alongside Android's widespread popularity and open attitude to development, drove me to develop the app for Android for this project, as a starting point on which to build alternative versions for other platforms in future.

2.1.2 – Xamarin

Xamarin launched in May 2011 [3], and is a framework which enables the development of native iOS, Android and Windows applications using C#. It was created by the members who previously worked on Mono, MonoTouch and Mono for Android [4], who reformed the team and project as Xamarin. Their work is essentially a cross platform conversion of the Common Language Infrastructure and Common Language Specifications, which are often referred to as "Microsoft .NET". In its previous iteration as Mono, it became quite popular due to the ability to reuse a great deal of the application logic between the three variants of supported applications, while retaining the ability to create and bundle native UI implementations to match the brand and feel of the target device.

One of the key features of Xamarin is the reusability, which is delineated by the project types. "Xamarin Portable Class Libraries" (PCLs) are the most compatible project type, and enable a great deal of cross platform reuse. Other types of projects in Xamarin enable Android, iOS and Windows views, and allow for the use of their respective platform specific features and types, where PCL's do not. These platform specific projects can use the logic developed in the PCL, by simply referencing the PCL project. This allows for the implementation logic in the PCL to only be written once, while enabling the development of an appealing native UI for each platform, and enables the app to be rapidly expanded across multiple platforms with minimal effort.



A screenshot demonstrating the reusability of the PCL project. All the targeted platforms are supported.

These factors, combined with my pre-existing experience developing applications in C#, made Xamarin a natural choice of tool to build this project.

2.2 – Market Research / Competing Applications

Prior to beginning my own design process, I decided to research other similar, successful apps to learn from. This section outlines my research into three chosen examples.

I selected the 3 applications based on their popularity on the store, primarily looking for apps with a high user count or high average review rating, settling on Flipboard, LinkedIn Pulse and Feedly. These choices were also influenced by the availability of users in my social circles, who I could poll to gain a greater understanding of the app's strengths and weakness. For each app I observed both my own and a sample of my peers' interactions with the app.

For each application I will focus on 4 core areas- the application' history, the USP/Key functionality, the app's weaker areas, and a summary of lessons learned and their influence on my project.

Lastly, in the following sub-sections, any quotes and statistics regarding downloads or ratings pertain only to the Google Play Store, and do not include other ecosystems.

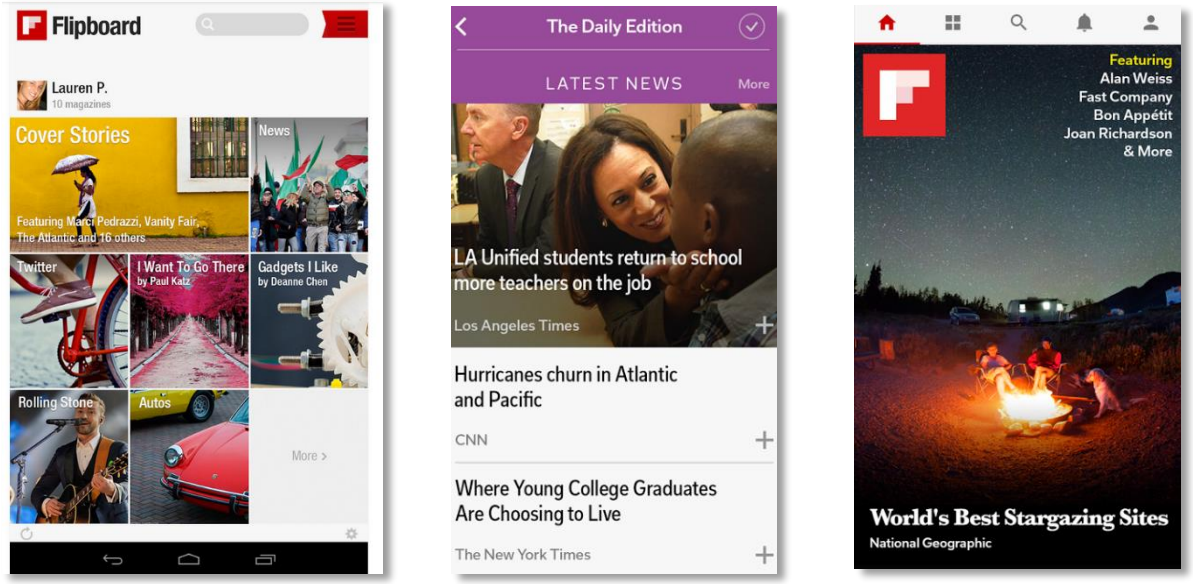
2.2.1 – Flipboard

2.2.1.1 - History

Flipboard is the most popular of the three news readers I researched, even being bundled with devices through partnerships with hardware vendors like Samsung [9]. The Play Store reports it has over 100 million downloads, and it is featured in the Editors Choice section. Being a commercial app which has been in development for many years, Flipboard boasts the most complete feature set of the three reviewed apps.

2.2.1.2 – USP & Key Functionality

Notably, Flipboard has a Web app (https://flipboard.com/) as well as an app for all 3 major platforms: Android, iOS and Windows Phone. I believe this platform ubiquity helps greatly with user retention, as fans who may wish to switch platforms will be able to continue using Flipboard regardless of which ecosystem they move to. This helps to justify my choice of using Xamarin for my project, as it speaks to the appeal of cross platform applications.



Various screens within the Flipboard application, illustrating groupings (left) latest news (center) and the homescreen's featured content (right).

News in Flipboard defaults to sorting by how "recent" it has been published, ensuring fresh content always floats to the top. Content is organised into "magazines" (groupings) which the users can modify at will, adding or removing individual sources or articles to each group. Additionally, the User can queue items to a custom magazine to read consecutively (using the '+' icon in the second image) which creates an extremely clean user action flow- generally the users (and myself) would browse through the sources to select articles to read before returning to the read queue to digest them sequentially.

The branding of the app centralises on the "Flip" theme. Light branding with plenty of high-quality images and colour splashes complete a very modern look for the app, which centralises on the widely known paradigm of "flipping through a magazine" to hook the user, and maintains this metaphor throughout the app with literal "flip" animations regularly occurring as the user navigates the app. These animations serve as great visual flair, and illustrate the quality of design.

2.2.1.3 – Weaknesses

The Flipboard app's primary weakness from what I discussed with my peers who also used the app lies within custom and non-standard UI. Certain elements of the UI, particularly the flow of pages as the user navigates the app, take a while to get used to, which can lead to some initial confusion for new users.

2.2.1.4 – Lessons Learned & Takeaways for my own project

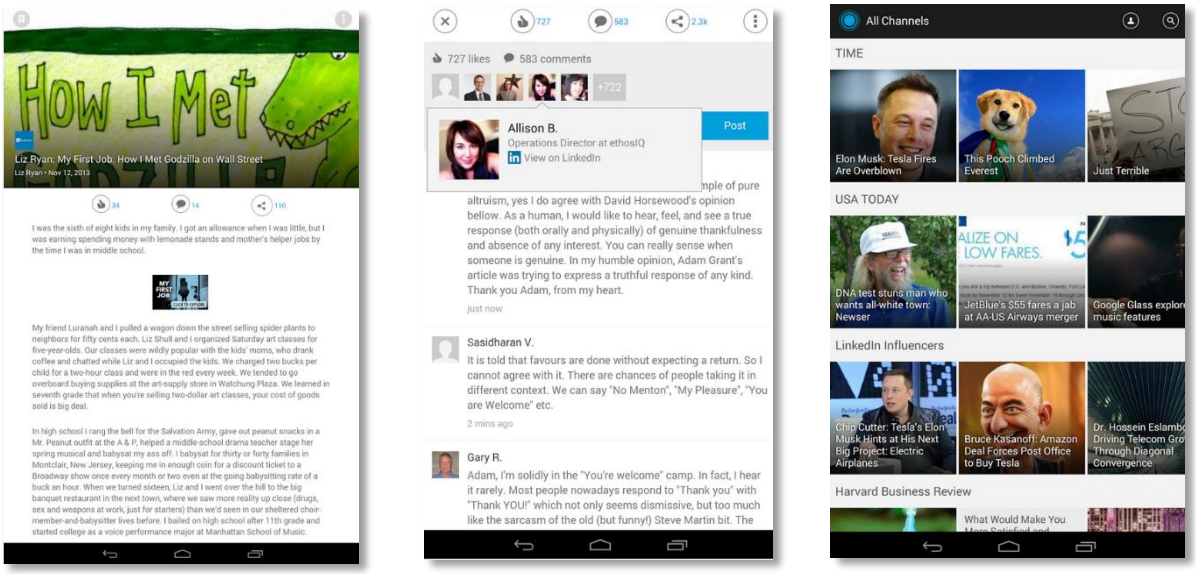
The clean design and ubiquitous availability of Flipboard are a key component of the applications success, and are two elements I will strive to facilitate in my own project. I will attempt to do so

without relying on custom design styles, opting instead to follow Google's Material Design standard, to guarantee a level of interaction familiarity with my app even for totally new users.

2.2.2 – LinkedIn Pulse

2.2.2.1 - History

LinkedIn Pulse (formerly Pulse) was recently acquired by LinkedIn for $90M [10] to become the mobile news platform of their brand. It has had over 10 million downloads since release.



The various views of the Pulse newsreader, showcasing its article view (left), social comments view (center) and article browsing view (right).

2.2.2.2 – USP & Key Functionality

LinkedIn Pulse differs from Flipboard greatly both in branding and feature set.

The application presents a minimalistic, fairly clean look which revolves around a more instant consumption content paradigm, where the user browses through a multitude of "Channels" at once (screenshot 3) then selects one to read (screenshot 2) and once finished begins again by returning to the "All Channels" view. The app leverages horizontal and vertical swiping gestures to allow the user to browse both sources and articles simultaneously without changing view, making for a very ubiquitous user flow and removing the need for complex navigation.

Feature wise, the application centres on a social theme, allowing users (under the banner of their LinkedIn profile) to comment, like and share articles with their network – this, paired with the viral proliferation of articles through more popular accounts, may be what lead to such a large user base for LinkedIn Pulse. Unfortunately, though unique and potentially appealing, I will be ruling out these features as they rely heavily on a social network backbone to provide the functionality, and as such are not very feasible features for a project with my timelines.

2.2.2.3 - Weaknesses

Notably however, my peers and I found the app feels much less premium and modern when compared to Flipboard, which I attribute to the chosen design- the contrasting black and white bars with minimal colour, while professional, do not convey the light airy feel of their competitors, or even apps which leverage Google's Material Design guidelines.

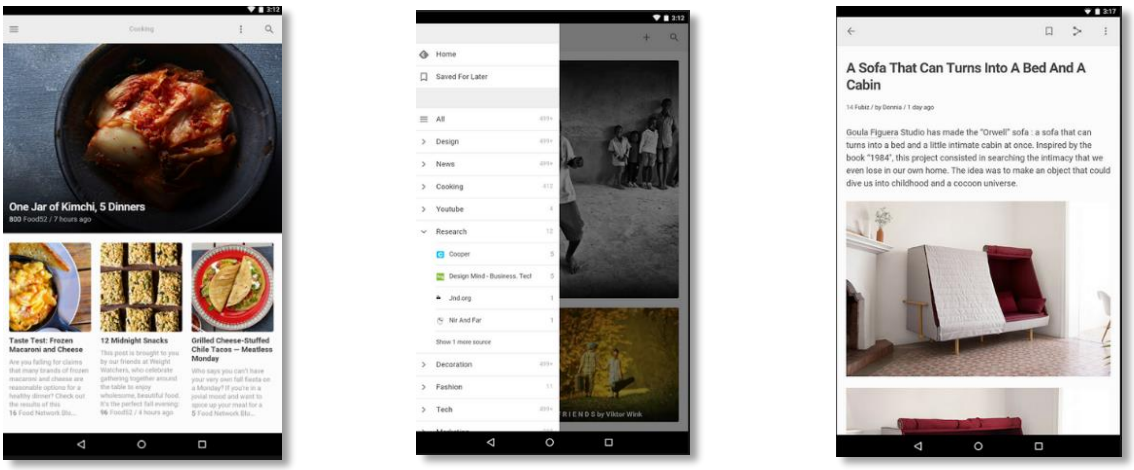
2.2.2.4 – Lessons Learned & Takeaways for my own project

The commentary of some of the user reviews on Play Store page for LinkedIn Pulse explicitly mentions the old/antiquated feeling UI not matching the look and feel of modern apps. This is something I'd like to avoid for my design. The interactive and engaging use of gestures to handle user actions however, is a real positive point for the app, and one which I would like to match in my design.

2.2.3 – Feedly

2.2.3.1 - History

Feedly is an app developed and managed by a much smaller team of people than the organisations who create the other apps in my comparisons [11], but still manages to claim an impressive user base of more than 5 million, which is why I chose to include it. The features and polish found in Feedly come with years of development, but on a much smaller scale, which allows me to contrast it to the above and hopefully use what I learn to create a set of realistic and attainable features for my project's specification later.



Feedly's source view, navigation drawer and article view respectively.

2.2.3.2 - USP & Key Functionality

Feedly curates some feed groups for people to subscribe to initially when they download the app, enables users to add their own sources and organise them in groups, and allows the user to share an article or open it in the browser for convenience. Lastly, it has a "saved for later" list which allows users to bookmark items they want to re-read later.

The app has a very visually appealing design which leverages a clever contrast of white and off-white to provide an aesthetic which, while somewhat bland, will not offend/put off any users. Navigation is handled ubiquitously using a gesture and menu recommended by the Google Material Design guidelines – the swipe-from-left "Navigation Drawer" pattern [12].

2.2.3.3 – Weaknesses

The Feedly feature list is relatively simplistic and non-unique in contrast to the other readers, which means it has to rely on it's clean aesthetic and simplicity for brand identity.

2.2.2.4 – Lessons Learned & Takeaways for my own project

The singular purpose of the Feedly app creates a nice minimalistic feel to the app, and one which welcomes users thanks to its simplicity. This is the initial impression I would like my users to have with my application. Additionally, the idea to allow users to save items to read for later is very useful, and something which I believe I can facilitate with relative ease.

2.2.4 – Summary

Seeing the reviews and comments from users on the Play Store commenting on the above features, and those of the other apps, has given me a great benchmark for which to base my project on, which will help greatly when I begin drafting a formal specification. I must be careful to remain realistic with my feature set and implementation choices due to my comparatively limited resources for time and development. Regardless, I will reference these apps and the lessons I learned from them to justify my specification and design decisions in the coming sections.

2.3 – Stakeholders

Having already defined my target audience in section 1.2, here I will outline my perceived requirements of stakeholders for whom the app must cater:

- Google
    - The aim of the project is to develop a prototype for an Android app using Xamarin which, with continued refinements, could be released on the Play Store in future.
    - Therefore, Google is a stakeholder, as they have a vested interest in the growth and quality of apps on their ecosystem, both on Android and in the Play Store.
    - Thus, Google's published recommendations and guidelines hold weight towards my decision making for this project, particularly those which are Android specific rather than Play Store specific.
        - For example, the need to declare any required application function privileges (for example, internet access or local storage access) is important as the feature will be blocked by Android without the declaration. The Play Store requirement to have image assets included with the app for listing on the store is less immediately important for the project, as it will not be listed on the Play Store upon completion, and the requirement can be fulfilled at a later date with ease.
    - Satisfying this stakeholder will thus enable the deployment of the app on Android devices, and in future could impact the potential growth of my user base by facilitating a continuous hosting and publishing platform – the Play Store.
- User testers
    - Although by necessity this group of stakeholders is a subset of "users", these individuals in particular play a key role in development and must be kept satisfied in a different way.
    - They have a vested interest in the app's ongoing development as they are giving up time to use and test the app, as well as provide useful feedback for its improvement.
    - As such, I should take heed and address their needs and concerns as needed, and take care to express how I value their opinion and input to ensure their continued participation.
- Project Supervisor
    - Similar to my User Testers, my project supervisor is a stakeholder in that they have a vested interest in the app's ongoing development, having had a say in the project's early decisions and development in a similar advice/feedback-giving fashion.
- Myself
    - As the developer, I am the fourth stakeholder group, as the time and effort I put into this project must meet my own aims and quality standards, as outlined in the Aims and Goals / Specification sections of my report. Doing so will help my own motivation and progress, which will impact the success of the project.

2.4 – Methods and Tools used

In this section I will outline the tools I have chosen to undertake the project along with the reasons/methodology for their use. This includes the Xamarin SDK, Visual Studio, Linq, MVC Pattern, OAuth2 and Team Foundation Server Online. The following sections outline the respective reasoning and thought process behind the selection of these tools.

2.4.1 – Xamarin SDK

I have chosen the Xamarin SDK as the base framework with which to develop my project.

This decision is driven by the following factors.

Firstly, I firmly believe that that cross platform applications fare better with regard to retaining and attracting users to the app in the long term, since it is more flexible, as potential users can install it on any of their devices. Therefore if I choose to continue to develop and later release the prototype in the future, it will stand a better chance of success by being a cross platform app. This ruled out the standard Java / Android Studio method of creating Android apps, as it is not cross platform compatible.

Secondly, the C# language also features an elegant implementation of Asynchronous operations called the "Async/Await" pattern, which will be described in detail in the Theory section below. This pattern was recently added to the C# language, and showcases the language's proficiency as a tool for developing modern, multi-threaded applications. This is especially useful due to the nature of my project – there are many places in which the app will be waiting for a response from a web or data source; these operations must be threaded and the calling contexts carefully managed in order to prevent race conditions, sluggish UI operations or even data loss. The "Async/Await" pattern facilitates this very effectively, and its use will allow me to pursue a more ambitious and interesting feature set as it speeds up development time of asynchronous code, thus allowing for my prototype to be further differentiated from similar apps.

2.4.2 – Visual Studio

Visual Studio will be my Development Environment (IDE) of choice for this project, specifically Visual Studio 2013 Update 4.

My choice of IDE has to conform to my use of Xamarin, and the only two options which Xamarin officially supports are Xamarin Studio and Visual Studio. I found this constraint acceptable given it did not conflict with any of my subsequent choices of tools. It also had to support my chosen Source Control provider, Team Foundation Server Online.

I opted for Visual Studio over Xamarin Studio due to the additional features which could help speed up my development time, such as UML and Class Diagram plugins as well as the very advanced real time debugger. This should speed up my iteration cycle time by enabling me to prototype more effectively.

I could have chosen to use the newer version of Visual Studio: 2015 CTP 5. However, I did not feel confident relying on a preview release to create the project given its importance to my degree. The risk of being unable to fix complications or instability in my project's time constraints was too high, so I opted for the older, but more stable 2013 release.

2.4.3 – Linq

Within the view and controller classes of my project, I will be leveraging C#'s Linq features, enabling the use of SQL-like query statements to find subsets of data. These can be filtered, sorted, and otherwise modified almost identically to SQL, and allows for data to be quickly and easily fetched for

display in the view. The benefits include being faster to develop with and only needing to maintain one (whole/unfiltered) set of data rather than persist various subsets for each view.

Although this is unorthodox in apps, I believe I am justified in using it because my project's data is static while displayed, and only changes when feeds update or the user navigates to a new fragment. In these cases I would have to update the subset of data being shown regardless, which means (theoretically) using a Linq query during the fragment's creation will perform adequately as well as enabling a simpler implementation in my views. I will reflect on this decision later in my report (Section 9.2) to confirm/revoke this theory.

2.4.4 – MVC Pattern

The Xamarin project model lends itself well to the Model View Controller pattern, where the Views and Controllers reside in the device-specific projects. In my project these will be in the "Freed.A" namespace and project, with Views being Layout ".axml" files and Controllers being their respective ".cs" counterparts in the Fragments folder, along with their helpers found in the Helpers sub-namespace.

The Models (data models and implementation logic) will reside in the PCL project ("Freed.Core") as they can be reused independent of target platform.

I chose to avoid naming my classes with typical MVC suffixes (e.g. "Controller") since the implementation is not typically MVC, as the Views and Controllers are separated from the Models by their project and namespace. Thus, for the purpose of this report, references to the "MVC" pattern are with regard to the implementation outlined here, rather than stricter interpretations of the pattern.

2.4.5 – OAuth2 & REST

For the Cloud Synchronization, I will be leveraging "OAuth2" & "REST" API's provided by Google. "REST" APIs appear to be the most generally available means of synchronising data to a Cloud Service, as well as the most widely-supported: Google, Facebook and Microsoft all have REST API's which developers can leverage. As such, implementing an architecture for a Google-REST based implementation should lead to a great deal of reusable code if I later decide to expand and support other service providers.

For authentication purposes, my research found that while it's possible (and simpler, implementation wise) to access Cloud services using a public, app-specific key [14], this limits me to the storage of application data to my own cloud resources. I have thus ruled out this method of authentication for my project as it isn't well suited to storing user data; it would require me to track and manage user accounts when syncing myself, which unnecessarily complicates the solution with an added layer of network and data structure complexity. Instead I will accommodate a user-authentication flow using a browser login page, to enable an OAuth2 handshake from which I can acquire an access token to access and use the user's personal cloud resources, specifically their Google Drive. After this authentication the interfacing and uploading with the Drive API is implemented via POST/GET HTTP messages to the REST API endpoint.

Although I might have to implement the file tasks myself (as the Google Drive.NET API is not supported in Xamarin), this should not be too difficult and Xamarin does provide a library ("Xamarin.Auth") which facilitates the authentication flow, which I find to be an acceptable solution.

2.4.6 – Team Foundation Server Online ("TFSO")

Finally, my choice of Source Repository will be Team Foundation Server online. This is because I already have a working development environment configured to use my TFSO account, and I did not find any significant benefits in moving to Git. The TFSO instance I am using is a secure private instance which only I can access, meaning I can safely store even sensitive data such as API keys in my code repository.

2.5 – Theory / Potential problems

This section outlines any theory or brainstorming done before beginning development to address basic functionality needs or potential development roadblocks. Each section provides a general overview of the challenge and proposed approach, and is later expanded upon with specific development plans in the Planning section. The key challenges I identified at this stage include:

- Modelling and Representing Data
- Efficient String Searching
- Stemming / Lemmatization for Word Suggestion Features
- Image Link Parsing
- OAuth2 & Cloud Synchronisation
- Using Asynchronous Operations

2.5.1 – Data Model Representation

The first problem I have to consider is how to handle the representation of feed data in my object hierarchy; essentially, what objects, properties and fields will I need to represent an RSS feed?

The varied nature and large amount of objects required to handle RSS data types poses an interesting challenge, as it would be very time consuming to create the vast number of properties and fields by hand, as well as requiring extensive knowledge of the RSS Schema and it's optional components. To overcome this, I believe I can instead generate an object model using Visual Studio's "Paste XML as Classes" functionality, by creating an example feed with as many optional feed elements as possible, to serve as an "extensive example" of sorts.

The generated object model will then serve as a starting point for the Data Model Representation, which can be augmented as development progresses with additional fields to store other, custom article-related data, such as whether the article has been read and the words (if any) which have triggered the kill-file filter.

Thus, as a starting point, I decided to find a few examples of RSS feeds to compile my "extensive example". Since the bulk of the information flow and scenarios (use cases) in my app will involve feed data, managing these objects efficiently is key, and I had to choose examples to represent various types of RSS feeds.

The various types of RSS feeds I identified and defined are of three types:

- "Full" feed
    - Definition: a news source which replicates content in full for the user to consume in any RSS reader
        - Chosen Example: Engadget RSS
- "Summarized" feed
    - Definition: a feed which contains only a snippet of the article and a link to the full content
        - Chosen Example: BBC News RSS
- "FeedBurner" feed

- Definition: an RSS feed created by the web feed management provider run by Google, and produces feeds which use a vast majority of the full RSS schema as outlined by on the Harvard Law RSS resource.[12]
    - Chosen Example: Creekworm Developer

Next, we have the user data object, which I have named “User”. It is the uppermost data structure in the object tree for my application, encapsulating the current feed data, the user’s preferences, kill file and cloud account information. To ensure data is correctly managed, I will use a singleton pattern to maintain a static user instance for use throughout the app, as for this application there should only ever be one user at a time and various functions within the app will require access to the aforementioned sets of data from a wide variety of code contexts, making it important that the user is publicly and globally accessible.

2.5.2 – Efficient String Search

One of the key challenges in implementing the kill file functionality is the ability to locate a substring within a string. Therefore I decided to research efficient string search algorithms, and I have outlined general details of a variety of these algorithms and their relevant pros and cons in the table below:

Algorithm	Short Summary	Pros	Cons
Knuth-Morris-Pratt[15]	Leverages word meta-information to infer where the next possible match may start.	- O(n+k) complexity - Skips previously matched characters	- Requires preprocessing (k) - requires fast evaluating heuristic failure function
Aho-Corasick[16]	Dictionary based algorithm – uses a finite state machine known as a "trie" to traverse during pattern matching.	- O(n) complexity - Matches all patterns simultaneously	- must be substantially modified to avoid substring matches
Rabin-Karp[17]	Uses hashing to find any one of a target string list in a string. Exploits the fact that if two strings are equal, their hash values will also be equal. Hash function greatly affects efficiency.	- Efficient for cases with large target-lists (large kill files) - simple implementation - can be very efficient with a good hash function	- Worst case time performance of O(nm) where m is the combined target string list length. - Relatively weak efficiency – does not skip target string length, unlike Boyer-Moore[18]
Bitap[19]	Uses precomputation of a bitmask from the target pattern to enable bitwise operations to perform the string search on the mask.	- bitwise operations are very fast - predictable O(mn) operations results in predictable runtime regardless of pattern or text structure	- performs best with patterns of length less than the system's WORD size

It is important to note however, none of these algorithms deal with the issue of detecting word separation – they are designed to detect strings within strings, nothing more. As such, I will select one to implement, and compare it's performance to the alternative, platform-provided string search which C# and the CLR (Common Language Runtime) provides: string.IndexOf(string)[20]. This built in search suffers the same issue of "word terminator" detection. Therefore, my selected implementation must outperform IndexOf() before it becomes worth investigating further, at which point I will have to devise and outline a means of handling the word terminator problem.

I believe the word termination problem is not a trivial one, since words can be "terminated" in the English language by a variety of means- certain punctuation, string termination and whitespace must all be accounted for, which is challenging both from a grammatical and algorithmic perspective.

Additionally, the user may wish to include feeds from other languages, which may have differing word termination characters and thus be substantially more difficult to accommodate.

To address this, I have researched a third, alternative option- Regular Expressions (Regex). Regex is a pattern matching tool which uses a query string language to define a target search pattern, and can then be used to find "matches" of the pattern within a string. Regex allows for word terminators to be handled as part of its query string language using the "/b" notation, for example, to match "Cat.", "Cat ", and " cat.", the query string "/bcat/b" and all cases would be accounted for. This is the simplest implementation option, but I must be careful regarding how I construct my query terms, as the speed of the string match is greatly affected by poor Regex syntax construction, particularly when an expression requires optional qualifiers or alteration constructs [21]. I do not anticipate having to use these, but it is something I should be aware of.

2.5.3 – Stemming / Lemmatization Algorithms

An idea I had discussed with my supervisor to improve the user experience as part of the Kill File Editing process was to present the user with a set of variant words after they had entered a word to ignore, with the aim of showing potential alternatives to also exclude. For example, if the user entered the word "computer", they could also be given the option to exclude "computers", "computing" etc. This seems very challenging at first glance, but potentially worthwhile, so I decided to do further research.

(F)	Rule			Example		
	SSES	→	SS	caresses	→	caress
	IES	→	I	ponies	→	poni
	SS	→	SS	caress	→	caress
	S	→		cats	→	cat

An example of Lemmatization logic rules and their applications.

The requirement for this to be feasible would be to find a way to generate the semantically linked children of the base (or "stem") word. This type of algorithm is referred to as Lemmatization, and it categorized as an NLP (Natural language processing) problem. The application I describe is the inverse to typical Lemmatization, which generally attempts to find a "lemma" from a lexeme, i.e. to take a derivative of a word and find its root. For this feature, I would require the reverse functionality.

The process is very complex, and generally involves tasks beyond the examination of the word itself, and can require context and grammatical input to be accurate. This is due to the potential for stem words to be common to many semantically distinct "lemmas", or variant words. Additionally, various lemma are very hard to generate, since they may contain no visible shared stem, but be semantically linked regardless – "is" and "are" for example.

The set of complex grammatical and linguistic challenges to overcome in order to accurately implement my own fully featured lemmatization algorithm fall far outside the possibility of realism for this project's timeline. With that said, I was keen to investigate similar or alternative routes, such as implementing an open source algorithm to enable the functionality or creating a very basic version of the algorithm which could leverage additional user feedback to be made more useful.

To that end, I discovered an open source implementation called "LemmaGen" [22] which supports the English language and requires no contextual input (which would rule out using it, since my user input for this feature is by design a single word only- there is no context to include!). However, unfortunately it is only a Lemmatization library, and does not provide any functionality for inverting the process. It may be theoretically possible to create this functionality by using the creating a map

of all possible input words and their outcomes, then inverting the map, but I do not believe this is ideal, as many words the user may enter are not dictionary-defined words ("iPhone" for example) and thus any implementation not based on grammatical rules would be unsuitable.

For a basic alternative, I had considered simply showing common English suffixes appended to the user's input. This would be algorithmically fairly trivial, however I decided against it. The user experience benefits of this feature hinge entirely on its usefulness, and showing a pop up dialog every time the kill file is edited only to present options which are potentially nonsensical could irritate the user, and that risk negates any value the feature may have.

Based on this research, I have excluded this feature from the project, and will enable the user to filter using specific words only.

2.5.4 – Image Link Parsing

Although parsing the image links from articles for viewing appears challenging, the Android TextView contains a built-in solution which facilitates this. It requires me to implement my own "ImageGetter" to download the images using the android "IImageGetter" interface, and then parses a body of HTML content to handle the downloading and viewing of images, all within a single TextView. I will attempt to implement this, however should this fail I believe I can solve the issue of Image link detection myself if necessary, provided that only valid HTML is considered for input.

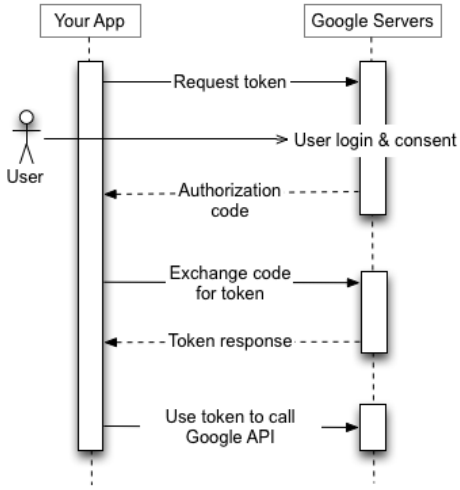
If necessary this will be discussed further in the design section (4.2.2) of the report.

2.5.5 – OAuth2: Access & Refresh Tokens

As mentioned in section 2.4.5, I will be using OAuth2 to facilitate the authentication of users who wish to sync data to their cloud account.

While the initial request should be fairly simple, as there are libraries and documented patterns to achieve the "handshake" flow required, I will also have to account for the expiry date of these access tokens. The tokens are valid for a set period, after which I will have to use a Refresh token to acquire a new access token. Since the Refresh token is static, this presents a challenge as it means I must provide long-term storage where it is readily accessible for when the access token expires, which necessitates the implementation of functionality allowing the serialization of data to the local disk.

Aside from this, I do not anticipate any other implementation concerns.



The Google OAuth2 handshake pattern.

2.5.6 – Cloud Synchronization via REST APIs

Once authenticated, I will be able to use the received token to access Google Drive REST API endpoints. The Drive REST API works on a principle of "scopes" – a request for authorisation includes a desired "scope" which it wishes to leverage, and the user verifies and accepts this before the application proceeds.

One challenge which must be addressed for reliable cloud synchronisation is the reliability of the synchronised file. If the user can accidently delete the app data, or other applications can modify the data, there is an inherent risk of unsanitised data entering the application flow by being synced and deserialized.

To address this issue I will be using the Drive API's "AppData" scope, which allows me to store data in the user's google drive in a hidden folder exclusive to my application. This prevents worries around users accidently deleting the app data (since it is hidden) as well as any concerns regarding other applications modifying the data, since only my app's API key can be used to access the application folder for my app.

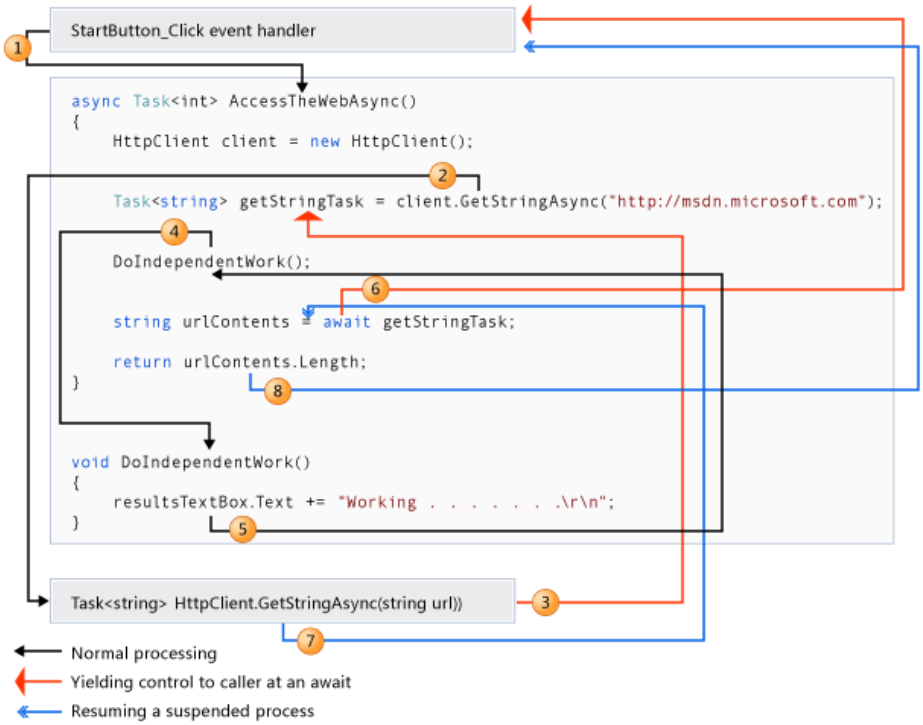
The Drive Rest API methods I will be utilizing are the POST and GET HTTP messages to allow for download and upload of files, as well as the DELETE method for file deletion (to delete stale data before a new file is uploaded).

Additionally, in the POST message type I will be using a format known as the "Multi Part" request to allow for the upload of metadata and data at the same time. This is in line with Google's recommendation [23] to do so for smaller file sizes, which my user-data will be, as it is mostly text.

2.5.7 – Asynchronous operations

The nature of Asynchronous operations in C# as of .NET 4.5 is a fairly elegant one, and one of the reasons I chose to use Xamarin for this project. For context, I will compare the implementation C# uses (the "Async/Await" pattern) to the standard practice for Java applications, since it is the more standard language used when developing on Android.

"Async/Await" can be understood best as a means of using the Task Parallel Library in .NET to simply write and execute long running operations in another thread, and easily access the results once they are ready. The most important part of this is understanding how the control flow moves from method to method, which is shown in the illustration below:



From my experience in past projects I have found this form of asynchronous tasking is much simpler to maintain than the android/java pattern of implementing the IAsyncTask interface and using multiple methods to handle the progress/result:

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

As such, it will be my default form of asynchronous task execution for this project, and will enable me to be confident in planning for the UI thread to be kept responsive, which will be a requirement in my specification.

2.6 – Conclusion

Having outlined the goals, audience and scope of the project, and taking into account the potential challenges in achieving the desired functionality, the next phase is to define the detailed requirements for my project's features and scope.

3 – Requirements / Features

My list of feature requirements is split into two categories; “critical” requirements, which represent features mandatory to the basic functionality of the app as defined in the Scope (Section 1.3), and “value-adding” requirements, which represent features that do not compromise the app’s usefulness and goals if left unfinished, but add a unique or useful feature for users if included.

The requirement numbers are later referenced in order to perform Correctness Testing in section 7.1.

3.1 – Critical Requirements

1) Create a functional UI with views and navigation ability for:
    - Editing the Kill File
        - Constraint: only needs to be able to add and remove individual words.
    - Viewing a specific source’s articles
        - Should be made available from the main navigation.
    - Reading an article
    - Editing user settings
        - Constraint: only Critical Feature settings required
    - Adding and Syncing to a Google Drive account
2) Fetch RSS XML data from a Web resource
    - Constraint: support only the required components of the RSS 2.0 specification[13]
3) Allow users to add and remove URIs to their subscribed feeds
4) Serialise/Parse this XML data to application objects
5) Filter out RSS data which contains specific, user-defined words (Kill File)
    - Constraint – critical requirement matches the exact word, not a substring (eg “cat” should not match “Concatenate”
6) Allow the user to modify the list of words to be filtered (Kill File)
    - Constraint, only add or remove single items, not direct file editing
7) Authenticate with Google Drive’s REST API
8) Synchronise user settings, data and state to their personal Google Drive account via HTTP REST APIs.
9) Allow the user to refresh/get new RSS data.
10) Load and display images from content
    - Constraint: Show images in contextually correct places within text
11) Allow the user to open the Link included in the RSS data in their default web browser.
12) Be performant and responsive by leveraging Tasks and the Async/Await pattern
    - This will be evaluated by the number of skipped frames on the UI thread and any delays in user flow while using the app (waiting for data etc.)

3.2 – Value Adding Requirements

13) Use material design guidelines to improve and modernize the functional UI prototype
    - I will leverage ideas and concepts from the peer-app research I documented in the section above to achieve this.
    - This requirement is based on lessons learned from competitor research: specifically, I would like to avoid the dated feel of the LinkedIn Pulse app.
14) Add a view to see the latest articles from all sources
15) Implement the android Refresher View to enable gesture based refresh ("overscroll" swipe down to refresh feed items)
16) Enable support for a broader range of the RSS specification's optional components.
17) Serialise and save the user's data to a local file on the device, and load data from this file for use when no internet connection is available (offline mode)
18) Load images asynchronously, enabling the user to read the article while images are fetched.
19) Serialise/Save images to disk to prevent re-download of any images the user sees while browsing content
20) Enable formatting and interaction with content using HTML tags (where the data source allows it)
21) Allow the user to add items to a "Reading queue" for later reference.
    - Based on Flipboard's popular "custom magazine" feature
22) Allow the user to track which items they've read
    - Provide visual feedback in lists to show this, and serialize it with other data to cloud and local storage
23) Allow users to mark items as unread/read
    - Should be possible via context menus during reading and list views
24) Flexible Scaling UI : The UI should scale to fit android devices of any screen size and resolution

4 – Design

This section is divided into User Interface / Visual Design (Section 4.1) and Systems Design (Section 4.2). The User Interface / Visual Design section houses the design and iteration of the UI Fragments and the app as a whole, while the Systems Design subsection outlines the structure and approach to implementing the various feature requirements as defined in section 3.

4.1 – User Interface / Visual Design

The wide target audience I described in section 1.2 presents various difficulties from a branding and UI design perspective, as the creation of a unique brand without a particular user subset in mind can be daunting.

To overcome this, I will leverage the research I performed on the various competing apps, especially regarding the “Feedly” app. The minimalist approach they have taken to designing their application’s look and feel is something which was universally accepted by the test group I showed the three applications to, and I believe this to be because the design elements were simple and clean, thus leaving nothing objectionable to any particular taste. In essence, the goal will be to make a UI which is acceptable to everyone, rather than making it specifically attractive to one subset of users and potentially alienating another.

Before I begin to discuss the concepts, prototypes, and final designs for my UI, I should explain the basic Android UI principles I will be using, especially the concept of “fragments”. Fragments in Android refer to sub-layouts which can be written in XML, and are page elements of varying size and content. These can then be individually maintained and reused in various contexts by “inflating” them within an activity, allowing a page to be comprised of multiple fragments and for fragments to be persisted and replaced as needed for app navigation. This creates a fluid and apparently seamless UI as fragment transitions and animations are handled by the android framework. The implementation of this is trivial, and simply requires the creation of AXML elements representing portions of the view rather than the whole view, which are then inflated using Android’s built in “FragmentManager”.

The user interface as a whole consists of 5 primary fragment types:

- Splash Screen Fragment
- Article List Based Fragments
- Edit List based Fragments
- Settings Fragment
- Navigation Menu Fragment

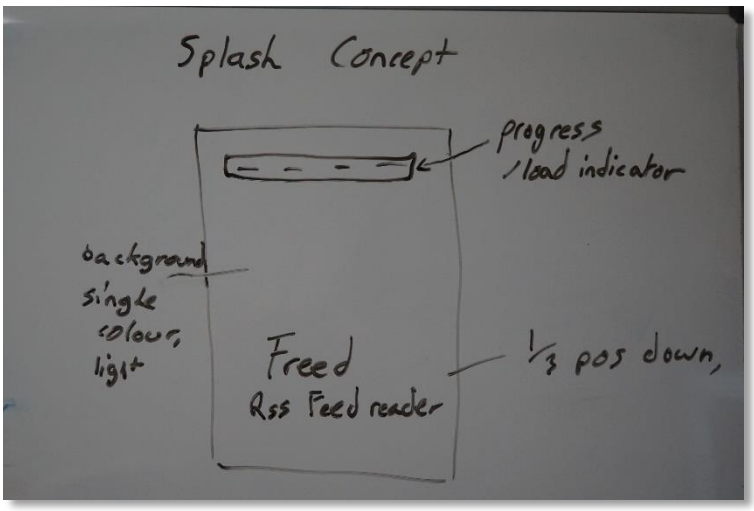
For each of these, I will illustrate the basic design and concept sketches, followed by the prototype implementation, and then any further refinements that were made along with justifications and feedback that were relevant to the decisions on each page visual.

4.1.1 – Splash Screen Fragment

The primary function of the splash screen is to provide visual flair along with a loading indicator to illustrate to the user that the application is starting up.

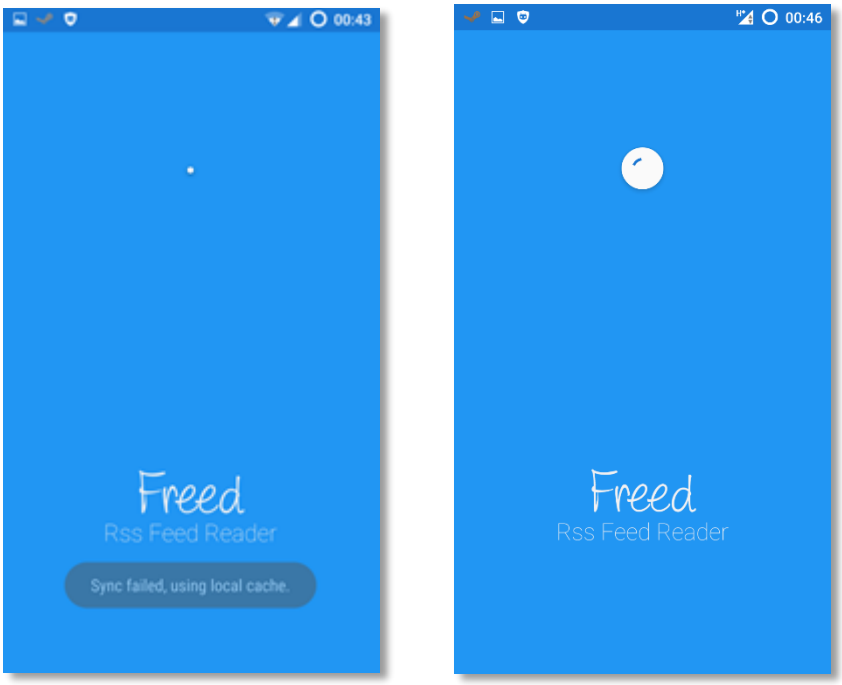
In my conceptual sketches for the splash screen, I wanted the user to see a coloured visual along with a loading indicator, simply to illustrate the app's status while fetching / desterilizing feed data.

I also wanted to show the branding for the app, a logo or title of some sort, along the bottom of the screen. This is mostly an aesthetic choice, but one that my users later mentioned they liked. I believe this to be a result of the more balanced page proportions compared to having a page dedicated to the loading bar.



Splash Screen Concept from early February, pre-development prototype.

My early prototype did not have the architecture in place to indicate when the app had finished loading feeds, and as such there are no development prototype screenshots of this fragment. The feature was during a usability test user suggested it during the Material Design UI iteration tests.



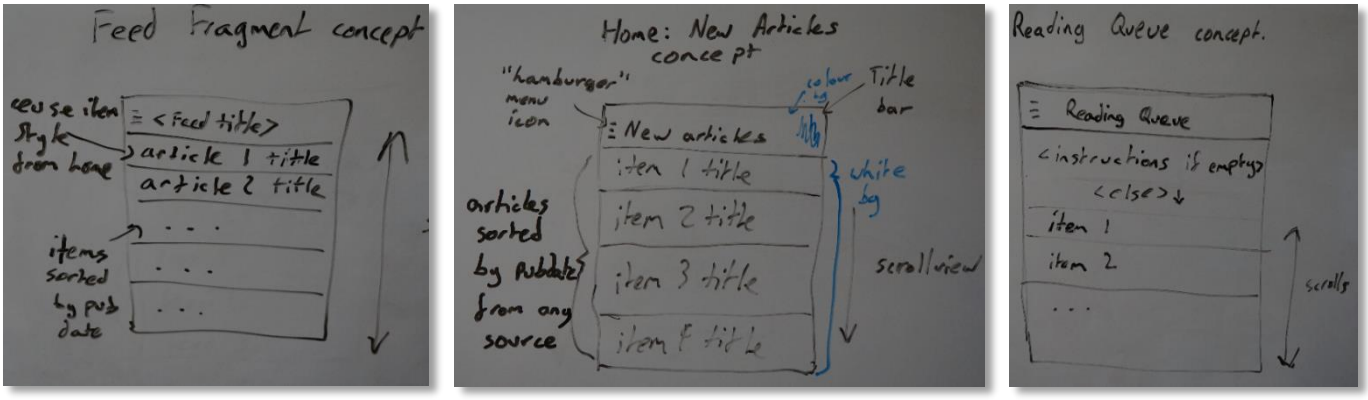
The finished, animated splash, spaced deliberately to allow for toast notifications during start up.

In the later development stages of the project, I had revised the implementation to use a Material Design progress indicator that was circular in nature, as well as applied brand colours and logo, resulting in a very polished splash screen, thus fulfilling the primary purpose of the fragment:

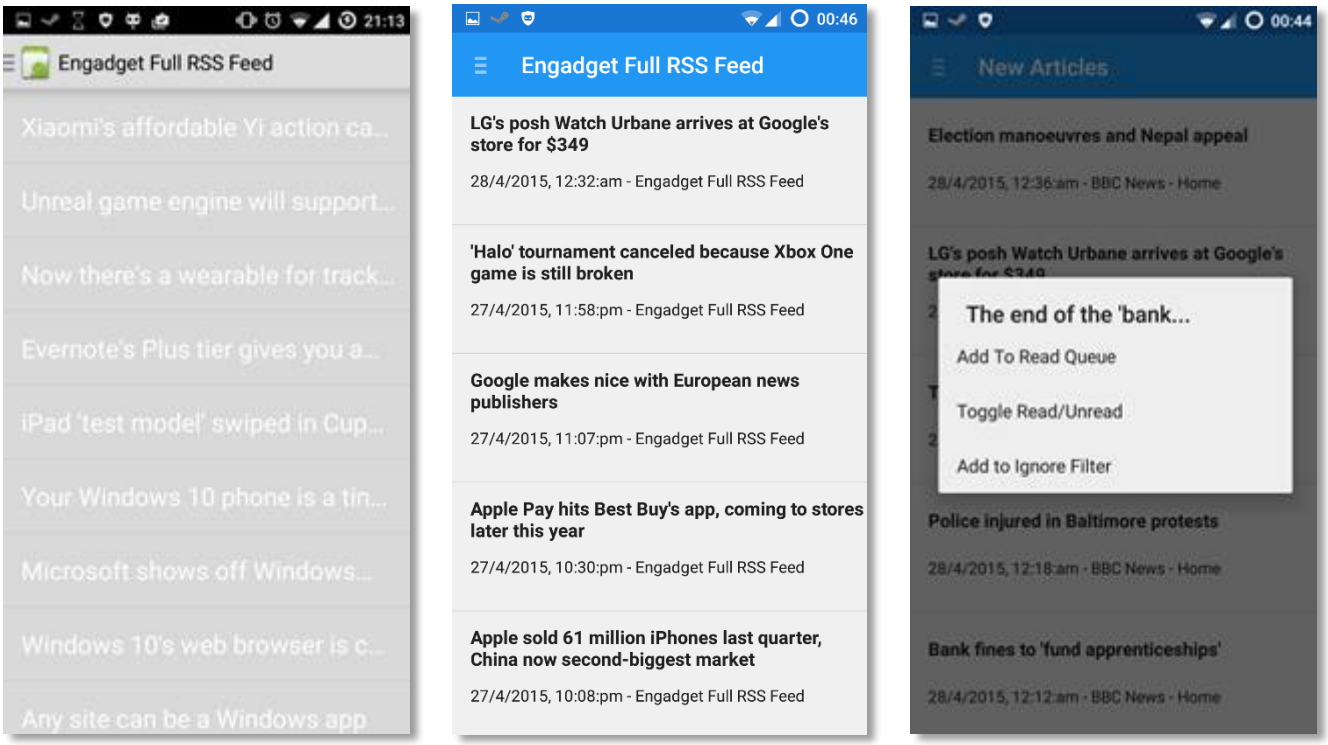
4.1.2 – Article List Based Fragments

The article list based fragments in my app refer to the fragments which contains a list of articles from a feed source, i.e. the New Articles, Feed and Reading Queue fragments. The primary purpose is to facilitate the display of this list of articles for the user to select and interact with.

The concepts outline initial designs for the article-list based pages, including interaction in scrolling and article item style, as well as the sorting of items (by publish date) and the page title. In general, fragments with articles enable scrolling vertically to browse, a long tap to open a contextual menu, and a "hamburger menu" icon in the title bar to allow users to tap a button rather than swipe to access navigation if they prefer. These are fairly standard android paradigms, which should make my app intuitive to learn and use for anyone familiar with the android platform.



These concepts received some changes while iterating on user feedback from usability testing sessions to complete the finished version: many people I showed the prototype to ask why the publish data and news source/publisher weren't also shown on the item, so I added these elements.

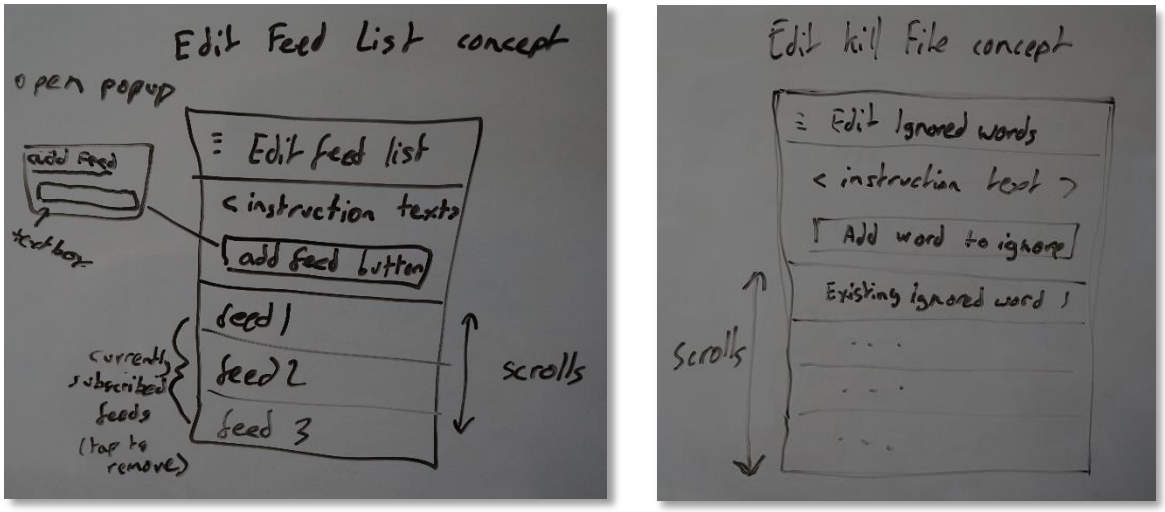


A comparison of the prototype article-list fragments (far left) to the finished version (centre, right).

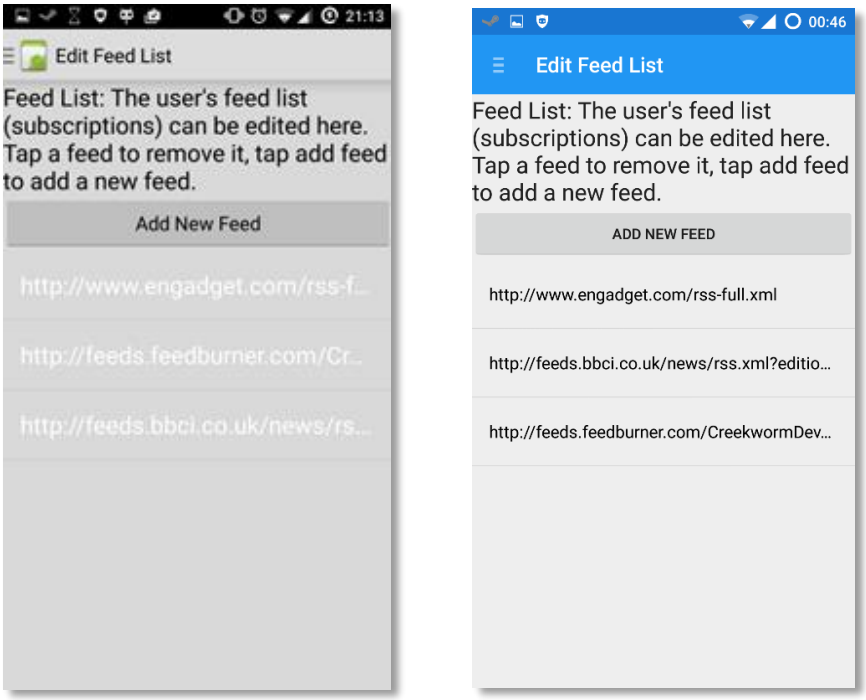
The items also visually indicate once they have been read, and have contextual menu options by long-tapping on them. Thus, the basic requirements of the fragment are met.

4.1.3 – Edit List based Fragments

These concepts outline initial designs for the fragments whose primary purpose is to allow the user to edit or manipulate collections of data, primarily the subscribed feeds list and the kill file "ignored words" list in this app. The design incorporates a small area for an option instructional text block, to ensure the user understands how to use the fragment correctly.



The edit list based fragment prototype looks very similar to the concept, and the prototype received no suggestions for improvement during usability testing. As such, in the final version it looks virtually identical, although notably improved by the styling and branding being applied.



The prototype and final edit list style fragments.

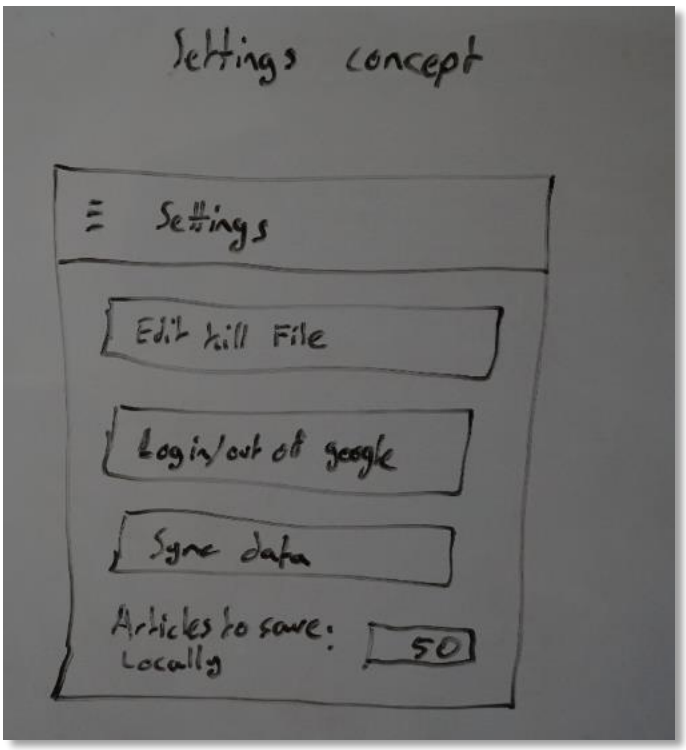
By using a ListView to house the list items and facilitate removal interactions by tapping, as well as including an add button to enable addition of data, the fragments' design is fit for purpose.

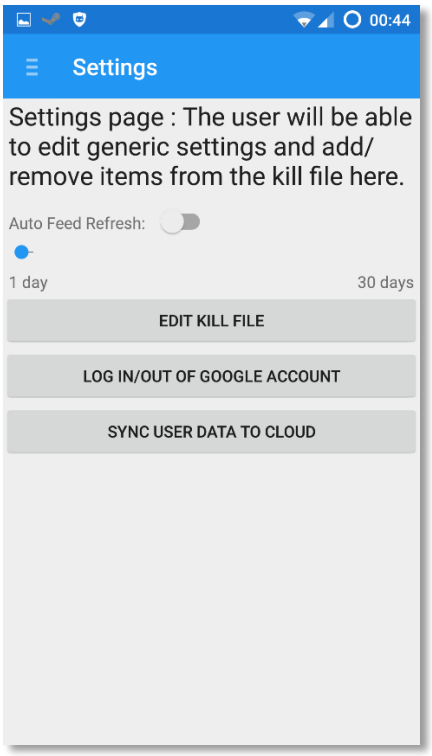
4.1.4 – Settings Fragment

The settings fragment design, which serves the purpose of facilitating the editing of various user settings. This is the last distinctly unique fragment I had to design, and a relatively sparse one – since I did not want to overcomplicate it and include visual elements for features which may or may not be added (specifically the Value Add Requirements), I opted to only include settings for require features in the concept, and to use basic UI elements to trigger their interactions (a set of buttons).

This way, if and when the prototype progressed to include any Value Add features which required their own settings, I could simply add buttons as needed without having to rearrange the other UI elements in the fragment.

This also applies to any items which can be grouped into a row- the layout is a linear list of view objects which stack vertically, so as long as the setting can be grouped into a rectangular layout itself, the same principle applies. This essentially excludes lists such as the Kill File and Subscriptions, which is why they have their own dedicated fragments.





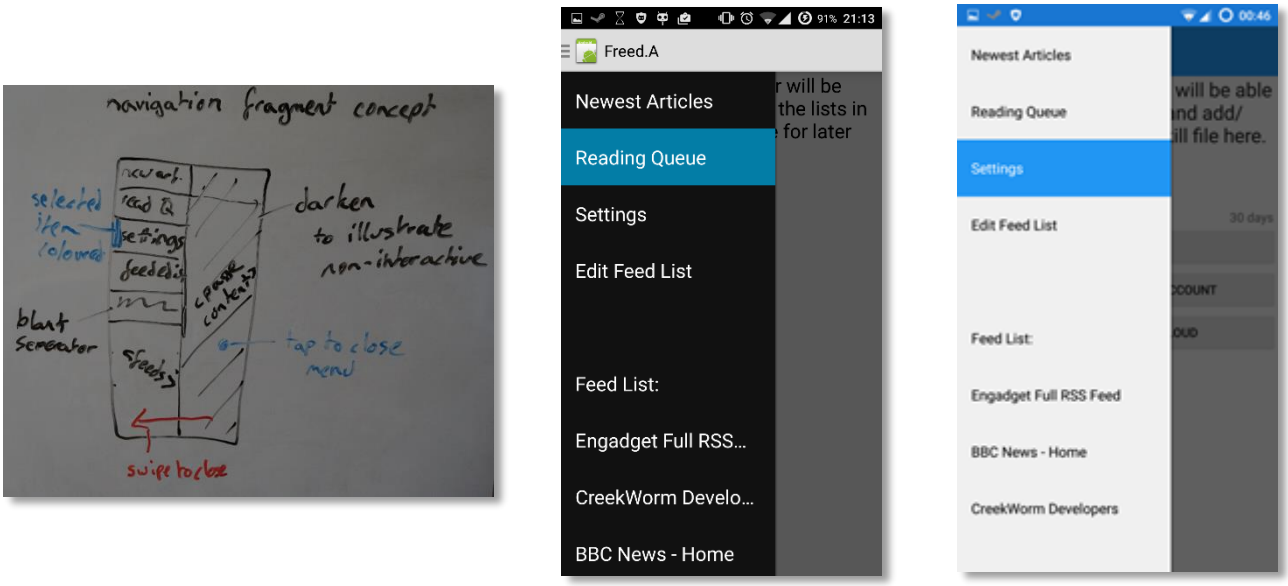
Left, the concept settings fragment, and right, the final version of the settings fragment, which suits the purpose of the fragment as initially described.

For brevity's sake I will not go into more details for the settings fragment, as it remained virtually identical to the concept throughout development.

4.1.5 – Navigation Menu Fragment

The navigation fragment concept is to enable the user to navigate between the primary fragments off the app, in this case the Newest Articles, Reading Queue, Settings, Edit Feed List and various feed fragments.

It is the most persistent in the app, being permanently available to the user through a swipe gesture and a “hamburger” menu icon in the title bar. This paradigm was introduced by google as part of their design guidelines for android apps as the “Navigation Drawer”. I selected it because it was especially well suited to handling navigation between multiple “top-level” views (views which the user should be able to access from the home screen) – since I wanted the user to be able to immediately navigate to any of their subscribed feeds upon app launch, it seemed ideal to use a Navigation drawer which contained the user’s feed sources as a variety of “feed fragments” to navigate to. The concept mock-up I created illustrates the key elements of the navigation drawer, though implementing it is much less complex as there are built in facilities to achieve the effects illustrated.



The navigation fragment concept, prototype and final implementation.

The only real change from prototype to final version is found in the layering of fragments, which makes the navigation fragment overlap the content fully in the background. This is part of the Google Material Design adjustments I applied when branding the app, as the guidelines illustrate that new fragments and objects should overlap content in a fashion which resembles a stack of physical objects in a space. While testing the app I felt that leaving the title bar uncovered broke this illusion somewhat, and adjusted the navigation menu fragment accordingly to have a larger height and start from the app’s origin, rather than the toolbar’s height value.

All in all, the navigation drawer is well suited for its intended purpose.

4.2 – Systems Design

This section outlines the systems design for various features implemented in the app. Where features have already been mentioned in the Theory section of this report, I will summarise the design with reference to the theory analysis, and prioritise highlighting any changes that were made for brevity.

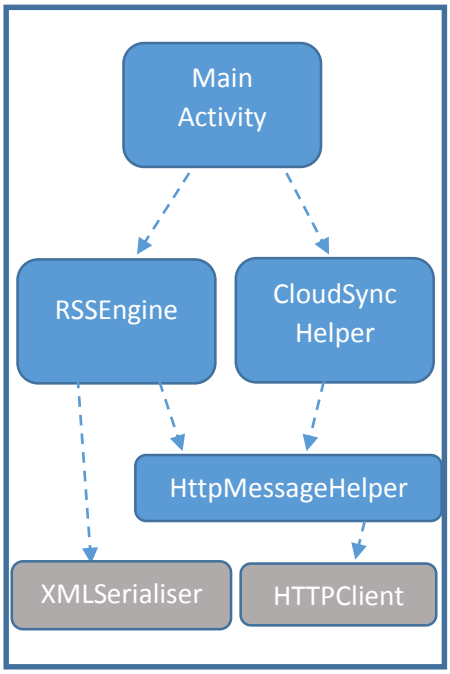
This section includes descriptions of the following features:

- RSS Fetching and Parsing
- "ImageGetter" and Image Downloading
- View representation and data sorting
- OAuth2
- Local Data Storage
- Cloud Synchronisation to Google Drive
- Kill File & String Search Engine

I have omitted the explanation of the fragment UI systems design, as these are self-explanatory given the context in the User Interface / Visual Design section.

4.2.1 – RSS Fetching and Parsing

Of primary design concern here is the fetching and parsing of RSS Feed data, and the object representation of the parsed data. I will be implementing RSS Fetching in my app is part of the Freed.Core reusable library. A flow diagram of the design overview is pictured on the right, where grey items indicate built-in .NET classes.



For the HTTP request, I decided to design a method which could later be reused to synchronise with the Cloud service, since both of them would use basic HTTP Get requests in some fashion. This follows the SOLID principle of having classes with only one responsibility each, and lead to the creation of the HTTPMessageHelper class. The class will use HTTPClient and HttpRequestMessage objects from .NET to retrieve data from the web, and include Using statements to ensure they are disposed of correctly to prevent memory leaks.

With the retrieval method designed, the next step is to devise a way of parsing the RSS data within the HTTP response. Since RSS is functionally adheres to an XML Schema, I decided to use .NET's built in "XMLSerializer" to parse the RSS data into custom objects. This makes the RSS fetching process a fairly simple two step process- a HTTP request is made to the Feed's URL to retrieve the body of content, which contains XML. This XML is then deserialized into objects of type "rss".

That leaves the RSS object. The design for this object must support the three required items from the RSS schema, and should also support any optional items from the schema which existed in my example data feedburner feed. I chose to implement the optional elements found in the feedburner feed because the examples provided by the RSS 2.0 specification were not complete in their inclusion of optional elements, and it fit my specification goal of supporting some of the optional RSS feed data elements.

The items will also contain custom data elements which are not deserialized from the XML, but rather generated by my own code and appended to the items on collection to provide various functionality, such as the filtering of RSS items using the Kill file. I decided that appending the details to the items themselves was favourable to encapsulating the items in an object with additional properties, since the types I needed to add were primitive in nature, and I did not wish to overcomplicate the "rss" object since it already fairly large. The new properties extending the RSS class are flagged to be ignored by the XMLSerialiser, which should preserve the reusability of the code for other applications if needed.

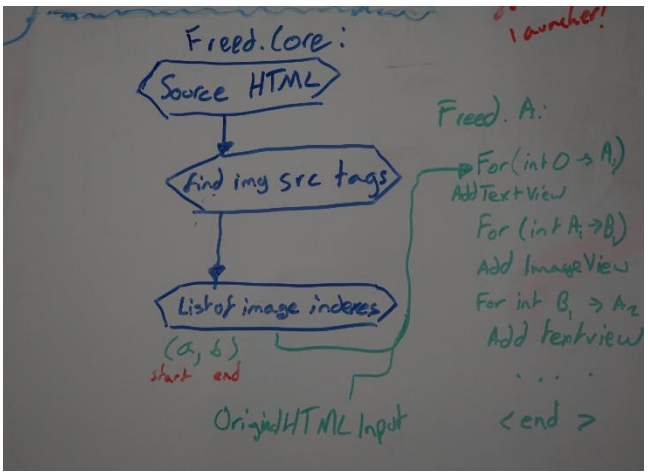
4.2.2 – "ImageGetter" and Image Downloading

The primary goal for this function is to enable images to be downloaded and shown in the correct contextual place.

The "ImageGetter" class' initial design and prototype originally leveraged the Android paradigm of implementing the "IImageGetter" interface, and passing an instance of this object to a TextView along with the HTML of the page in question. The TextView then creates ImageViews to host the images as needed whenever it encounters an image tag within the HTML body.

Theoretically this would accomplish the primary goal of the method. However, I was unable to successfully implement this in an asynchronous fashion, which meant using it would compromise the performance of the app greatly as the UI thread was blocked until images were downloaded. This problem is explained in detail in the Development Problems section, and I pursued an alternative, original solution instead.

Although my design will require a far more involved development effort, I am confident in its feasibility, and believe that (with some testing) the performance and reliability have the potential to be good.



My brainstorming of a workflow for parsing HTML for images and building a ViewGroup for them.

To start with, I thought about how best to detect image links within a HTML body.

The "img" tag is used for images in HTML, and the required attribute "src" determines the resource to be displayed. This means my first challenge is to devise a means of finding all instances of the "img src" attribute in the response body. There are a few considerations to make however;

- The ordering of attributes is not guaranteed for the img tag in the HTML specification, meaning the img tag may appear followed by another attribute before src. This rules out a simple string search for "img src"

- Whitespace does not affect HTML tags, meaning “img src = <link>” and “img src= <link>” are both equally valid tags, and my algorithm must handle this.
- The closing of the image tag is not required in HTML, but is in XHTML. It can also take multiple forms, being either self-closed (<img />) or distinctly closed <img></img>. This must also be handled, as it rules out finding the tag using a simple “img” search.

Lastly, I decided to handle only valid HTML and XHTML inputs, since that was the scope of the “ImageGetter” which my solution is replacing, and to aim for a linearly complex algorithm, since I see no reason why the HTML should need to be parsed multiple times in its entirety just to find image tags. (Note: It may have to be reparsed to construct the ViewGroup, but this is outside of the scope of this algorithm’s design.)

With this scope concretized, I begun to design the architecture of the implementation. Firstly, if we must we must parse the HTML input to find our images for display, this can be handled in Freed.Core, since it is platform independent. This will be done in a class called HTMLParser, so that if other HTML parsing algorithms are ever needed, they can be housed in the same class – it has a Sole Responsibility and is Extensible, as per the SOLID principles of class design.

This brings us to our second part of the implementation, the Android-specific portion which constructs the image and textviews to show the content in the layout. This class must be housed in the Freed.A namespace, as it deals with Android specific view types.

This class, which I will call TextImageViewCreationHelper, will contain only one major method, “CreateViewGroups”. CreateViewGroups should leverage the HTMLParser to create a series of View type objects, which can then be added to the layout as a single ViewGroup.

4.2.3 – View representation and data sorting

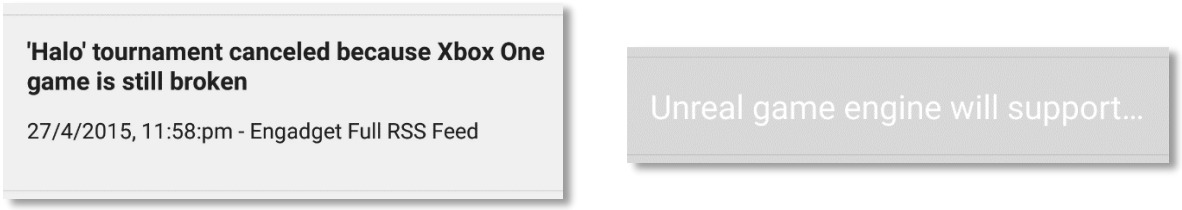
This section’s topic is used in almost every view fragment, so I will use the new articles fragment as a general example for the sake of brevity. The explanation of the design can be applied to any fragment which shows a list of feed data.

In order to represent my views in the fashion outlined in my specification, three things are required:

- A view object capable of containing the collection
- A means of displaying the individual items in a sub-view of the collection
- A subset of data which is to be shown in the view

For the View object, a typical Android ListView would suffice. Android provides a type of fragment which embeds a ListView by default, which is called a ListFragment. This provides the collection container.

Next, the individual item layout and display. To meet the specification “value-add” requirement of showing the user a read/unread state, this item layout must be of a custom design. It also necessitates altering aspects of the individual item container, as well as access properties from the item itself. Both of these are possible using a custom adapter by overriding methods, so this design seems suitable. The design I have chosen to address this is to use a Custom Adapter for the fragment’s ListView, as it is the only means by which all the requirements of the feature can be addressed. I believe the best design is to override the included ArrayAdapter class, with an additional type requisite to ensure it is only used for items which are article types. The adapter implementation will have to be housed in the Freed.A namespace, since it is Android specific. Thus the means of displaying data is designed, and can be implemented.



The altered individual item container (left) vs the standard ArrayAdapter item container (right).

This is an elegant design thanks to ListView's adherence to the Liskov Substitution Principle, which allows me to rely on the fact that the ListView will still function when given a subtype of the adapter instance. Seeing the SOLID principles applied in the Android codebase helps to validate my efforts to adhere to them.

Lastly; subsets of data. As noted in section 2.4.3 (Methods and Tools) I decided early in the project to use Linq to implement the bulk of the work of the controller in the MVC pattern; which includes this requirement: to sort and select segments of data to use in response to various view commands and inputs. The query can be executed upon the fragment's creation, but specific details of the Linq queries will depend on the fragment, and thus this will be discussed later in the Implementation section. This handles the design for the data subsets, and completes the feature design for the majority of the UI.

4.2.4 – OAuth2

My cloud synchronisation to Google Drive requires the use of an OAuth2 token to function, which necessitates the implementation of an OAuth2 flow, both from a systems and UI perspective (since the user must first login to the third party resource and confirm the permissions for the token to be granted). Since this login will be "persistent" – i.e., the user should not have to log in every time the sync feature is used, I will also need to handle a refresh token.

The authorization will be for the Google Drive "Scope" of the App Folder. This means the token will be valid only for operations pertaining to the storage of my app's data in its own dedicated (hidden) folder in the user's Google Drive.

My design for the implementation of this feature can be broken down as follows:

- Facilitate POST and DELETE message types
    - I will extend the HTTPMessageHelper class to handle these.
- Show a Web Login page for Google Authentication
    - I will use a third party library, Xamarin.Auth, to handle this
- Retrieve the token once sign in has occurred
    - I will use the properties in the User class to store the access token provided by Xamarin.Auth
- Store the Refresh Token and use it as needed
    - I will design a form of Local Storage (caching) to accommodate the refresh token's long term storage
    - I will ensure the "expiry timer" within the access token is checked before attempting to use it
        - In the event the timer has expired, I will use the refresh token to acquire a new access token for use before proceeding.

The classes I will create to handle Google authentication will be:

- GoogleSyncUIHelper, which will make calls to the authentication library Xamarin.Auth.
    - Will facilitate login and initiation of OAuth2 handshake. The creation of UI to handle the authentication is platform specific, so this class must be created in Freed.A. It can be reused to wrap any required UI elements around other sync tasks too, such as file upload or download.
- CloudSyncHelper, which will handle using a refresh token to acquire a new access token.
    - This is not platform specific, and can thus be held in Freed.Core. The class can then be extended later to facilitate other cloud sync tasks such as file upload/download.

4.2.5 – Local Data Storage

The Authorization requirement of handling a refresh token long term can be addressed by serializing data to disk, which brings us to the design for Local Data Storage.

The design for this will be cross-platform, and thus housed in Freed.Core, was facilitated by the PCL storage library to map various platform's local storage paths using an object provided by the library. This is discussed in detail later in the report in the Development problems section (6.1), and will not be elaborated here.

The Local data storage design will be such that the user data (the singleton instance of the User Class) is serialized to JSON (using the JSON.NET Serializer) and stored to disk.

To enable this functionality, classes in the Freed.Core library will be needed to handle Storage/IO operations, which leads to the creation of a StorageIOHelper class.

This class will open or create files as needed in the app's data folder and write the JSON data to them. It will also be responsible for reading from the file to an object (deserializing). Although this seems like it is handling multiple responsibilities in this description, the grouping of Input/Output operations in classes is fairly common practice due to their similarity in nature, so I feel comfortable that this design does not violate the Single Responsibility principle.

It is important that we do not open the same file for reading and writing at the same time, so files in use will be kept in a list to check this before a new file stream is created for use.

I will also use C#'s "generics" language feature in this class. Given the files will be serialized/deserialized to/from JSON regardless of type, there is no reason to specify a specific input type. By using generics, the code becomes much more reusable and extensible in future, since the code-caller can simply specify what object type they are working with and rely on the StorageIOHelper to handle it. I will doubtlessly be able to import this class into future Xamarin projects I may undertake when reading/writing to disk for this reason.

4.2.6 – Cloud Synchronisation to Google Drive

The design of my synchronisation to Google Drive is heavily dependant on the API endpoints and methods used, and as such the Drive REST API documentation [24] was very useful in designing this implementation.

My Cloud Synchronisation feature is essentially a variant of the local JSON storage outlined above, with the additional requirement to upload/download the file to and from a web resource, rather than to the local disk. Therefore this section will focus on the upload and download of the file, rather than the serialisation, which is identical (JSON based).

To facilitate cloud sync, the REST API for Google Drive I will be using is entirely for the App Data scope, my application's own folder in the user's google drive. As such, the design and creation of REST requests must be such that they only address the App Data folder, since any other will return an error from the Google API.

This means my code must be designed as follows:

- The upload of data must create a message consisting of:
    - A HTTP POST request
    - A set of Metadata (the desired filename to upload, the location to upload (which must be the App Data folder)
    - A set of data (the content of the file to upload)
    - A valid access token
- It must also serialize the data to JSON from a user object.
- The download of data must create a message consisting of
    - A HTTP GET request
    - A valid access token
- It must also deserialized the data from JSON to a user object.

One additional consideration must be made to file upload. After reading the API Documentaion, I discovered that files uploaded with the same filename as an existing file will not replace the file. As such, I must also delete the existing file if it exists in the user's google drive before uploading. This means creating additional requests to get the file list of the app data folder, and to delete the file if it exists.

As with the majority of my code for the project, I will attempt to implement the design using the Async/Await task pattern to maintain the main thread's responsiveness.

4.2.7 – Kill File & String Search Engine

The design for the kill file is one which I put considerable thought into, and supplements the background efficient string searching research outlined in section 2.5.2.

In order to efficiently parse and filter the kill file, a multitude of problems must be addressed to avoid blindly searching massive articles for potentially large lists of ignored terms.

As such, before discussing the implementation of the search itself, we should consider the kill file's interactions and implications regarding the string search in general, and devise a strategy to approach the task with these considerations in mind.

Firstly, since every article must be either filtered or unfiltered before it can be included in a view, it is important to store the result of the filter application within the article data. Let us call this storage of filtered words the "filterMatches array". This design is ideal because as long as the kill file has not been changed, the same articles will be hidden/shown on every view for each relevant source, so the design avoids having to re-process the filter's string search when a view is created, making it

more efficient. Instead, we only need to hide items which have a non-empty filterMatches array, which is a relatively trivial filter to create using Linq.

```
.Where(item => item.filterMatches.Count == 0)
```

A Linq clause which could be used to show / return only items which have an empty filterMatches array.

Now, let us assume when a new article is added, we will scan the article for all words which exist in the kill file, and update the filterMatches array of the article accordingly. This is unavoidable for the desired functionality.

With that said, how can we address the addition and removal of items to filter to optimise efficiency? Given our knowledge that all existing filter items already exist in the item's filterMatches array, when a word is added to kill file we should be able to avoid having to reprocess every item in the kill file and article collection, and instead we can search the article collection for only the new word instead. If it is found, it can be appended to filterMatches for the relevant articles.

If a word is removed from the kill file, we can simply update the article collections' filterMatches arrays to remove that word if it exists. The result is that if that word was the only item in the filterMatches array, once it is removed the article will become unhidden / shown. If that word is removed and other words still exist in filterMatches, then the article remains hidden, as other words from the kill file are still preventing its display. This is exactly the functionality we desire.

```
if (!bannedStrings.Any())
{
    return false;
}

// check if item has already been scanned
if (item.hasBeenFilterScanned)
{
    if (item.flaggedFilterItems == null || !item.flaggedFilterItems.Any())
    {
        return false;
    }
    else
    {
        return true;
    }
}
else // not yet scanned, scan and update item's content
{
    return DoesItemContainFilterMatches(item, bannedStrings);
}
```

An overview of the kill file processing flow described above, the result of which determines if the article is shown or hidden using a Boolean return.

In summary by storing filter results in an array within the article data model itself, the required workload for the kill file filtering is greatly reduced when a view changes, as we only need to find items with an empty filterMatches array rather than perform the actual string search. Additionally, the workload for adding or removing a kill file term is greatly reduced, and the addition of completely new article collections is the only case in which we must fully process the kill file's string search multiple times consecutively. Lastly, since any article which has been processed will have its results stored/synchronised to the Cloud along with the rest of the item data, this means any items which have been scanned previously will already have the correct filterMatches populated, and if the user synchronises to another device they need not be reprocessed.

Since kill file alterations and mass article additions in the app are comparatively scarce in contrast to view changes, I will pursue this implementation for my project.

Now that the application of the kill file to articles has been addressed, we can evaluate the string search methods available to us to perform the filtering. As outlined in section 2.5.2, I decided to

implement the BITAP algorithm for string search, compare it to the built in .NET string search, and then evaluate my options. I tested this in a separate project to evaluate my options before proceeding to design, and the results are outlined in section 7.3.2 - String search performance testing (Practical). After seeing the results, and researching my options regarding word termination detection, I have opted to design the string search using Regex instead. This is because there are simply too many character variants which could potentially be classed as a word terminator, and writing such an algorithm does not make sense when Regex has a very basic means of achieving the same result, and is widely available for use (including as part of .NET).

Therefore, my chosen string search design will use Regex and the "/b" query string language notation to perform string searches for user inputted kill file terms. Combined with the design above I believe it will be both suitable for purpose and performant.

```
var matches = new List<string>();

if (!string.IsNullOrEmpty(domain))
{
    foreach (var pattern in targetStrings)
    {
        if (!string.IsNullOrEmpty(pattern))
        {
            Regex r = new Regex(@"\b" + pattern + @"\b", RegexOptions.IgnoreCase);

            if (r.IsMatch(domain
            {
                // pattern was found
                matches.Add(pattern);
            }
        }
    }
}

return matches;
```

class System.String
Represents text as a series of Unicode characters.

The Regex String Search method.

5 – Implementation

This section will outline the project's structural overview, Class Modelling, UML Diagram and Major class summary.

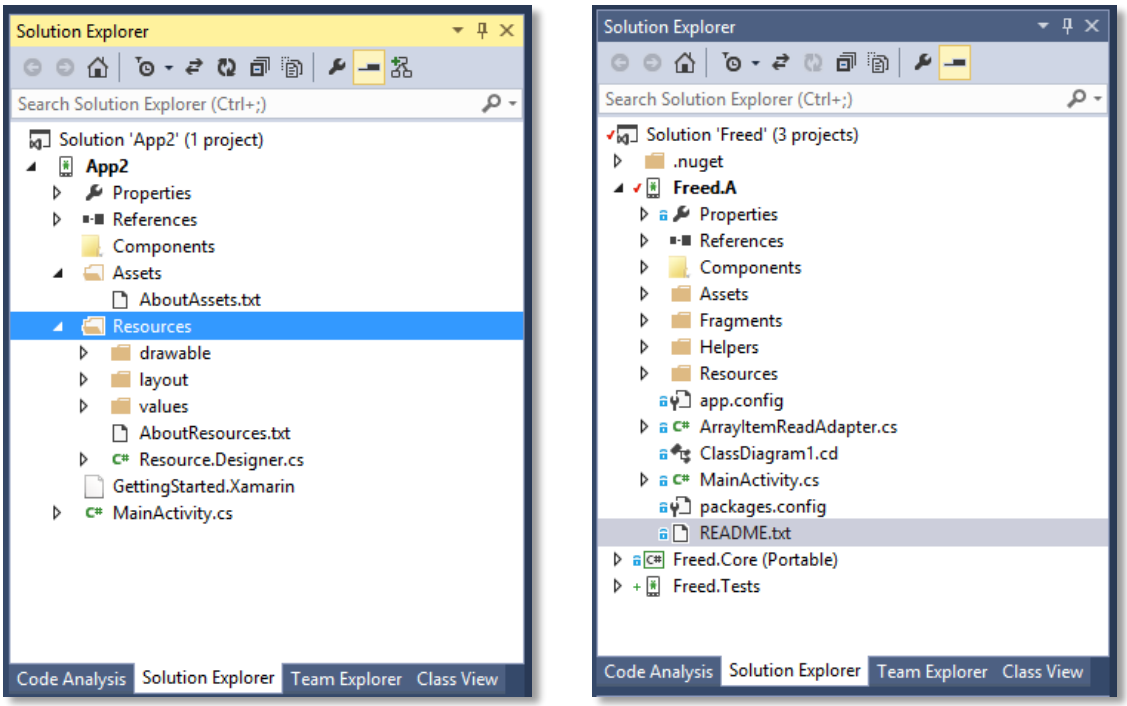
5.1 – Project Structure / Overview

Now that the systems and solutions have designs in place, we can begin to discuss the structure of the project as a whole and how I have organised it. In simple terms, the project consists of two key components, the Freed.A project and Freed.Core project, where Freed.A houses any Android specific content (the View and Controller layers) and Freed.Core houses the portable implementation logic (the Model layer). The following two sections outline the composition of these projects.

5.1.1 – Freed.A: Xamarin.Android Project Template & Hierarchy

The starting point for my project implementation revolves around creating the project hierarchy, which I based on the first requirement in my specification: a functional basic UI to build functionality on and enable testing in an android app. As such I will be using the "Xamarin.Android" template project. It allows me to begin with the basic components of an Android app already implemented – a main activity and starting page are included, which I can then go on to modify.

To facilitate my fragment-based UI, I will use the structure created by the project, with a few modifications:



A comparison of the stock Xamarin.Android profile (left) and my project setup (right).

The main addition of note is "Fragments". This folder contains the "code-behind" (.cs, C#) files associated with each fragment. Since the fragments themselves are .AXML files, it is necessary to place them in the Layout folder within Resources, just like other visual elements. This allows them to be referenced using the Android Resources collection and inflated when needed.

It's important to note the absence of any non-view related logic, which sit within the "Model" component of the MVC pattern. As mentioned when describing my goals for the project, I will be striving to separate any model code into its own portable class library (Freed.Core) to facilitate the

reuse of this code on other platforms. This means the Freed.A project is essentially designed to only contain the view and its controllers (Helpers).

5.1.2 – Freed.Core: Xamarin Portable Class Library Project Hierarchy

Any aspects of the logic, from data model and objects to HTTP access methods and RSS parsers, will be implemented in the Freed.Core project where possible to maximise the potential for reuse.

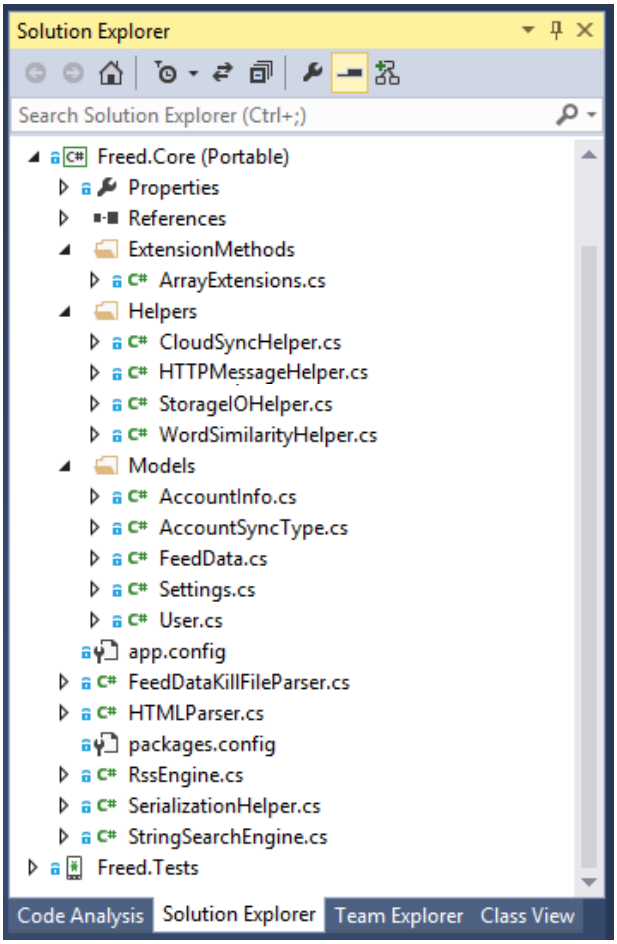
Freed.Core is distinct from Freed.A in that it has no ability to reference anything which cannot be interpreted on other platforms, i.e. most types or methods from the Android or Java namespaces cannot be used, only those in the basic implementation of the .NET and CLR (Common Language Runtime), and selected Xamarin ported classes.

Though this may sound limiting, a great deal of the applications logic can often be implemented in this library environment, and the benefits of doing so are that the library can then be used in any other Xamarin project time – be it Windows, iOS or Windows phone.

This saves a great deal of development effort in future for developers wishing to branch out to other platforms, something I am keen to do should my Android prototype be successful after the conclusion of this project.

With that said, the Freed.Core project has an architecture as follows:

There are 3 primary namespaces within Freed.Core; ExtensionMethods, Helpers, and Models.



The Freed.Core PCL project hierarchy.

ExtensionMethods holds any methods which I have written leveraging C#'s Extension Method feature, which allows commonly used methods to be rewritten such that they can be accessed from any object type declared in the method's arguments using a "(this objectType name)" fashion. They are a useful tool which greatly enhance the extensibility of code, which I have leveraged in this case to enable some additional custom Array modifications. These (and all other class functions) are outlined in detail in a later section.

"Helpers" serves a similar purpose to the Helpers found in Freed.A, and holds (non-platform specific) implementation logic. This namespace has one distinction to Freed.A's – the logic held in these helpers pertains to methods which require external libraries to function. By organising them in this fashion, I was able to maintain a greater understanding regarding the impact of any changes made during development, as working in the "Helpers" namespace immediately informed me I was working with code that was more heavily coupled, and should be more careful to avoid making any breaking changes. Any implementation logic which was not heavily coupled to third party frameworks is kept in the base "Freed.Core" namespace.

Lastly, "Models". Models refers to objects which serve primarily as data structures, and make up a crucial part much of the applications data flow. The objects here represent data for feed content,

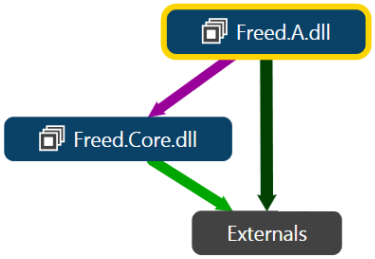
user state, preferences, and cloud synchronisation information. Changes to this namespace are extremely impactful and far reaching, and would need to be thoroughly tested before checking in.

5.2 – Class Modelling / UML and Dependency overview

To illustrate the class modelling process I took for my project, I will take a nested approach, progressing in level of detail at each step, culminating in the next section which explains the implementation details of the individual classes themselves.

5.2.1 – Namespaces and container projects

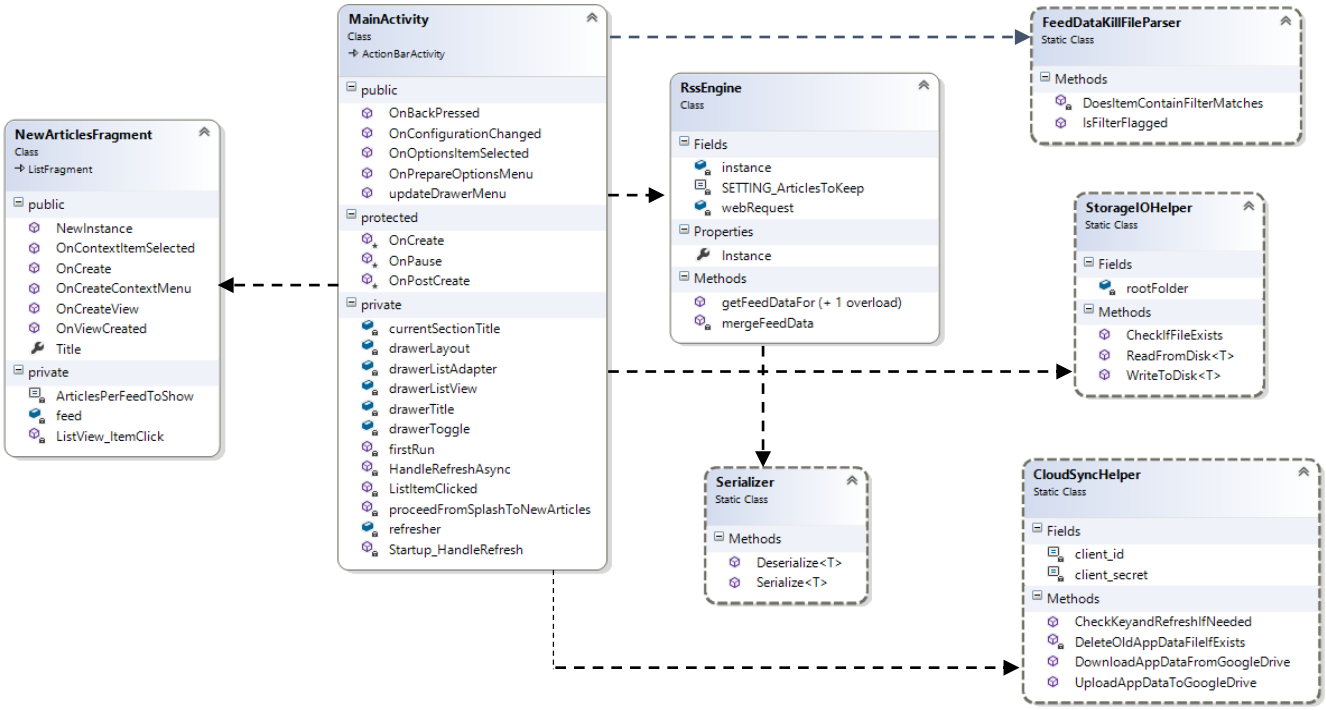
As I mentioned in the project description above, the first step was to create the projects for the Android specific and the portable components. These are Freed.A (Android exclusive components) and Freed.Core (Portable / cross platform components) in my solution design.



These are organised as the diagram opposite illustrates. The lines indicate dependencies between the projects, with "Externals" representing the third party and inbuilt tools and namespaces.

5.2.2 – UML Diagram Overview

The extended class diagrams and UML for my project is very large in size, as such I have heavily simplified the UML for inclusion in this report. The UML diagram below represents the core components of both Freed.Core and Freed.A combined, and make up the Model View Controller pattern. The View and Controller is represented by the ArticleFragment and MainActivity relationship, and the Models which handle data and logic are to the right of MainActivity. The flow of the MVC pattern is such that the Controller (MainActivity) handles the initiation of logic tasks in the Model, such as RSS Fetching, Kill file filtering, cloud sync and saving to local storage, and then facilitates the display of results in the View (in my project, the current Fragment).



5.3 – Major Class Summary & Interactions

This section will outline the classes relevant to the major app flow and functionality therein. This includes some classes which are already outlined in the UML diagram and Design portion of the report, however will focus on describing the significant elements of the implementation details within the classes, rather than the general design or problem context. In the case of fragments, the fragments will be described either as a group (as per the Design section groupings) due to their similarity, or with reference to a specific example which can be extrapolated to other fragments of the same type.

Given my Xamarin interpretation of the MVC pattern, classes labelled as Data Model and Implementation Logic fall under the model category, and are held in Freed.Core. Controller logic and View (fragment) groups represent the Controller and View categories, and are held in the (platform specific) Freed.A namespace.

The order of the summaries is such that the classes that interact with each other follow one another, to better illustrate the code flow. I will firstly begin with the two major Data Models, then proceed to the entry point of the app (MainActivity), and from there proceed to explore the major class interactions in the order in which they are called.

5.3.1 – Data Model: "User" Class

The data model representing the User serves as the encapsulating object for the bulk of the app's data. Any feed data, user settings and other such properties are encapsulated within the User class. This class is widely used throughout the app, and since a great deal of the app's code requires access to the user object, for example, the fetched feed data is stored the User object under the CurrentFeedData property, and the Cloud and Local Data sync serialize the User object as a means of persisting state for later use.

This widespread use makes it important to ensure a shared, static user instance is used throughout the app to avoid multiple conflicting instances of the user object, which is why I chose to implement the singleton pattern to facilitate this widespread access and single-instance guarantee.

The model uses generic interfaces for its properties wherever possible (for example, IList for any collection type properties) in order to abide by the SOLID principles of relying on abstract types and interfaces rather than concrete implementations.

Other specifics of the user class are fairly simple, basic attributes and do not require explaining, and as such will not be outlined here.

5.3.2 – Data Model: "rss" Class

The implementation of the rss class represents feed data objects. Its main function is the encapsulation of any data relating to a Feed source, be it articles, source titles, links etc.

The creation of most of the class' properties was completely automated using the "Paste Special" facility provided by Visual Studio, which allows for the pasting of valid XML data as classes. I chose to leverage this, and merged RSS data from combining my example feeds to create the foundation of this data model using "paste special". This was further manually augmented as needed, for example, to add a Boolean flag indicating if the object has already been filtered or not.

The use of a generated object gave me a robust foundation on which to build my object design and was a very useful development shortcut provided by my choice of tools.

The class uses "Data contracts" to mark various classes and fields with notation aiding the .NET Xml serializer, indicating that various properties should be handled in certain ways or ignored entirely during serialization. For example, the `hasBeenFilterScanned` property which facilitates kill file

scanning is flagged as `[System.Xml.Serialization.XmlIgnoreAttribute()]` to signify it should not be serialised/deserialized to when using the XmlSerialiser. This allows the class to be reused as needed in other rss applications, while including elements and properties which are specific to the implementation of my app.

5.3.3 – Controller Logic: MainActivity (and Navigation Drawer)

The MainActivity is the entry point to the app, starting at the "OnCreate" method when the app is first launched. The class handles initialization tasks, inflating the splash screen, starting a cloud sync or local data load of User data, and on completion facilitating the progression of the splash screen to the NewArticles fragment, the first user interactable portion of the app's execution flow.

Once the initialization is complete, the class remains heavily active even when a fragment is in use, as it handles the events from the Navigation Drawer, the "Overscroll Refresh" gesture, and the activity events for the app Pausing and Resuming, which call for local disk serialization / cloud synchronization.

Before moving on to the fragments, I will outline what these events entail.

- The navigation drawer ItemClick events are bound to a method in MainActivity, `ListItemClicked`. This method handles the app's navigation from the navigation drawer, by creating and inflating fragments based on the index of the clicked item in the navigation drawer.

- The overscroll refresh gesture is handled by the RefresherView the MainActivity, which has a "Refreshing" event. When this event is fired, the MainActivity method `HandleRefreshAsync` is called, which performs a set of functions similar to that of the app's initialization: the subscribed feeds are refreshed, the navigation drawer items are (re)generated, and the fragment view is updated – either rebuilding the current view to populate the new content or taking the user to the NewArticles fragment, depending on their current active fragment. This is by design, as the implication of a user requesting new data is that they will want to see the new data which has been retrieved once the refresh is complete.

- Lastly, the automatic Local Disk / Cloud synchronisation. To enable a seamless user experience for local caching and cloud sync, the app will automatically perform these functions via MainActivity events. For example, when the app is "pausing," the MainActivity overrides this using the `OnPause` method, which creates a new thread to serialize the User data object to a JSON file on disk. If the user has authenticated with Google Drive, the thread will also upload the data to the user's Google Drive account. Similarly, on creation of the MainActivity, any available data is downloaded from the authenticated cloud account, and this data is set as User singleton instance for the app. If this fails (for example because there is no cloud data), the code flow falls back on deserializing from the app data folder on disk, enabling local or unauthenticated use of the app. The third and final fallback is to initialize a default user object, which usually only occurs if the app has been freshly installed or the user data has been manually deleted.

This summarises the major functionality of the MainActivity. The class contains a great deal of additional methods to facilitate the initialization and startup of the app, but these fall outside of the scope of this report, as their explanation is not particularly relevant to any of my app's feature implementations or requirements.

5.3.4 – Implementation Logic: RssEngine

The RSS engine is called by the MainActivity both on launch and when a feed refresh has been requested. It has only two major functions - `getFeedDataFor`, which updates the User object with current feed data for a specified list of sources, and the `filterAndMergeFeedData` method, with ensures new data is merged with existing content and not removed until it exceeds the specified allowed article count per feed (currently hardcoded in my app to a value of 50).

The refreshing of feed data iterates though the provided source list, making a HTTP GET request to fetch the content and then deserializing this to the `rss` object type using the .NET XML Serializer. Once this is done, a string is allocated to represent the feed in the navigation drawer, and then the fetched feed data is filtered and merged by the `filterAndMergeFeedData` method.

The `filterAndMergeFeedData` method filters the articles by PubDate in order to make them more readable in the view fragments later, as well as merging the fetched data with any data with a matching source URL which already exists in the current user object. This is done by taking the Union of the new and old instance of the feed data.

5.3.5 – View / Fragment Group: Article List based Fragments (including NewArticleFragment)

The NewArticleFragment is the next step in the natural progression of my applications launch, and is inflated by the MainActivity to replace the SplashFragment once initialization of the app is completed. The fragment showcases the latest articles from all of the users subscribed feeds, selecting the top 5 newest from each to form the “Newest Articles” list. Interacting with the items allows for the opening of the article in an ArticleDetailFragment for reading.

In my application, the article list based fragments (such as NewArticleFragment) implementation comes down to three elements; the ListFragment, the custom adapter, and the result of a Linq query.

- The ListFragment is a type of Fragment which is provided by Android, and houses ListView within a fragment, which is naturally suited to containing collection types. It includes click events, which can be bound to in the OnCreate override and used to navigate / perform an action when the user clicks an item. The event includes details about the clicked item (e.g. the position in the list) to facilitate these actions.

- The custom adapter is the adapter (ArrayItemReadAdapter) is an extension of the ArrayAdapter base type which I created to represent individual articles within the ListView, and allows me to modify the styling of the item dynamically based on various events. In the case of Article Lists case, the adapter has multiple TextViews to accommodate the additional publish date and author information my users requested during a feedback session, as well as housing view logic to modify the font and style of the item when an item has been read, to provide a visual indicator to the user. This is achieved by overriding the GetView method within the adapter to check if the item it is hosting has been read, and then altering the properties of its child TextViews to modify their formatting.

- This brings us to my use of Linq to create data subsets. I chose to use Linq to create the subsets of data for views dynamically, like the example below:

```
feed.channel.Articles = User.Instance.CurrentFeedData
    .SelectMany(x => x.channel.Articles.OrderByDescending(y => y.pubDateTypeFormatted).Take(ArticlesPerFeedToShow))
        .Where(z => !FeedDataKillFileParser.IsFilterFlagged(z, User.Instance.UserSettings.KillFileEntries))
        .OrderByDescending(x => x.pubDateTypeFormatted) // reorder to adjust for multiple sources mixing date sorts
        .ToArray();
```

The Linq query used to subset and sort the data for the New Articles fragment.

This is probably the most complicated Linq query of all the view fragments, and essentially comes down to four lines of code, which I find very elegant. The query selects the subset in the following fashion:

- SelectMany – selects all the articles from all the feeds based on the given delegate:
    - Delegate – selects all the articles in the feed based on the descending order of their publish date, and takes X number of items from the result (in this case, ArticlesPerFeedToShow = 5)
- Where – filters the results of the above enumerable to those where the delegate is true:
    - Delegate – inspects the item to check if the Kill file filter has been applied, and returns true if the item should be hidden. This is negated, resulting in only the shown items to be used as the result.
- OrderByDescending – sorts the results by publish date once more, so that the latest articles from each feed are listed amongst each other in order of newest->oldest
- ToArray: evaluates the enumerable to fetch the desired objects as specified above, returning the results in an Array.

This condenses a great deal of iteration into a minimal amount of code, thus improving the maintainability and readability of the classes greatly. It also sped up my implementation (and subsequent testing) a great deal. I have further analysed the general use of Linq later in this report, in the Reflection section (9).

5.3.6 – View / Fragment Group: Settings Fragment

The settings fragment is inflated by the Main Activity when the user selects it from the Navigation drawer, and enables access to various user settings, including the editing of the kill file, the authentication with Google Drive, and the manual synchronisation of data using the authenticated account.

The implementation of this class is fairly simple, as we enable these three functions using three buttons and their bound “Click” event handlers.

- For the edit kill file button, we create and inflate a new instance of the EditKillFileFragment, which handles the functionality for us.

- The Log in/out of google account functionality is similarly offloaded away from the settings fragment, since it is the responsibility of an existing class: the GoogleSyncUIHelper. This is implemented in this fashion in order to enable the authentication with google to occur from other places in the app with ease in future, if further functionality were to be added which required it.

- The sync user data button checks that the user object contains information which allows us to sync, namely the token information provided by the OAuth2 flow. If this information is present, we perform the sync using the GoogleSyncUIHelper, which uses the implementation logic from Freed.Core with some additional UI feedback for the user. If the

information is absent, the GoogleSyncUIHelper notifies the user that the required information is not present, and they should first log in before attempting to sync. The Freed.Core library first checks for existing data in google drive, which is downloaded if found, and overwrites local data. If it is not found, the local data is uploaded to the cloud. This ensures that users who have the app on multiple devices are synchronising to/from a single data file, and prevents merge conflicts.

Finally, if the result of the sync was reported as a success, the settings page reminds the user to refresh their feed in order to merge the latest content with the (potentially old) synchronised data.

5.3.7 – Controller Logic: GoogleSyncUIHelper

The GoogleSyncUIHelper is a controller-level wrapper for the Cloud Synchronisation feature. It is called to perform Cloud synchronisation tasks, from the SettingsFragment when the sync button is pressed and from MainActivity's onCreate event. This helper provides UI feedback for the Cloud Sync feature, which is why it exists in the Freed.A namespace, since UI operations are not cross platform implementations.

The class has separate methods pertaining to the file list, upload, download and authentication operations required for cloud sync, all of which are implemented in Freed.Core classes or the Xamarin.Auth library, with only UI elements being handled by this class. The primary function of this class is to wrap the `CloudSyncHelper` class' functionality with UI feedback to keep the user informed.

- The file list method (`CheckForExistingDataInGoogleDrive`), upload method (`UploadUserDataToGoogleDrive`) and download method (`DownloadUserDataFromGoogleDrive`) function in a very similar fashion. All methods use the app's Google API key and the user's access token. If the access tokens requires a refresh, this is handled first. Then the upload/download/find functionality itself is handled, which returns a status Boolean: true for success and false for failure. This return value determines the content of the UI dialog box informing the user of the results, but only if this was requested by the calling method using the optional parameter "`confirmNeeded`".

- The login functionality in the `GoogleAccountLoginToggle` method differs greatly to the upload and download in its implementation, and uses the Xamarin.Auth library to provide the required functionality, as mentioned in section 4.2.4. If the user has not logged in, it uses the Xamarin.Auth "`OAuth2Authenticator`" object to build an initial request to Google's OAuth2 API, which is then started in a WebView by opening a new activity to handle the Authentication flow. The completion of this activity is bound using an event handler to the "`auth_Completed`" method in the class, which then stores the information in our app's user data for use, and provides feedback to the user indicating success or failure.

5.3.8 – Implementation Logic: CloudSyncHelper

The `CloudSyncHelper`, as mentioned in section 4.2.4 and 4.2.6, will be responsible for the handling of various cloud sync tasks. It is called by MainActivity and the SettingsFragment when these classes initiate an operation which uses Google Drive. This includes refreshing the access token, finding existing files, deleting files, uploading files, and downloading files.

All of the methods leverage a variety of HTTPRequest types to perform their functions by building a URL based on the Google Drive REST API endpoints.

- To refresh the access token, the `CheckKeyandRefreshIfNeeded` method checks the current datetime against the token's expiry time. If a refresh is needed, a new POST message is created using the app API keys and the user's refresh token. The response to this message contains a new access token, which is then returned to the caller. If a refresh is not needed, a null return is provided.

- Finding existing files uses the access token, app API key and REST API endpoint to form a HTTP GET request. The URI includes a query string to specify what files are desired, and the response is the metadata of any files found using the desired query.

- The upload of files is handled by the `UploadAppDataToGoogleDrive` method. Firstly, we must verify that user data in the cloud is cleared, to prevent duplicate files existing in Google Drive. This is because files of identical names being uploaded do not get overwritten when using the Drive API. Next, we can create our request URI using the access token, app API key, and the desired REST API endpoint, and form our `MultipartContent` object, which contains the file metadata (filename and containing folder) and the file data. This is then uploaded using a HTTP POST message, and the Google API responds with the uploaded file's metadata if the operation is successful. The last step is to save the Drive "FileId" to our user object, as a reference to where the cloud data is stored.

- The download of files is handled by the `DownloadAppDataFromGoogleDrive` method, which forms a HTTP GET message using the user access token and the REST API endpoint URI, which contains parameters for our desired file. The response to this is the file specified by the endpoint URI. The method then handles serialization of the response to our User object using the Json.NET serializer to parse the stream. I chose to use a stream to improve the memory performance, something which is discussed in detail in the Performance Testing section of my report in section 7. Before storing the downloaded User object, we transfer the current value of the Drive's synchronised data location, which updates the resource for the next sync with the correct fileID to download from Google Drive. If this was not done, we would not be able to delete the old version of the file before uploading, as the fileID would be out of date and no longer exist. Once this is done, the current `User.Instance` data is replaced.

- File deletion from Google Drive is fairly simple, we simply make a HTTP Delete message with the desired fileId included in the Drive API REST endpoint parameters, and provide a valid access token for the request header. There is no response from the server on successful deletion, so nothing more needs to be done.

5.3.9 – Implementation Logic: StorageIOHelper

`StorageIOHelper` is a Freed.Core class which facilitates the local serialization of user data to disk, and is called by ImageDownloadHelper during image download and MainActivity during onPause and OnCreate.

This is important both for offline use of the app, and for the preservation of a user's Refresh token if cloud sync is enabled, to prevent them having to re-authenticate with each sync. The cross platform implementation of this functionality is achieved using the PCLStorage library, which maps the local app data path of various platforms to a C# object. Without this, the class would have had to be housed in Freed.A, and been less reusable.

The class provides two fundamental functions, reading objects from disk and writing them to disk. These are facilitated by the Json.NET serializer, which takes a provided object and creates a JSON representation, which is then directly written to a file stream for optimal efficiency. Both functions check that the file is not in use before starting the operation, and update the list of files in use with their current file target. This is to prevent Stream exceptions that occur when opening the same file in multiple stream variables.

The class can also determine the existence of a file using the PCLStorage library, which is useful for performing app logic such as determining if there is any cached data or if the app has been run before. This is done by attempting to get the file in question, and then handling any `FileNotFoundException`s and `DirectoryNotFoundException`s explicitly to return false.

My implementation of this class uses C# Generics functionality to allow the calling code to dictate which object type should be serialised from/deserialized to as a type reference, i.e. `WriteToDisk<string>` which would deserialise a string, whereas `WriteToDisk<int>` would serialise an int. This makes the class extremely reusable, even in projects completely unrelated to mine, and is a great feature of the C# language.

5.3.10 – View / Fragment Group: Edit-List based Fragment

The edit-list based fragments include the Kill File Editing fragment and the Edit Feed Subscriptions fragment, and are called by the SettingsFragment and the MainActivity respectively. The kill file-specific implementation details have already been outlined in the section 4.2.7, and as such this summary will exclude those details.

The edit list based fragments have to provide the ability to show the list of data, remove items from the list and add new items to the list. Adding items is done using a button to pop up an edit textbox to add new items. Removing items is done by tapping items in the list display, and the display of items is achieved thanks to the ListFragment's Adapter property, which makes it easy to bind and display collection of data in the list fragment.

5.3.11 – Implementation Logic: FeedDataKillFileParser

The `FeedDataKillFileParser` class is used to simplify the application of the kill file filter using Linq, and is called by the RSSEngine.

The primary method, `IsFilterFlagged`, first ensures that any filter items exist to be applied. If there are none, we need not filter and can immediately return false to indicate the item should not be hidden. If there are items however, we proceed to check whether the item has already been scanned. This property of the item is updated whenever a scan is required, as outlined in section 4.2.7, and is needed to ensure that view creations do not needlessly cause the item to be rescanned while applying an existing filter. In this case, the `flaggedFilterItems` array in the item will already have the values causing the filter to be triggered, so we do not need to scan again, we can simply return true if the array is not empty, and false if it is. Lastly comes the case which indicates the item has not yet been scanned. In this instance, we call the class' `DoesItemContainFilterMatches` method to search the items title, description, source title and encoded fields for any instances of the filter string. A Union with the results and the item's `flaggedFilterItems` array ensuring only distinct filter matches are kept in the array, and the method returns true or false based on the item's filter array content being empty or full respectively in order to complete the filtering.

5.3.12 – View / Fragment Group: Article Detail Fragment

The article detail fragment is called by any of the Article List-based fragments, which create a new instance of this fragment to host the article which was clicked in their ListView. implementation handles the View logic for individual articles. This means showing a body of HTML formatted content which comes from the RSS feed, including images and text with various formats and styles. To do this, when the view is created, we call the `HtmlParser` class to find our image resource indexes in the content, and then the `CreateViewGroups` method in `TextImageViewCreationHelper` is called, and the results are given to the fragment's `AddViewGroupsToFragment` method.

This method iterates through the results, which are pairings of strings and ViewTypes. If the viewtype enum is a TextView, then a TextView is created with the given string input assigned as a HTML parsable input (using the built in `Html.FromHtml` method) and the type of text in the view is defined as a SpannableString to handle formatting. If the ViewType is an ImageView, the string is handled as an Image resource to download and the `URLImageParser` asynchronously downloads it and creates a Bitmap image from the data stream while the method continues to iterate and build view objects. Once the method has iterated through all of the results, and irrespective of the state of the image downloads, all of the views will have been added to the fragment's linearlayout container. If imageviews have pending downloads, the imageview will redraw it's canvas automatically as soon as the download is completed to show the image.

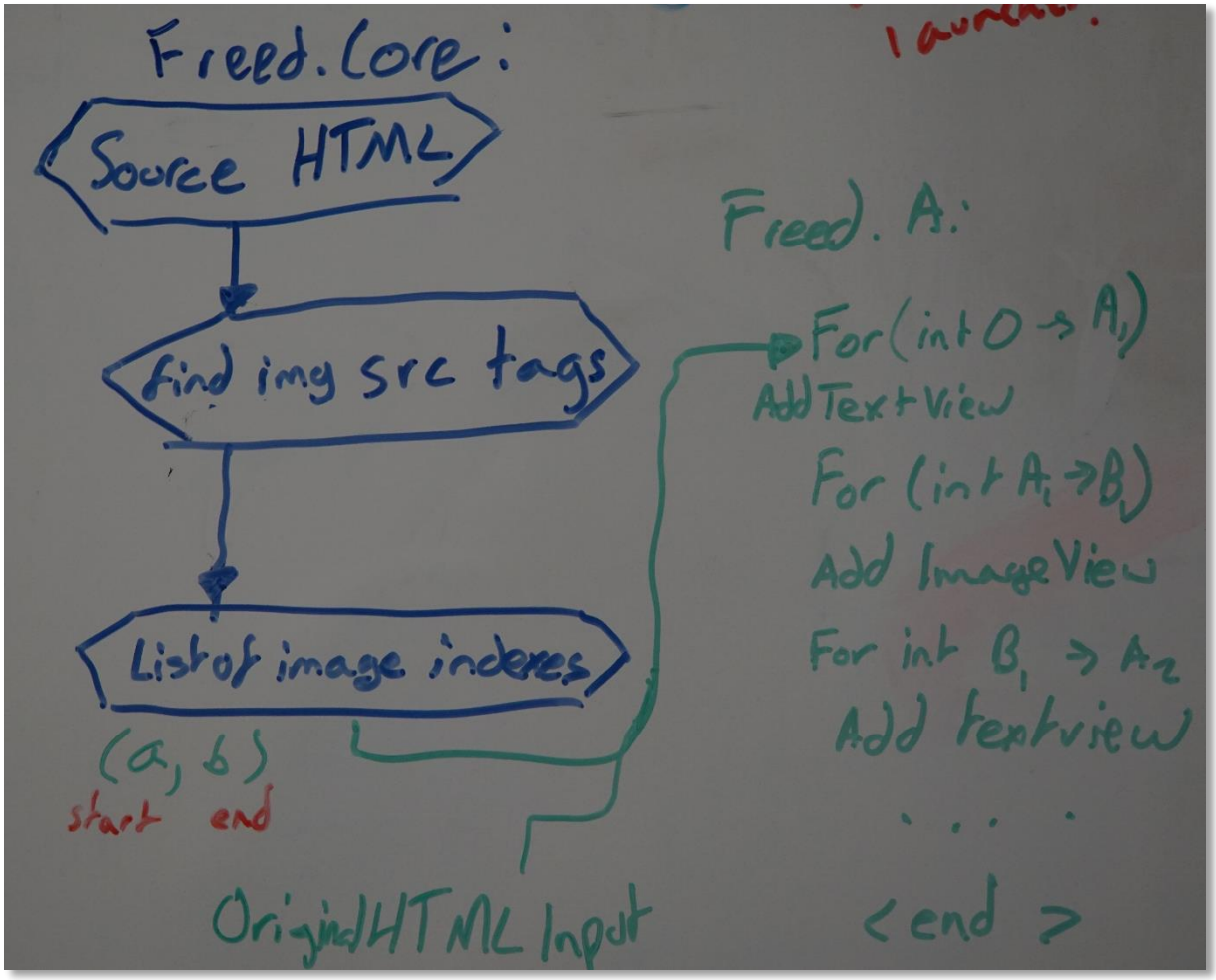
5.3.13 – Implementation Logic: HTMLParser

`HtmlParser` is called by the ArticleDetailView and TextImageViewCreationHelper. It's goal (and the class' sole responsibility) is to parse a body of HTML text for various desired results. The two implemented methods in the class are designed to facilitate the location of images.

- In the ParseHtmlForImgRanges method, we iterate through the character array of images, and when a series of characters matching "<img" is seen, we search for the closing tag ">" and add this to our results list. Since the image tag can be self closed ("/>") or distinct closed (</img>) as part of the XHTML standard, we must handle both of these cases, which I do by excluding all closing tag from my search (by checking if the next non-trivial (whitespace) character after the "<" is "/"). The return of this function is made up of the indexes of the character array array on which the image URL starts and finishes. This enables us to show

the image in the text's correct contextual position later during view creation in Freed.A's TextImageViewCreationHelper.

- HTML parser will also facilitate locating the src attribute in the method ParseImageUrl. This is a similar character-array search, where we find the consecutive characters "src". Then we search for the 2nd non-trivial character following. This is to handle whitespace, since attributes of "src =", "src=" and "src= " are all valid. Therefore the first non-trivial character will be =, and the next will be the start of our image – a " character. If this " character is not found, the string can be safely marked as malformed, and the function returns an empty string. This is because in both HTML and XHTML this is the only valid character to denote the start of a URI for the src attribute, so anything else is out of our defined scope of acceptable input.If the character is found, we can search for the next " character, and add anything that appears in between to our list of image sources, to be returned at the end of the loop.



Flow diagram planning of the HtmlParser functionality.

5.3.14 – Controller Logic: TextImageViewCreationHelper

TextImageViewCreationHelper's responsibility is to create ViewGroups made of Text and ImageViews. It is called exclusively by the ArticleDetailFragment to facilitate the contextual and asynchronous creation of images within a HTML text body.

The main method of the class (CreateViewGroups) uses a list of "imageRanges" (which represent array indexes, between which lies an image source) to create a LinearLayout ViewGroup comprised

of Text and Image views. These views will be created in alternating fashion such that images appear in their correct contextual place, as they do in their original HTML body.

By iterating through the HTML body string from 0 until the start index of the first image range, we can safely mark this as text (HTML) to be shown in a TextView. This is created and added to the result view group, along with an Enum to signify it is a TextView.

Now by taking the range from the current index to the end-index of the current ImageRange value, we have an Image source URL, which I am able to download and create an ImageView with, and the following index is treated as text content until reaching the next start index value from the imageRanges list. This process loops until the entire HTML body is processed, creating a series of views which can then be added to the LinearLayout ViewGroup by a fragment.

This implementation enables proper awaiting of the download method since I am no longer limited to the Drawable return type of the IImageGetter interface, which allows the view to correctly update once the image is downloaded. This addresses the problem I outlined briefly in the Design section of the report, and is discussed in more detail in the Development Problems section (6.3).

5.3.15. – Implementation Logic: URLImageParser / ImageDownloadHelper

This class handles image resource fetching for the application. It existed at first in the Freed.A namespace as it initially was the implementation of the Android specific IImageGetter interface, but now exists in the Freed.Core namespace as ImageDownloadHelper, as the interface is no longer used for the functionality as described in section 4.2.2. The class' implementations are identical. Moving the implementation to Freed.Core enables its reuse in future versions on other platforms.

`ImageDownloadHelper` is called only by the ArticleDetailFragment to facilitate the asynchronous download of images, and has only one method: `GetDrawable()`. This method is asynchronous in order to maintain UI responsiveness while images are being downloaded, and as such returns a Task<Bitmap> which can be awaited to retrieve the Bitmap result once it is available.

The method initialises a Bitmap object with the desired settings, and then starts a new thread to perform the work using Task.Run. This task is immediately returned to the calling method, and when completed will return a bitmap containing the image data.

Within the new thread, the process of fetching an image begins. Firstly, we check if the file exists locally on the device, and load it from disk if it does using the `StorageIOHelper` class, then return the bitmap. If it does not, we download the image from the given URL, and once it is downloaded we store the value to disk so that if the user views the article again, they can see the image without downloading it. The image result is then returned.

5.3.16. – Conclusion

This completes my description of the major implementation segments of my project. Other details are fairly minor, however the full source code is included with this report for review as needed.

6 – Development problems

The key development problems encountered revolved around A) Code Reuse in PCLs, B) Threading, Web Requests & Context Yielding, and C) Asynchronous Image Loading with Contextual Placement. The summaries of these issues and any solutions I devised are found in the sections below.

6.1 – Code Reuse & Portable Class Libraries in Xamarin (PCLs)

A great deal of challenge arose from attempting to include certain elements of functionality within Freed.Core – specifically the ability to write files to disk and handle HTTP Messages.

These proved difficult due to the nature of the implementation typically being device specific – on Windows devices, the implementation differs to on Android in that files must be written to a different path, and a different HTTP client is used as default. Therefore including them in Freed.Core was initially not possible.

With further research however, I was able to solve both of these issues, and both in a similar fashion.

For the storage requirement, my only limitation was that Xamarin does facilitate writing files to disk cross-platform, since the file path for where applications are allowed to store data vary from OS to OS. After searching for a solution, I was found a Xamarin NuGet package called PCL Storage.

To clarify, NuGet packages are extensions which can be easily added to a code project through Visual Studio. They enable the use of third party libraries within your project in a fashion which guarantees they are kept up to date as updates are made to the library.

PCL Storage was a library written by Xamarin Staff to facilitate exactly what I was trying to achieve by mapping the default file path of the system at runtime. This means by constructing my file paths in a relative manner to their "`FileSystem.Current.LocalStorage`" object, at runtime the object is resolved to the system's default folder for local storage (/data/app/<package-name> in Android), allowing me to write files to disk without encountering problems arising from my application not having sufficient privileges or running into invalid file path exceptions.

The HTTP message implementation was solved in a similar fashion, leveraging Microsoft's Portable HTTP Client Libraries package to enable the sending and receiving of HTTP messages.

6.2 – Threading, Web Requests & Context Yielding

During the projects development, I encountered an issue while using the Async/Await pattern to make requests to web resources. Although the pattern worked well, I had also been manipulating the context yielding of Async/await with the .ConfigureAwait(true) syntax to allow for additional tasks to be executed once the web request had finished – usually to perform changes to the UI once the data had been received.

However, I encountered strange, hard to reproduce bugs where portions of data would be missing. Initially I was inclined to investigate the methods responsible for fetching data from the web, but eventually I realised that the problem disappeared when running the same code synchronously, which means the data fetching could not be at fault.

After further research and debugging I realised that I was using the wrong tool to achieve my goal of following up on task execution. ConfigureAwait is designed to alter the continuation context, not the order of processing, and as such what my code was actually doing was marshalling the thread back to the calling context on completion. At first glance this appeared to be what I needed, as it compiled and did not throw exceptions (since the UI code was correctly being run on the UI thread after being marshalled), but on closer inspection I realised that the code following the

ConfigureAwait call was not always running after the thread had completed its task, but sometimes beforehand.

This was problematic as it caused UI elements (such as the navigation drawer) to sometimes be populated with blank elements if the network was still fetching data when my UI code was called. The bug therefore, was a race condition problem.

This issue was the hardest to identify during the course of my project, as race conditions can be very hard to debug since they do not always occur reliably. In my case, the app worked fine while on WiFi, where the speed of the task's execution was such that my UI code being marshalled took longer than fetching the data from the Web, causing the code to function as desired. On 3G/Mobile data however, the UI code executed first, and revealed the race condition. While this is trivially summarised after the fact, the problem was actually fairly major and difficult to trace at the time.

To solve the issue, I studied various ways to follow up thread execution, and learned about the ContinueWith() method that all Tasks in .NET 4+ have access to.

ContinueWith allows for the provision of a delegate to be queued for execution after the attached task has completed. It can explicitly be marshalled back to the UI thread using an optional parameter, making it ideal for my use case, and is supported as part of the same Xamarin package that enables the Async/Await pattern. This is how I resolved the race condition.

6.3 – Asynchronous Image Loading with Contextual Placement

During development one of the problems I ran into was with implementing the standard android TextView's architecture around Image containers, as I outlined in my research section (2.5.4).

I was determined to have images included in article content be placed as they are in their RSS, rather than load them separately, as many feeds (Engadget from my test feeds included) have text which references images placed contextually, and become nonsensical when the images are moved or absent. As I mentioned, to achieve this I implemented Android's Html.IImageGetter interface to download images, which then get rendered within a textview which has been passed a HTML source string and an instantiated implementation of the interface.

This worked to some extent, however the images would not be drawn in the view once download was complete, and even forcing a total view refresh left the images missing from the TextView. Upon closer inspection of my implementation, I noticed the IImageGetter interface does not return a Task type, which is required for the Async/Await pattern to return an object as a result (it returns as Task<some_object_type> immediately, which can then be awaited to retrieve the encapsulated object when needed).

This meant that the pattern was not implemented properly, and caused the view not updating after the asynchronous call completed. I verified the issue by removing the Async pattern from my interface implementation, which led to images being shown, and proving my assumption that the issue was a compatibility problem between the Async handling in C#/.NET and the Xamarin/Android IImageGetter interface type.

Unfortunately, manually redrawing the nested view was also not a viable solution, as there is no means of accessing the encapsulated ImageViews within the TextView once they are created – they are obscured from public modification.

This was problematic, as my synchronous workaround wasn't an acceptable permanent solution; any articles with large / multiple images would cause the app to freeze until the image downloads were complete, a problem which would be greatly exacerbated if a user were to use a slow/mobile data connection rather than WiFi.

This forced me to redesign the implementation, and write a great deal of code to create my own implementation of similar functionality. By leveraging my knowledge of C# and the Async Await pattern, researching and testing my HTML parsing thoroughly, and using the more basic view types provided by Android (Imageview and TextView without an ImageGetter), I was able to overcome the problem. The resulting helper class constructs a view hierarchy which dynamically loads images into the view as they complete downloading, and successfully met the requirements of my specification. Implementation details are outlined in section 4.2.2.

Although I am happy with the result, as it is very robust, performant and visually pleasing, this problem does exemplify some of the challenges that can unexpectedly arise when developing cross platform on Xamarin. I was unable to rely on third party support, as no-one (so far as I could research) had undertaken a similar task before, and thus I was forced to rely on my own C# knowledge to devise an alternative, equivalent solution. Had I been working in a standard android app, the issue would not have occurred, and even if it had, there would have been a greater chance of being able to utilise forums or documentation and examples from the Android SDK to help fix the issue.

7 – Testing

My testing process consists of three major sections; correctness testing, usability testing and performance testing.

7.1 – Correctness testing

This section will assess various features and requirements from my specification, and outline to what extent my finished project meets them.

To determine if my implementation meets the goals and features laid out by my specification, I have referenced the requirement numbers in my specification mapped the section of the project which fulfils the requirement/feature to its number. Every feature in the specification has been successfully addressed, and the below list outlines which class or part of the project facilitates the feature.

1) Fulfilled by the Prototype UI shown in the design section.
2) Fulfilled by the RSS engine implementation outlined in the Implementation section.
3) Fulfilled by the User data model and the Edit Feed List fragment shown in the design and implementation sections.
4) Fulfilled by the use of XMLSerialiser as outlined in the design section.
5) Fulfilled by the Regex parser and FeedDataKillFileParser.
6) Fulfilled by the Edit Kill file list fragment.
7) Fulfilled by the GoogleSyncUIHelper.
8) Fulfilled by the CloudSyncHelper and GoogleSyncUIHelper.
9) Fullfilled by the RSSEngine.
10) Fulfilled by the ImageDownloadHelper class.
11) Fulfilled by the final UI implementation of the ArticleDetailFragment and Article List based view.
12) Fullfilled by the practical verification performed in the Performance testing section (7.3)
13) Fulfilled by the Final UI design.
14) Fulfilled by the newest articles fragment.
15) Fulfilled by the MainActivity use of RefresherView, as seen in the Final UI design.
16) Fulfilled by the Data Representation design using Paste Special via Visual Studio, as elaborated in section 5.3.2.
17) Fulfilled by the StorageIOHelper.
18) Fullfilled using the Async Await Pattern in conjunction with HTMLParser and TextImageViewCreationHelper.
19) Fulfilled by StorageIOHelper and ImageDownloadHelper.
20) Fulfilled by Android's Textview and the Html.FromHtml method.
21) Fulfilled by the Read Queue Fragment
22) Fulfilled by the ArrayItemReadAdapter
23) Fulfilled by the final UI and contextual menus as seen in the Design section
24) Verified in the Testing section for UI scaling (section 7.5.1)

7.2 – Usability / Usefulness Testing

The usability and usefulness testing in my project refers to the testing of the app with a variety of users with the aim of receiving feedback to iterate on and improve the quality of the app. It was performed constantly throughout the development process, which is illustrated throughout the User Interface section of the report (4.1).

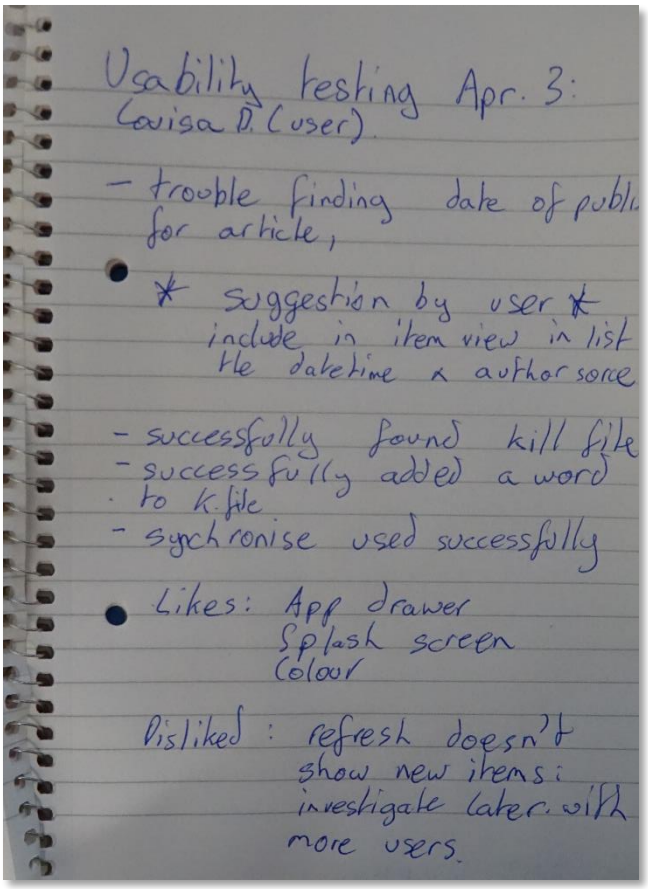
This process was conducted with 3 distinct groups of people; my family, friends and fellow students. The approach I took to testing was a combination of methods. Firstly, a method commonly referred to as "Hallway testing" – testing in which the users were simply asked to try the app and observed

during use. After the app's use I asked them to list the things they disliked, if any, as well as 3 things they enjoyed about the app.

This allowed me to both identify bugs and weaknesses in the app to improve on as well as gain a better understanding of the app's strengths, to ensure future decisions do not compromise my user's favourite areas.

The second type of testing I performed was what I call "Usability" testing, where I would ask the user to perform a task which I knew the app was capable of, and observe if and how they were capable of doing so. This form of testing is useful for evaluating the UI design in terms of feature accessibility, and drove decisions during the project such as which menu items to include in the Navigation Drawer, what the article items should look like in a list and more.

The feedback from these testing sessions was generally taken face to face, as I was always present during the testing. I noted down any points made by the user or observations I came up with myself in a notebook to ensure I did not forget any feedback.



An example of one of the usability test session notes.

7.3 – Performance Testing

This section outlines details behind my General Performance testing, String Search Component testing, UI Thread performance testing and Automated Unit Testing.

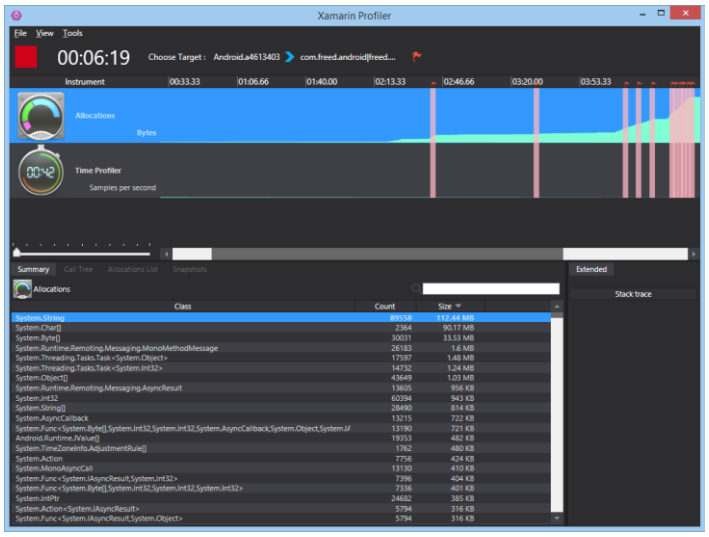
7.3.1 – General Performance Testing with Xamarin Profiler

During development, I used Xamarin Profiler to assist in performance testing the app as a whole. By using the object allocation and time profilers, I was able to visualise the impact of various user actions on the system. In using the app and testing new functionality with the profiler running, I was

able to ensure that functionality did not needlessly use memory or processor time by analysing which components of my code were causing the most usage and then evaluating alternatives.

To illustrate this further, below is an explanation of how I reduced the memory usage of my implementation of JSON serialization using the tool.

While testing the app after implementing Json.NET serialisation, I noticed a great deal of allocations being caused by String types. This struck me as odd, since I hadn't directly written any code which could cause such an outcome.



Xamarin Profiler showing the memory leak in question. The green graph indicates allocations over time.

Upon further inspection, I saw that the stack trace indicated the NewtonSoft.Json serializer was allocating the strings in question, which lead me to believe that the images I was converting to strings and serializing to disk when viewed were not being disposed of.

This lead me to re-evaluate my solution and discover the cause was a nuance of Garbage collection in C# - objects greater than 85kb in size are assigned to the large object heap. To summarise a complex topic, this essentially means they are garbage collected much less frequently and can cause heavy memory usage. [25]

After learning this, I went through my code and changed any implementation logic which might allocate a large object to work with a "using Stream" block instead, which uses buffers to allocate small portions of the data at a time and then disposes when reaching the end of the using block. This alleviated the memory leak.

```
using (Stream s = await file.OpenAsync(FileAccess.ReadAndWrite))
using (StreamWriter sw = new StreamWriter(s))
{
    JsonSerializer ser = new JsonSerializer();
    ser.Serialize(sw, obj);
}
```

Stack trace

```
(wrapper managed-to-native) objec
Newtonsoft.Json.JsonTextReader:Re
Newtonsoft.Json.JsonTextReader:Re
Newtonsoft.Json.JsonTextReader:Re
Newtonsoft.Json.JsonTextReader:Pa
Newtonsoft.Json.JsonTextReader:Pa
Newtonsoft.Json.JsonTextReader:Re
Newtonsoft.Json.JsonReader:ReadA
Newtonsoft.Json.JsonTextReader:Re
Newtonsoft.Json.Serialization.JsonS
Newtonsoft.Json.Serialization.JsonS
Newtonsoft.Json.JsonSerializer:Dese
Newtonsoft.Json.JsonSerializer:Dese
Newtonsoft.Json.JsonConvert:Deser
Newtonsoft.Json.JsonConvert:Deser
Newtonsoft.Json.JsonConvert:Deser
Freed.Core.Helpers.StorageIOHelpe
System.Threading.Tasks.AwaiterActi
System.Threading.Tasks.Task:Proces
System.Threading.Tasks.Task:Finish
Unknown
Unknown
PCLStorage.FileExtensions/<ReadAl
(wrapper unbox) PCLStorage.FileExt
System.Threading.Tasks.AwaiterActi
System.Threading.Tasks.Task:Proces
System.Threading.Tasks.Task:Finish
Unknown
Unknown
Unknown
Unknown
System.Threading.Tasks.AwaiterActi
System.Threading.Tasks.Task:Proces
System.Threading.Tasks.Task:Finish
```

Above, an example of a "using stream" block, which self-disposes on close. Right, the stack trace of the excessive object allocation, showing code calling Newtonsoft serialiser.

7.3.2 – String Search Performance Testing (Practical Comparison)

During my implementation of the bitapSearch, I decided to investigate whether the C# language's built in .Contains() method was more performant. To accomplish this, I created a small demo app,

which levearges a system diagnostics stopwatch to count the number of milliseconds and ticks the code requires to execute.

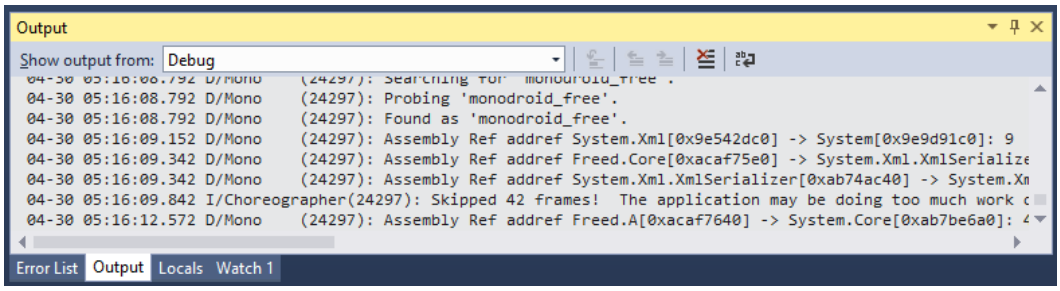
```
0 references
static void Main(string[] args)
{
    var a = args[0];
    var b = args[1];
    Stopwatch x = new Stopwatch();
    x.Start();
    var p = bitapSearch(a, b);
    x.Stop();

    Stopwatch sx = new Stopwatch();
    sx.Start();
    var f = b.Contains(a);
    sx.Stop();
}
```

Using a secondary console app to generate a random string of length 5000 to serve as a mock “article”, and inserting the string “aabbcc” into the article at a random index, I ran this test in a loop and averaged the results.

I was very surprised by the results, so much so that I decided to forgo using my bitapSearch function in the app and use the Contains method for my kill file filtering. Although my bitap search functioned well, and was fast enough to be virtually unnoticable in terms of processor load (averaging only (1646 ticks and 0.0004812 seconds to find the 6 charater sequence), the Contains() method C# provides was far superior, measuring in at only 17 ticks and 0.0000049 seconds using the same methodology. Given such drastic differences, though on quite minute scales, I reverted to the Contains() method as seen C#. This was then replaced as well, in favour of the Regex methodology outlined in previous sections, in order to efficiently resolve the issue of “word termination” as described in my planning stage.

7.3.3 – UI thread performance testing

To test the UI thread performance of my app, I configured my IDE to report when the UI thread was blocked, along with the duration of the block. I could then evaluate to what extent the responsiveness of the app was delayed with a numerical figure, if there were any delays at all.



The output window of Visual Studio showing a slight UI block of 42 frames on app launch.

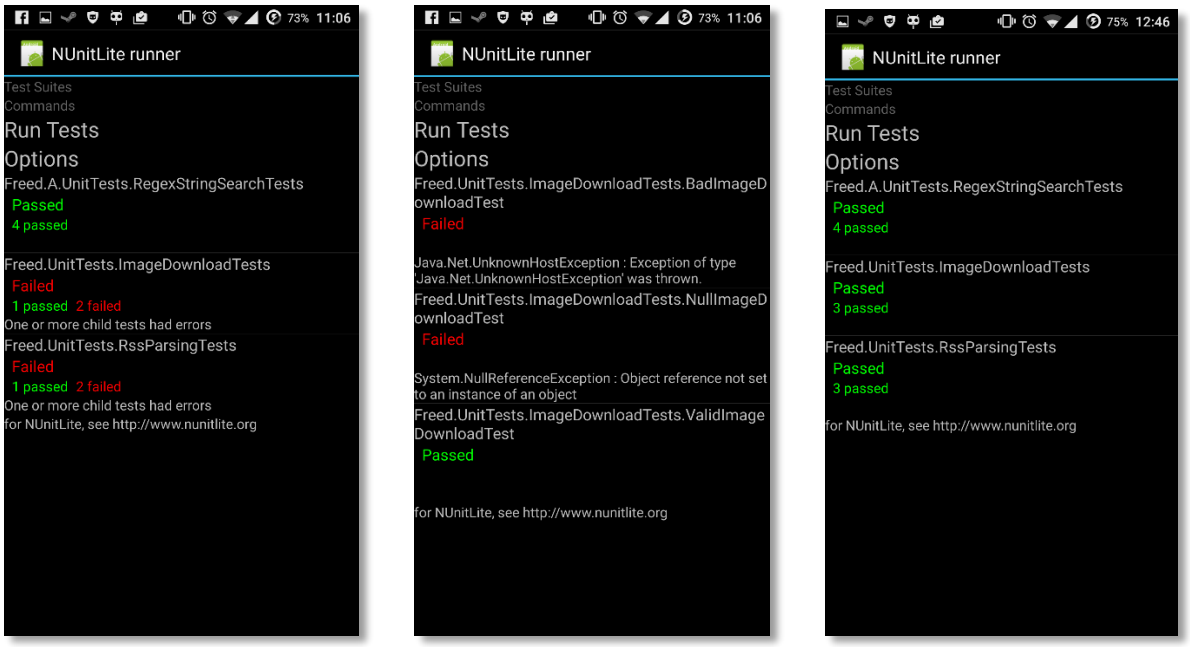
Throughout my use of the app over a half hour period, the only registered UI block was during application start, which I believe is the debugger attaching itself. The delay varied between 20 and 50 frames, and was never noticeable as a user. I am very happy with the performance of the final app, and believe it has met the responsiveness requirement in my specification.

7.4 – Automated Unit Testing

This section outlines the Unit tests written for various sections of functionality in the project. I elected to write tests for the sections of functionality which are core to the project, namely RSS feed

aggregation, Image downloading and kill file processing. Given more time to work on the project, I would like to expand these to cover a greater percentage of the project's codebase, as it is an efficient way of verifying functionality as well as preventing any regressions during code changes in future.

The unit tests in this section are contained in a project of their own, which runs on the android device. The tests and their reported success or failure can be seen on the screen. This is ideal as it helps to rule out device-specific bugs, which although does not currently apply to my project since I only have one test device, will doubtlessly be useful in future testing.



The Freed.Tests project facilitating automated Unit Testing on device.

7.4.1 – String Search: Regex

The tests for the regex string search verify that the search is correctly:

- Identifying the target strings
- Adding them to the result collection
- Returning the result collection

Additional clauses also verify that the search does not erroneously match non-contained target strings.

The edge cases tested for by this unit test include null inputs and empty strings / lists for the domain string and the list of target strings respectively. These tests are found in the `RegexStringSearchTests` class in the Freed.Tests Unit Test project.

7.4.2 – RSS Parsing

The tests for RSS parsing verify that the method is correctly:

- Returning a Boolean completion indicator
- Populating the User object with feed data

Additional tested edge cases include the handling of a malformed feed link and a null feed link. The tests for this functionality can be found in the `RssParsingTests` class.

7.4.3 – Image downloading

The tests for Image Downloading include verification that:

- The downloading of an image with a valid source
    - Ensuring the image objects data, height and width properties are correctly populated

The edge cases tested also test the download implementation's ability to handle a malformed/fake URL, as well as the ability to handle a null or empty source input. The tests for this functionality can be found in the `ImageDownloadTests` class.

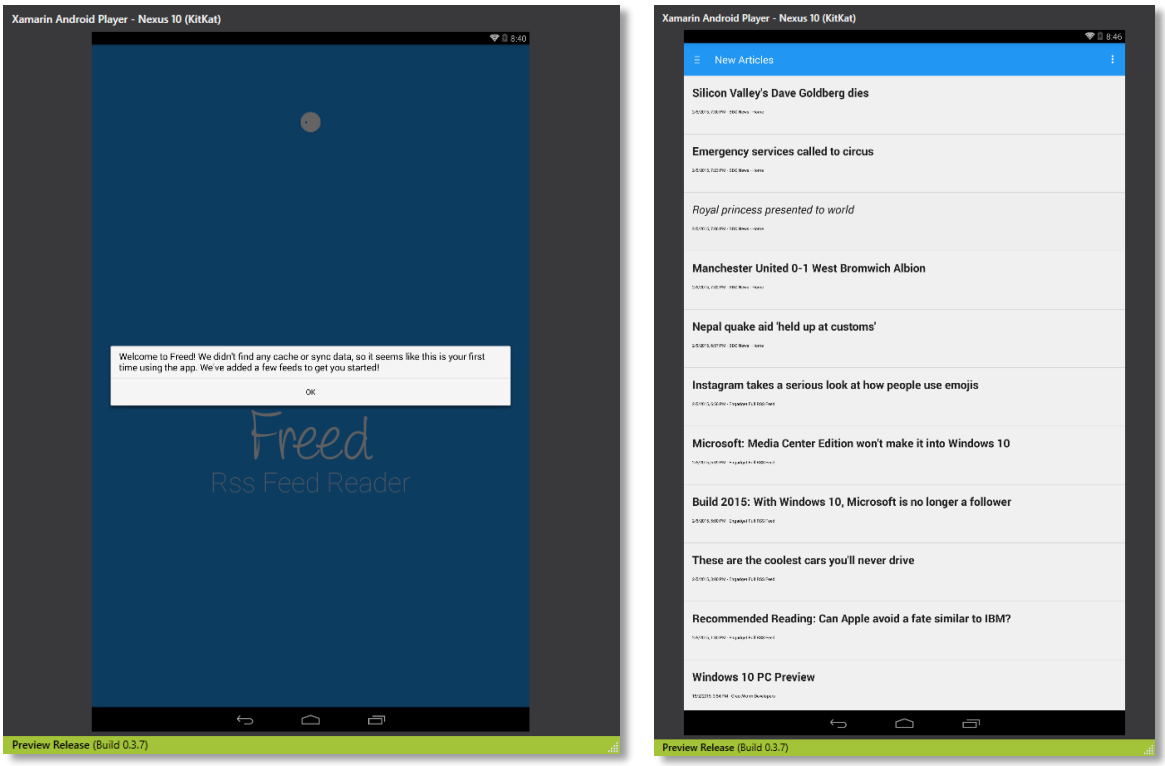
7.5 – UI Scalability Testing

Due to the limitation of not owning a 2nd android device to be able to test my software with, I was forced to rely on emulation software to test the scaling of my UI design. This is not ideal, as emulators can have quirks and problems that wouldn't be seen on a physical device, but it serves as a good general test of the function, which I would then supplement with actual device testing if I had more resources available.

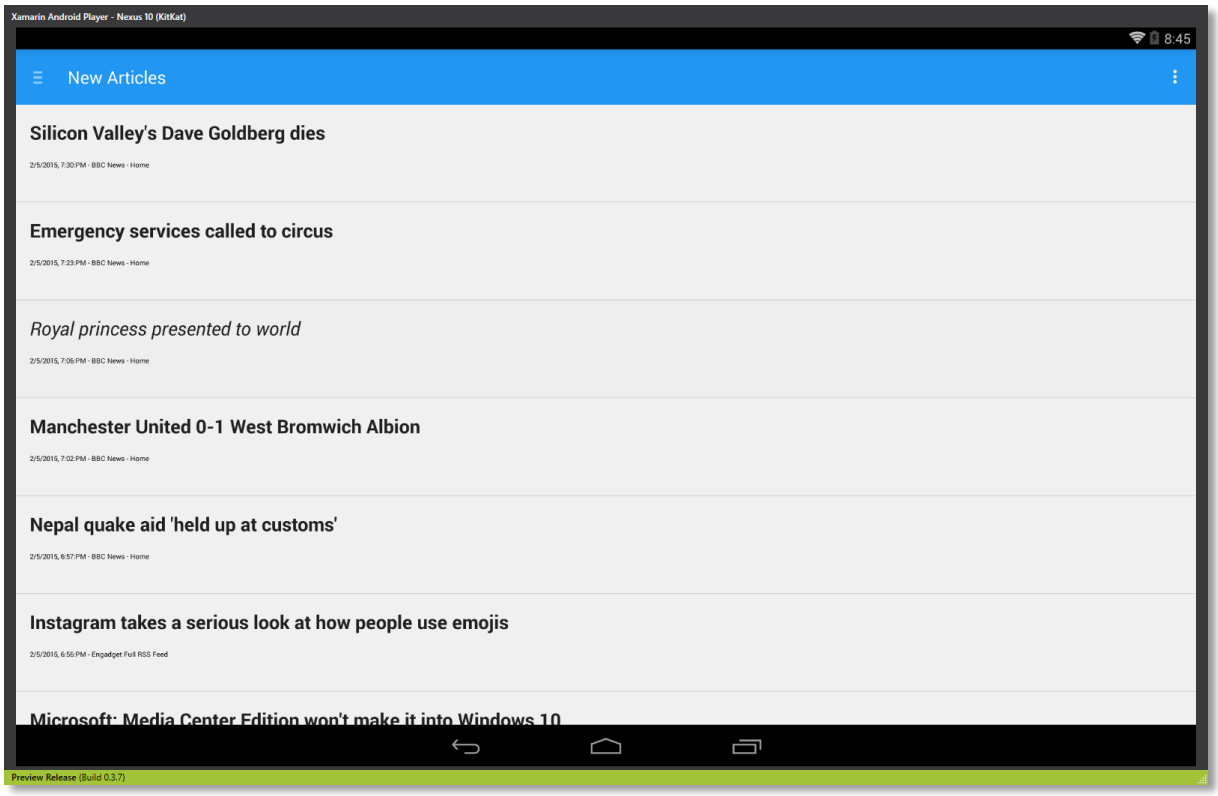
I am using the Xamarin Android Player emulator to perform these tests, with a device and image setup configured to mimic that of a nexus 10 tablet running Android KitKat. The emulated screen resolution is 1600x2560, which is larger than my display, therefore screenshots shown are a scaled down version of this resolution.

For the test, I mimicked the procedure for Usability Testing, Navigating to each screen in the app, and then reversing the steps using the back button, before generally using the app for a few minutes.

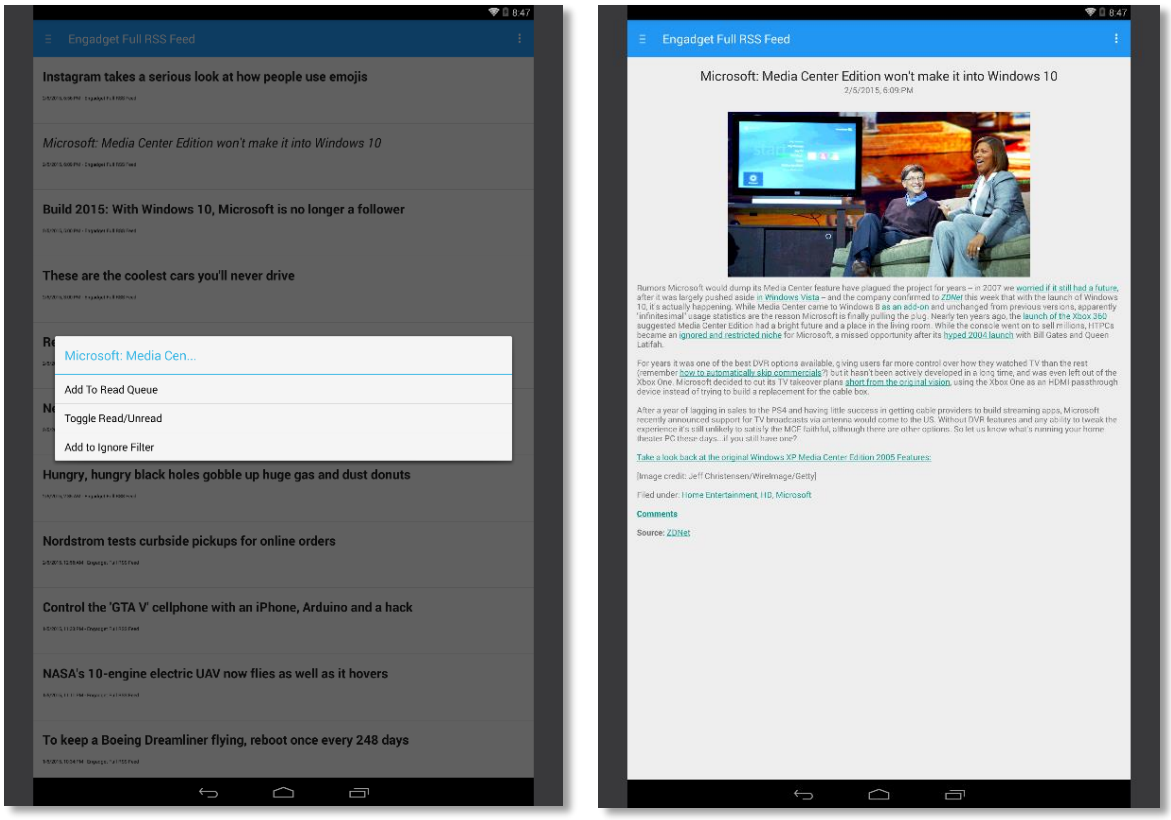
The following two pages contain some screenshots of the resulting scaling, which I found perfectly functional. Although the text appears small in the screenshots, this is because they have been scaled down for inclusion in this report, and the size was in fact perfectly readable when testing in the emulator.



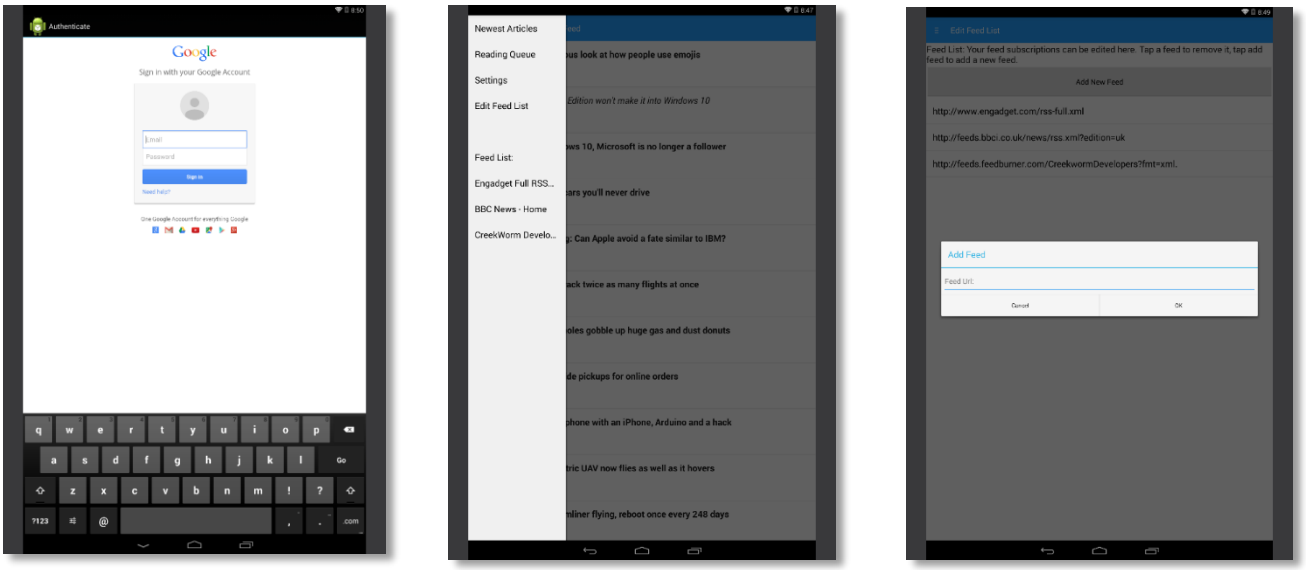
The app splash screen and new articles fragment, portrait view during emulator UI scaling testing.



The new articles fragment in landscape in the emulator.



The contextual pop up and Article detail fragments.



Google Authentication UI, navigation and Edit-List based fragments all scaled equally well.

8 – Future Work

The future work I envision for the project varies in scope and ambition, but addresses a general theme of pursuing greater polish, establishing app identity via unique feature additions, and expanding availability across platforms. As such, if I were to continue developing the project, the following sections outline my goals for each theme.

8.1 – Polish

In my creation of the projects current prototype, I have implemented a coherent and appealing UI design, however elements of the design lack refinement and could be improved upon given additional time.

As an example of this, the resource values used for various elements are sometimes inconsistent, be it for layout margins in certain sub-views or the colour resource used for the page background. Minor elements like this, while not majorly detracting from the usability of the app, can be off-putting subconsciously to the user, as they convey a lack of quality in the UI. This is a paradigm present in all forms of product creation, from the “door gap” issue car manufacturers often face problems with to the stitching quality of cloth a tailor may be critiqued upon – consumers prefer a sense of uniformity to a design, and this is something I believe I could improve on with more time to work on the project.

This also applies to non-UI elements, such as expanding the available user preferences. For example, the application currently supports the local and cloud storage of 50 articles per feed, as a hardcoded value which I settled on after user feedback. However, it would be great to take the time to build a user setting for this value instead, to facilitate all preferences.

Finally, this applies to the project codebase. Although I did my best to pursue the SOLID principles, I believe with more time I could further refactor some methods into more isolated and reusable segments to improve the abstraction and maintainability of the codebase. An example of this would be the ImageDownloadHelper in Freed.Core, which is a fairly large method with distinct segments (download and serialization) which could be broken down into isolated methods.

8.2 – Unique features and App Identity

The kill file feature found in my app serves as a great unique feature with which to attract potential users, so it stands to reason that the addition of more unique useful features would increase the attraction. As such, given more time on the project I would investigate and pursue the development of these, after validating their usefulness with a focus group or feedback session.

An example of an idea I had during development that falls into this category would be the concept of an “interests list”, where a user would add topics to an interest list and the app would fetch interesting related content by using a web search API, thus allowing the user to discover new content and feed sources without explicitly having to seek them out by adding an RSS feed subscription.

8.3 – Cross platform availability

One of my project’s goals was to enable the app to be used across multiple platforms with as little additional development effort as possible. As such, the future development goals I had in mind naturally include the creation of additional View and Controller implementations for iOS and Windows platforms. Given the reusability of a great deal of this project’s implementation (model) logic via the “Freed.Core” library, I am sure this is a viable and worthwhile future pursuit.

9 – Reflection

In this section, I will briefly reflect on some of the assumptions and decisions I made at the start of project, and evaluate them now that I have experienced the consequences and benefits during development.

9.1 – Xamarin

Reflecting on my choice to use Xamarin, I am very pleased with the outcome of the decision. Although initially quirky in areas, and harder to find support discussions for on community sites, overall the project suffered no negative consequences from the use of Xamarin. On reflection I believe the choice to use Xamarin was an appropriate one given the context and the basic requirement of cross platform availability to ease future development.

9.2 – Linq

Early on in the project I decided to use Linq to facilitate the handling of a variety of functionality in the View/Controller layers of my MVC pattern interpretation, for example where the view requests a subset of data for display, amongst other things.

Reflecting on this from a performance perspective is somewhat beside the point – although Linq is quite performant, an explicit loop is almost always faster computationally. The benefits of Linq are found in the cleanliness and ease of implementation, which pays dividends in the maintenance cost of the project decreasing, as well as improving readability of complex loops for better class clarity. As an example, I took a typical Linq statement from my application, and attempted to implement it without using Linq:

```
frag1.feed.channel.Articles = User.Instance.CurrentFeedData
    .SelectMany(x => x.channel.Articles.OrderByDescending(y => y.pubDateTypeFormatted).Take(ArticlesPerFeedToShow))
        .Where(z => !FeedDataKillFileParser.IsFilterFlagged(z, User.Instance.UserSettings.KillFileEntries))
        .OrderByDescending(x => x.pubDateTypeFormatted) // reorder to adjust for multiple sources mixing date sorts
        .ToArray();
```

The Linq example, taken from the NewArticlesFragment.

```
List<List<rssChannelItem>> result = new List<List<rssChannelItem>>();
foreach (var feed in User.Instance.CurrentFeedData)
{
    int index = 0;
    List<rssChannelItem> sortedArticlesByPubDate = new List<rssChannelItem>();

    foreach (var article in feed.channel.Articles)
    {
        if (!sortedArticlesByPubDate.Count > 0)
        {
            sortedArticlesByPubDate.Add(article);
        }
        for (int i = 0; i < sortedArticlesByPubDate.Count; i++)
        {
            if (article.pubDate > sortedArticlesByPubDate.ElementAt(i))
            {
                sortedArticlesByPubDate.Insert(i - 1, article);
            }
        }
    }

    var subSort = new List<rssChannelItem> { sortedArticlesByPubDate[0], sortedArticlesByPubDate[1], sortedArticlesByPubDate[2], sortedArticlesByPubDate[3], sortedArticlesByPubDate[4] };
    result.Add(subSort);
}

foreach (var list in result)
{
    foreach (var feed in list)
    {
        int index = 0;
        List<rssChannelItem> sortedSubsetByPubdate = new List<rssChannelItem>();

        foreach (var article in feed.channel.Articles)
        {
            if (!sortedSubsetByPubdate.Count > 0)
            {
                sortedSubsetByPubdate.Add(article);
            }
            for (int i = 0; i < sortedSubsetByPubdate.Count; i++)
            {
                if (article.pubDate > sortedSubsetByPubdate.ElementAt(i))
                {
                    sortedSubsetByPubdate.Insert(i - 1, article);
                }
            }
        }
    }
}
```

My uncompleted attempt at implementing a similar functionality using loop constructs.

The length of the implementation and nesting of loops should illustrate the point here- linq helps to greatly simplify complex object management for a small performance tradeoff. Although I would

hesitate to use Linq for extremely performance critical scenarios such as enterprise code, for client-level applications such as my project it is a great tool. The added simplicity of development allowed me to speed up my iteration cycles greatly. I would not hesitate to use Linq for similar applications in future.

9.3 – Android

I am happy with the projects performance, style and interaction mechanisms on the Android platform. Though arguably the same results would have been possible on Windows Phone or iOS platforms, the use of Android enabled much more practical testing (as I do not own a device for the other platforms) as well as the potential to target a larger user base if I choose to release the app in future. Therefore I think it was the right decision to start the development on Android.

9.4 – Regex

The use of Regex in my project to implement the Kill File functionality was arguably the only reasonable method of implementation in the given timescale. It provided a great solution to the problem of word termination, and allowed for the use of a built in (and thus self-maintaining) string search instead of my custom-implemented “Bitap” search algorithm. The only downside to the regex implementation is that it was not possible to implement the “suggestion” feature for adding words to the kill file due to the low quality of suggested results that would have been provided by blindly appending various suffixes to the user input, however since there are no available libraries which provided the reverse-lemmatization needed, I do not see this as a Regex issue as much as a general feasibility one. As such I am happy with the use and functionality of Regex in my project.

9.5 – Material Design

The adherence to Material design guidelines was a time consuming one (as showcased in Section 4.1), but one which proved extremely valuable. During usability testing all of the testers responded that the design notably improved the readability and intuitiveness of the UI, and even at a glance it is obviously far superior to the prototypes outlined in the design section, both aesthetically and functionally.

On reflection, I would have spent a little more time researching and fully understanding how best to create and apply the styles in a reusable fashion, as looking back I did not make much effort to enable reuse of layout files and resources. This is something I will learn from and address in future work on the project.

10 – Conclusion

The project's aims and goals outlined that the end result would be a News Reader app with Kill File and Cloud Synchronisation capabilities. By the end of the project, this was successfully implemented along with a great deal of additional features and design iteration to produce what I believe is a quality application. The UI follows material design principles and uses interesting control gestures such as swipe to refresh, feed data can be added, removed and refreshed as desired, the kill file successfully filters articles based on user input and the app's entire dataset is serialised to and from the cloud on app close and open respectively.

The app is also produced this in a manner that is built on a solid foundation for future development and expansion by using the Xamarin platform to enable me to easily scale the app to iOS and Windows platforms, which is something I hope to do in the near future.

In conclusion, I believe the project overall was very successful, and I learned a great deal from the challenges and experiences of implementing it.

11 – References

[1] – Xamarin – http://xamarin.com
(as seen on 23rd March 2015.)

[2] – The #1 Google Play Store Region by Most Downloads is the United States - http://blog.appannie.com/app-annie-index-market-q1-2014
(as seen on 23rd March 2015.)

[3] – Announcing Xamrarin - http://tirania.org/blog/archive/2011/May-16.html
(as seen on 23rd March 2015.)

[4] - .NET Alternative in Transition - http://www.informationweek.com/architecture/net-alternative-in-transition/d/d-id/1098050
(as seen on 23rd March 2015.)

[5] – Downloading the Source - https://source.android.com/source/downloading.html
(as seen on 24th March 2015.)

[6] - Announcing the Android 1.0 SDK, release 1 - http://android-developers.blogspot.in/2008/09/announcing-android-10-sdk-release-1.html
(as seen on 24th March 2015.)

[7] - Google's Android becomes the world's leading smart phone platform - http://www.canalys.com/newsroom/google%E2%80%99s-android-becomes-world%E2%80%99s-leading-smart-phone-platform
(as seen on 24th March 2015.)

[8] – Apache Cordova is a platform for building Native Mobile Applications using HTML, CSS and Javascript - https://cordova.apache.org/
(as seen on 24th March 2015.)

[9] - Samsung and Flipboard team up to bring its signature magazine-style newsfeed to the Samsung Galaxy Note - http://www.cnet.com/pictures/flip-through-samsungs-flipboard-for-phones-and-tablets-pictures/
(as seen on 25th March 2015.)

[10] - LinkedIn Acquires Pulse For $90M In Stock And Cash - http://techcrunch.com/2013/04/11/linkedin-acquires-pulse-for-90m-in-stock-and-cash
(as seen on 25th March 2015.)

[11] – The Feedly Team - http://feedly.com/about.html
(as seen on 25th March 2015.)

[12] – Navigation Drawer - http://www.google.com/design/spec/patterns/navigation-drawer.html
(as seen on 25th March 2015.)

[13] – RSS 2.0 Specification, Harvard Law - http://cyber.law.harvard.edu/rss/rss.html
(as seen on 27th March 2015.)

[14] – Drive REST API - https://developers.google.com/drive/web/query-parameters
(as seen on 1st April 2015.)

[15] - Knuth, Donald; Morris, James H., jr; Pratt, Vaughan (1977). "Fast pattern matching in strings". *SIAM Journal on Computing* **6** (2): 323–350. doi:10.1137/0206024. Zbl 0372.68005.

[16] - Aho, Alfred V.; Corasick, Margaret J. (June 1975). "Efficient string matching: An aid to bibliographic search". *Communications of the ACM* **18** (6): 333–340.

[17] - Karp, Richard M.; Rabin, Michael O. (March 1987). "Efficient randomized pattern-matching algorithms". *IBM Journal of Research and Development* **31** (2): 249–260. doi:10.1147/rd.312.0249. Retrieved 2013-09-24.

[18] - Boyer, Robert S.; Moore, J Strother (October 1977). "A Fast String Searching Algorithm.". *Comm. ACM* (New York, NY, USA: Association for Computing Machinery) **20** (10): 762–772. doi:10.1145/359842.359859. ISSN 0001-0782.

[19] - Bálint Dömölki, An algorithm for syntactical analysis, Computational Linguistics 3, Hungarian Academy of Science pp. 29–46, 1964

[20] – String.IndexOf Method – https://msdn.microsoft.com/en-us/library/k8b1470s(v=vs.110).aspx (as seen on 14th April 2015.)

[21] - https://msdn.microsoft.com/en-us/library/dsy130b4%28v=vs.110%29.aspx#linear_comparison_without_backtracking (as seen on 14th April 2015.)

[22] – LemmaGen – http://lemmatise.ijs.si/Home/Overview (as seen on 14th April 2015.)

[23] - https://developers.google.com/drive/web/manage-uploads (as seen on 16th April 2015.)

[24] - https://developers.google.com/drive/web/ (as seen on 16th April 2015.)

[25] - https://msdn.microsoft.com/en-us/magazine/cc534993.aspx (as seen on 24th April 2015.)

Acknowledgements

I would like to extend my utmost gratitude to those who supported me in the creation of my project, including:

- My supervisor, Professor Ralph Martin, for his advice and encouragement during our review meetings, and the idea upon which this project was based.
- Nabil Chaaban and Rob Hamilton, for their feedback and opinions on the drafts of this report.
- My usability testers, for contributing their time and efforts to test the app during development.
- My friends and family, for their support and guidance throughout the project.

Additionally, I would like to thank the teams who develop the tools and programs this project used during its creation.

Appendix

The appendix to this report is provided as an electronic attachment, which includes the compiled and signed Android .APK package, as well as a .ZIP file containing the full source code.