

*Stock portfolio manager for Android*

Final Year Project

Interim Report

Author: David Gordon – 11108215

Supervisor: Michael Coughlan

B.Sc. Computer Games Development

Department of Computer Science and Information Systems

University of Limerick

Ireland

**Contents**

**1. Project Summary**

1.1 Project Summary.................................................................................... 3

1.2 Abbreviations.............................................................................................3

**2. Introduction**

2.1 Project Overview/Background.................................................................................... 4

2.2 Objectives of Proposition............................................................................................. 4

2.3 Motivating Factors………..............................................................................……….5

**3. Research**

3.1 Implementation Technologies………………………………………………….......6

3.1.1 IDE (Eclipse vs Android Studio)………………………………………………6

3.1.2 Financial APIs………………………………………………………………….6

3.1.3 SQLite vs MySQL……………………………………………………………...7

3.2 Benchmarking………………………………………………………………………..7

3.2.1 Previous FYPs…………………………………………………………………7

3.2.2 Existing Products……………………………………………………………..8

3.3 Exploratory Program……………………………………………………………...9

**4. Description of Current Progress**

4.1 Design………………………………………………………………………………10

4.2 Implementation……………………………………………………………………....11

4.2.1 Retrofit Interface Implementation and Issues…………………………………..11

4.2.2 JSON to POJO………………………………………………………………….11

4.2.3 Environment Volatility…………………………………………………………12

4.2.4 The Observer Design Pattern…………………………………………………13

4.2.5 Networking Issues……………………………………………………………13

**5. Project Plan**

5.1 Project Plan……………………………………………………………………14

**6. References**

6.1 References………………………………………………………………………15

**1.1 Project Summary**

This project aims to create a stock portfolio manager for a mobile device. The application will allow users to keep information about the current stocks in their portfolio as well as add additional tranches of stocks to this portfolio. The application will provide the user with multiple tools in order to analyse the potential of each individual stock item. These will range from basic charts and graphs to more advanced share selection tools, such as the dividend yield or Fools Ratio calculator. The application will show the current value of your portfolio by downloading the current prices from the Yahoo Finance API. One limitation is that the project should apply only to the Irish stock exchange (ISEQ). The application will provide a basic view of each individual stock item. This will consist of the stock’s name, ticker id, current price and an indication whether the stock has risen or fallen. Further analysis of each individual item can be performed as required by the user.

This project aims to keep track of the user’s favourite stocks in one place. This allows the user to make the most of their resources and aids them in gaining the highest return on investment possible once they enter the stock market in earnest.

**1.2 Abbreviations**

SDK: Software Development Kit

ISE: Irish Stock Exchange

API: Application Programming Interface

UI: User Interface

GUI: Graphical User Interface

JDK: Java Development Kit

IDE: Integrated Development Environment

JSON: JavaScript Object Notation

XML: Extensible Markup Language

POJO: Plain Old Java Object

ADT: Android Development Toolkit

adb: Android Debug Bridge.

AVD: Android Virtual Device

REST: Representational State Transfer

JAR: Java Archive

YQL: Yahoo Query Language

URL: Uniform Resource Locator

MVP: Minimal Viable Product

**2.1 Project Overview/Background**

This project is designed to take advantage of the Android SDK **(Google, 2008)**.

This project is aimed at people interested in the Irish stock market who own an Android device. The target audience will be quite diverse as the application is not exclusively aimed at those coming from a business background. During my research I found myself getting lost in some of the financial applications due to the sheer number of stocks and stock exchanges they encompassed. As my application will be limited to the Irish Stock Exchange I believe that it will serve as a perfect introductory application for those who are interested in the stock market yet inexperienced when it comes to how it works.

Although my application is not exclusive to business people, it can be presumed that those with some sort of experience in stocks would be most likely to use the application. Therefore the tone of the applications UI is very important. Many well-known applications have a lighter subject matter than found in stock portfolio managers. Instagram and Twitter for instance deal in sharing opinions and photos between friends. Therefore, many design principles espoused by the Android community, such as flat design etc. may not be suitable in this particular context, with the user more interested in functionality than the ‘bells and whistles’.

**2.2 Objectives of Proposition**

The objectives of my project are as follows

* Create a stock portfolio manager for Android
* Create an application that uniquely focuses on the Irish Stock Exchange (ISEQ)
* Provide an alternative application to the many found on the Google Play Store.
* Combine common functionality found in stock portfolio applications with an attractive, intuitive user interface.
* Upload my finished application to the Google Play Store.
* Improve my skills as a designer, developer and tester of Android applications.

**2.3 Motivating Factors**

My desire to become a junior Android developer after graduating proved to be one of the major motivating factors behind choosing an Android application as my final year project. For my work placement, I interned as an Android developer for a mobile company in Barcelona, Spain. This experience proved to be highly rewarding and led me to consider Android development as a possible career path. Enhancing my Android application portfolio will improve my employability in this regard.

During my internship I had developed software as part of a team in an agile environment. However, I have never had the opportunity to develop a large scale application from the ground up encompassing the design, development, testing and deployment stages. Being involved in all stages of producing a successful mobile application will hopefully increase my knowledge of the process as a whole and not just the development cycle.

I have always wanted to work with the most cutting edge technologies and developing for Android allows me to do just that as Google constantly introduces new features on top of its software stack. Android ‘Lollipop’ or 5.0 **(Google, 2014),** was released in November 2014 and is available to developers free of charge. This may potentially give me an advantage on other stock portfolio managers available on the Goggle Play store, as they were developed with older operating systems in mind.

This project also gives me the chance to learn more about the particularities of the stock exchange as a whole. I have always had an unexplored interest in the markets and I feel like this project will guide me towards a greater understanding of the subject at hand. Hopefully by the projects conclusion I will not only have a fully functioning mobile application, but a clear understanding of how the exchange works as a bonus.

All the software and hardware required are available to me for this project. The Android SDK **(Google, 2008)** as well as Java runtime environment (JDK 1.70\_15) installed on my device are free to download and use. The ‘Genymotion’ **(Genymotion, 2013)** emulator I will use also comes free of charge. The testing device I will use is a LG Nexus 4.

Finally, the well-defined scope of the project description was a real motivating factor in choosing it. Being inexperienced when it comes to developing large scale software applications, I am wary of the risk associated with maintaining a project like this. Increasing the scope without due cause tends to be one major pitfall of the overenthusiastic developer. In the project description, presented was a log of requirements detailing what is expected from the application, more detailed than what was found for other projects. Fulfilling these requirements will be my primary goal and will allow me to stay focused. The scope can then be increased depending on time constraints. This will also mimic the workflow of a mobile application company, allowing me to prepare for future employment. In this scenario the client comes to the development company with a set of requirements. The developer delivers said requirements, along with any additional features they have time to add.

**3 Research**

**3.1 Implementation Technologies**

For this project I used the JDK (jdk1.7.0\_15) along with the Android SDK. When developing native applications for Android there is no other option than using this combination of software packages.

* *IDE (Android Studio vs Eclipse)*

Android Studio (**Google, 2013**) is the official IDE for Android application development, based on IntelliJ IDE. Although still in beta, it has many advantages over its older competitor, Eclipse (**Eclipse Foundation,** 2004). I have sufficient experience using Eclipse with the Android plugin. However, I feel as though its user interface is a bit dated. For this project I have decided to move forward with Android Studio. This decision was made simple for me during the development of a prototype (discussed in section 3.3). When trying to integrate a particular library from GitHub (**Github 2008**) into my application, Eclipse was having trouble recognizing the JAR file in its build path. Even more worrying was the fact other similar libraries suitable for my prototype did not even provide a JAR file. It became apparent many GitHub contributors were primarily accommodating Android Studio users. Android Studio uses a build automation tool called Gradle (**Gradle 2009**). This allows the user to add libraries to their application by adding a dependency to their Gradle structure as one single line of code. The simplicity of integrating new libraries was a major motivating factor in choosing Android Studio over other competitors.

compile 'com.squareup.retrofit:retrofit:1.8.0'

* *Financial API’s*

My application will need to interact with an API in order to receive the stock information needed. I initially believed this would be a relatively simple task. However, a large proportion of the financial API’s researched have major drawbacks associated with them. The Google finance API **(Google 2006)** for instance has been depreciated since 2011. Other examples, such as the Open Exchange Rates API have a subscription fee associated with them. The API I choose to develop my prototype with was the same one mentioned in the project description. The Yahoo Finance API **(Yahoo!, 2008)** returns data as either JSON or XML depending on the developer’s preference. I was impressed by the amount of detail received from each API call as well as its ease of use. However, the Yahoo API was similarly not without its flaws. As the API is maintained by the community, some of the data it returns is corrupt or even non-existent. For instance a call to the current price of Bank of Ireland may return 1.97, while the same call to Allianz Insurance might return ‘null’. This caused me some problems during development of the prototype; however these were easily rectified by multiplying the change in percent with yesterday’s closing price. Another API I have been looking at recently is the Bloomberg Financial API. Once subscription based this is now available to developers free of charge. The API is extremely well documented and robust and could serve as a viable alternative to the Yahoo Finance API during the development process ahead.

* *SQLite vs MySQL*

As a developer of a stock portfolio manager, one must find a way to store the stock information after a purchase so that it can be retrieved and manipulated at a future point in time. “SQLite (**General Dynamics 2000**) is an open source database embedded in android that supports all standard relational database features such as SQL syntax and transactions“(**Mobisys, 2012**). It is easy to use, as it does not require any database setup or administration. The database is automatically managed for the developer. This is sufficient if no social aspect is expected from the application. However, if the application was to evolve at a future stage so that users could compete against each other on public or private leader boards for example, than using an MySQL**(Oracle, 1995)** database server side would be a better solution. Designing for change obviously has its own advantages, however the potential added complexity is the reason why I decided to go with SQLite.

**3.2 Benchmarking**

**3.2.1 Previous FYPs**

I found several FYPs that dealt with developing native Android applications while researching previous FYP reports. While there had been attempts to develop a stack portfolio manager before, the project was left unfinished with no report or final application submitted. I therefore had to investigate projects that made excellent use of the platform rather than relying on projects that had direct relevance to stock portfolio management. One example of such is the “Fall-down monitor for accelerometer-based mobile phones” **(Curtin 2012)**

The purpose of the application is to send “notification policies to alert emergency contacts in the event of a detected fall and invoke a distress call to summon help when required.” This project is slightly irrelevant to mine when it comes to implementation specifics. I will not be required to work with the phones hardware, particularly the accelerometer at any stage of development. However the report has a full section detailing how to publish and monitor an app on the Google Play store. As I have no prior experience in publishing applications I feel as this will be of great value nearing the projects conclusion.

I have often struggled to efficiently test and optimize my code. This is something I believe many inexperienced developers have problems with. Working with Eclipse means developers often come across tools belonging to the ADT, without ever truly understanding their purpose. ProGuard for instance, I have heard used numerous times. However it was not until reading this report that I was given a simple to understand summary of what it actually does. This report does an excellent job of describing some of the lesser known elements of the Android development environment. These range from the adb to the icon set wizard.

The report also brings to my intention a source code analyzer named PMD. In the past I have struggled to find a code analyzer which I am comfortable using. I am keen to get in contact with Eric to discover just how useful PMD was during the testing phase of his project.

**3.2.2 Existing Products**

The Google Play Store provides numerous financial applications similar to the stock portfolio manager I will develop. I believe that it was important to research not only mobile applications but also some web based ones. Web based applications tend to have a more advanced feature list than those found in mobile devices. This meant I could potentially implement some features not found in competitor applications. The five applications I researched were TheStreet, JStock and MarketWatch (Android), as well as the Financial Times and Bloomberg portfolio managers (Web).

* *Financial Times Stock Portfolio Manager*

This application contains much of the functionality I hope to provide in my final application. On startup the user is greeted by a dashboard which contains all your portfolios and any relevant news relating to owed stock items. The dashboard itself is slightly plain when compared to Bloomberg for instance; however it never serves as a distraction to the main goal of portfolio management. It provides an options menu to view all your past transactions. This is something that was not included in the original project spec for my application. However, I feel as though incorporating functionality to keep track of transactions would be a worthwhile addition to the application as a whole. In depth graphs are also provided for each stock item in the portfolio. However, as this is a web based tool it is likely that html, css and JavaScript were used to develop it. Visual data representations such as bar and line charts are much more common in web pages than they are in mobile applications. This is due to the abundance of graph APIs available for that platform. On the other hand a google search for “Android graph APIs” gave me the following result. <http://www.stackoverflow.com/questions/6806537/graphs-api-for-android> **(Stack Exchange Network, 2008).** It seems a lot of the dynamic Android Graph APIs are either depreciated or poorly documented. For that reason I will implement simpler versions of what is found in the Financial Times and Bloomberg web applications.

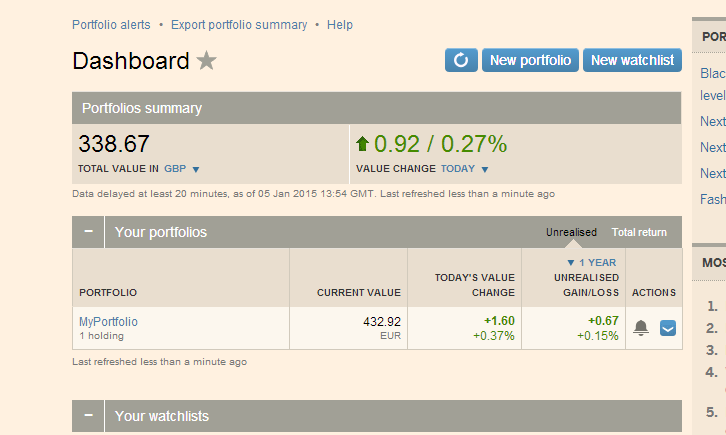


Fig 1.1

**3.3 Exploratory Program**

Although I have had previous experience developing android applications for mobile devices, my supervisor thought it would be beneficial to build a simple prototype. This prototype uses some of the technologies which will be incorporated into the final version of my application.

Initially, there were two major areas of concern. Where exactly would I find the data required to populate my application, and how would this data be delivered quickly and safely to my application? Developing a prototype allowed me to become familiar with the YQL API **(Yahoo!, 2008**) which I will be using as the source of the detailed stock data required by my application. I have also gained some knowledge in using a RESTfull client called Retrofit **(Square Open Source, 2013)** as a means of delivering this data to my application.

On start-up, the program would simply query the YQL API **(Yahoo!, 2008)** for the required data. The aforementioned data would then be delivered to the application by Retrofit **(Square Open Source, 2013)** and displayed hierarchically as stock items. Each stock item would be represented by a stock name, ticker id, as well as a change in percent from the previous day’s closing price. A colour, green or red represents a positive or negative change respectively.

Developing this prototype also gave me the opportunity to work with some of the new API’s released in Android 5.0 **(Google, 2014)**. The list was implemented using RecyclerView rather than the standard ListView found in most mobile applications. RecyclerView was introduced in androids newest release as a replacement for ListView. This will soon become a fundamental part of android application development; similar to ListView before it. Therefore, I saw this prototype as the perfect opportunity to implement it for the first time.

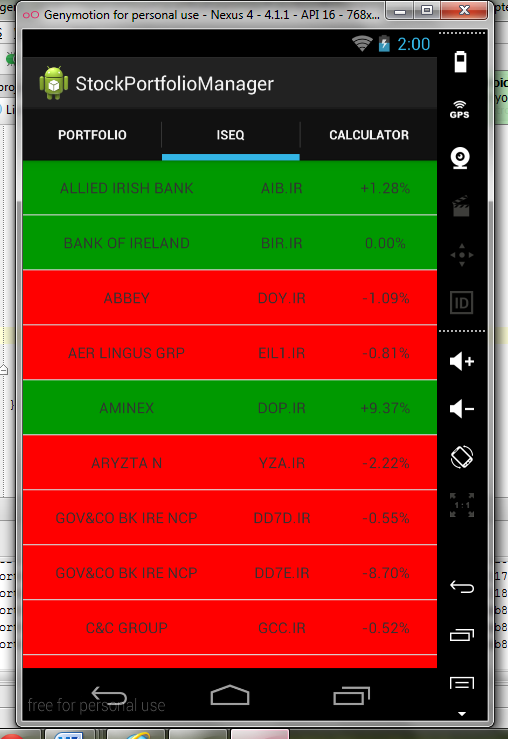


Fig 1.2

**4 Description of Current Progress**

**4.1 Design**

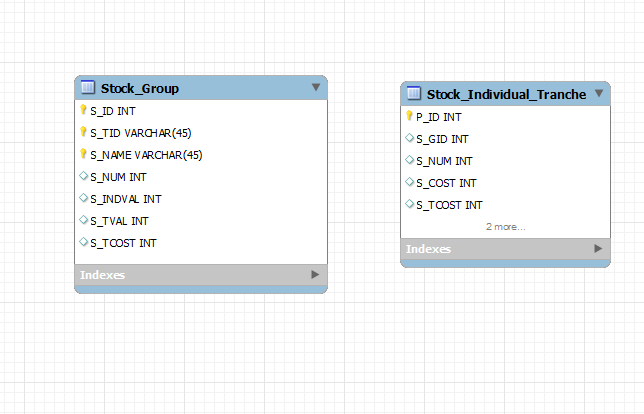


Fig 1.3

Database schema for stock portfolio manager. The left table, Stock\_Group represents each stock item in the ISEQ. Therefore Stock\_Group will consistently have fifty seven data entries, one entry for each stock, e.g. Bank of Ireland, Allianz Insurance etc. S\_ID is an auto incrementing integer which stands for Stock Id. S\_TID represents ticker id (e.g. BIR.IR) while S\_NAME defines the stock name (e.g. Bank of Ireland). All three of these values will be Primary keys, unique to that entry. S\_NUM shows the number of each stock that that the user has bought i.e. is in their portfolio. Representing the current individual value of that particular stock is S\_INDVAL. This value is taken from the YQL API **(Yahoo!, 2008)** on startup and so is constantly updated and ever changing. S\_TVAL is used to show the total value of that stock item owned by the user. This is calculated by multiplying the current individual value of the stock (S\_INDVAL) by the number of stocks owned (S\_NUM). S\_TCOST is the amount spent purchasing S\_NUM amount of that particular stock item. To clarify things, let’s take Bank of Ireland stocks as an example. Let’s say the user has bought 100 Bank of Ireland stocks. S\_TVAL represents how much these 100 Bank of Ireland stocks are now worth while S\_TCOST represent how much they were purchased for. S\_TCOST is calculated from the Stock\_Individual\_Tranche table.

The Stock\_Individual\_Tranche table represents each individual tranche of stocks bought by the user. P\_ID is an auto incrementing integer which stands for Purchase Id. S\_GID represents the Stock Group Id. There is a direct correspondence between S\_GID and S\_ID found in the Stock\_Group table. If for instance Bank of Ireland stocks have an S\_ID of 1, then purchasing a tranche of Bank of Ireland stocks will correspond to a S\_GID of 1 for that purchase. S\_NUM represents the number of stocks bought in that purchase. S\_COST refers to the cost of an individual stock for that purchase. S\_TCOST is calculated by multiplying S\_COST by S\_NUM and its represents the total cost of that purchase.

To calculate S\_TCOST for the first stock item in Stock\_Group we could use a SQL query similar to the following

*SELECT* *SUM* (S\_TCOST) *FROM* Stock\_Individual\_Tranche *WHERE* S\_GID = 0;

**4.2 Implementation**

Android mobile applications are developed in Java **(Oracle, 1995)** while making use of some APIs specific to the Android platform. XML is also used for the user interface. Android binds the Java code to its XML counterpart in a way that the user interface and the business logic underneath it are loosely coupled.

Here are some specific implementation scenarios I have come across during development thus far.

* *Retrofit Interface Implementation and Issues*

“Retrofit is a type-safe REST client for Android and Java” **(Square Open Source, 2013)** Retrofit needs you to set an endpoint for its request, which I set to <https://query.yahooapis.com/v1/public>. Then I had to define the interface itself with a relevant Http annotation (Get, Post etc.) As I wanted to read information from a server rather than write to a server, I used @Get. Then you specify the request URL. I initially believed that I needed queries for this, however in foresight the request URL will never be changed at runtime (Application will always require all data on all Irish stocks) and so a static request URL was sufficient.

**public interface StockApiInterface {**

**@GET*(/\*****STATIC* *REQUEST URL\*/)*

**ResultWrapper listQuotes(); //** Explained in next section

**}**

* *JSON to POJO*

The idea of the Retrofit interface is to receive JSON (or XML) from a web server and convert it to relevant POJOs. However, the JSON I was receiving from the Yahoo API was very verbose; needing over ten different Java classes to encapsulate the response. The JSON had five levels of nesting, making it hard to visualise the data as JSONArrays and JSONObjects. To overcome this I only had to change the diagnostics in the static request URL from true to false. This returned a much more manageable JSON response.

@GET("/yql?q=select%20\*%20from%20yahoo.finance.quotes%20where%20symbol%20in%20(" + Constants.STOCK\_INDEXES +")%0A%09%09&**format=json&diagnostics=false**&env=http%3A%2F%2Fdatatables.org%2Falltables.env&callback=")

As Retrofit **(Square Open Source, 2013)** has a built in Gson **(Google 2008)** converter, It will automatically parse the JSON request however the developer much specify the Java classes that the JSON will be converted to. For this I simply used an online JSON to POJO converter. ‘jsonschematopojo.org’ **(Github, 2011)** was my preferred choice. By simply inputting JSON data into an editor the program would output all needed Java classes to parse such data.

However even after receiving a more manageable JSON response I was still having problems converting the data to my Java objects. For example, when trying to access the quotes data members such as symbol, I would be given a ‘*NullPointerException’*. This was due to the fact Retrofits built in Gson converter was not correctly converting the JSON data to objects. In the end I needed a wrapper class that would encapsulate all JSON data. Then by using:

Quote[] quotes = wrapperClass.getQuery().getResult().getQuotes();

I could access the quote. Then using:

quotes[i].getSymbol();

I was able to access the specific Quotes symbol.

//Retrofit Interface

public class RetrofitInterface {

private static StockApiInterface sStockService;

public static StockApiInterface getStockApiClient() {

if (sStockService == null) {

RestAdapter restAdapter = new RestAdapter.Builder()

.setEndpoint("https://query.yahooapis.com/v1/public")

.build();

sStockService = restAdapter.create(StockApiInterface.class);

}

return sStockService;

}

public interface StockApiInterface {

@GET(***(/\*****STATIC* *REQUEST URL\*/)*

ResultWrapper listQuotes();

}

}

//MainActivity

ResultWrapper result = RetrofitInterface.getStockApiClient().listQuotes();

Quotes[] quotes = result.getQuery().getResults().getQuotes();

* *Environment Volatility*

I consider the Android software stack to be volatile when compared with developing for Java desktop applications. During the development process I had to restart the adb on several occasions from the command line using

**adb kill-server**

**adb start-server**

The adb “is a command line tool that allows a developer to communicate with an Android device or emulator in a client server architecture” **(Curtin 2012).** The Eclipse IDE (**Eclipse Foundation,** 2004) also seems to be relatively unstable when developing for Android. For example my application was eventually unable to run through the GUI ‘run’ button. Instead I had to run it from the programs menu. I have never before witnessed this problem when developing Java applications using Eclipse. This was one of the reasons I changed my development environment to Android Studio (**Google, 2013**).

* *The Observer Design Pattern*

My application currently has a RecyclerView that is populated by a list of stock items displaying some basic information (illustrated in fig 1.6) Clicking on a particular stock item will provide further, more in-depth analysis about the stock in question (Not yet implemented) When the user decides to buy a tranche of a particular stock the application must notify the portfolio that it’s state has changed. The portfolio must then update its internal state accordingly, taking into account the recent purchases made by the user. I use the Observer design pattern described in the ‘Gang of Four’. The Observer DP “defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically” **(Gang of Four, 1994).** My fragment containing the ISEQ RecyclerView acts as the ‘subject’ in this case. The subject is observed by the ‘observer’. In my application the fragment representing my portfolio plays this role. By using the Observer DP the internal state of my portfolio stays consistent by updating every time a new tranche of stocks are bought.

public interface Observer {

public void update(ISEQFragment subject);

}

public interface Subject {

public void attach(Observer observer);

public void detach(Observer observer);

public void notifyObservers();

}

* *Networking Issues*

As memory allocation on Android devices is limited in comparison to desktop, it is vital to keep the main thread free of any high intensity computations. Ignoring this rule can cause considerable amounts of lag in your application. One example of a high intensity computation is networking. Android will throw a *‘NetworkOnMainThreadException’* runtime error if your application tries to perform networking tasks on its main thread. I ran into this problem during the early stages of development. Retrofit **(Square Open Source, 2013)** for example, requires network access as well as the ‘android.permission.INTERNET’ permission. In order to be use Retrofit I had to introduce an asyntask and put all my networking functions in its ‘doInBackground(Void…params)’ method. The asyntask allows the code to run in parallel and ensures that networking is never done on the main UI thread.

**4.3 Application Functionality**

* *Loading screen fragment*

The loading or ‘splash’ screen is a fragment contained within an activity which can be defined in the Android manifest as my launcher activity using

**android.intent.category.LAUNCHER**

The splash-screen animation will be discussed under the ‘Third party libraries’ section of this report. Here the business logic underneath the UI will be discussed exclusively.

Splash-screens are worthless unless there is a specific purpose other than visual presentation behind them. In this application, the call to the Yahoo API had originally taken place in the ISEQ fragment. This caused two major problems

1. Repeated API interaction
2. UI instability

If the client code used to connect with the API was written in the ISEQ fragment then every time a new instance of this fragment was created (i.e. when the user selected ‘iseq’ in the navigation menu) the application would need to make a RESTfull call to the server. This would not serve as application loading times would increase significantly, ultimately leading to critical UX problems. This is an example of how fragments are created in the application.

//Container Activity

**Fragment fragment = null;**

**switch(position){**

**case 0:**

**fragment = new ISEQFragment();**

**break;**

**case 1:**

**fragment = new CalculatorFragment();**

**break;**

…

Another reason for moving the client code from the ISEQ fragment is that a lot of the UI elements that make up an ISEQFragment row do not rely on the result of the API call in order to be initialised and displayed. These UI elements include the company logos, the ‘favourites’ button as well as the ‘more info’ button illustrated in (FIG … HERE GIVE REFERECE TO SCREENSHOT. REPORT WILL HAVE POTENTIALLY THREE PAGES COVERING ALL SCREENSHOTS) These UI elements are displayed automatically while others take more time. One such UI element is the arrow drawable. This relies on the API call to decipher which drawable to load, whether it is positive, negative or neutral (Fig…). This is an example of poor UI design as the process should be atomic, either all or nothing. Moving the API call to the splash-screen fragment means that when the ISEQFragment is initialised, the result of the API call is already available. This means that the initialisation and display of the UI elements can take place simultaneously.

*How does the application perform the API call?*

As the splash-screen fragment encompasses a lot of animations and other visual effects, it was necessary to abstract the networking specifics of calling an API into its own class. An interface is defined in its own package called ‘RestCall’ which contains the abstract method ‘doTask()’

Public interface RestCall{

Public ResultWrapper doTask();

}

The reason for defining an interface is that although the chosen REST client is currently Retrofit, this may not be the case in the future. Volley, another RESTfull client, or perhaps a more optimized API may become available down the line. As long as these varying API’s return a ResultWrapper object (Yahoo API result at highest level of abstraction, contain query, quote, result etc.) then different implementations can be swapped in and out of the client code. This takes advantage of Java’s polymorphic capabilities.

RestCall request = new RetrofitRestCall();//OR

RestCall request = new VolleyRestCall();//

This is an example of ‘designing for change’. The implementation of the REST call is defined in the RetrofitRestCall class. This class simply creates a new asynchronous thread using androids ‘asyntask’. As android does not allow networking to be done on the main UI thread, creating a new thread using ‘asyntask’ allows a developer to integrate concurrency into his design. The call to the API is made using the Retrofit interface.

ResultWrapper result = RetrofitInterface.getStockAPIClient().listQuotes();

return result;

*//Clint code*

Private ResultWrapper performAPICall(){

RESTCall task = new RetrofitRESTCall();

ResultWrapper result = null;

result = task.doTask(); //Abstract method from RestCall interface, implemented in

RetrofitRestCall.

return result;

}

*How is this information passed to the MainActivity?*

At the moment the result from the API call is stored in the SplashScreen activity. However, this activity is destroyed once a new activity is launched, in this case the MainActivity. Therefore a solution must be found to pass the result (an instance of ResultWrapper class) to the new activity where it can be used. Passing data between activities is usually done using a bundle.

Intent mIntent = new Intent(this, Example.class);

Bundle bundle = new Bundle();

Bundle.putString(key,value);

mIntent.putExtras(bundle);

startActivity(mIntent);

A bundle is suitable for passing any type of raw data between activities (Ref to where this is said). This includes strings, doubles etc. However, it cannot pass an instance of a class between activities, for example. To overcome this problem the ResultWrapper instance must be serialized to make it compatible with the bundle. The solution was found using stackoverflow (Ref to solution). Many possible solutions were mentioned, however a Google Gson based solution was most suitable in this case due to the fact the library had already been imported in the application. Retrofit uses Gson to parse an API’s JSON response into instances of its corresponding POJOs.

Intent mIntent = new Intent(getActivity(), MainActivity.class);

mIntent.putExtra(“ResultWrapper”, new Gson().toJson(mResult);

In this case mResult is an instance of a ResultWrapper. The putExtra() function is a simple shortcut that converts it’s parameters into a bundle which can be passed between activities so that explicitly creating an instance of a bundle is not necessary.

In order to receive the result in the MainActivity the ResultWrapper instance passed in the bundle must be deserialized. This is also done using Gson.

String bundleResult = “”;

Bundle bundle = getIntent().getExtras();

If(bundle!=null){

bundleResult = bundle.getString(“ResultWrapper”);

}

result = new Gson.fromJson(bundleResult,ResultWrapper.class);

The MainActivity now has an instance of the ResultWrapper class which contains all the data relating to the ISEQ stock items. Although this is a relatively simple solution to implement, it seems like more of a hack than a proper implementation. Therefore, one way to optimize the application in the future would be to implement a standardized way of passing objects between activities.

*How does the MainActivity provide this data to its child fragments?*

The container activity (MainActivity) now has an instance of the ResultWrapper class which contains all relevant stock data. The MainActivity will be maintained throughout the applications runtime, as the fragments it contains will be created and recycled. Even if the activity is destroyed due to screen orientation changes, the application can simply save the ResultWrapper instance in the onDestroy() method and retrieve it when the MainActivity is recreated using the ‘savedInstanceState’ bundle in onCreate(). Therefore, having the result stored in the MainActivityand letting it’s child fragments query it seemed to be the best implementation. Providing a simple ‘getter’ method in the MainActivity would allow the fragments access to the ResultWrapper instance.

*//MainActivity*

public ResultWrapper getRestResult(){

return result;

}

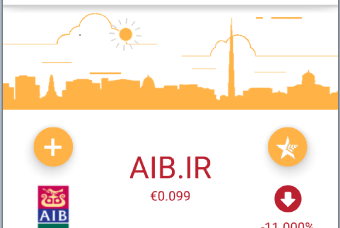
*//ISEQFragment*

this.result = ((MainActivity)this.getActivity()).getRestResult();

//result is the local ResultWrapper

Design (Will be in different section in finished report (ie Design section)

* *Refresh ISEQFragment design*



**Fig 0.0**

The stock data is downloaded on application start-up, as mentioned in section (…here ref section ‘loading screen fragment’). This means that the applications stock information may become outdated if the application remains open for a sustained period of time. Therefore it was necessary to implement some sort of ‘refresh’ functionality to update the data. There are numerous ways to implement this. One such way is to use the ‘SwipeRefreshLayout’ which was released by Google in their newest API called ‘Lollipop’ as part of their V4 support library. Developers are generally encouraged to use standard android libraries when available due to the rigorous testing they have been subjected to prior to release. Using standard android libraries also minimises APK size and supports consistency within the application. (Ref hsshshjdjdasjdhdadhajsdjsad) Despite these advantages this application uses a library developed by a third party vendor. His GitHub account can be found at ‘https://github.com/Yalantis/Phoenix’. This library was chosen simply for aesthetic purposes and runs the risk of introducing stability issues in the application, as is the case with most third party libraries. The library aims to ‘provide a simple and customizable pull to refresh implementation’ and is included in the applications as an external module (More in ‘Importing projects (Gradle, external module, code copying)’ section…).

*Usage frag\_iseq XML*

<com.yalantis.pulltorefresh.library.PullToRefreshView

android:id="@+id/pull\_to\_refresh"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

>

<android.support.v7.widget.RecyclerView

android:id="@+id/recyclerView"

android:layout\_height="match\_parent"

android:divider="@drawable/list\_selector"

android:dividerHeight="1dip"

android:layout\_width="match\_parent">

</android.support.v7.widget.RecyclerView>

</com.yalantis.pulltorefresh.library.PullToRefreshView>

The idea behind the library is to wrap the UI element that requires ‘refresh’ functionality in a custom ‘PullToRefresh’ view. That view can then be referenced in and a ‘OnRefreshListener’ can be attached.

mPullToRefreshView = (PullToRefreshView) android.findViewById(R.id.pull\_to\_refresh);

mPullToRefreshView.setOnRefreshListener(new PullToRefreshView.OnRefreshListener() {

@Override

public void onRefresh() {

mPullToRefreshView.postDelayed(new Runnable() {

@Override

public void run() {

mPullToRefreshView.setRefreshing(false);

…

}

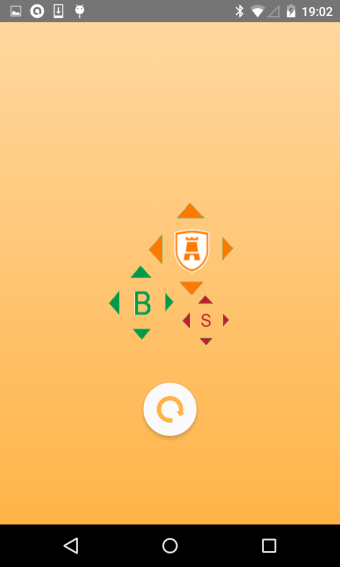
}, 2000);

}

});

The refresh image, as illustrated in Fig (…) is customizable by overwriting three drawables. These three drawables consist of a sun, sky and building png. If you do not include your own custom drawables the library will use default images for all three. The default refresh image is shown in Fig (…image of GitHub picture). The images used by the application remain largely the same except for a change of colour mirroring the look and feel of the whole application. The buildings png now shows a silhouette of Dublin rather than a generic city skyline.

* *Loading screen design*

**

**Fig 0.0**

In this section the design of the applications loading or ‘splash’ screen will be discussed. The business logic behind the loading screen is discussed in another section of the report (Ref)

The ‘splash’ screen introduces two new third party libraries. One of these libraries animates the ‘filling’ effect of the screens logo and background. The other library is used to compose a new material style circle progress bar. The first library can be found at ‘https://github.com/fedestyla/FillingEffect’. As this library does not support gradle integration (More section ‘importing libraries’) it must be included as a package in my application. This increases the size of the applications code base which is not desirable. Another problem with the filling effect library is that it is based on a ‘ScrollListener’. This is suitable when the user is expected to interact with the application i.e. to unlock a screen. However, a ‘splash’ screen is simply used to convey a sense of progress to the user and is not expected to be interactive.

The proposed solution was to replace the ScrollListener’ with a Timer.

There were two functions of major interest within the client code. These two functions were setLevelOfMask(int percentageOfMaskFill) and scrollTo(int x,int y). These functions were responsible for filling the application icon and background respectively. Whereas the unaltered GitHub library would call these functions whenever a ‘ScrollListener’ event was fired, this application needed to start a timer on initialisation and bind these function calls to the timer. In order to run timed events in android, an instance of the handler class must be created. We can set the length of each timer iteration using ‘timerHandler.postDelayed(context, timeInMilliseconds)’. However, updates to the UI cannot be done from outside the main UI thread and in this case the code is run from a worker thread. One solution is to implement the Runnable interface which is “Often used to run code in a different [Thread](http://developer.android.com/reference/java/lang/Thread.html).” (Google, 2008)

public void initialize() {

final Handler timerHandler = new Handler();

observableScrollView.post(new Runnable() {

int count = 0;

@Override

public void run() {

if(count <= 100) {

setLevelOfMask(count);

scrollTo(count);

count += 1;

timerHandler.postDelayed(this, 25);

}else{

timerHandler.removeCallbacks(this);

BusProvider.getInstance().post(new Event());// Animation finished event.

}

}

});

}

The level of progression of both the icon and background fill are synchronised with the count variable that increments on every iteration of the Timer.



**Fig 0.0**

The second library is used to compose a material style circular progress bar (Fig 0.0). It can be found at ‘https://github.com/lsjwzh/MaterialLoadingProgressBar’ and it supports gradle integration. The library includes many customizable attributes for the circular progress bar.

app:mlpb\_show\_arrow="true"

app:mlpb\_arrow\_height="5dp"

app:mlpb\_arrow\_width="10dp"

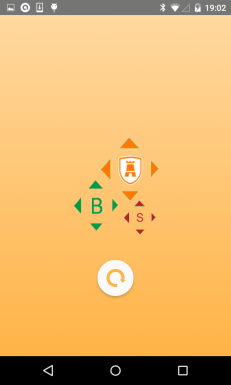
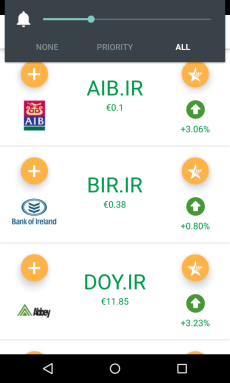
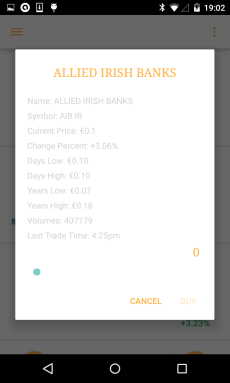
app:mlpb\_enable\_circle\_background="true"

app:mlpb\_progress\_stoke\_width="5dp"

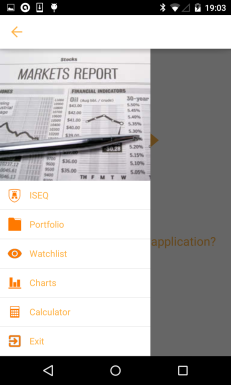
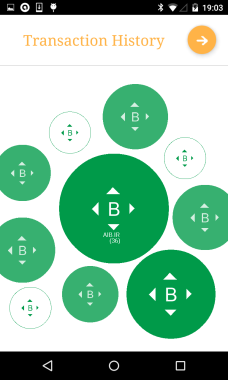
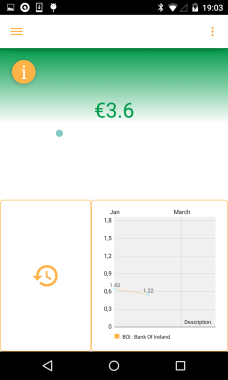
The colour scheme of the progress bar can also be set programmatically.

mCircularProgress.setColorSchemeResources(R.color.app\_orange);

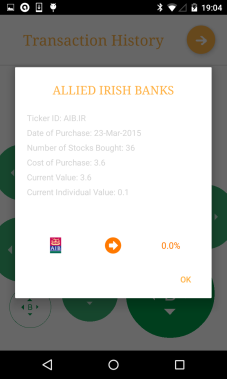
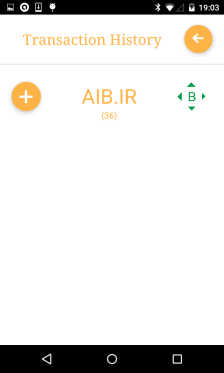
Mockups

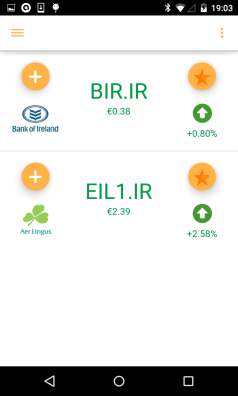
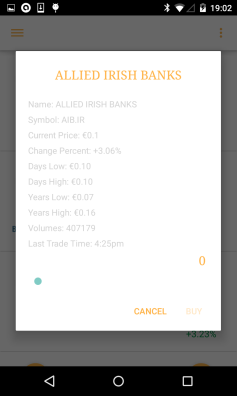
   

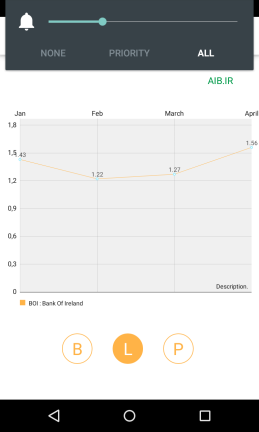


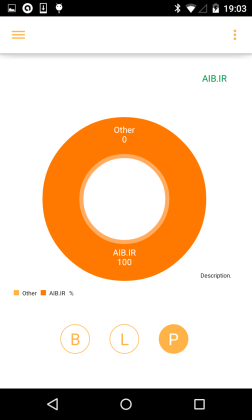
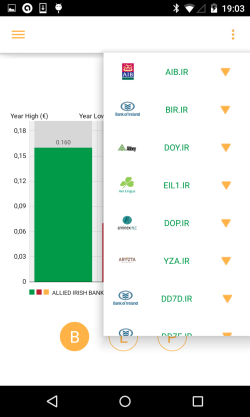
 

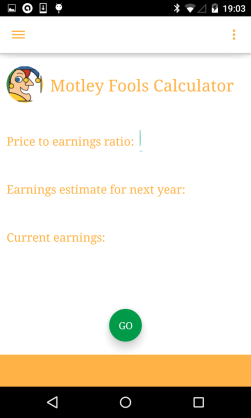


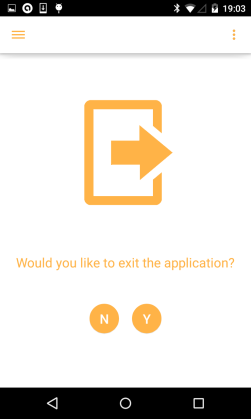
  

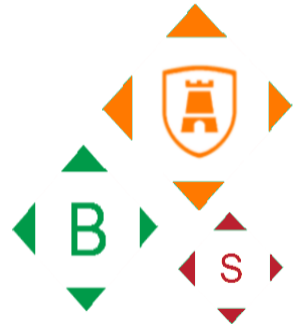
 

**** ****



 ****

**App Logo**

****

**Fig 0.0**

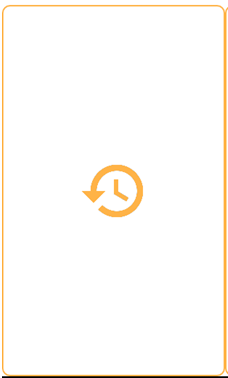
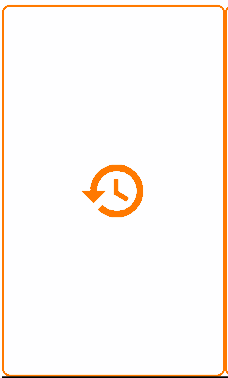
Fig 0.0 illustrates the menu icon used by the application. It is composed of three separate images. The orange image contains the ‘ISEQ’ badge and represents the dominant colour scheme of the application. The green and red images symbolise ‘buy’ and ‘sell’ within the application. Many examples of these ‘buy’ and ‘sell’ icons can be found during the applications runtime.

In order to change the menu icon of an application some changes must be made to the AndroidManifest.xml file. “The manifest file presents essential information about your app to the Android system” (Google 2008) The following application characteristics must be modified under the ‘application’ tag of the xml file.

**android:label="@string/app\_name"**

**android:icon="@drawable/icon"**

**Drawable selectors**

 **** 

**Fig 0.1 Fig 0.2 Fig 0.3**

Android allows you to change the background image of a UI element such as a ‘Button’ or ‘ImageView’ depending on its state. First the necessary resources must be added to the drawable folder. The example illustrated above consists of four different resources, two png files and two ‘border’ drawables. A selector drawable can then be written to bind these resources to the different states of the UI element. Examples of UI states include pressed, focused and non-focused. Here is an example of a selector drawable written for Fig 0.1’s border.

<?xml version="1.0" encoding="utf-8"?>

<selector xmlns:android="http://schemas.android.com/apk/res/android">

<item android:drawable="@drawable/button\_border\_portfolio\_pressed"

android:state\_pressed="true" />

<item android:drawable="@drawable/button\_border\_portfolio\_pressed"

android:state\_focused="true" />

<item android:drawable="@drawable/button\_border\_portfolio" />

</selector>

A selector is also written for Fig 0.1’s background image (Fig 0.3). These selectors can then be set to the corresponding UI element (ImageButton in this case) in the Layout xml file using the ‘background’ and ‘src’ attributes.

<ImageButton

android:id="@+id/ib\_trans\_history"

android:layout\_width="0dp"

android:layout\_height="match\_parent"

android:layout\_weight="0.4"

***android:background="@drawable/border\_selector"***

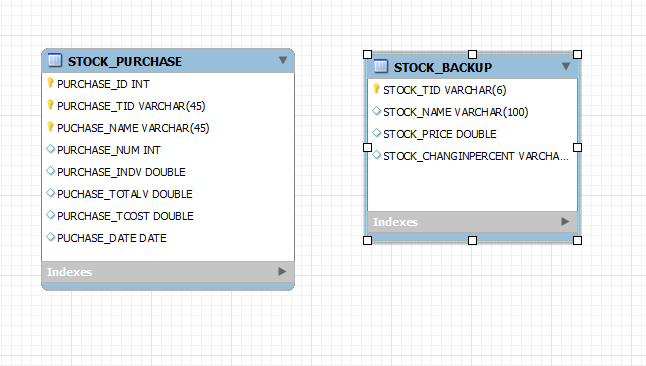
***android:src="@drawable/trans\_icon\_selector"***

/>

This is one example of where selectors are used in this application. Further examples can be found bellow.

**SQLite Database Design**



**Fig 0.0, Database schema for stock portfolio manager.**

The left table, STOCK\_PURCHASE represents each stock item purchase made by the user. This table is used as the data source for the applications portfolio. PURCHASE\_ID is an auto incrementing integer which keeps track of how many purchases have been made. PURCHASE\_TID represents ticker id (e.g. BIR.IR) while PURCHASE\_NAME defines the stock name (e.g. Bank of Ireland). PURCHASE\_NUM shows the number of stocks bought in a solitary purchase. Representing the current individual value of that particular stock is PURCHASE\_INDVAL. This value is taken from the YQL API **(Yahoo!, 2008)** on startup and so is constantly updated and ever changing. PUCHASE\_TVAL is used to show the total value of that particular stock purchase. This is calculated by multiplying the current individual value of the stock (PURCHASE\_INDVAL) by the number of stocks purchased (PURCHASE\_NUM). PURCHASE\_TCOST is the amount spent purchasing PURCHASE\_NUM amount of that particular stock item. To clarify things, let’s take Bank of Ireland stocks as an example. Let’s say the user has bought 100 Bank of Ireland stocks. PURCHASE\_TVAL represents how much these 100 Bank of Ireland stocks are now worth while PURCHASE\_TCOST represent how much they were purchased for. PURCHASE\_TCOST is calculated by multiplying PURCHASE\_INDVAL by PURCHASE\_NUM on purchase. This value is identical to PURCHASE\_TVAL when first added to STOCK\_PURCHASE. However, unlike PURCHASE\_TVAL which is updated constantly, this value will never change.

To calculate the total value of the user’s portfolio we could use a SQL query similar to the following

"SELECT SUM(" + PURCHASE\_TVAL + ") FROM " + “STOCK\_PURCHASE”, null);

The STOCK\_BACKUP table represents the last updated state of each stock item. Connection to a reliable internet source cannot always be relied upon during application start up. For this reason a stock backup SQLite table was composed to store the last updated value of each respective stock item. Data from this table can be utilized when connection to the API is not possible. This table is updated on every successful connection to the data source. STOCK\_TID represents a stocks ticker id while STOCK\_NAME represents its name. STOCK\_PRICE and STOCK\_CHANGEINPERCENT are used to show a stock’s current price and change respectively. This data is only sufficient to compose the ISEQFragment, as stock id, name, price and change is all that’s required. It is not extensive enough however, to perform an in depth analysis of each stock item without sufficient network access. This is a potential point of improvement for the future; enlarging the table to account for entries such as ‘LastTradeTime’, ‘YearHigh’ and other information required for an in depth analysis.

**Transitions/Animation**

**RotateAnimation Class**



**Fig 0.0 Fig 0.1**

The application contains two transaction history screens. One screen is used to illustrate recent purchase history while the other encompasses a complete log of past transactions in list form. Traversal between both screens is done using the ImageButton shown in Fig 0.0 and 0.1. As the number of screens is limited to two, navigation options are either ‘go forward’ or ‘go back’. Therefore a rotating button, illustrated by Fig 0.0 on the first screen and Fig 0.1 on the second was introduced.

It is currently not possible to introduce rotation in UI elements using XML and so rotation must instead be done programmatically. To do this you must create an instance of the ‘RotateAnimation’ class introduced in the new Lollipop API.

RotateAnimation rotateAnimation = new RotateAnimation(rotateFrom,rotateTo,mArrow.getWidth()/2,mArrow.getHeight()/2);

The parameters needed to create the instance are as follows.

* rotateFrom: “Rotation offset to apply at the start of the animation”(Google, 2013)
* rotateTo: “Rotation offset to apply at the end of the animation” (Google, 2013)
* pivotX: “The X coordinate of the point about which the object is being rotated” (Google, 2013)
* pivotY: “The Y coordinate of the point about which the object is being rotated” (Google, 2013)

In this case, the rotateFrom and rotateTo parameters are dependent on which of the two screens is currently active. The X and Y pivots are at the origin of the ImageView (mArrow).

The duration of the animation can also be set.

rotateAnimation.setDuration(500);// 500 milliseconds

The last requirement is to set the animation to the corresponding UI element, in this case the ImageButton ‘mArrow’. All the aforementioned tasks are done within the mArrow’s ‘OnTouchListener’

mArrow.startAnimation(rotateAnimation);

**ViewPager Class**

ViewPager: “Layout manager that allows the user to flip left and right through pages of data. You supply an implementation of a [PagerAdapter](http://developer.android.com/reference/android/support/v4/view/PagerAdapter.html) to generate the pages that the view shows.” (Google, 2008)

This application uses a ViewPager to implement a ‘thumb slide’ between different graph fragments (Bar, Line and Pie). To do this, a custom ‘PagerAdapter’ must be passed to the ViewPager. In this example, the ‘PagerAdapter’ is an instance of a ‘FragmentPagerAdapter’. The android SDK provides two types of ‘PagerAdapters’, one being the aforementioned ‘FragmentPagerAdapter’ while the other is a ‘FragmentStatePagerAdapter’

“Therefore FragmentStatePagerAdapter should be used when we have to use dynamic fragments, like fragments with widgets, as their data could be stored in the savedInstanceState. Also it wont affect the performance even if there are large number of fragments. In contrary its sibling FragmentPagerAdapter should be used when we need to store the whole fragment in memory. When I say the whole fragment is kept in memory it means, its instances wont be destroyed and would create a memory overhead. Therefore it is advised to use FragmentPagerAdapter only when there are low number of fragments for ViewPager.”( <http://www.truiton.com/2013/06/android-fragmentpageradapter-vs-fragmentstatepageradapter/>)

The memory overhead of maintaining three static fragments would not be sufficient enough to warrant the use of a ‘FragmentStatePagerAdapter’ and so a custom ‘FragmentPagerAdapter’ was the chosen implementation.

Custom FragmentPagerAdapter Class

private class ScreenSlidePagerAdapter extends FragmentPagerAdapter {

public ScreenSlidePagerAdapter(FragmentManager fm) {

super(fm);

chartFragments[0] = new BarChartFragment();

chartFragments[1] = new LineChartFragment();

chartFragments[2] = new PieChartFragment();

}

@Override

public Fragment getItem(int position) {

return chartFragments[position];

}

@Override

public int getCount() {

return NUM\_PAGES;

}

}

In order to get a working implementation, we must create an instance of this ‘FragmentPagerAdapter’ and set it as our ViewPager’s adapter.

mFragmentPagerAdapter = new ScreenSlidePagerAdapter(getChildFragmentManager());

mViewPager.setAdapter(mFragmentPagerAdapter)

**Custom Animations**

Slide in and slide out animations are integrated into the fragment transitions in the ‘transaction history’ section of the application. As mentioned in the ‘RotateAnimation Class’ section, the activity only contains two fragments which are swapped when the user hits the arrow button (mArrow, section 0.0(RotateAnimation class’)

The desired animation is identical to that found using the ViewPager class (section 0.0 (ViewPager). However, ViewPager animations cannot currently be used for transitions based on button clicks, they are only viable for transitions based on a ‘thumb slide’. One solution is to write custom animations to account for sliding one fragment out and sliding a new one in. Four animation files must be written, slide-in-left, slide-out-left, slide-in-right and slide-out-right. Here is an example based on the slide-in-left animation file.

<set xmlns:android="http://schemas.android.com/apk/res/android" >

<translate

android:duration="500" //Corresponds to ‘RotateAnimation’ duration seen in section 0

android:fromXDelta="100%"

android:toXDelta="0%" >

</translate>

</set>

These animations are then passed to a SupportFragmentManager which is responsible for committing a transaction. In this case, the animIn and animOut parameters of the ‘setCustomAnimations’ function are dependent on which of the two ‘transaction history’ screens is currently active as the animations for both will be opposites of each other.

getSupportFragmentManager().beginTransaction()

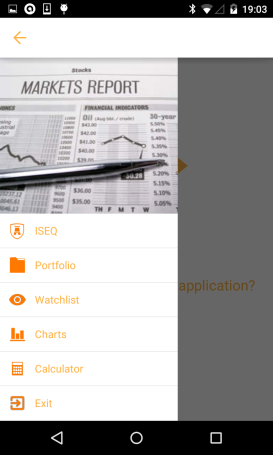
.***setCustomAnimations(animIn,***

***animOut)***

.replace(R.id.fl\_trans, nextFragment)

.commit();

**Navigation Menu ViewFlipper**

****

**Fig 0.0**

ViewFlipper: “Simple [ViewAnimator](http://developer.android.com/reference/android/widget/ViewAnimator.html) that will animate between two or more views that have been added to it. Only one child is shown at a time. If requested, can automatically flip between each child at a regular interval.” (Google, 2008)

The application uses a slideshow of images to give context to its navigation menu (Fig 0.0). The ViewFlipper API was chosen to implement these flipping images. To take advantage of the ViewFlipper API, each image must be enclosed within a ViewFlipper tag as an ImageView. This application contains three distinct ImageViews which are flipped after a certain interval. The interval is specified programmatically, rather than in the xml file and can be written as follows.

flipper = (ViewFlipper) findViewById(R.id.switcher);

flipper.setAutoStart(true);

flipper.setFlipInterval(10000);

**4.5 Open Source**

* *GitHub*

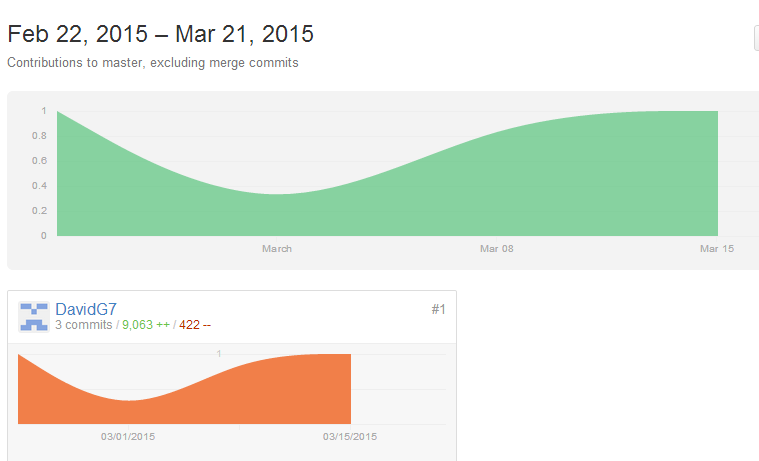
A GitHub repository was set up specifically for this project. Git allows you to make commits at any time, with or without internet connection. To do so simply issue the command.

**$ git commit –m ‘Title’**

To push this commit to the remote repository the user must also issue the following command

**$ git push –u <name of branch>**

Before this however a git repository must be initialized using **$ git init**. Then you must add your files to this local repository with **git add <filename>**. Git is not constrained to just code, as you can also push other files such as .txt and image files to the repository. As this project only involved one contributor, the repository was mainly used as a means of backing up previous work. Code would be pushed to the repository once every week after its creation on the 22nd of February. The program is no longer machine specific as GitHub repository code bases can be forked and replicated from anywhere.



**Fig 0.0**

* *Square libraries used by application* *(Square Open Source, 2013)*

ButterKnife (<http://jakewharton.github.io/butterknife/>)

ButterKnife is a view injection library for android. As android is a relatively new platform it requires writing some ‘boilerplate’ code that has not yet been abstracted into suitably simplified API’s. For example, most developers write numerous ‘findViewById’ calls in each application component. In order for a view to be used it first must be defined using ‘View v;’ and then initialised by explicitly binding it to a view found in the inflated XML layout file. That is done through the following.

**View v = (View) findViewById(R.id.view);**

This is suitable if the number of UI elements is limited. However, as the number of UI elements increase so too does the amount of ‘boilerplate’ or repetitive code. This can be reduced by using ButterKnife to inject views, listeners and other UI related objects. ButterKnife also allows us to bypass the need to inflate views just to set a listener on them as listeners can now be automatically configured onto methods.

//Standard Android

**View view = inflater.inflate(R.layout.fragment\_login, container, false);**

**mUserNameEditText = (EditText) view.findViewById(R.id.username);**

**mPasswordEditText = (EditText) view.findViewById(R.id.password);**

//ButterKnife Alternative

**@InjectView(R.id.username)**

**EditText username;**

**@InjectView(R.id.password)**

**EditText password;**

//ButterKnife Listener Injection

**@OnClick(R.id.username)**

**public void submit(View view) {**

**…**

**}**

The main advantage of using ButterKnife in your application is to increase the readability of your code base as well as minimise its size. ButterKnifes main competitor is RoboGuice (Google 2014). This application chose ButterKnife as its preferred view injection library due to its perceived performance gains. While ButterKnife runs at compile time, RoboGuice uses reflection at runtime to inject its views, causing a performance impact.

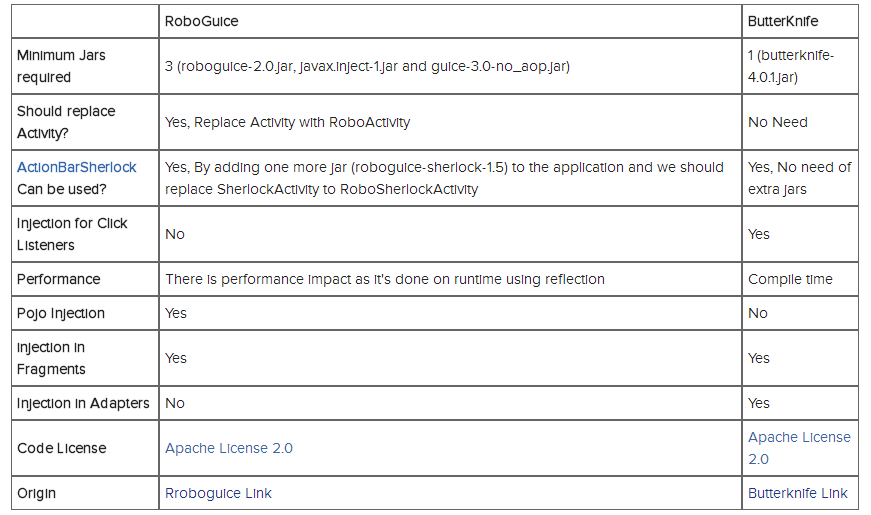


Fig 0.0 (http://java.dzone.com/articles/dependency-injection-roboguice)

Picassso

Picasso

**5.1 Project Plan**

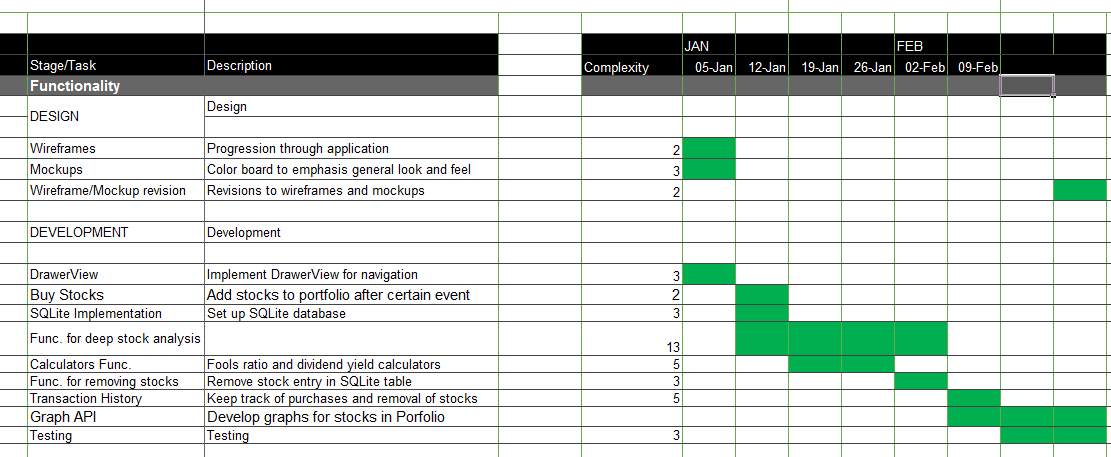


Fig 1.4

In order to ensure my progress goes as smoothly as possible I have laid out an eight week plan which I hope will leave me with a MVP or ‘minimal viable product’ by its conclusion. The green boxes represent what I will work on for the duration of the 8 weeks. For instance, the week starting on the 5th of January I will be working on finishing wireframes and mockups as well as implementing the DrawerView in my application. A DrawerView is a type of menu commonly found in Android applications. Testing will be done during the last two weeks of the plan. This plan only covers designing, developing and testing the application itself, it is not concerned with the FYP report.

The complexity rating uses Fibonacci numbers rather than going from one to ten. This was something I was thought during my internship. When using a 1-10 scale a five rating might be given to a medium difficulty task, while nine could be given to a very difficult task. This poorly represents the time requirements of one task in relation to another. In programming a difficult task may take three times longer to finish than a mundane one. Using Fibonacci numbers three may be given to the mundane task and thirteen to a very difficult one. This gives you a better idea of how long one task will take in relation to another which will in turn allow you to plan your time more efficiently.

Some of the tasks I anticipate taking a considerable amount of time are as follows

* Functionality for deep stock analysis.
* Graph API.

As I previously stated, it seems a lot of the Android Graph APIs are underdeveloped. From my research I have learned that working with dynamic graphs is no simple task when using the Android platform <http://www.stackoverflow.com/questions/6806537/graphs-api-for-android> **(Stack Exchange Network, 2008)**. Therefore I have given myself a three week period where I can work on implementing the graphs in my application. I will also be testing the application and implementing the transaction history during these final three weeks.

* 1. **References**

1. Curtin, Eric (2012) “Fall-down monitor for accelerometer-based mobile phones”
2. Stack Overflow <http://stackoverflow.com/> (accessed 3 January 2015)
3. Google (2008) ‘Android SDK’, [online] available:

<http://developer.android.com> [accessed 5 October 2014]

1. Google (2014) ‘Android Lollipop’, [online] available:

<http://www.android.com/versions/lollipop-5-0> [accessed 5 October 2014]

1. Genymotion (2013) ‘Genymotion emulator’, [online] available:

<http://www.genymotion.com> [accessed 10 November 2014]

1. Google (2013) ‘Android Studio’, [online] available:

<http://developer.android.com> [accessed 10 November 2014]

1. Eclipse Foundation (2004) ‘Eclipse IDE’ [online] available:

<http://www.eclipse.org> [accessed 25 October 2014]

1. Github (2008) ‘Github git repository’, [online] available:

http.//www.github.com

1. “Graphs API for Android” - Stack Overflow
2. Gradle (2009) ‘Gradle Build automation tool’, [online] available:

<http://www.gradle.org> [accessed10 November 2014]

1. Yahoo! (2008) ‘Yahoo Finance API’, [online] available:

<http://www.developer.yahoo.com> [accessed 12 November 2014]

1. General Dynamics (2010) ‘SQLite database engine’, [online] available:

<http://www.sqlite.org> [accessed 5 December 2014]

1. Oracle (1995) ‘MySQL’, [online] available:

<http://mysql.com>

1. Gang of Four (1994) ‘Design Patterns: Elements of Reusable Object-Oriented Software’
2. Mobisys (2012) ‘Introduction to Android SQLite Database’, [online] available:

<http://www.mobisys.in> [accessed 5 December 2014]

1. Square Open Source (2013) ‘Retrofit, a type-safe REST client for Android and Java’, [online] available: <http://www.square.github.io> [accessed 15 November 2014]
2. Oracle (1995) ‘Essentials of the Java programming language’, [online] available

<http://www.oracle.com>

1. Google (2008) ‘Gson, A Java library to convert JSON to Java objects’, [online] available: <http://www.code.google.com/p/google-gson> [accessed15 November 2014]