**Contents**

**1. Design**

1.1 Mock-ups………………………………………………………………………………3

1.2 Design Specific Implementation……………………………………………....5

1.2.1 Loading Screen Design…………………………………..5

1.2.2 Refresh Screen Design………………………………………………….8

1.3 App Logo………………………………………………………………….10

1.4 Drawable Selectors…………………………………………………..11

1.5 Transitions/Animations……………………………………………………….12

1.5.1 RotateAnimation Class……………………………………………………12

1.5.2 ViewPager class……………………………………………………………….14

1.5.3 Custom Animations………………………………………………………….15

1.5.4 Navigation Menu ViewFlipper…………………………………………..17

1.6 SQLite Table Design………………………………………………………..18

**2. Development**

2.1 Open Source…………………………………………………………… 20

2.1.1 Square…………………………………………………………………20

2.1.1.1 ButterKnife………………………………………………………………20

2.1.1.2 Picasso………………………………………………………………….21

2.1.1.3 Retrofit………………………………………………………………22

2.1.1.4 Otto……………………………………………………………………25

2.1.2 GitHub Personal Repository………………………………………………..26

2.2 Library Importation………………………………………………………28

2.2.1 Gradle………………………………………………………………….28

2.2.2 JAR…………………………………………………………………28

2.2.3 Module Integration……………………………………………………..29

2.3 Design Patterns…………………………………………………………30

2.3.1 Singleton………………………………………………30

2.3.2 Observer…………………………………………………..31

2.4 Application Functionality………………………………………………………32

2.4.1 Loading Screen…………………………………………………..32

2.4.2 Navigation Menu…………………………………………………..34

2.4.3 ISEQ Screen……………………………………………………………..35

2.4.4 Watchlist…………………………………………………………….37

2.4.5 Portfolio………………………………………………………..38

2.4.6 Charts…………………………………………………41

2.4.7 Calculator…………………………………………43

**3. Testing**

3.1 Android Debugger/Logging……………………………………………44

3.2 Proguard………………………………………………………….44

3.3 Usability Testing…………………………………………………..45

3.4 Java Yourkit Profiler………………………………………………..47

**4. Further Work**

4.1 Screen Optimization…………………………………………………….48

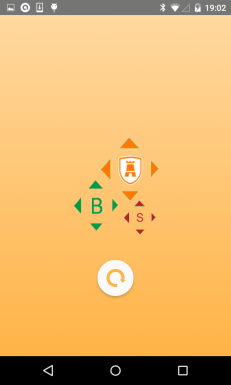
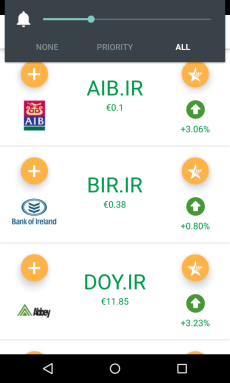
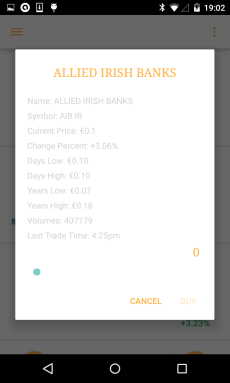
4.2 Further Testing…………………………………………………………..49

4.3 App Store Deployment………………………………………………….49

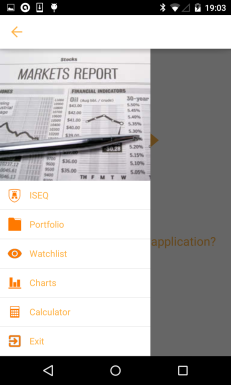
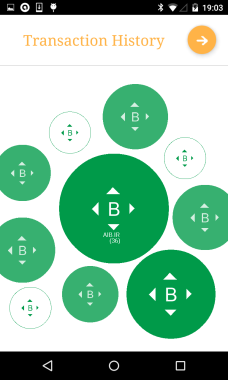
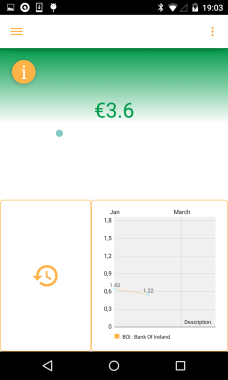
4.4 Google Analytics………………………………………………………..50

**1 Design**

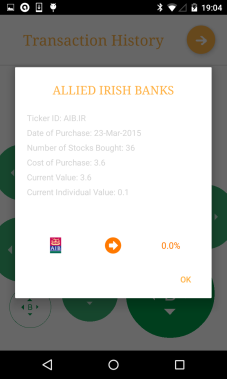
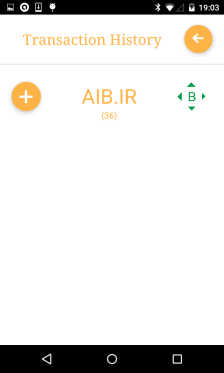
**1.1 Mock-ups**

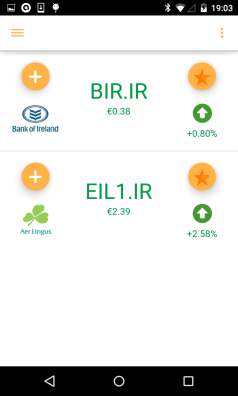
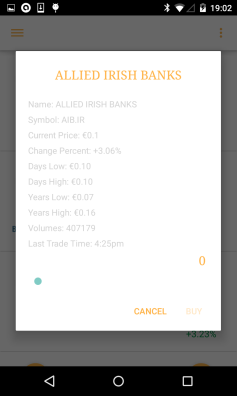
   

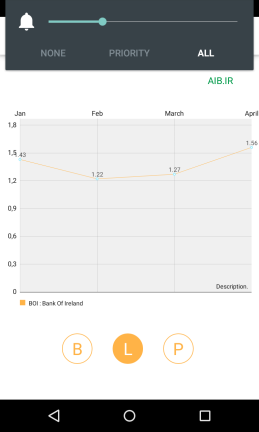


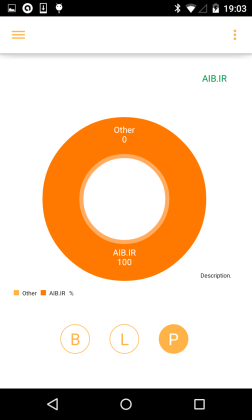
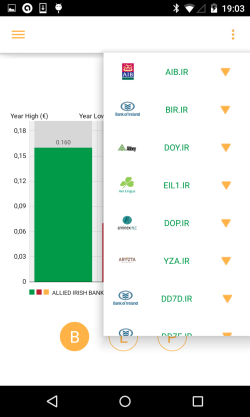
 

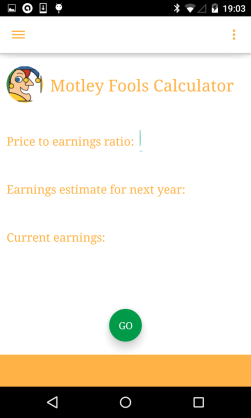


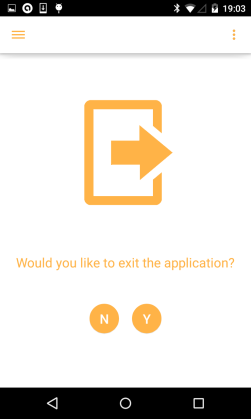
  

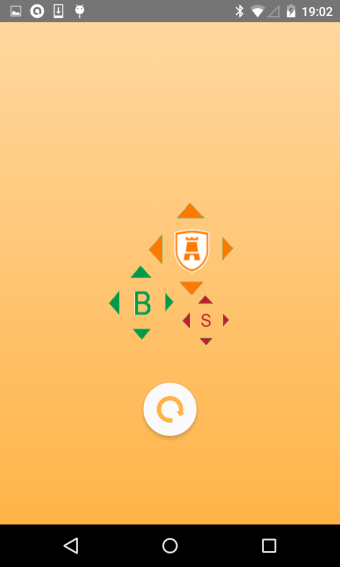
**** ****



 ****

**1.2 Design Specific Implementation**

**1.2.1 Loading Screen Design**

**

**Fig 1.1**

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 (Section 2.4.1)

The ‘splash’ screen introduces two new third party libraries. One of these libraries animates the filling effect of the screen’s 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’.(**GitHub, 2008**) As this library does not support Gradle integration (Section 2.2.1) it must be included as a package in the application. This increases the size of the application’s code base which is not desirable. Another problem with the filling effect library is that it is based on a ‘ScrollListener’. This is suitable only 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 therefore not expected to be interactive. The proposed solution was to replace the ‘ScrollListener’ with a ‘Timer’.

There are two functions of interest within the client code. These two functions are *setLevelOfMask(int percentageOfMaskFill)* and *scrollTo(int x,int y).* These functions are responsible for filling the application icon and background respectively. Whereas the original GitHub library calls these functions whenever a ‘ScrollListener’ event is fired, this application needs to start a timer on initialisation and bind these function calls to that 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 running in parallel from a worker thread. One solution is to implement the Runnable interface which is “Often used to run code in a different Thread.” (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 is synchronised with the count variable that increments on every iteration of the timer.



**Fig 1.2**

The second library is used to compose a material style circular progress bar (Fig 1.2). It can be found at ‘https://github.com/lsjwzh/MaterialLoadingProgressBar’(**GitHub, 2008**) and it supports Gradle integration. The library offers many customizable attributes for the circular progress bar.

*app:mlpb\_show\_arrow="true"*

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

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

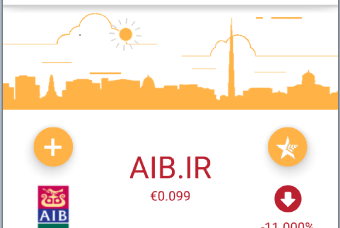
*app:mlpb\_enable\_circle\_background="true"*

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

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

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

**1.2.2 Refresh Screen Design**



**Fig 1.3 Fig 1.4**

The stock data is downloaded on application start-up, as mentioned in section 2.4.1. This means that the applications stock information may become outdated if the application remains open for a sustained period of time. Therefore it is 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 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”. (**Google, 2008**) 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’(**GitHub, 2008**). This library was chosen 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 application as an external module (More in section 2.2).

*//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 elements that require refresh functionality in a custom ‘PullToRefresh’ view. That view can then be referenced in the code and an ‘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);*

*…*

*}*

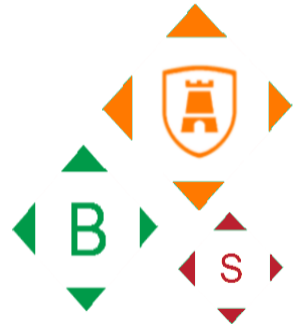
*}, 2000);*

*}*

*});*

The refresh image, as illustrated in fig 1.3 is customizable by overwriting three drawables. These three drawables consist of a sun, sky and building png image. 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 1.4. 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.

**1.3 App Logo**

****

**Fig 1.5**

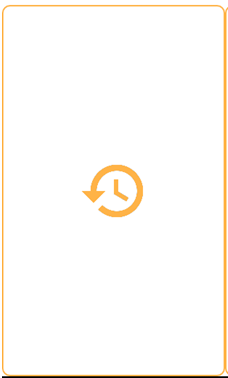
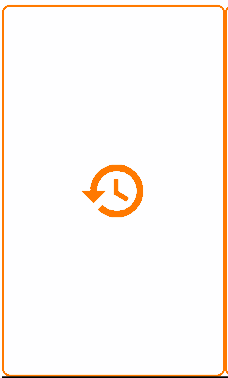
Fig 1.5 illustrates the menu icon used by the application. It is composed of three separate images. The orange image contains the ISEQ badge and represents the dominant colour scheme of the application. The green and red images symbolise ‘buy’ and ‘sell’ within the application. Many examples of these ‘buy’ and ‘sell’ icons can be found throughout the application.

In order to change the menu icon of an application some modifications 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 AndroidManifest.xml file.

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

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

**1.4 Drawable Selectors**

 **** 

**Fig 1.6 Fig 1.7 Fig 1.8**

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 1.6’s border.

*//border\_selector.xml*

*<?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 1.6’s background image (fig 1.8). 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 (fig 1.9).



**Fig 1.9**

**1.5 Transitions/Animation**

**1.5.1 RotateAnimation Class**



**Fig 1.10 Fig 1.11**

The application contains two transaction history pages. One page is used to illustrate recent transaction history while the other contains a complete log of past transactions in list form. Traversal between both screens is done using the ImageButton shown in Fig 1.10 and 1.11. 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 1.10 on the first screen and Fig 1.11 on the second was introduced.

It is currently not possible to introduce rotation in UI elements using their corresponding xml attributes 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. **(Google, 2013)**

*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 (fig 1.10/1.11). All the aforementioned tasks are done within the ImageButton’s ‘OnTouchListener’

*mArrow.startAnimation(rotateAnimation);*

**1.5.2 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 ‘FragmentStatePagerAdapter’. 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/>)

A custom ‘FragmentStatePagerAdapter’ was the chosen implementation as the application deals with dynamic fragments that change depending on what stock item is currently selected.

*//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 ‘FragmentStatePagerAdapter’ and set it as our ViewPager’s adapter.

*mFragmentPagerAdapter = new ScreenSlidePagerAdapter(getChildFragmentManager());*

*mViewPager.setAdapter(mFragmentPagerAdapter)*

**1.5.3 Custom Animations**

Slide in and slide out animations are integrated with the fragment transitions found in the ‘transaction history’ section of the application. As mentioned in the ‘RotateAnimation Class’ section (section 1.51), the activity only contains two fragments which are swapped when the user hits the arrow button (fig 1.10)

The desired animation is identical to that found using the ‘ViewPager’ class (section 1.5.2). However, ‘ViewPager’ animations cannot currently be used for transitions based on button clicks, they are only viable for transitions based on a ‘thumb slide’, “Layout manager that allows the user to flip left and right through pages of data.” (**Google, 2008**). 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*

*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 pages is currently active as the animations for both will be opposites of each other.

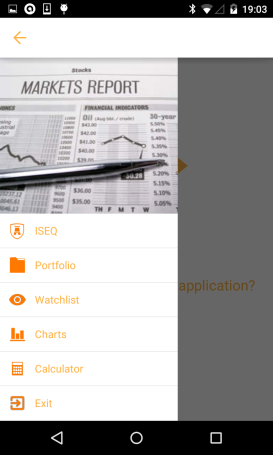
*getSupportFragmentManager().beginTransaction()*

*.****setCustomAnimations(animIn, animOut)***

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

*.commit();*

**1.5.4. Navigation Menu ViewFlipper**

****

**Fig 1.12**

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**)

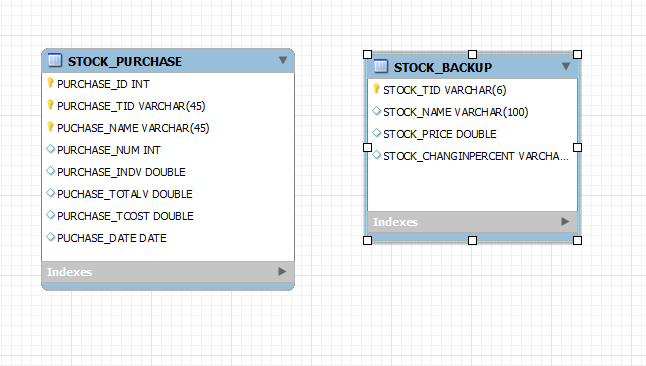
The application uses a slideshow of images to give context to its navigation menu (fig 1.12). The ‘ViewFlipper’ API was chosen to implement these flipping images. To take advantage of the ‘ViewFlipper’ API, each image must be enclosed within a ‘ViewFlipper’ tag as an ‘ImageView’. This application contains 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);// 10 seconds*

**1.6 SQLite Database Design**



**Fig 1.13, 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, PURCHASE\_TCOST 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 an internet source cannot always be relied upon during application start up. For this reason a stock backup SQLite table is 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 is 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 last trade time, year high and other information required for an in depth analysis.

**2 Development**

**2.1 Open Source**

**2.1.1 Square Open Source Libraries**

**2.1.1.1 ButterKnife**

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. **(Harsha Vardhan. (2014). *Dependency Injection: RoboGuice vs. ButterKnife.* Available: http://java.dzone.com/articles/dependency-injection-roboguice. Last accessed 30th March 2015.)**

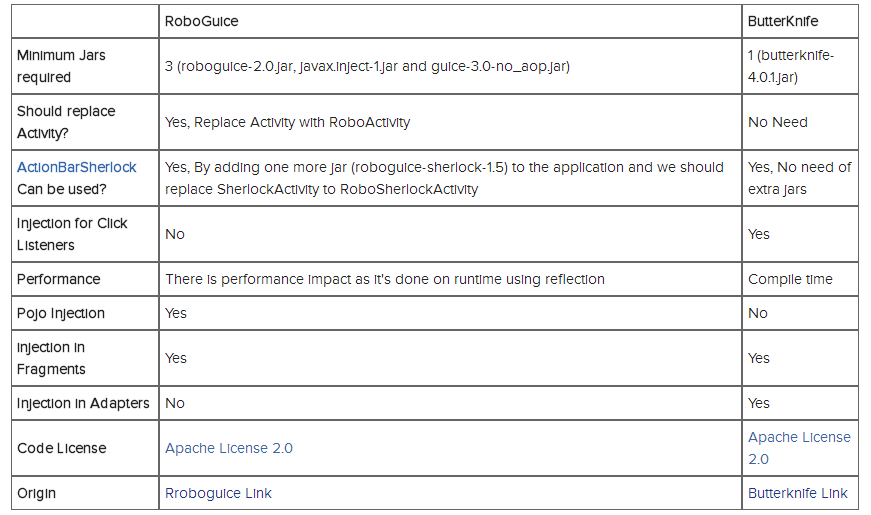


Fig 2.1 (http://java.dzone.com/articles/dependency-injection-roboguice)

**2.1.1.2 Picasso**

Picasso (http://square.github.io/picasso/)

“Picasso allows for hassle-free image loading in your application—often in one line of code!” **(Square Open Source, 2013).** Picasso simplifies the process behind displaying images in an application. Its three main functions are as follows.

1. “Handling ImageView recycling and download cancelation in an adapter.”
2. “Complex image transformations with minimal memory use.”
3. “Automatic memory and disk caching.”

Picasso is used for all image loading within the application. One particular example of its use is within the ISEQ fragment screen. This screen contains fifty seven different stock items divided into distinct rows. Each row also has a company logo associated with it (Fig 2.3)



Fig 2.2

As this application does not load its images ‘lazily’ (Only when they become visible), a lot of processing is required to instantaneously load fifty seven images into their respective ‘ImageViews’ when the ISEQ fragment is first instantiated. Prior to the use of Picasso, there was a lot of ‘lagging’ when opening and closing that particular fragment (ISEQFragment). The use of Squares Picasso library minimised this lag substantially within the application. This was all done using just one line of code

*Picasso.with(mContext).load(ISEQIcons.getResourceId(position, -1)).into(viewHolder.icon);*

‘ISEQIcons’ is a typed array containing references to all of my company logo drawables while ‘viewHolder.icon’ is my ‘ImageView’ that these drawables will be set to.

**2.1.1.3 Retrofit**

Retrofit (<http://square.github.io/retrofit/>)

“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 was set to <https://query.yahooapis.com/v1/public> in this application. Then the interface itself had to be designed with a relevant Http annotation (Get, Post etc.) As the application intends to read information from a server rather than write to a server, @Get was used. Then the request URL is specified. It was initially believed queries would be needed for this, however in hindsight 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 this application is receiving from the Yahoo API is very verbose; needing over ten different Java classes to encapsulate the response. The JSON has five levels of nesting, making it hard to visualise the data as JSONArrays and JSONObjects. To overcome this, a change to the diagnostics in the static request URL from true to false was required. 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 must specify the Java classes that the JSON will be converted to. For this an online JSON to POJO converter was used. ‘jsonschematopojo.org’ **(Github, 2011)** was the 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 there was still some considerable problems converting the data into the required Java objects. For example, when trying to access the quotes data members such as symbol a ‘*NullPointerException’* would be given. This was due to the fact that Retrofits built in Gson converter was not correctly converting the JSON data to objects. In the end, a wrapper class that would encapsulate all JSON data was needed. Then by using:

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

The quote could be accessed. Then using:

*quotes[i].getSymbol();*

The specific Quotes symbol could also be accessed.

*//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();*

**2.1.1.3 Otto Event Bus**

Otto(<http://square.github.io/otto/>)

“Otto is an event bus designed to decouple different parts of your application while still allowing them to communicate efficiently”. **(Square Open Source, 2013).** The Otto event bus is an example of the publish-subscribe paradigm commonly used as a means of communication. It is in many ways similar to the Observer DP discussed in section 2.3.2.

The event bus is also an example of the Singleton DP discussed in section 2.3.1.

The main use of an event bus in android development is for IPC, or inter process communication between system components. Android components include activities, services, content providers and broadcast receivers. As mentioned in section 2.4.1, the ‘MainActivity’ is maintained for the duration of the applications run time. For this reason the Otto event bus was chosen to handle communication between the activity and its child fragments. The process involves one component subscribing to a certain type of event while another component publishes an event of this type. In order to subscribe to an event, a component must first register itself with the event bus.

*BusProvider.getInstance().register(****this****); // ‘this’ refers to component (activity)*

To subscribe to an event you must also annotate a function with ‘@Subscribe’ The function name is not important as it will never be explicitly called anywhere in the applications code base. Its only purpose is as a callback method to a published event. However, the method must take only a single parameter corresponding to the event type the component wishes to subscribe to.

*@Subscribe*

*public void adapterCallback(ExitEvent event) {*

*…. do stuff}*

The following is used to publish an event in a separate system component.

*BusProvider.getInstance().post(new ExitEvent());*



Fig 2.2

An example of the event bus’ use in the application is to communicate changes made to the user’s stock item favourites. Favourites are added and removed by clicking on the star icon (Fig 2.2) in the ‘ISEQFragment’. The ‘MainActivity’ requires to be informed of any changes made to the favourites list as it needs to communicate these changes to other fragments as and when they are created. An example of this is the watchlist, which needs to know which stock items are in the favourites list in order to display them. In order to communicate these changes, the Otto event bus is utilised.

The ‘ISEQFragment’ publishes an event of type ‘FavouritesEvent’. This is done inside the star icon’s (Fig 2.2) OnClickListener and the updated list of favourites is passed to the ‘FavouritesEvent’ which is then published .The ‘MainActivity’ must subscribe to this event in order to synchronise its own class level instance of the favourites list with the new list published each time the star icon is interacted with.

*// ISEQFragment*

*BusProvider.getInstance().post(new FavouritesEvent(favourites));*

*// MainActivity*

*@Subscribe*

*public void favouritesChangedCallback(FavouritesEvent event) {*

*this.favourites = event.getFavourites(); } // updates class level favourites list*

*// Favourites Event*

*public class FavouritesEvent {*

*ArrayList<Integer> favourites = new ArrayList<Integer>();*

*public FavouritesEvent(ArrayList<Integer> favourites){*

*this.favourites = favourites; }*

*public ArrayList<Integer> getFavourites(){ return favourites; }*

**2.1.2 Github Personal Repository**

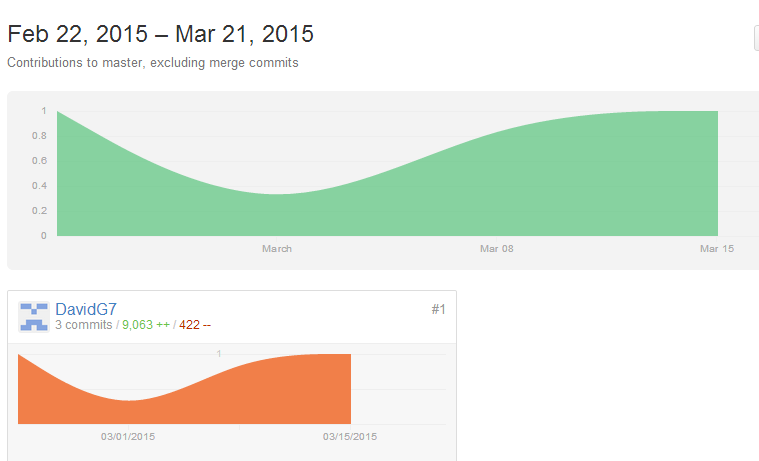
A GitHub repository was set up specifically for this project. Git allows you to make commits at any time, with or without internet connection. To do so simply issue the command.

**$ 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 2.3**

**2.1.2 Singleton Design Pattern**

The Singleton design pattern is used to implement the Otto event bus from Square **(Square Open Source, 2013).** The intent of the Singleton design pattern is to “Ensure a class only has one instance, and provide a global point of access to it”, **(Gang of Four, 1994).**

According to Joshua Bloch, in ‘Effective Java’, there are two ways to implement the Singleton design pattern in an implementation. One approach is to make a public static final field and initialise it inside a private constructer.

*public static final Bus bus = new Bus();*

The other approach, the one taken by the application, provides a public static factory method which allows access to the private static final field.

public final class BusProvider {

**private static final Bus BUS = new Bus();**

**public static Bus getInstance()** {

return BUS;

}

private BusProvider() {

}

}

The second approach has the advantage where it gives you flexibility to change your mind without changing the API. As one of the desired quality attributes of the application is flexibility and extensibility, the second approach was chosen for the implementation.

“It makes sense to use the first approach if you’re absolutely sure that the class will forever remain a singleton. Use the second approach if you want to reserve judgement on the manner” **(Effective Java First Edition, 2001**)

**2.1.2 Observer Design Pattern**

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).**

The application currently has a RecyclerView that is populated by a list of stock items displaying some basic stock information (ISEQFragment). Clicking on a particular stock item will provide further, more in-depth analysis about the stock in question. When the user decides to buy a tranche of a particular stock the application must notify the portfolio that its state has changed. The portfolio must then update its internal state accordingly, taking into account the recent purchases made by the user. The Observer design pattern described in the ‘Gang of Four’ is used. The fragment which contains the ISEQ RecyclerView acts as the ‘Subject’ in this case. The subject is observed by the ‘Observer’. In this application, the MainActivity class plays the role of Observer as it updates the portfolio values in the database when notified of stock purchases. By using the Observer DP the internal state of the portfolio stays consistent by updating every time a new tranche of stocks are bought.

*public interface Observer {*

***public void update(ISEQFragment subject); // Updates Portfolio’s internal state***

*}*

*public interface Subject {*

*public void attach(Observer observer);*

*public void detach(Observer observer);*

***public void notifyObservers(); // Called when stocks are purchased***

*}*

In order to register an ‘Observer’ with a ‘Subject’, the observer instance must be passed to the subject’s constructer so that it can be attached on subject initialization.

*public ISEQFragment(****Observer observer****) { // Subject Constructer*

***this.attach(observer);*** *// Attaching Observer}*

**2.1.2 Gradle**

Gradle is an automation tool that allows you to define and manage dependencies in one line of code. As stated in the ‘Research’ (Link) section of this paper, one of the reasons Android Studio (**Google, 2013**) was chosen over Eclipse as the IDE of choice was due to the simplicity of integrating libraries using the Gradle build system, something that Eclipse (**Eclipse Foundation,** **2004**) does not offer.

*// ’ISEQ Stock Exchange’ Gradle dependencies.*

*dependencies {*

*compile fileTree(dir: 'libs', include: ['\*.jar'])*

*compile 'com.squareup.picasso:picasso:2.5.0'*

*……..*

*compile project(':library')*

*compile files('libs/mpandroidchartlibrary-1-7-4.jar')*

*}*

**2.1.2 JAR**

Although most large scale third party libraries are available through the Gradle build system, some must still be downloaded as JAR files and integrated into the application by the developer. ‘MPAndroidChart’ is an example of such library used within the application.

In order to integrate a third party library without Gradle support you must

1. Download library JAR from website (<https://github.com/PhilJay/MPAndroidChart>)
2. Add to your projects ‘lib’ folder
3. Go to ‘File’ and then ‘Project Structure’
4. Under the desired module, add a ‘File dependency’ pointing to the JAR file added to the ‘lib’ folder in part 2.
5. Sync your Gradle files by building project.

The library is now integrated with the project and the following line of code is added to the corresponding ‘gradle.build’ file.

*compile files('libs/mpandroidchartlibrary-1-7-4.jar')*

**4.3 Loading Screen**

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 ‘Loading Screen Design’ 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 0.0. 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 (**Square Open Source, 2013**), 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’ within the application. 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. 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 (http://stackoverflow.com/questions/4249897/how-to-send-objects-through-bundle). 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 MainActivity and letting its 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

**4.3 Navigation Drawer**

The navigation drawer is used as a means of traversing the application and was chosen over other common navigation tools such as tabs etc. It can be described as “a panel that displays the app’s main navigation options on the left edge of the screen.” (**Google, 2013**). In order to implement a navigation drawer in an activity a ‘DrawerLayout’ must be placed as the root view of the corresponding xml layout file (activity\_main.xml). The application’s ‘DrawerLayout’ contains the container for the main activity’s fragments as well as a ViewFlipper and ListView to represent the navigation menu itself.

The ‘ViewFlipper’ and activity fragments are discussed in separate sections of the report (Sections 1.5.4 and 2.4 respectively). This section will primarily focus on the implementation of the ‘ListView’ as well as other appropriate navigation drawer implementation issues.

In order to implement a ‘ListView’ in android, a ‘ListAdapter’ must be passed to the corresponding ListView instance. A ListView can only display data that is suitably wrapped in an adapter. Android provides a standard adapter that can be initialised as follows.

*ArrayAdapter adapter = new ArrayAdapter(this,* ***android.R.layout.simple\_list\_item\_1****, list);*

An adapter such as this can be used to display a basic list of text. However, it cannot be used for more complex ‘ListView’ items, ones that contain an ‘ImageView’ or different fonts for example. A custom adapter must be written in order to convey such detail.

The custom adapter used to implement the navigation drawer’s ‘ListView’ is derived from the ‘BaseAdaper’ class. The ‘BaseAdapter’ class is one which can be extended in order to override the methods which tell the ‘ListView’ what to display. The most important method to override is the getView() method, in which you return the ‘View’ to display at a particular position of the ‘ListView’.

*public class NavDrawerListAdapter extends BaseAdapter {*

*….*

*public NavDrawerListAdapter(Context context, ArrayList<NavDrawerItem> navDrawerItems){*

*this.context = context;*

*this.navDrawerItems = navDrawerItems;}*

*@Override*

*public View getView(int position, View convertView, ViewGroup parent) {.*

*convertView = LayoutInflater.inflate(R.layout.drawer\_list\_item, null); // Custom row layout file*

*…*

*return convertView; } }*

As can be seen from the code snippet above, the custom ‘ListAdapter’ requires some data to be passed to its constructer. In this case, the data required is an ‘ArrayList’ of type ‘NavDrawerItem’, a custom class which individually represents one row of the navigation drawer’s ‘ListView’. A ‘NavDrawerItem’ comprises of a title, icon and a ‘selectedVisible’ boolean. The title and icons are statically stored in the strings.xml file of the values folder and can be retrieved programmatically as follows.

*private boolean[] selectedVisible = new boolean[]{true,false,false,false,false,false};*

*navMenuTitles = getResources().getStringArray(****R.array.nav\_drawer\_items****);*

*navMenuIcons = getResources() .obtainTypedArray(****R.array.nav\_drawer\_icons****);*

*navDrawerItems = new ArrayList<NavDrawerItem>();*

*navDrawerItems.add(new NavDrawerItem(****navMenuTitles[0], navMenuIcons.getResourceId(0, -1),selectedVisible[0]****));*

This data is then passed to the custom ‘ListAdapter’ on initialisation which uses it to build each row of the ‘ListView’ in its overridden getView() method.

*adapter = new NavDrawerListAdapter(getApplicationContext(),* ***navDrawerItems****);*

*mDrawerList.setAdapter(****adapter****);*

One major implementation issue that occurred while designing the navigation drawer was that the drawer itself would stutter when switching between fragments. This was due to the fact the application was trying to both close the navigation drawer and switch fragments concurrently after firing the ‘ListView’s’ OnitemClickListener ().

*mDrawerList.setOnItemClickListener(new AdapterView.OnItemClickListener() {*

*@Override*

*public void onItemClick(AdapterView<?> parent, View view, int position, long id) {*

*mDrawerLayout.closeDrawer(mDrawerLinLay); // Closing Drawer*

*// Switch Fragment Implementation } });*

According to official documentation, applications should “Avoid performing expensive operations such as layout during animation as it can cause stuttering; try to perform expensive operations during the STATE\_IDLE state.” (**Google, 2013**).In order to abide by the documentation, the code to switch fragments was moved to the onDrawerClosed() callback method which is fired after setting the current fragment in the ‘ListView’s’ OnItemClickListener().

mDrawerList.setOnItemClickListener(new AdapterView.OnItemClickListener() {

@Override

public void onItemClick(AdapterView<?> parent, View view, int position, long id) {

setCurrentFragment(mFragments[position];

mDrawerLayout.closeDrawer(mDrawerLinLay); }

});

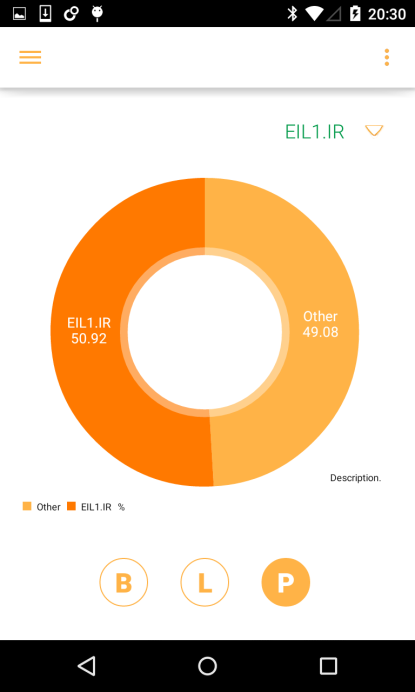
*public void onDrawerClosed(View view) {*

*// Switch fragment implemention*

*} // fragment imp. moved to OnDrawerClosed.*

With this new implementation, the navigation drawer is closed before any fragment transactions take place. This means that the stuttering which occurred when both operations were done concurrently is no longer an issue.

**4.3 Charts**

**Fig 0.0 Fig0.0 Fig 0.0**

MPAndroidChart (<https://github.com/PhilJay/MPAndroidChart>)

As android has no built in native chart APIs to develop with, a third party library called ‘MPAndroidChart’ was used to implement bar and pie charts in the application. These charts are used in the application to illustrate all-time highs and lows (bar) and percentage of the users portfolio made up of a particular stock icon (pie). The line chart illustrated in fig 0.0 is an image downloaded from a webpage to represent trends of a particular stock icon. Navigation between the different charts is done using the ViewPager class discussed in section 1.5.2 as well as by binding ‘OnClickListeners’ to buttons named B,L and P. The chart can then by swapped inside these ‘OnClickListeners’ using

*mPager.setCurrentItem(position,true);*

In order to create an instance of either the bar or line chart, the current ‘active’ quote must be passed as a parameter in its constructer. The current ‘active’ quote is visible as a ‘TextView’ above the respective chart fragment (AIB.IR (Fig0.0,0.0), EIL1.IR (Fig 0.0))

*new BarChartFragment(mActiveQuote);*

The data entries are then built using this ‘Quote’ instance that represents the current ‘active’ or selected stock item.

*// LineChart*

*BarEntry yHigh = new BarEntry((float)Double.parseDouble(mQuote.getYHigh()), 0);*

*BarEntry yLow= new BarEntry((float)Double.parseDouble(mQuote.getYLow()), 1);*

*BarEntry current = new BarEntry((float)mQuote.getLastTradePriceOnly(), 2); values.add(yHigh); values.add(yLow); values.add(current);*

These ‘BarEntries’ are used to build a ‘BarDataSet’ which the chart then sets as its data using *‘chart.setData(BarDataSet)’.* As mentioned above, the data set by the applications bar chart revolves around what the stock items current, year high and year low values are. The pie chart on the other hand uses the current ‘active’ stock item to compare the value of that particular stock within the user’s portfolio to the value of the portfolio as a whole. For instance, as illustrated in Fig 0.0, EIL1 or Aer Lingus stocks make up over 50% of the users portfolio while other stocks only amount to 49.08%. Selling these Aer Lingus stocks in the portfolio section of the application (discussed in section 2.4.5) would reduce the majority percentage of EIL1 stocks within the user’s portfolio.

*// PieChart*

*MySQLiteHelper stock\_port\_value = new MySQLiteHelper(getActivity());*

*stock\_port\_value.open();*

*totalPortfolioValue = stock\_port\_value.getPortfolioValueFromSQLLiteDB();*

*quoteValue = stock\_port\_value.getStockItemPortfolioValue (mQuote.getsymbol());*

*stock\_port\_value.close();*

Some custom SQLite methods were written to retrieve the portfolios total value as well as the value of all purchased stock items of type ‘active’ stock. These values are then converted into percentages and used to build a dataset similar to the one used by the bar chart discussed above.

By benchmarking other similar stock applications (section 0.0) it becomes apparent that line charts are generally used to illustrate stock value over a certain period of time, whether it be daily, monthly or even yearly. One restriction with this application is that the Yahoo API does not provide enough data to build charts similar to this. There is no array of stock values representing price over different periods of time. The API is limited to current price, yesterday’s price as well as daily/yearly highs and lows. For this reason the application uses an external webpage (Bloomberg give ref) to display information on stock items trending values.

* WebView (However introduces scrolling which is counter intuitive to viewpager)
* Retriving image was more suitable