Technical Guide for Final Year Project

# AutoSync Android Mobile Application

| | |
|---|---|
| Name: | Eoghan McMullen |
| Supervisor: | Monica Ward |
| Student ID: | 11442938 |
| Blog: | http://blogs.computing.dcu.ie/wordpress/eoghanmcmullen/ |

# Table of contents

# 1. Introduction

## 1.1 Overview

The system I developed is an automatic local backup system. The system comprises of a SyncBox and a mobile application named AutoSync which is installed on an android device. The SyncBox was created using a Raspberry Pi which acts as a wireless router, it also has a usb storage device plugged into it. The user uses the mobile application to connect to the SyncBox. The user then uses the mobile application to choose which folders they would like to have automatically upload to the SyncBox. Once the initial sync process is complete the user will receive a notification on their device any time their mobile device is within range of the SyncBox.

The system has a number of other features including single file upload, a shared file upload for multiple users, android memory clean up and a viewer that allows the user to see what is currently stored on the SyncBox and download the files they see.

The main idea behind this project is to provide a very low cost way to automatically back up any data on a mobile device with very little effort. Cloud services provide similar functionality but this system provides the functionality without recurring bills. In short the product developed is a home cloud that provides automated functionality through an Android application

## 1.2 Glossary

- **SyncBox:** A raspberry pi which is configured to act as a wireless router.
  This in combination with a usb storage device makes a SyncBox
- **RaspberryPi:** A credit card sized computer
- **Sync:** sending files between the android application and the SyncBox
- **JSCH:** a java implementation of SSH2 api
- **SFTP:** Secure file transfer protocol
- **Xamarin:** A studio for developing mobile applications in c#
- **Android studio:** An IDE for developing android applications
- **IDE:** Integrated development environment
- **Sqlite**: An in-process library that implements a self-contained, zero-configuration, serverless, transactional SQL database engine.
- **Async thread:** thread that runs separately to the user interface thread, used for performing background work.
- **SSID:** String name of a wireless access point
- **Persistent data:** In context of an it's data that is maintained in the application even when the app is closed i.e. local databases etc.

# 2. Motivation

There are many reasons I decided to undertake the AutoSync project. I have a huge interest in mobile development, in particular Android. I wanted to create an application that was useful, and easy to use. There are many people that don't avail of cloud services, particularly when they have to pay for it. AutoSync allows a user to make a one time purchase of the SyncBox and keep their files secure at home.

The user doesn't have to worry about running out of space, they can simple insert a new usb storage device whenever they like. Usb type storage tends to be a lot cheaper than cloud options, particularly over an extended period of time.

I also wanted to experiment with transferring data between a mobile device and a micro computer. The possibilities for micro computer related applications are endless and I wanted to experience what working between them was like first hand.

## 2.1 Inspiration

The inspiration for this project came from the number of times I'd heard people had lost their mobile devices which had all of their files. My usual question would be "Why didn't you back it up in the cloud" and the response I would nearly always receive: "I can't afford another bill" or "It's too much hassle". I decided to create a system that would give users the peace of mind of knowing that all of their files are secure in the cheapest, easiest way possible.

# 3. Research

There were a great number of items that needed to be researched when carrying out this project, I will list the ones below that I feel were most important and helped me to learn the most.

## 3.1 Xamarin Studio

When I began working on AutoSync I thought it might be possible to use Xamarin to create a cross platform deployable application. I began the application by coding in C# and put a good few week work into it. I later learnt that when dealing with different operating systems file structures that this would not be possible. Too much would have to change for each application. It simply wasn't viable in the available time to create a cross platform application.

## 3.2 SFTP

As my project involved a lot of file transfers I needed to use a secure reliable method of transferring files. I looked into many ways of doing this but sftp seemed the most reliable and the most likely to work with the Raspberry Pi. I discovered that Xamarin had no SFTP libraries available, this was another major drawback to coding the app in C#, there were only libraries that needed to be paid for. In Java JSCH is available for free. This was another of the main reason I decided to develop the application natively using Android Studio. I was able to confirm online that JSCH is usable with Android which was great news. It was apparent that the online documentation for JSCH was not very good. I spent a good deal of time writing simple java programs to learn how to use the JSCH libraries.

## 3.3 Android

In order to understand Android I decided to try and develop some simple applications before tackling my own one. I searched online for some simple apps and found that most people started by developing calculator apps etc. I developed a number of single page apps that performed some calculations, some of which I found quite useful. I still didn't feel comfortable with how much knowledge I had for tackling my own application so I signed up to www.teamtreehouse.com for a month. This gave detailed walkthroughs on developing a number of apps which I found very useful. By following tutorials and online help pages I was able to gain enough knowledge to understand how to go about tackling my own project

## 3.4 RaspberryPi

The initial plan for communication between the mobile device and the raspberry pi was to use Wifi direct. This turned out not to be possible. I researched a number of different ways to communicate between the two devices and found that the best way for me was to use the raspberry pi as a wireless router. I needed to learn how to configure IP tables and turn a wireless chip which generally serves as a wifi receiver into a wireless hotspot.

## 3.5 wireless chips

In order to connect the raspberry pi to the android device I needed to turn it into a wireless router. I needed to get a chip that would work as a broadcaster as well as a receiver. There are a number of wireless chips available for the raspberry pi, not all of them will provide this functionality. I purchased one and after hours of trying to configure it I discovered that it would not be possible.After some research I ordered a wifi dongle with the following chip, rtl871xdrv. This turned out to work exactly as required.

## 3.6 Raspberry pi 3

The  I started my project only the raspberry pi 2 was available. Unfortunately without some customisation the Raspberry pi 2's usb ports do not out put enough power to power an external usb drive. Fortunately when I saw the Raspberry Pi 3 was released I noted that the usb power had been upgraded. There was also now an onboard wireless chip that I could use to turn the raspberry pi into a router. This was a much better piece of hardware for my project so I upgraded.

## 3.7 broadcast receivers

One of the main features of the AutoSync application is to detect when a SyncBox is in range. This is vital, when it is detected a notification is sent to the user to allow the automatic sync to begin. I put many hours into finding the correct way to do this. I tried multiple solutions, one of which was a system service. A service is an app function that can run in the background even if the user has quit the app. The only way to turn it off is to uninstall the app. That is exactly the functionality I needed as it is likely the user will quit the app when not using it. Unfortunately the service didn't perform as expected. I found another solution using a Broadcast receiver. I set it to listen to network changes and check router names. Only when a SyncBox is detected does the notification fire. The broadcast receiver runs in the background even when the app is quit.

## 3.8 custom adapters

Custom adapters were one of my main research areas. There are many components needed to successfully implement a custom adapter and there is quite a lot of understanding needed to implement them, especially in a way which you would like them to look. A custom adapter is what allows a list to be customised to your own specification i.e. images, scrollable, multiple headings, buttons etc. I needed custom adapters throughout my app. There are currently four unique adapters in the AutoSync application, each providing unique functionality and design.

## 3.9 webpages

I wanted to create an easy to read help page within the application for the user. This was not easy using android methods. It was difficult to create something that looked good with good structure using java. I researched how one might go about adding a manual to an application. Eventually I discovered Androids webview class. This class allows you to embed a webpage within the application. I was able to create a mobile friendly webpage that would open within the app. It is very easy to navigate. I believe it was the best solution to the problem.

## 3.10 action bar

The Android developers webpage recommends giving users the ability to navigate the application using the action bar. Again this was more custom coding that had to be implemented to get the desired results. I added the action bar navigation as was recommended. I had to follow a number of guides when implementing this feature.

## 3.11 user interface

One of the main goals of my project was to create an application that was as user friendly as possible. This meant designing a user friendly interface that users would not find difficult to navigate. I spent time researching different apps and how they presented data to users. I eventually decided to emulate a windows tile app which meant the screen is filled with chiles which act as button, the buttons all have images overplayed on them to help describe to the user what functionality the button will provide.

### 3.10 accessibility

When designing a user friendly app it is necessary to put a lot of research into accessibility. Fortunately android have a  information available to make applications as accessible as possible. This includes a developer checklist which will be highlighted in the testing section of this guide.

# 4. System functionality

## 4.1 Initial connection

The user uses the mobile application to make the initial connection to the SyncBox. The user enters the password which is provided on a sticker on the SyncBox. When doing this the user enters a username. This username is used to create a parent folder on the SyncBox's storage device. If the mobile device successfully
communicates with the SyncBox then a database entry for the new user is created on a SQLite database on the mobile device. The  database holds all the necessary information to reconnect automatically to the SyncBox. The user is then added to the list of available users in the "My SyncBox's" menu. From there the user can select the user they just created and utilise all of the apps functionality.

## 4.2 Folder chooser

The folder chooser allows the user to browse the phones file system and select folders. These folders are used in the automatic sync. This folder chooser had to be created by me as Android has no in built functionality for this. The user can see what files are in each folder as they move through the different folders.

## 4.3 Automatic sync

This is the main functionality of the system and the most important. This is what allows the mobile device to automatically sync all of the user chosen directories to the SyncBox. When the users mobile device is within range of the SyncBox a notification is sent to the user. The user can click the notification which begins the automatic sync. The automatic sync gets all of the files in the chosen directories, including child directories and places them on whatever storage device is currently plugged into the SyncBox. The folder structure from the mobile device is used to create the structure on the storage device, the storage device therefore has the same layout as the mobiles directory.

Files which are already on the SyncBox are ignored except if that file has been updated. As the Sync is in progress any file that is found to already exist on the SyncBox is checked for an update. This is achieved by checking the number of bytes in the file. If an update is found to have occurred the file is replaced with the new version.

In the case where a user has set up multiple users on the same SyncBox from the same mobile device then both of the users will be backed up using the automatic sync functionality.

Users are able to use multiple SyncBox's, for example one at home and one at work. If the users has two SyncBox's the Automatic sync functionality ensures that the correct SyncBox is being accessed.

## 4.4 manual sync

A user can choose to perform a manual sync at anytime by navigating to the users menu and clicking manual sync. This will back up all of the user chosen directories to the SyncBox while maintaining the file structure of the mobile device. This is useful functionality in the case where a user needs to quickly ensure all files are backed up. This will only back up whichever user account is currently accessed.

## 4.5 Single file upload

This allows a user to browse through the phones memory and select a single file to upload to the SyncBox. If the file resides within a number of directories then those directories will also be placed on the SyncBox maintaining the phones file structure.

## 4.6 Shared folder upload

The shared folder functionality allows a user to upload a file to a shared folder on the SyncBox. The shared folder can be accessed by other users who have synced with the SyncBox.

## 4.7 Cleanup

The cleanup functionality allows the user to free up space on their mobile device. the user chooses a folder and a date. Any files that were created before the date chosen are backed up to the SyncBox and deleted from the mobile device.

## 4.8 Access SyncBox

This functionality allows the user to browse what is currently backed up to their account on the SyncBox. The user can download any file they like from here, it is saved to their downloads folder.

## 4.9 Access shared folder

This functionality allows the user to browse the shared folder on the SyncBox, the user can download any files they wish from here.

## 4.10 Help page

There is an embedded webpage with instructions on how to use all of AutoSyncs functionality within the app. The user can access it by pressing the “Help” button on the home screen.

## 4.11 Notifications

Notification functionality was added to the app to notify the user when an automatic sync could begin. The notification performs as a heads up notification which is certain to grab the users attention. Then can ignore it if they wish. The notification will show even if the user has quit the app.

## 4.12 Allow multiple SyncBoxes and users

AutoSync was designed to allow multiple users use the same SyncBox. That includes multiple accounts on the same phone if the user wishes. The user can also have multiple SyncBox's. The users accounts when created are assigned to the SyncBox they were initially synced with, this allows the user to use multiple SyncBox's e.g. home and work SyncBox.

## 4.13 Navigation

After the first round of acceptance testing it was evident that some users were unsure of how to get from one are of the app to the next. the action bar was customised to allow the user to navigate more fluidly. The action bar now includes a description of the current activity, A home button to allow the user to navigate to the home screen from anywhere in the app and a back button which returns the user to the previous activity.

# 5.Implementation

## 5.1 SyncBox

I decided to use a Raspberry Pi to create the SyncBox, the main reasons being it's size, linux based operating system, on board USB connection and price. By using this micro computer I knew I would be able to create an affordable system.

The SyncBox's main job is to act like a wireless router. It allows wireless devices to connect and provides internet to them. It also allows SFTP to a synced mobile device. In a market situation the SyncBox would come with a sticker on it that has the SyncBox's password. This would be used to sync the two devices.

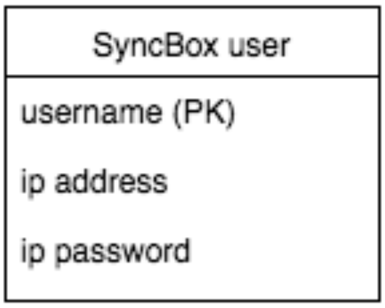
To turn the SyncBox into a wireless router there were a number of steps to perform:

1. Configure the Raspberry pi with a static IP address
    - This allows the mobile device to ensure it's connecting to the correct device. The IP address is saved to a database entry when the pi is first synced.
2. Install and configure a DHCP server
    - This serves up IP addresses to connecting devices.
3. Install and configure the access point daemon
    - This sets up the password protected wireless network that is broadcast from the Raspberry Pi.
4. Configure IP routing between the wireless and Ethernet
    - This was configuring the IP tables to allow the Raspberry pi to share it's internet connection with any connected wireless devices.

## 5.2 User Database

To allow the user to sync with multiple sync boxes and create multiple accounts on the same box , a database was needed. This also ensured that the system was not reliant on a certain ip address. The ip address is saved to the database when the initial connection is made between the two devices i.e. no hard coding values.

I decided to implement a SQLite database on the mobile application that would hold all of the relevant information for a connection. An image of the simple database is shown below.



The username is the primary key, this means that each of the usernames connected to the SyncBox must be unique. When the user is making the initial connection to the SyncBox a check is performed on the SyncBox to see if that username already exists on the usb storage device. The ip address and ip password are used to open a connection between the SyncBox and the mobile device whenever a function is being used.

## 5.3 Main functionality code

All of the main functionality for the application was developed using Java with Android studio. This meant using Android studios framework. The user interface aspects were created using xml. resources like strings were also created using xml. A screen in android is represented by an Activity. Classes are written to add business logic to the activities.

## 5.4 Creating a usable product

When creating AutoSync I wanted to create a product which would be usable on any android device, this meant no hard coding variable etc. This was quite a difficult task as communicating with the SyncBox needed a number of variables.
For example an ip address is needed for the SyncBox, the SyncBox has a static Ip address but each SyncBox's ip address is different. This meant identifying the ip address at runtime. I achieved this by setting the routers name to SyncBox, this meant a user can easily identify what network they should connect to. In the background when a user makes the initial connection the SyncBox's ip address is retrieved and saved to a database for that user account.
A user is also able to use any usb drive this meant that at runtime the android application would have to determine the usb drives name.

# 6. Sample code

As my application contains over 20 class files it is not possible to include samples from every class here. Below I will explain what I believe are the most important classes to this applications core functionality.

## 6.1 Async task

My application deals with transferring lots of data, this meant that I would need to use multiple threads. It is not possible or recommended to transfer data on an Android user interface thread as it can cause crashes and lagging on the users device. I needed to use an Async thread.

The Async task has a number of jobs and I will list what I used them for below.

```
void onPreExecute()
```

This was used to display different dialogs to user depending on the task being performed, e.g. if a user was performing a sync it shows a loading progress bar to the user. If a user is uploading single files it will show a different loading progress bar etc.

```
void onPostExecute()
```

This is fired when the work of the Async task is complete. Mainly I used this to turn the loading dialogs off and display a message to the user indicating the status of their request. i.e. whether their request had successfully completed or not

```
String doInBackground(String… params)
```

This is where the background thread does its work. I used this method to create an instance of a class CommunicatiorSSH and call appropriate functions.

The issue I had with Async task was that the doInBackground function performs whatever is in the method. I needed to choose what functions to run on the background thread from user input. Originally I thought that I might have to create an Async task for each piece of functionality of the system but this seemed like a bad path to take. It would mean lots of the same code.

If you look at the above method declaration for doInBackground you will see the parameter String… params. I was able to utilise this to let the doInBackground method know which functionality I wanted it to perform.

```
//username here represents the currently requested filepath
SSHBackgroundWorker sshWorker = new SSHBackgroundWorker(v.getContext(), username);
                                        sshWorker.execute("checkGetFilesOnPi", username);
```

In the above code you can see the string “checkGetFilesOnPi”, this is used in the background thread to determine which method the CommunicatorSSH class should run. This saved lots of typing and turned out to be a solution that worked really well. username is passed to let the communicator know which user account it is currently dealing with

Below I have included some of the code from the Async task, I have only included the doInBackground method as this is the most important one. I have removed most of it but left enough to show the overall idea.

```
        @Override
    protected String doInBackground(String... params)
    {
        //pass a string as a parameter, depending on that
        //string run the required method from CommunicatorSSH
        //pass the username too, get user info from db

        methodToRun = params[0];
        username = params[1];

        final SQLiteNameStorer db = new SQLiteNameStorer(mContext);
        SyncBoxUser sbu = db.getUser(username);

        if(sbu != null)
        {
            ip = sbu.getIp();
            ip_password = sbu.getIp_password();
        }
        //load values in from sub
CommunicatorSSH communicator = new
CommunicatorSSH(ip,ip_password22,username,mContext);


        if(methodToRun.equals("createUserDirectory"))
        {
            taskCompleted = communicator.createUserDirectory(username);
        }

        //Backup all user chosen directories to pi
        else if(methodToRun.equals("backUpUserChosenDirectories"))
        {
            //false here is for cleanup mode
            //empty string is for miliseconds for clean mode only
            taskCompleted = communicator.backUpUserChosenDirectories(this,false, "");
            if(taskCompleted)
            {
                numFilesUploaded = communicator.getNumFilesUploaded();
            }
        }
```

## 6.2 CommunicatorSSH

CommunicatorSSH is responsible for performing all of the data transfers to and from the SyncBox from the mobile device. This class is over 700 lines long so I only want to include the functions responsible for the automatic sync functionality because I believe this is some of the most complicated code in the application.

```
public Boolean backUpUserChosenDirectories(SSHBackgroundWorker backgroundWorker,
                                           Boolean cleanupMode, String
timeInMilisString)
{
    Boolean taskComplete = false;
    UsersDirectoryChoosingHandler directoryHandler;
    Long timeInMilis = 0L;

    //FOR CLEANUP can change the filename here to be the cleanup file with an if
statement
```

```
    if(cleanupMode)
    {
        //get the directories from the directory file for the user
        directoryHandler =
        new UsersDirectoryChoosingHandler(context,username + "CleanupFile");

        timeInMilis = Long.parseLong(timeInMilisString);
    }
    else
    {
        //get the directories from the directory file for the user
        directoryHandler = new UsersDirectoryChoosingHandler(context,username);
    }
    //populate directory path array
    ArrayList<String> directoryPaths = directoryHandler.readDirectoryList();
    //remove all unnecessary child directories
    ArrayList<String> cleanedDirectoryPaths =
directoryHandler.removeChildDirectories(directoryPaths);

    //open up the sftp connection
    Boolean connectionEstablised = openSshCommunication();

    if(connectionEstablised)
    {
        //for each directory traverse and do appropriate actions
        File file;

        String [] pathSplit;
        String path;

        //Should ensure user directory is created
        createUserDirectorySafe(username);

        //get the total number of files to upload.

        for(String s:cleanedDirectoryPaths)
        {
            file = new File(s);
            countFiles(file);
        }
        percentageToAddToProgress = 100.00/totalFileCount;

        //get the usb drive path to use
        if(getUsbDrivePath())
        {
            //traverse each parent directory and add files to device
            for (String directory : cleanedDirectoryPaths)
            {
                createParentFolders(sftp, directory,usbDrivePath,username);
                file = new File(directory);
                traverse(sftp,
file,usbDrivePath,backgroundWorker,cleanupMode,timeInMilis);
            }
        }
        taskComplete = true;
        closeConnection();
    }

    return taskComplete;
}
```

The above method has a number of tasks:

- Determine if we are running cleanup mode, if we are the users directories must be changed to only deal with the folder to be cleaned
- Get all of the users chosen directory by creating an instance of the UsersDirectoryChoosingHandler class.
- Ensure a connection is open to the SyncBox so SFTP can begin
- Ensuring a folder for the user is created on the usb drive that is currently in the SyncBox
- Ensuring the correct USB drive name is being used

- Ensuring that directories won't be dealt with multiple times by the transfer by removing child directories
- Notifying the asynctask that some data is transferred so the AsyncTask can update the progress bar
- For each directory calling the Traverse method.

This class was created following an object oriented approach, all of the tasks listed above were split into methods.

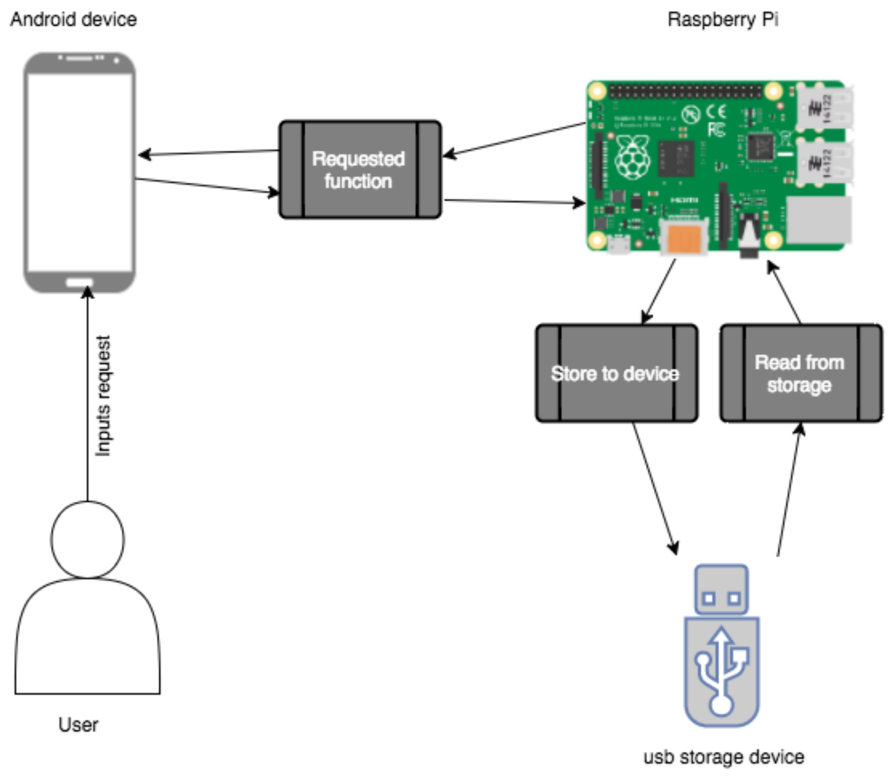
The other main method for this functionality is the traverse method shown below:

```
public void traverse (ChannelSftp sftp, File dir, String usbDrivePath,
                      SSHBackgroundWorker backgroundWorker,Boolean cleanupMode, Long timeInMilis){
    String EnvironmentPath = Environment.getExternalStorageDirectory().getAbsolutePath() + "/";
    SftpATTRS fileAttributes = null;
    Vector<ChannelSftp.LsEntry> fileList;
    String pathBeforeFolders = Environment.getExternalStorageDirectory().getAbsolutePath() + "/";

    //Long for getting file size
    long fileSizeInMB;

    if (dir.exists(){
        File[] files = dir.listFiles();
        for (int i = 0; i < files.length; i++){
            File file = files[i];
            if (file.isDirectory()){
                try{
                    String s = file.getCanonicalPath();
                    String folderPath = s.replace(EnvironmentPath,"");
                    //create the directory on the pi
                    createDirectory(sftp, "/media/pi/" + usbDrivePath + "/" + username + "/" + folderPath);
                } catch (IOException e){e.printStackTrace();}

                //recursively move into the next directory
                traverse(sftp, file, usbDrivePath, backgroundWorker, cleanupMode, timeInMilis);
            }
            //or else its a file
            else{
              try{
                    //remove filename from path
                    String filePathOnAndroid = file.getCanonicalPath();
                    String folderPathOnAndroid = filePathOnAndroid.substring(0,

filePathOnAndroid.lastIndexOf(File.separator));
                    String folderPathOnPi = folderPathOnAndroid.replace(pathBeforeFolders, "");
                    String fileNameOnly = FilenameUtils.getName(file.toString());

                    try{
                        //check if file exists on pi
                        fileList = sftp.ls("/media/pi/" +usbDrivePath + "/" + username + "/" + folderPathOnPi);

                        //check file size against current file
                        Boolean inPiStorage = false;
                        String lsFileName ="";

                        for(ChannelSftp.LsEntry ls: fileList){
                            lsFileName = ls.getFilename();

                            if(lsFileName.equals(fileNameOnly)){
                                inPiStorage = true;
                                fileAttributes = ls.getAttrs();
                                break;
                            }
                        }
                        if(inPiStorage){
                            //check file for update
                            long piFileSize = fileAttributes.getSize();
                            long androidFileSize = file.length();

                            if(piFileSize != androidFileSize){
                                //remove  old file, write new one
                                removeFile(sftp, "/media/pi/" + usbDrivePath
                                           + "/" + username + "/" + folderPathOnPi + "/" + fileNameOnly);
                                copyOverFile(sftp, filePathOnAndroid, "/media/pi/" + usbDrivePath
                                             + "/" + username + "/" + folderPathOnPi + "/" + fileNameOnly);

                            }

                        }
                        else if(!inPiStorage){
                            //add the file
                            copyOverFile(sftp, filePathOnAndroid, "/media/pi/" + usbDrivePath
                                         + "/" + username + "/" + folderPathOnPi + "/" + fileNameOnly);
                        }

                        //CLEANUP MODE DELETE HERE
                        if(cleanupMode){
                            Date lastModDate = new Date(file.lastModified());
                            Long dateModified = file.lastModified();

                            if(dateModified < timeInMilis){
                                //get file size
                                fileSizeInMB = file.length() / (1024 * 1024);
                                //Delete the file
                                deleteFile(filePathOnAndroid);

                                mbDeleted += fileSizeInMB;
                                numFilesDeleted++;
                            }
                        }
                        //update progress here
                        backgroundPercentageTotal += percentageToAddToProgress;
                        backgroundWorker.doProgress((int) Math.ceil(backgroundPercentageTotal));

                        //Increse number of files handled
                        numFilesUploaded++;
                    } catch (Exception e){
                        //e.printStackTrace();
                    }
                } catch (IOException e)
                {
                    // e.printStackTrace();
                }
```

The above traverse method is responsible for moving through each file in each directory (including sub directories) in a users chosen directories. When this method is run a number of things happen:

- Any directories that are found are created on the SyncBox's storage device, maintaining the same structure as the mobiles file structure.
- When a directory is found the traverse method is called recursively to handle all files within that directory.
- If a file is found:
- The file is checked if it already exists on the SyncBox, if it does already exist then it is checked for an update, the update is checked by comparing how many bytes the file on the SyncBox has and how many bytes the file on the mobile device has. If an update is found the newer version of the file is loaded onto the pi
    - If the file doesn't exist on the SyncBox it is copied over to the SyncBox.
    - If cleanup mode is enabled, if the file is found on the SyncBox it is deleted from the mobile device.
    - A count of the number of files and the amount of data transferred is kept so that the user can be notified.
- The traverse method will run until all files in the directory that the backUpAllUsers method passes to it.

# 7. System Architecture



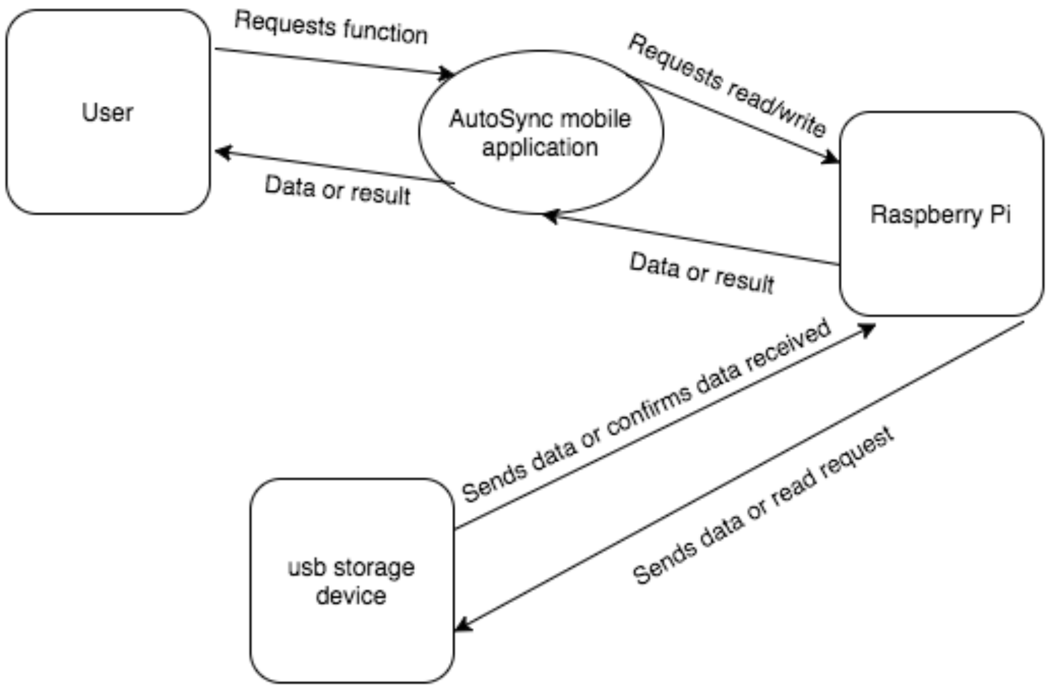
The above diagram shows how the system interacts. the user inputs a request into the mobile device, the mobile device then performs the action by interacting with the raspberry pi. Data is passed both to and from the raspberry pi depending on the function being used.

The Raspberry Pi provides access to it's USB drive wirelessly and the mobile device interacts with this. All functionality for transfers is coded within the app. The Raspberry pi only serves as a wireless access point with port 22 open for SFTP transfers. The user interacts with the system using the application on their mobile device.

# 8. High level design

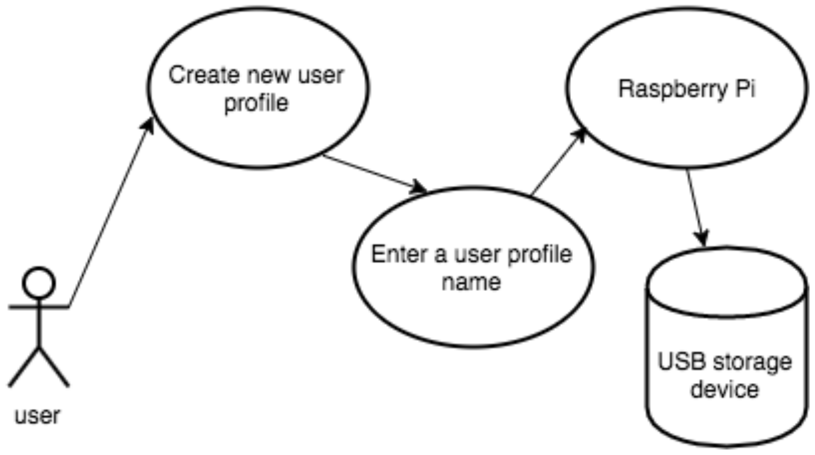
## 8.1 Context diagram

The below context diagram shows the highest level view of the system, data flows and passes to external entities are shown.



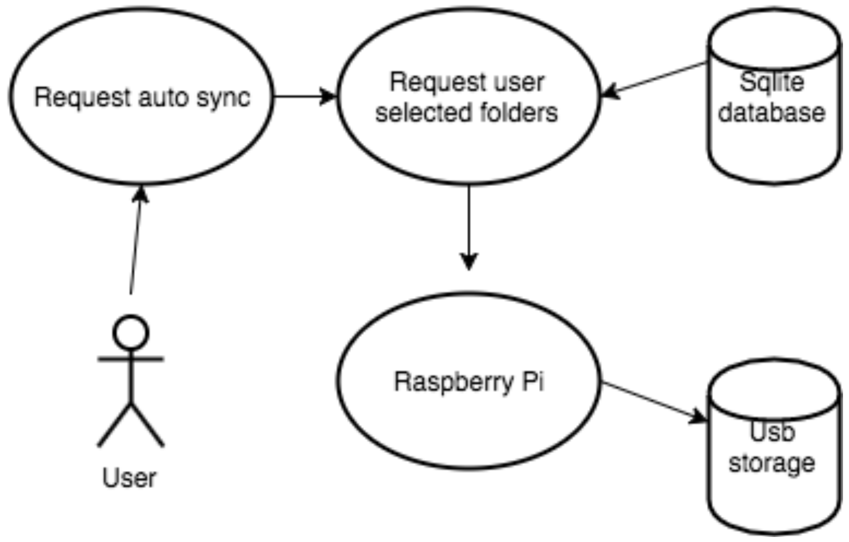
## 8.2 Initial connection

Here the user is creating a new user profile. The user enters a username, the username is sent to the Raspberry Pi which checks the username against existing ones. If the username is found to already exist on the pi the request is refused and the user notified. If the request is successful the username is used to create a folder for that account on the USB storage device. this folder is where all of the users files will be stored.



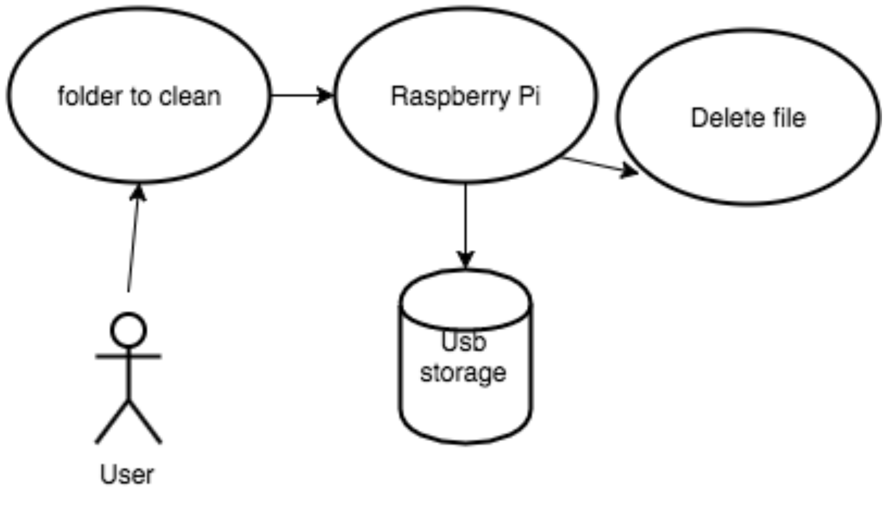
## 8.3 Automatic Sync

Here the user is requesting an automatic sync. The application checks the folders that the user has included in the automatic sync. The mobile app then establishes a connection with the Raspberry pi and checks the files currently on the usb storage device. If a file or folder is not on the usb storage then it is created. The code was written to handle usb devices being changed, if a new usb storage device has been input then the user profile is recreated on the new device. Files which are already on the usb drive are checked for updates, if no updates are found the file is not copied over again. This is more efficient.



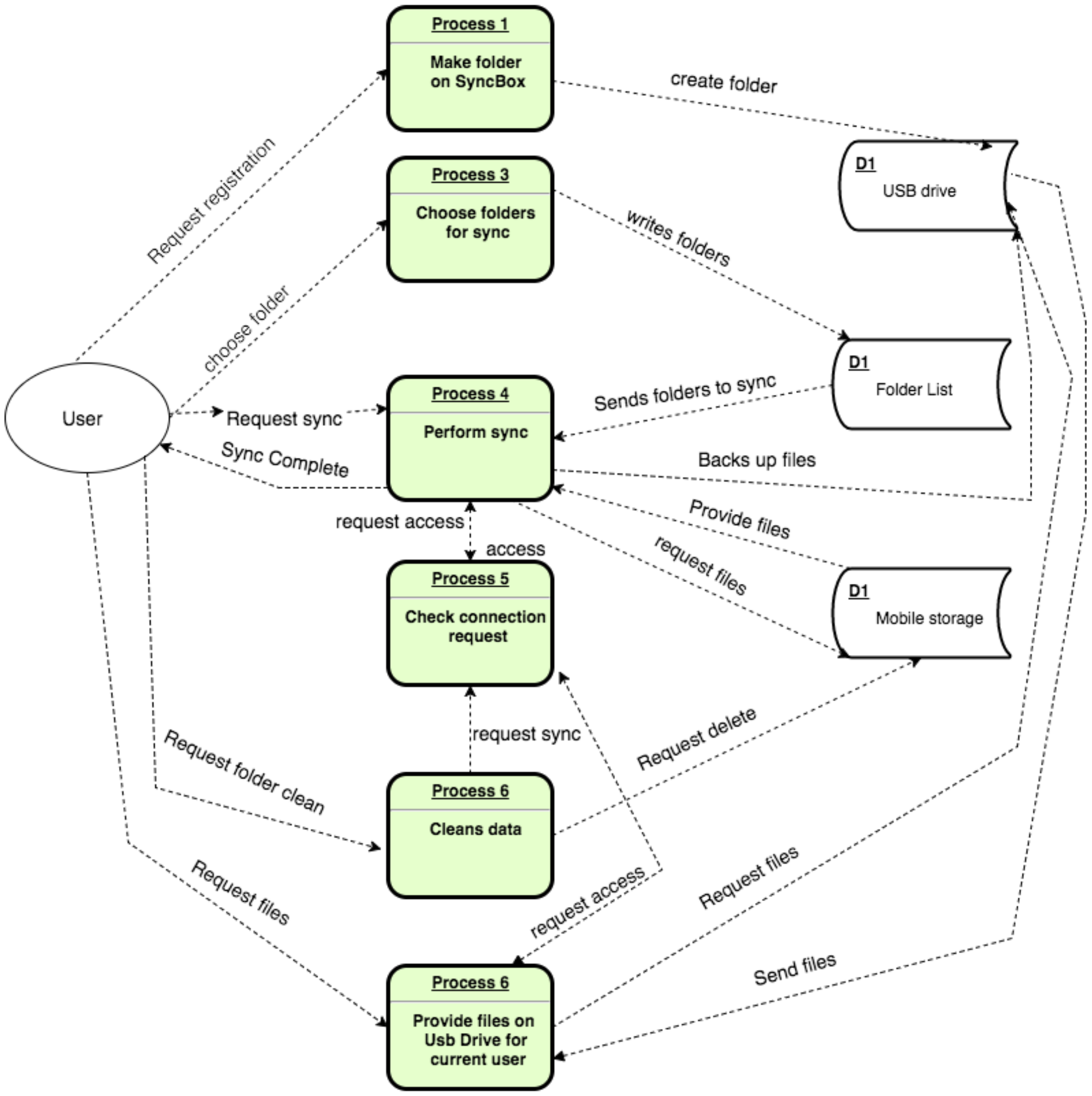
## 8.4 Data clean

j Here is the user requesting a data clean, the files in the folder selected for the cleaning process are saved to the USB drive in the raspberry pi. If the most recent version of the files are already on the pi they are not transferred. Any file that is successfully transferred is then deleted from the mobile device freeing up memory.



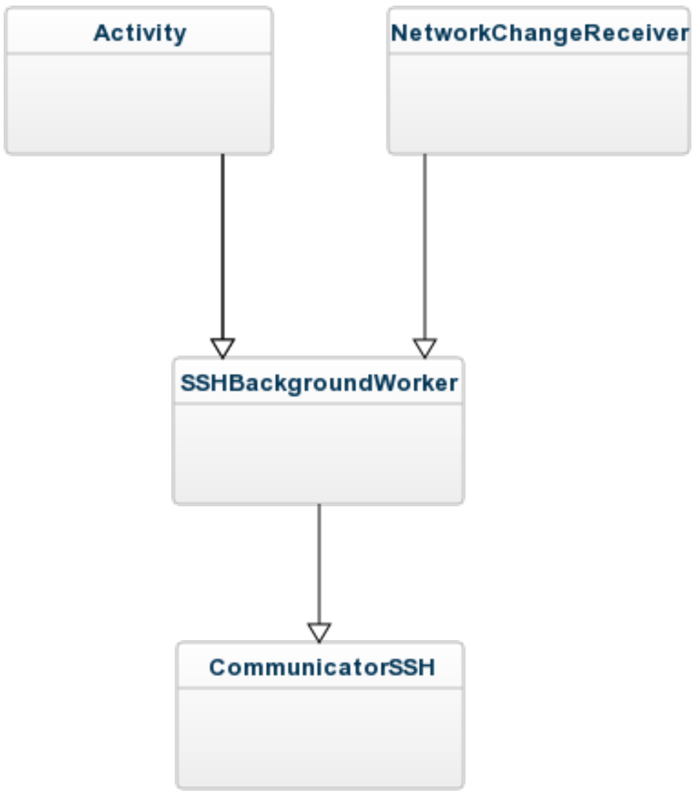
## 8.5 Data flow diagram

Below is a data flow diagram depicting how the information required for the system flows between the user , processes and data storage. Nearly all of the system functions(single file upload, manual sync, automatic sync etc.) will all require the same data flow.



## 8.5 Main classes interacting

The following diagram depicts how the main classes of the system interact. A user can choose a function which would come from an Activity(as shown below). There are 7 activities the user can use to interact with the SyncBox, The activity passes the required information to the SSHBackgroundworker which then uses the passed information to call an instance of the communicator on another thread. The communicator then performs all interactions with the Raspberry Pi. The NetworkChangeReceiver is responsible for determining if a Raspberry Pi(SyncBox) is in range. This is how the automatic sync functionality works.



## 8.6 Class descriptions

### 8.6.1 class HomeScreen

This is an activity class. It is responsible for the functionality behind the Homescreen of the application.

**Main jobs:**
- Allow user to access help webpage activity
- Allow user to access SettingsActivity activity
- Allow user to access the SyncedDeviceChooser activity, provided users are available
- Allow user to access the SelectRaspberryPi activity

### 8.6.2 class InstructionsWebView

- Provides an in app instructions page to the user.
- Provides navigation functionality back to the Homescreen.

### 8.6.3 class SettingsActivity

- An activity class responsible for allowing the user to enable and disable notifications.
- Provides navigation functionality back to the Homscreen.

### 8.6.4 class SelectRaspberryPi

- An activity class responsible for displaying a list of wireless hotspots to the user.
- Wireless hotspot list entries are customised with the CustomWifiRowAdapter class.
- Clearly identify a SyncBox to the user with an image.
- Allow refreshing of the page, SyncBox may not be ready yet.
- When a SyncBox is clicked by the user provide a dialog for entering a password.
- Creating an instance of BackgroundDialog, an async task to create a connection to the SyncBox.
- Turns wifi of the mobile device on if it is currently off.
- Accept a return from BackgroundDialog class, the return determines if the connection to the SyncBox was successful or not.
- In the case of an unsuccessfully connection the password dialog is shown to the user again indicating an incorrect password was entered
- Provides navigation functionality back to the Homscreen.
- If a user clicks an item other than a SyncBox displays a message to them indicating it is not a SyncBox
- On successful connection of a SyncBox, activity CloudRegistrationActivity is started, the connected ip address, password entered and access point SSID are passed to CloudRegistration activity.

### 8.6.5 class CustomWifiRowAdapter

- Responsible for customising the list on the SelectRaspberryPi class
- Assigns a wifi image to any routers, highlights a SyncBox with the AutoSync icon.
- returns the row that was clicked to the SelectRasperryPi class.

### 8.6.6 class BackgroundDialog (Async task)

- Responsible for connecting the access point(SyncBox) that is passed to it by the SelectRaspberryPi class
- Determines the SyncBox's security type i.e. (open, psk, wep, eap)
- Display a dialog to the user when the connection is in progress
- Return a boolean to the SelectRaspberryPi class to indicate whether connection to the SyncBox was successful or not.

### 8.6.7 class CloudRegistrationActivity

- Activity class responsible for allowing a user to enter a name that will be used to create a user account for them.

- Only allows a user name between 3 and 12 characters long, ensures no special characters are used
- Checks if the same username is being used for another account on the device by accessing the SQLiteNameStorer database.
- Starting an instance of SSHBackgroundWorker async task. Passing the new username and a string to the async task “createUserDirectory” indicating it to create a new directory on the SyncBox.

---

### 8.6.8 class SyncBoxUser

- Provides access to the Sqlite database on the mobile device SQLiteNameStorer.
- Provides functionality for getting and setting database entries

---

### 8.6.9 class SQLiteNameStorer

- Creates a Sqlite database to the application for persistent data.
- Stores the data required to connect to the SyncBox i.e. password of SyncBox, Ip address of sync box

---

### 8.6.10 class SyncedDeviceChooser

- Activity class, provides a list of currently synced user accounts.
- Populates the list using the CustomSyncedDevicesRowAdapter adapter class.
- CustomSyncedDevicesRowAdapter handles the clicks on the row items.
- Provides navigation functionality back to the Homscreen.

---

### 8.6.11 class CustomSyncedDevicesRowAdapter

- Responsible for customising the list in the SyncedDeviceChooser activity class
- Adds different colours to every row to enhance user experience.
- Adds icon to each row
- When user clicks on a row the username associated with that row is passed to the SyncedDeviceMenu class and the class is started.

---

### 8.6.12 class SyncedDeviceMenu

- Provides the syncing functionality for each user account. deals with the user that was passed to it by CustomSyncedDevicesRowAdapter
- Any functionality on this activity that requires SyncBox connection (e.g. “Single file upload”) gets checked for SyncBox connection before allowing the user access. The user is notified if they are not currently connected. This is checked by checked the current networks ip address against the ip address stored in the current users database entry.
- The Sync Now button is responsible for executing SSHBackgroundWorker async task passing “backUpUserChosenDirectories” as the parameter. This tells SSHBackgroundWorker to run a sync to the SyncBox.
- The Upload file and shared folder buttons are responsible for starting the ChooseDirectory class. They pass the selection mode they require the ChooseDirectory class to perform and the name of the function that called the class.
- The Cleanup device button is responsible for starting the CleanupFolderPicker class, the current username is passed.

- The Access SyncBox button is responsible for starting a dialog and asking the user whether they would like to access their own folder or the shared folder. The appropriate request is then started with the SSHBackgroundWorker async task.
- The Folder chooser button is responsible for starting the SelectFolders class and passing the current username to it.

---

### 8.6.13 class ChooseDirectory

- This activity class is responsible for providing the functionality for allowing a user to choose files and folders. The activity has a grid view layout which is styled from CustomDirectoryChooserAdapter adapter class.
- This class is always started for a result meaning that it will pass the calling class a result of some sort(a file name or folder name).
- The class behaves differently depending on what string the class that called it passes. The calling class passes a string indicating whether it should behave as a folder chooser or a file picker.
- When behaving as a folder chooser the class allows the user to choose a folder and passes the result to the calling class.
- When behaving as a file picker the calling class is passed a filename.
- The class blows the navigation of the whole filesystem on the mobile device i.e. into sub directories etc.
- When a user enters a directory the class allows them to see all of the files and folders within it.
- The back button  for this class has been specially overridden to allow navigation back to previous folders.
- Each time the users enters a new directory the adapter must be reloaded with the current directories files and folders

---

### 8.6.14 class CustomDirectoryChooserAdapter

- Adapter class that handles clicks on the ChooseDirectory activity.
- Styles all folders and files, assigns folder images to folders, file images to files etc.
- Identifies images in the file structure and displays the images in the ChooseDirectory activities gridview.
- passes a string of the file or folder path that was clicked to the ChooseDirectory class which then takes appropriate actions.

---

### 8.6.15 class CleanupFolderPicker

- Calls the ChooseDirectory class to allow a user to choose a folder to pick
- Allows user to use a calendar to select any date previous to the current days date.
- Takes the date chosen and parses it to a string value of milliseconds.
- Starts a SSHBackgroundWorker async task, passes the current username, the method to run (cleanupFolders) and the string representation of the date chosen in milliseconds.

---

### 8.6.16 class SelectFolders

- Activity class responsible for showing users what folders are currently selected for the automatic sync process.
- Add folder button launches the ChooseDirectory class to allow a user to add a folder to the list.
- The list is styled using the CustomFolderRowAdapter adapter class.
- The done button is responsible for writing any directories chosen to the apps persistent data.
- Any folders in the list when the done button is pressed will be written to a text file for that particular user. The text file stores all of the folders the user has chosen.

- Any folder that was removed by the user is removed from the persistent data file when the done button is pressed.

---

### 8.6.17 class CustomFolderRowAdapter

- Controls the bin button on the SelectFolders activity list view.
- Passes the id of the bin icon clicked to the SelectFolders class so that it can deal with the row the bin was pressed on i.e. remove it from the list.

---

### 8.6.18 class MimeUtils

- Determines the file type of a file i.e. image,video,music etc.

---

### 8.6.19 class NetworkChangeReceiver

- BroadcastReceiver extension class
- This class is responsible for listening for wifi changes on the mobile device
- The class determines if a SyncBox has been connected to
- It will run in the background even when the app has been quit by the user
- When a SyncBox is in range this class generates a notification and sends it to the mobile device.
- When the SyncBox is out of range the notification will disappear if the user has not interacted with it.
- The notification contains a button that allows the user to start the auto sync process.
- The notification is only sent if a user account has an ip address associated with that SyncBox.

---

### 8.6.21 class UsersDirectoryChoosingHandler

- Used to add, remove, retrieve any directory files associated to a particular user.

---

---

### 8.6.22 class SSHBackgroundWorker

- Async thread class.
- Responsible for creating threads that will perform any operations that will interact with the SyncBox.
- The calling class of the thread passes a string that relates to the method the SSHBackgroundWorker should run. This string is used to call a method using the CommunicatorSSH class
- While SSHBackGroundWorker is performing an action it displays a dialog to the user, depending on the action different progress bars are shown to the user.
- The dialogs are turned off when the thread finishes it’s work and displays different result messages to the users e.g success,fail, number files transferred, memory transferred etc.

---

### 8.6.23 class CommunicatorSSH

- SSHBackGroundWorker creates an instance of CommunicatorSSH and chooses the function it will run.
- CommunicatorSSH always runs on an async thread

- Performs all interaction between the SyncBox and the mobile device.
- Opens/ closes connections to SyncBox, sends files, receives files, downloads files etc.

### 8.6.24 class SyncBoxDirectoryBrowser

- Allows the user to browse the files currently on the SyncBox's memory device(usb).
- Allows the user to download files from the SyncBox shared folder or current users folder.

### 8.6.25 class AutomaticSyncInProgressActivity

- This activity is started when the notification is pressed
- Retrieves all of the user accounts on the mobile device and attempts to back their chosen folders to the SyncBox.
- Only user accounts that are linked with the SyncBox in range are dealt with.

# 9. Testing

## 9.1 unit / integration

Unit and integration testing was carried out throughout the project. As soon as a class was developed or even a method rigorous testing was performed on it. Instead of using an automation based testing system like Junit, I opted to test the code manually. For each method a list of inputs was drawn up along with expected results. The same process was used for integration testing. Fortunately while performing this testing plenty of bugs were found. the reason I opted for manual testing was the nature of the project being developed. When transferring files I needed to know whether those files had successfully transferred to the SyncBox, it was easier to perform these kinds of tests myself. Lots of user input is required for the system to run, these inputs can be emulated by Junit but I preferred using the debugger to step through each piece of code. Android studio allows you to use a physical device while debugging so I was able to input the appropriate gestures until reaching a breakpoint and then viewing results. Appendix B contains a table with a number of the unit and integration tests that were carried out. The full table is available at request. It was far too long to include.

**SOME RESULTS**

## 9.2 Monkey Testing

Monkey is used to stress test an application. It helps to identify whether the application is stable or not. It sends a random number of keystrokes to the application. Errors are reported back to the application. Ideally it should find any keystroke combinations that will cause screen freezing or app crashing. The Monkey tests I carried out can be seen in Appendix A. The defects found are at the end. This testing highlighted some defects that I missed in unit/integration testing, these defects were causing app crashes. To effectively use Monkey the tests should be performed on each activity of the application. Record any bugs found or errors thrown.

## 9.3 user acceptance

For my project I carried out two rounds of user acceptance testing. The first round was to try and identify any areas of the application that I could improve upon or functionality that users might like to have implemented. This was carried out early in the development process to give myself time to make any significant changes that were highlighted. The method I chose for user acceptance testing was a survey. In the first round of testing the users allowed me to deploy the app to their devices, it was not yet available for download. The first round of user acceptance testing yielded some great results that I will highlight below.

**User acceptance testing round 1 key notes**

The application was far from finished when user acceptance round 1 was undertaken. It was a much more simply design with a lot less functionality. I requested that the participants undertake a survey after using the app. My main goal here was to find out if the users found the app useful and if they had any suggestions for improving the app.

After reviewing the results of the survey these were the key elements that needed to be addressed:

- Users didn't understand the file browser. The original file browser I implemented was made using a simple list and text with the folders identified by a "/". Nearly half of the users noted that they could not clearly understand this. I decided then to implement a file browser that was image based.
- When the users were asked what they could imagine themselves using the app for, the majority of responses were "photos". I knew my file browser would have to show photos for the app to be a success.
- Many users noted that they found it difficult to know their current location in the app. I decided to follow androids best practises. For the next version I implemented the action bar with navigation.
- Some users couldn't understand how to use some of the functionality or even what it did. I decided to add an instruction manual to the next version.
- A number of users noted that they would like to be able to upload a single file if they were in a hurry.
- My supervisor gave me an idea to implement a shared folder that would be accessible to all users from their mobile device. I added a question to the survey to see if users might find the functionality useful. Over half of the testers thought it would be a good feature.
- Users requested that they could see what files were uploaded to the SyncBox. I decided to implement a file explorer to view the files on the SyncBox.
- When asked about the overall look of the app most users responded that they would like more colour to make the app more attractive.

I took all of the feedback from user acceptance testing round 1 on board and tried to implement as many of the features as possible before the next round of testing.

# 10. Problems and Resolutions

## 10.1 Wifi direct

I attempted to get a wireless chip which would allow Wifi direct access. Unfortunately this was not possible. The Raspberry Pi would not accept connection using Wifi direct. I had many different options I had to explore. I thought about using bluetooth connectivity. I didn't want the limited range of bluetooth and it's slower upload speeds in comparison to Wifi. Connecting through an existing network was an option but that would mean that the SyncBox was dependent on a wireless router. I decided to turn the Raspberry pi into a wireless router itself. This meant that the pi could be taken anywhere and used by simply plugging it in, no additional hardware necessary and this was the solution I liked best. This solution also gave another massive advantage. I was able to configure the Raspberry Pi to provide internet access so it can also replace a home router. My initial idea of Wifi direct didn't work out but I believe the solution I came up with was even better.

## 10.2 Xamarin

I initially wanted to build the application using Xamarin studio. The idea behind this was to allow for deployment to different mobile platforms. Unfortunately I discovered that too much code would need to be written to allow for different file systems across different mobile platforms. Another huge drawback here was that Xamarin had no SSH apis. This is what would be used to transfer data between the mobile application and the SyncBox. The only api's available for Xamarin were paid, in  excess of $300, this wasn't an option for me. I decided to go native and restart the application in android studio. JSCH, a java implementation of SSH2 was available which was a lot more promising for me. There were no real benefits left to coding using Xamarin Studio so I decided to cut my losses. I did however learn some C# coding which may be useful in the future.

## 10.3 File explorer

A file explorer was something I imagined Android would provide to developers, I was mistaken. This turned out to be one of the much more challenging aspects of the project. I initially developed an explorer that had no images and directories were identified with a "/". After my first round of user acceptance testing it was evident that this would not be good enough. Nearly all users couldn't understand how to navigate the filesystem. Towards the end of the project I was much more experienced with custom adapters(this is what allows you to have a list of items within an app that have images, multiple headers etc.) I developed a filesystem which was much more user friendly using images. The user acceptance round 2 showed that users were much happier with it.

## 10.4 Images from phone

One of the main reasons users told me they would like to use the AutoSync app was for backing up photos. In round 2 of user acceptance testing it was evident that users wanted to see what photo they were choosing rather than going off the photos name. I decided to add functionality that would allow photos to be seen in the file explorer. this proved to be a difficult task. For many hours the app was crashing from overloads. This happens when the system tries to load too many photos into memory at once. I tried limiting the images size but it did not work. Eventually I discovered a third party library called Picasso. This handles the memory overload by only loading photos that are being displayed to the user. The end result was very pleasing.

## 10.5 Automatic sync - notification

The initial plan for the Automatic sync functionality was to have the application perform the sync in the background without notifying the user. The user wouldn't have to press anything. A number of testers pointed out flaws in this logic. Including scenarios where users would only be within range for less than a minute and then gone again(i.e. if a user ran into the house and back out). The user would also not have the satisfaction of knowing their files are backed up. The notification functionality was introduced as an alternative. Whenever the users mobile device is within range of the SyncBox they will receive a notification allowing them to decide whether to perform a sync or not. They only need to press one button. I think this alternative is better to the original idea, the user is made aware that their data is backed up.

## 10.6 marshmallow

To understand this issue I'm going to describe the scenario that happened to me. The AutoSync application was being developed and many functions working. I was happy with the progress. I updated my phone(which is the main device for testing) to Android marshmallow. The next morning I tried to deploy the app to the phone. None of the functionality was working. The app seemed to be broken. I spent hours with the debugger looking for the cause but could not find it. I then plugged in a tablet and deployed the app and it was working. I discovered that I had mistakingly aimed my app at API 23 which is Android Marshmallow. The device I was testing on was API 22. Marshmallow introduced new security features which meant asking the user for permissions within the app instead of when it was being installed. I was able to aim my app at API 22 which was the correct version for me. In future I will know to be more careful when deciding which API level to use.

## 10.7 Raspberry pi usb power

When I was using the Raspberry pi 2 as the SyncBox it was not possible to use a usb hard drive as  the usb ports did not provide enough power. I was able to route power from one of the other usb ports to the one the usb hard drive was plugged in to but this was not an ideal solution, the hard drive would turn off from time to time. When I saw that the Raspberry pi 3 had increased the power available in the usb ports I purchased one which solved the issue.

## 10.8 Multiple Notifications

The broadcast receiver was sending multiple notifications to the user when the SyncBox was in range. For an unknown reason the Broadcast receiver was registering the SyncBox as connected multiple times, this is why the notifications were pouring through. I decided to use a static variable in the code which locked the notification from publishing again once it was published. This solved the issue in an efficient manner

## 10.9 Multiple usb devices

Finding a way to determine the name of the usb plugged into the the Raspberry Pi proved very difficult. I realised that all devices that were not storage were registered in the "/media/pi/" directory of the Raspberry Pi as Settings i.e. the names always began with "Settings". I used that to my advantage by only accepting a usb drive path that didn't contain the "Settings" keyword. This has proved to be a perfect solution and allows the user to change what usb device they are using at any time without any configuration necessary.

# 11. Results

The results for this project were overall a success. The idea I originally came up with became a reality. I developed an Android application that allows very easy transfer and backup of files to a usb drive which is plugged into a Raspberry Pi. I had to program things differently to what I originally imagined, for instance I originally thought that the code to perform the back up would be located on the Raspberry Pi, it turned out to be a much better option to code all functionality from the mobile device. One aspect of the application that I am disappointed with is the lack of cloud connection. This turned out to be much more difficult than I originally thought it would. I am aware now of how to tackle the cloud connection feature, with more time available I think I could have implemented it successfully. However my functional specification stated that cloud functionality would only be implemented if time allowed. I am happy to have a working product which performs all of the vital functional requirements.

The original sales pitch was to sell a SyncBox with a hard drive hooked up to it e.g. a user could purchase a terra byte SyncBox etc, Using user feedback it was determined that allowing the user to source and use any usb drive they like was a much better option. This also means the SyncBox could be sold at a much lower cost.

# 12. Future Work

## 12.1 Cloud connection

One of the features I attempted to add to the system was cloud access to the SyncBox. This proved to be more difficult than originally expected. In order for the SyncBox to be available over the internet the router the SyncBox is using for internet connection would need to have port forwarding enabled to allow the SyncBox to be accessed. This wasn't a viable option for the SyncBox as it would mean users would have to enable port forwarding themselves which most users could not do. It would make the product overly difficult to use. I did discover what I thought would be the solution. www.dataplicity.com provides functionality to allow your raspberry pi to be accessible through an internet connection without port forwarding enabled. Unfortunately they do not provide functionality for SSH. I did host a website on the SyncBox which is accessible using dataplicity. Unfortunately I did not have enough time to implement a website which would serve up the SyncBoxes data. It was outside the scope of this project. I would definitely like to continue on this aspect of the project and get it working.

## 12.2 User requested features

When performing user testing a number of features were requested. I managed to implement a few of them but not all were covered.

- Some user would like the option to password protect their folders in a situation where multiple users are using the same SyncBox.
- users would like to be able to select multiple files instead of just single files when uploading to  the SyncBox
- Users would like to be able to access and download their files from a computer.
- Search functionality, for finding a particular file quickly.

## 12.3 Features I would like

I noted a number of features I feel would make the application better

- The ability to open any file from the apps file explorer in the mobile devices in built app. e.g. if a user selects an audio file it would open in google music
- For the Raspberry Pi I would like a receiver with better range. The range isn't awful but could definitely be increased.

# Appendix A: Monkey Testing

**Monkey Tests Table**

| Test ID | Activity | Results | Issue | Actions to take |
|---|---|---|---|---|
| 1 | HomeScreen | Success | | None |
| 2 | HomeScreen | Monkey connected to a syncbox with wrong password | Only connected because the SyncBox had been connected successfully previously, not really an error. The device was connected to the SyncBox before starting. | None, not a critical issue |
| 3 | HomeScreen | Tried to connect to wireless devices that were not SyncBox's, System blocked it and showed toast. Success | None needed | None |
| 4 | Instructions | No issues | | None |
| 5 | Instructions | No issues | | None |
| 6 | Instructions | No issues | | None |
| 7 | Connected SyncBoxes | No issues, ran 5000 keystrokes | | None |
| 8 | Connected SyncBoxes | No issues, ran 5000 keystrokes | | None |
| 9 | Connected SyncBoxes | Uploading dialog is closed on back pressed | Need to disable back button while progressdialog is showing | dialog.setCancelable(**false**); added to each progress dialog. |

| | | | | |
|---|---|---|---|---|
| 10 | Connected SyncBoxes | No issues, 5000 strokes | | None |
| 11 | User Menu | No issues, noticed that Monkey often turns off the Wifi, safeguards in place for no connection are working | | None |
| 12 | User Menu | No issues, 5000 touches ran | | None |
| 13 | Choose a file | No issues, Noticed that no touches went through once progress dialog was showing, good fix applied | | None |
| 14 | Cleanup Device | No issues, 5000 strokes | | None |
| 15 | Cleanup Device | No issues, 5000 strokes | | None |
| 16 | Registration | No issues, safeguards for correct email format etc. working | | |
| 17 | Registration with correct email typed already | No issues | | None |
| 18 | Cleanup device | java.lang.ArrayIndexOutOfBoundsException: length=1; index=2<br><br>On getdateInMilis | Found issue, when "Cleanup now" button is pressed with no data entered the app crashes | Was trying to use a value that could be NULL |
| 19 | Cleanup Device | No issues | | |
| 20 | Cleanup Device | No issues | | |
| 21 | AutoSync Folders | No issues | | |
| 22 | Instructions | No issues | | |
| 23 | Instructions | No issues | | |
| 24 | Homescreen | No issues | | |

| | | | | |
|---|---|---|---|---|
| 25 | Select Syncbox | No issues | | |
| 26 | | | | |
| 27 | | | | |

**Monkey tests defects found**

| Test ID | Issue | Fix | Completed? |
|---|---|---|---|
| 9 | Progress dialog gets closed on back pressed | dialog.setCancelable(**false**); added to each progress dialog. | Yes |
| 18 | java.lang.ArrayIndexOutOfBoundsException: length=1; index=2<br><br>On getdateInMilis | Was trying to use a possible null value | Moved the code to a null safe location |

## Appendix B: Unit/Integration tests

| **Class** | **Method** | **Input** | **Expected Output** | **Actual Output** | **Result** |
|---|---|---|---|---|---|
| HomeScreen | Instructions Button | User click | Go to help page | Help page displayed | Pass |

| | | | | | |
|---|---|---|---|---|---|
| HomeScreen | SyncedDevices Button | User click | Go to Synced Devices list activity | Synced devices activity displayed | Pass |
| HomeScreen | syncNewDeviceButton | User click | Go to sync new device activity | New device activity displayed | Pass |
| Homescreen | OnBackPressed | User clicks back button | No action | No action | Pass |
| | | | | | |
| SelectRaspberryPi | generateWifiSSIDs | Available wireless devices | List of available wireless hotspots available populated | All wireless device SSIDs in List | Pass |
| SelectRaspberryPi | turnOnWifi | Wifi disabled | Wifi is turned on and user is notified | As expected | Pass |
| SelectRaspberryPi | wasSyncboxClicked | User click on wifi list item | A. If SyncBox was clicked then Password dialog shown to user<br>B. If other device clicked show user Toast | As expected | |
| SelectRaspberryPi | checkConnectionSuccessful | Async Task returns boolean and the connected SSID | If a SyncBox is connected the Registration activity is started | As Expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| SelectRaspberryPi | checkConnectionSuccessful | Async Task returns boolean and the connected SSID | If a SyncBox Is not connected the wrong password dialog is shown to user | As Expected | Pass |
| SelectRaspberryPi | onBackPressed | User clicks back button | HomeScreen Activity is started | As Expected | Pass |
| SelectRaspberryPi | Action bar back button | User clicks action bar back | HomeScreen Activity is started | As Expected | Pass |
| SelectRaspberryPi | Action bar home button pressed | User clicks action bar home | HomeScreen Activity is started | As Expected | Pass |
| SelectRaspberryPi | Password entry Dialog cancel | User clicks cancel on password entry dialog | Dialog disappears returning control to activity | As Expected | Pass |
| SelectRaspberryPi | Password entry dialog ok button | User clicks ok button | Launches Background task, user shown loading dialog | As Expected | Pass |
| SelectRaspberryPi | swipeRefreshListener | User pulls down on listview | List is refreshed | As Expected | Pass |
| | | | | | |
| Instructions webview | onBackPressed | User clicks back button | HomeScreen Activity launched | As Expected | Pass |
| Instructions webview | onBackPressed | User clicks back button | HomeScreen Activity is started | As Expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| Instructions webview | Action bar back button | User clicks action bar back | HomeScreen Activity is started | As Expected | Pass |
| Instructions webview | Nav list item clicked | Nav list item clicked | User brought to correct heading | As Expected | Pass |
| | | | | | |
| CloudRegistration Activity | onCreate | SelectRaspberryPi intent | String ip populated with ip of wireless network chosen in previous activity | As Expected | Pass |
| CloudRegistration Activity | onCreate | SelectRaspberryPi intent | String ipPassword populated with ipPassword entered in previous activity | As Expected | Pass |
| CloudRegistration Activity | onCreate | SelectRaspberryPi intent | String ssid populated with ssid of SSID chosen in previous activity | As Expected | Pass |
| CloudRegistration Activity | Done Button if statements | Users enters wrongly formatted email address | Error message appears on email edittest | As Expected | Pass |
| CloudRegistration Activity | Done Button if statements | Users enters Username too short or long | Error message appears on username edittext | As Expected | Pass |
| CloudRegistration Activity | Done Button if statements | User doesn’t enter username but email is correct | Error message appears on username edittext | As Expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| CloudRegistration Activity | Done Button if statements | User enters username with space | Error message appears on username edittext | As Expected | Pass |
| CloudRegistration Activity | Done Button if statements | User enters empty password having entered correctly formatted email and username | Error message appears on password | As Expected | Pass |
| CloudRegistration Activity | Done Button if statements | User enters too short or long password (5,8) having entered correctly formatted email and username | Error message appears on password | As Expected | Pass |
| CloudRegistration Activity | Done Button if statements | User enters password containing characters other than letters or numbers having entered correctly formatted email and username | Error message appears on password | As Expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| CloudRegistration Activity | Done Button if statements | User has entered correctly formatted username, email and password | Boolean validation stays true | As Expected | Pass |
| CloudRegistration Activity | isEmailValid | String email | Return true if email ok format | As Expected | Pass |
| CloudRegistration Activity | isEmailValid | String email | Return false if email format wrong | As Expected | Pass |
| Choose Directory | onBackPressed | directoryPath != Enviroment root<br><br>Case directoryPath is a directory | Counter = 1<br><br>Edittext moves back one directory<br><br>itemClickHandler called with direct parent directory | As expected | Pass |
| Choose Directory | onBackPressed | directoryPath != Enviroment root<br><br>Case directoryPath is a file | Counter = 1<br><br>Go back a directory and file removed | FAIL<br><br>Messes up | FAIL |
| Choose Directory | On cancelButton click | User click<br><br>Previous Activity: SyncedDeviceMenu | Previous activity is called | As expected | Pass |
| Choose Directory | On cancelButton click | User click<br><br>Previous Activity: Cleanup | Previous activity is called | As expected | Pass |

| | | | | | |
|---|---|---|---|---|---|
| Choose Directory | On cancelButton click | User click<br><br>Previous Activity: Choose file | Previous activity is called | As expected | Pass |