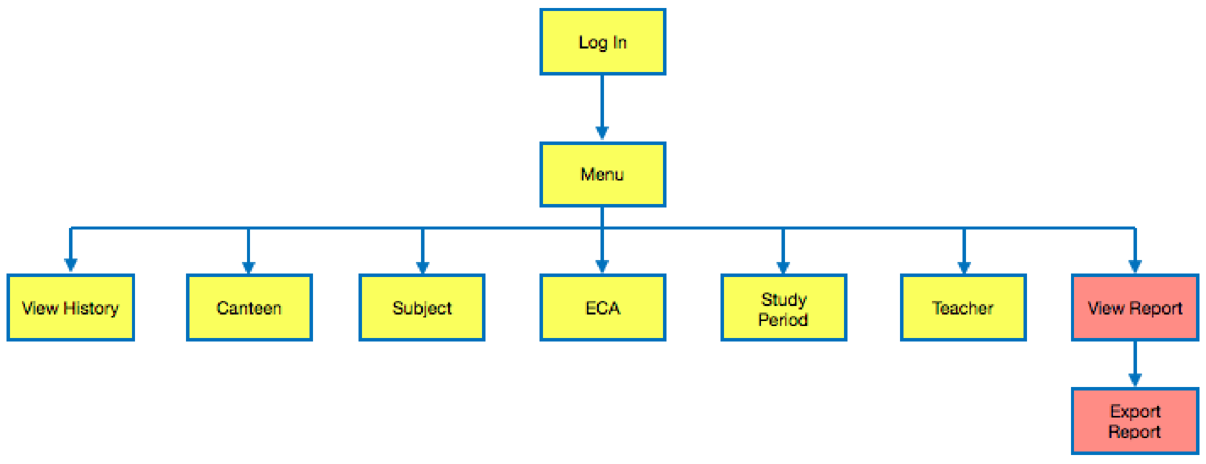
SECTION B

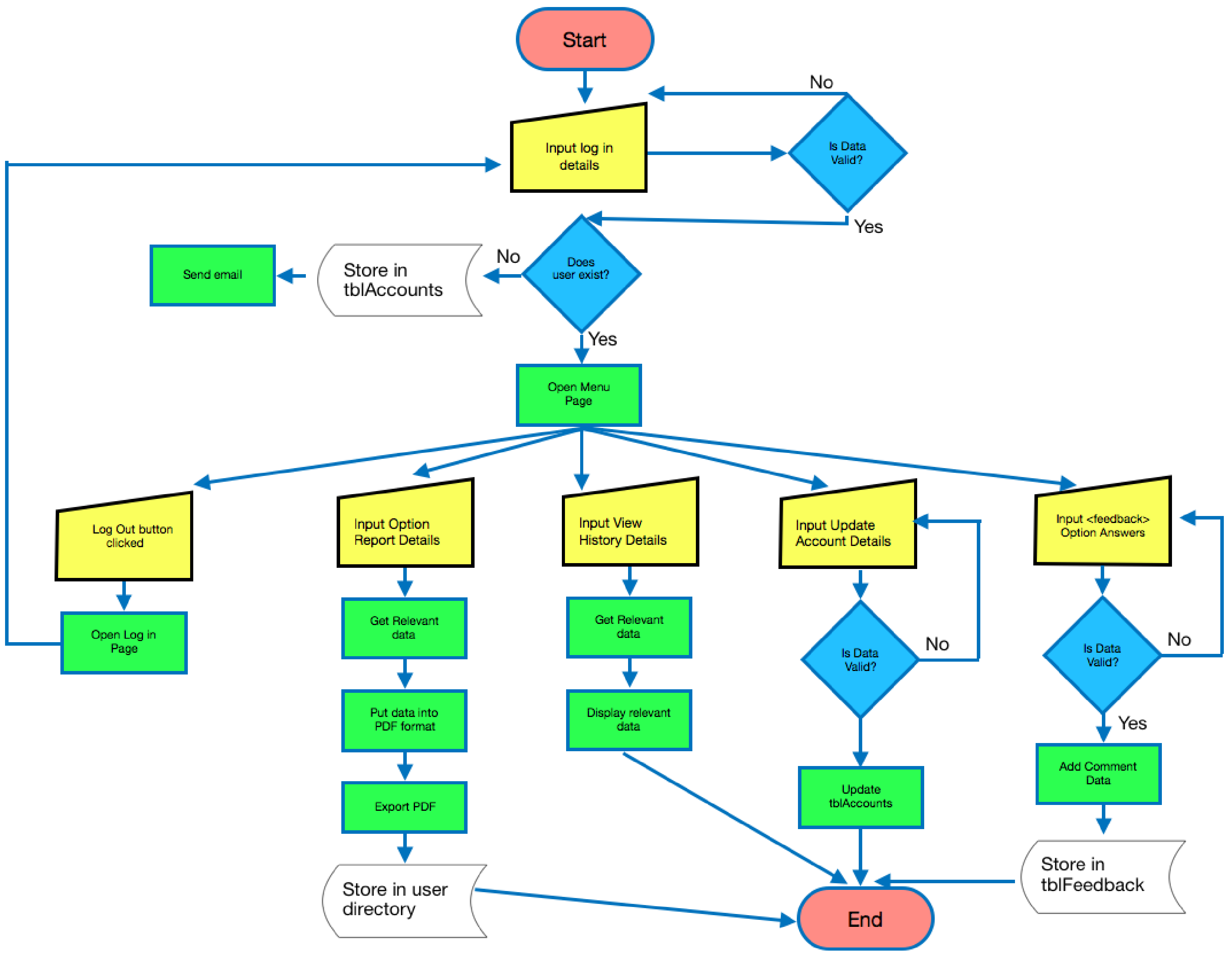
OVERALL STRUCTURE

MENU NAVIGATION (01/05/18)



Above is a picture of my “Menu Navigation Diagram”. The yellow boxes show all the menus visible to both students and administrators, whilst the red boxes are menus visible to administrator only.

SYSTEM FLOW CHART (02/05/18)



The diagram above shows my system flow chart. The diagram follows the key that can be seen below:

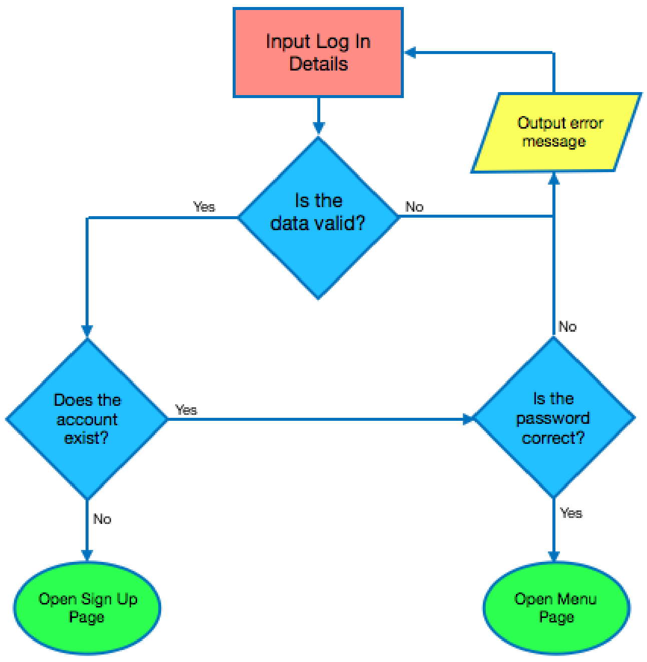
Symbol	Meaning
⬭	Start or End for the System Flow Chart
▱	Manual Input
◆	Decision
⊂⊃	Store in Database
↓	Flow of Data

DATA FLOW DIAGRAMS (10/05/18 - 11/05/18)

Data flow diagrams show how the user data travels through the app. The key can be seen below:

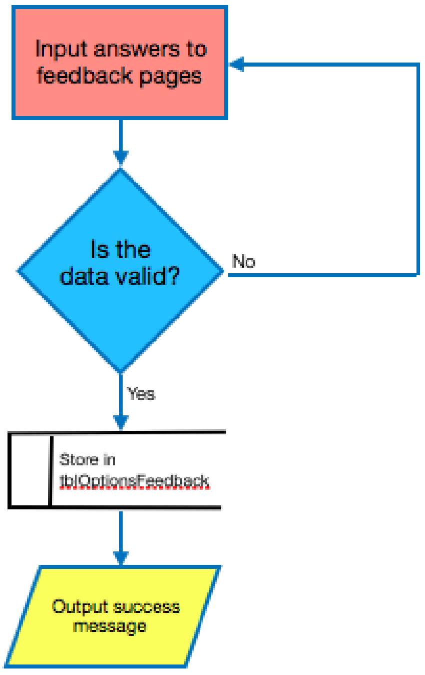
Symbol	Meaning
🟥	Source of Data
🔷	Decision
🟢	Process
▱	Output in Application
⊏	Store in Database
↓	Flow of Data

LOG IN PROCESSES



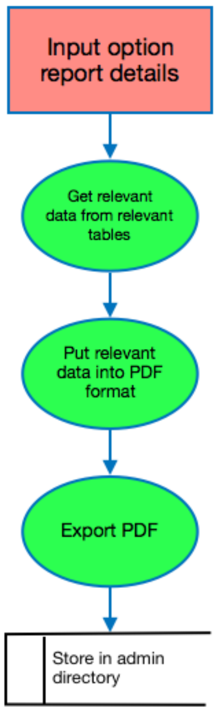
This diagram shows my Data Flow Diagram for the Log In process. First, I validate the inputs to ensure the log in details match my requirements (as mentioned in Appendix B). Then, if the account exists, I check the password. If the password matches the one stored in the database, then the menu page is opened.

COMMENTING PROCESS



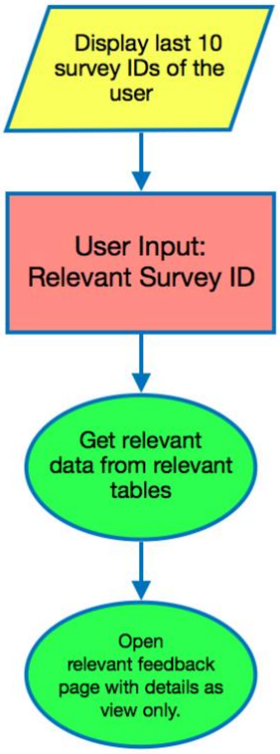
This diagram shows how comments can be added from the relevant feedback pages (eg: “Canteen”, “Teachers” etc.). Before storing, the entries are timestamped and a unique survey ID is also generated.

EXPORT OPTION REPORT PROCESS


Input option report details
Get relevant data from relevant tables
Put relevant data into PDF format
Export PDF
Store in admin directory

The DFD above shows how the PDF report will be created and stored on the admin’s phone.

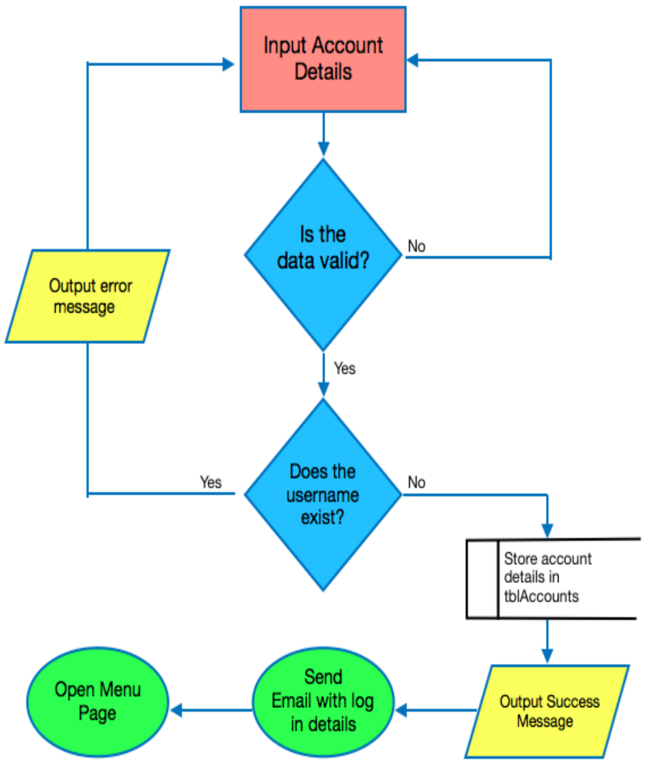
CHECK HISTORY PROCESS


Display last 10 survey IDs of the user
User Input: Relevant Survey ID
Get relevant data from relevant tables
Open relevant feedback page with details as view only.

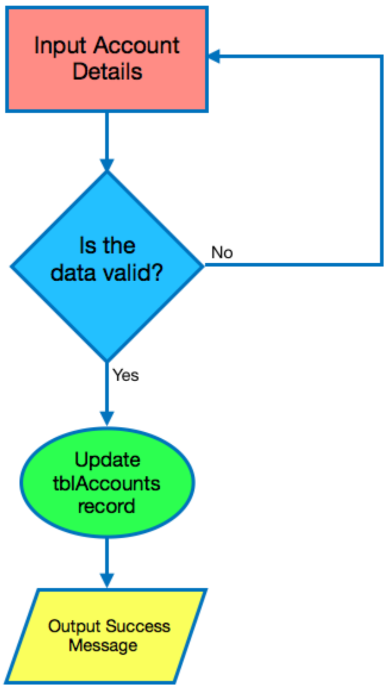
This DFD shows how users can see what they have posted previously across all the feedback areas. As can be seen in the design of the "view history page", a table view displays the Survey IDs along with the category and time of submission. Upon clicking on the survey ID, the user will be taken to a view only copy of the survey form where they can see their previous inputs.

SIGN UP PROCESS

Once the relevant details have been filled out, the username is checked to see if that username already exists in tblAccounts. If it doesn't, then the account details are stored.



UPDATE ACOUNT PASSWORD

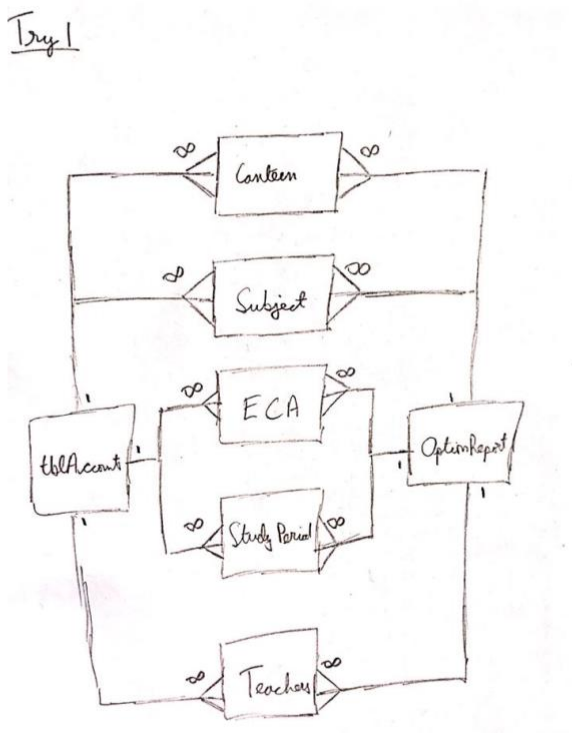


This diagram shows how the accounts are updated.

INTERNAL STRUCTURE

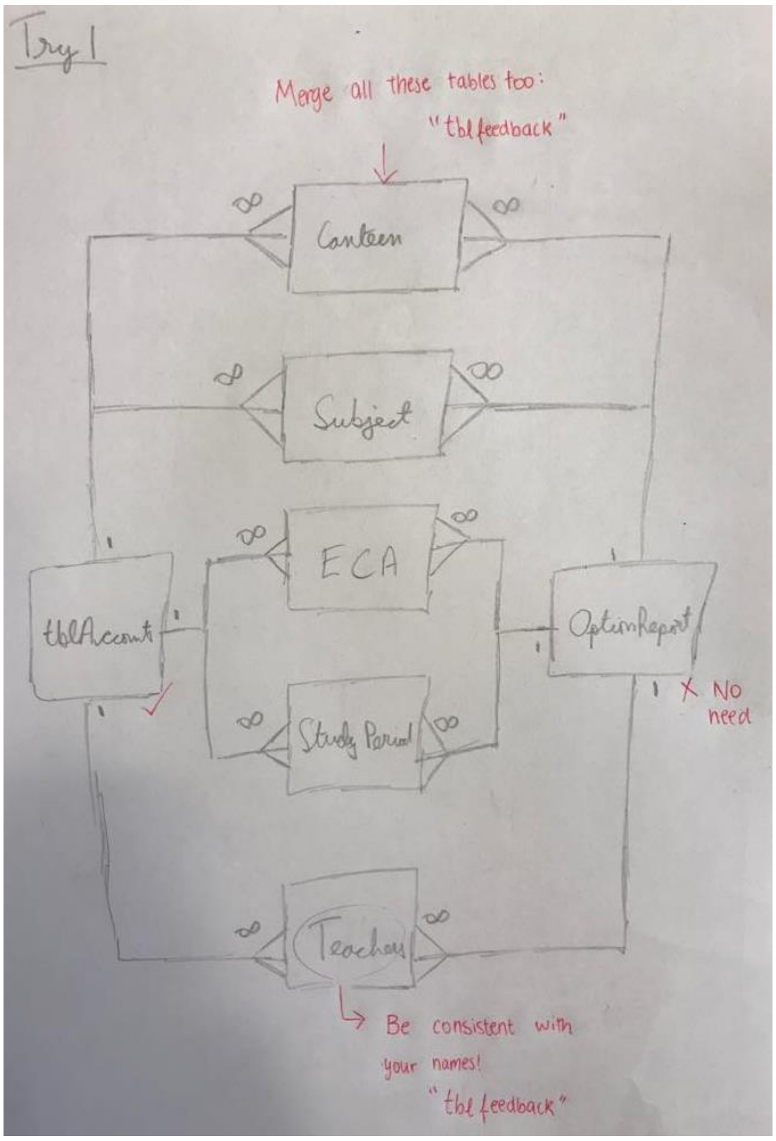
ENTITY RELATIONSHIP DIAGRAM (15/05/18 – 25/05/18)

INITIAL DESIGN



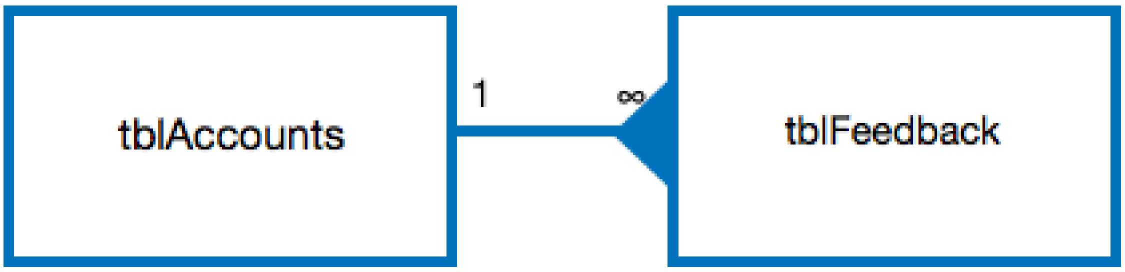
My initial attempt at an entity relationship diagram can be seen on the left. After having a discussion with my advisor (Appendix B) we came up with a new design as can be seen below.

ADVISOR FEEDBACK (APPENDIX B)



Above is a picture of the feedback I received from my advisor. I acted upon this to create the final ERD below.

FINAL ERD



Surprisingly, after discussing with my advisor (Appendix B), I realized that I only needed 2 tables: accounts and feedback. The rest of the functionalities like reports, history, and signups revolve around the use of these two tables only.

DATA DICTIONARIES (01/06/18 – 02/06/18)

tblAccounts

Field Name	Data Type	Default Value	Validation Rule	Extra	Required
ID	int(11)			AUTO_INCREMENT	Yes
UserName	varchar(10)		4 Letters then 2 digits		Yes
Password	varchar(15)		Max 15 characters		Yes
Admin	tinyint(1)	0			Yes
YearGroup	varchar(13)		More than 1, Less than 13		Yes
DateOfCreation	timestamp	CURRENT_TIMESTAMP			Yes
Access_Token	varchar(45)	null			

The table above shows how the user details will be stored. Note that each user will be assigned an "**ID**", which serves as a **unique primary key**. The entry is also **timestamped** to further ensure there is **no redundancy**. Note that each user is given a unique "**Access Token**" too, whilst this will be further explained later, it ensures that the user can only submit their feedback once they have logged in and the database has provided them with the token as a form of **authentication**.

tblFeedback

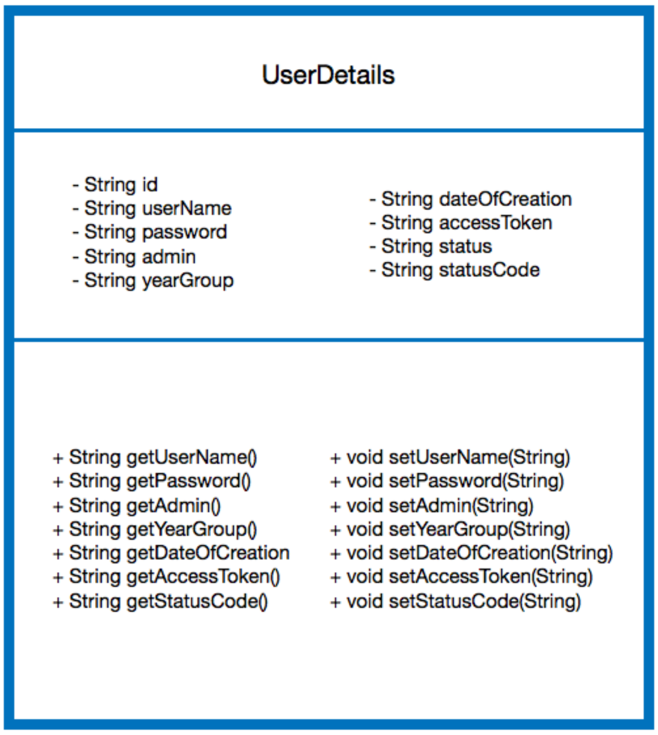
Field Name	Data Type	Default Value	Validation Rule	Extra	Required
ID	int(11)			AUTO_INCREMENT	Yes
CategoryType	varchar(100)				Yes
UserID	int(11)				Yes
Question1	varchar(100)		Value from 0-3		No
Question2	varchar(100)		Value from 0-3		No
Question3	varchar(100)				No
Question4	varchar(100)		Max 100 Characters		No
Question5	varchar(100)		Max 100 Characters		No
Question6	varchar(100)		Max 100 Characters		No
DateCreated	timestamp	CURRENT_TIMESTAMP			Yes

The table above shows how the information for the feedback given will be stored. Once again a **unique primary key**: "**ID**" is used. Note that when a feedback is given, the **category** of the feedback (eg: Canteen/ subject etc.) is stored as well. The **userID** is stored to keep track of **accounts**.

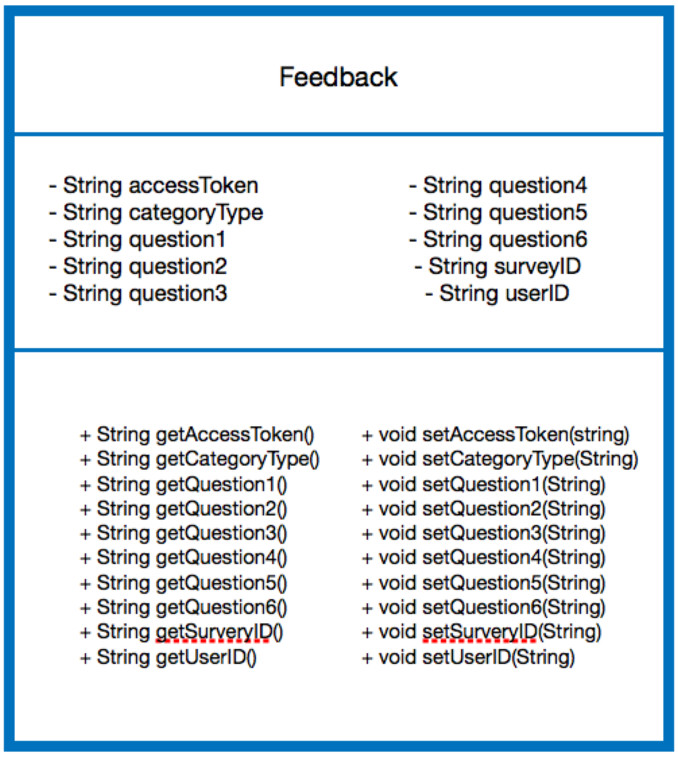
CLASS DIAGRAMS (06/06/18 – 10/06/18)

USERDETAILS

This class stores the user details throughout the time they use the app. The main use of this app is for database querying as I have to pass its member variables as arguments for my query functions.



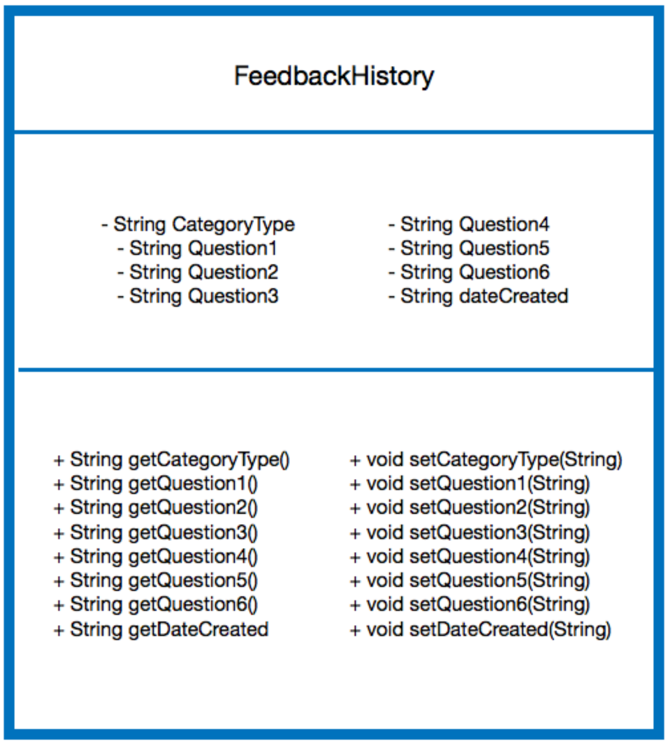
FEEDBACK



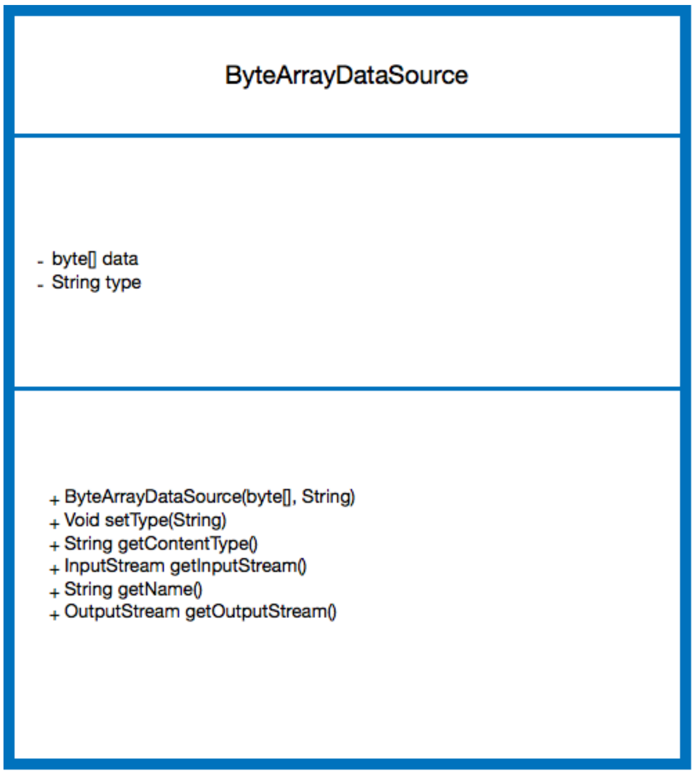
This class stores the feedback of the user BEFORE it is stored in tblfeedback in the database. Note that the accessToken allows authorization for the writing on the online database. A surveyID is also generated to ensure there is no redundancy in the database structure.

FEEDBACKHISTORY

This class stores the feedback of the user BEFORE it is stored in tblfeedback in the database. Note that the accessToken allows authorization for the writing on the online database. A surveyID is also generated to ensure there is no redundancy in the database structure.



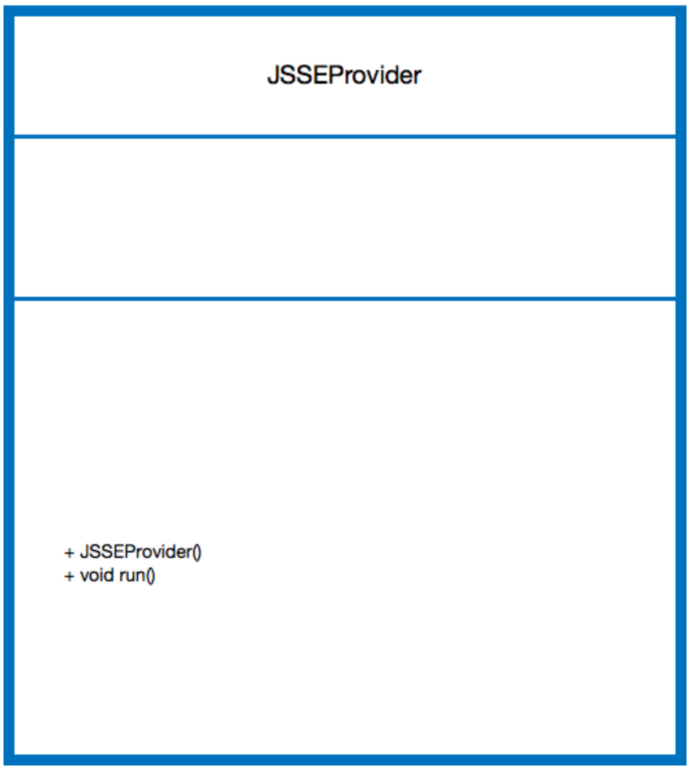
BYTEARRAYDATASOURCE


ByteArrayDataSource
- byte[] data
- String type
+ ByteArrayDataSource(byte[], String)
+ Void setType(String)
+ String getContentType()
+ InputStream getInputStream()
+ String getName()
+ OutputStream getOutputStream()

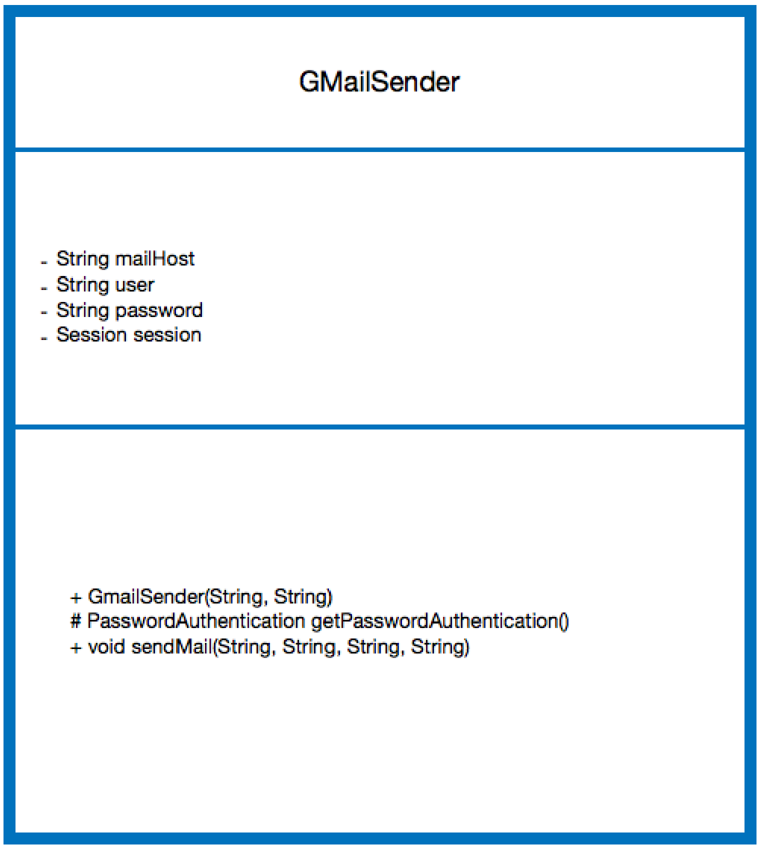
This is a class which helps with sending emails.

JSSEPROVIDER

This is provider class helps with sending emails

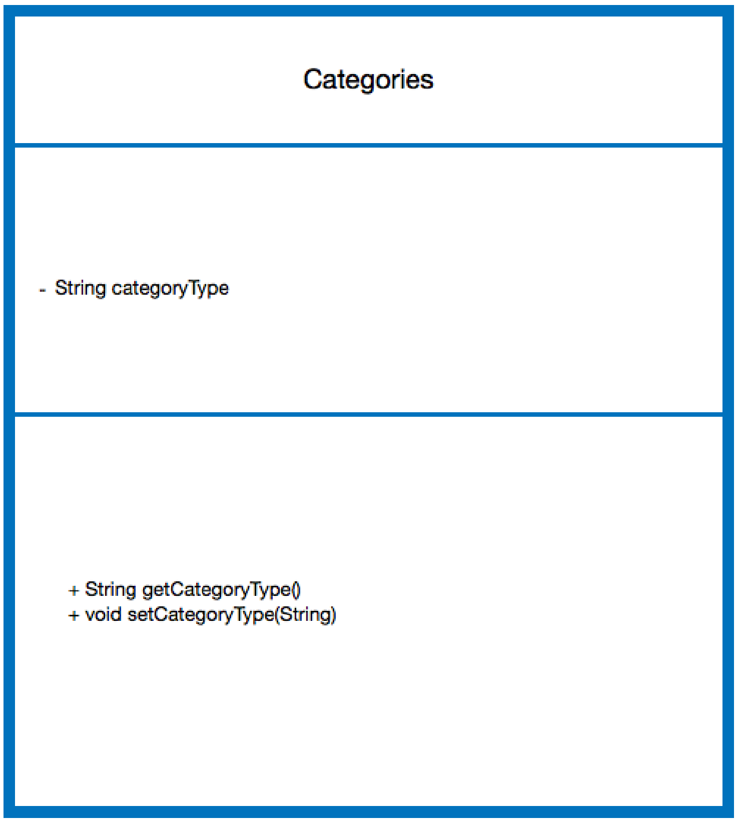


GMAILSENDER


GMailSender
- String mailHost
- String user
- String password
- Session session
+ GmailSender(String, String)
# PasswordAuthentication getPasswordAuthentication()
+ void sendMail(String, String, String, String)

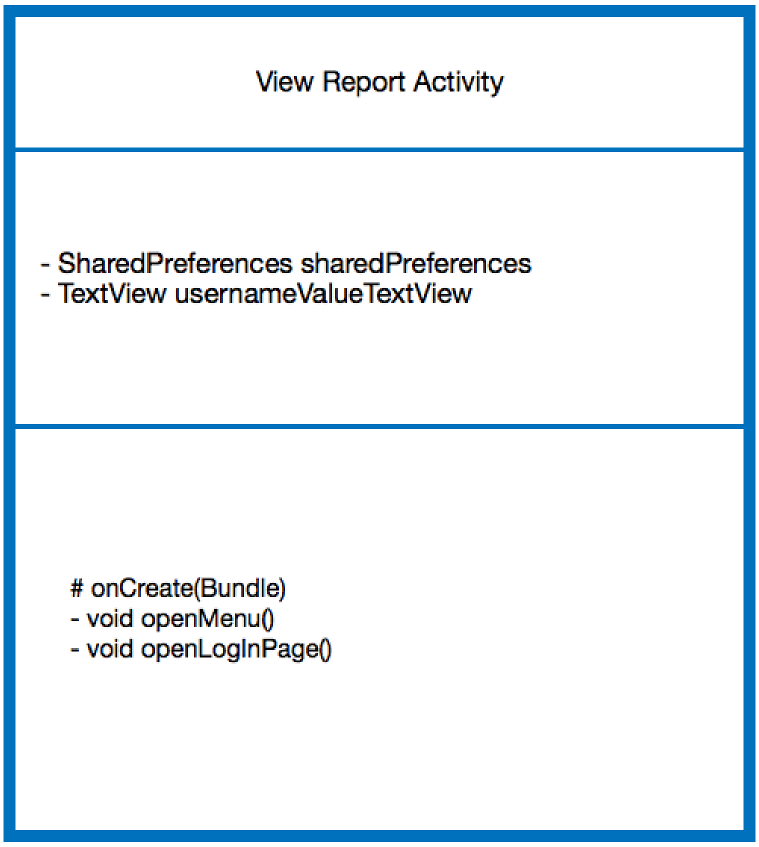
This is the main helper class which aids in sending emails from the sign-up page.

CATEGORIES



This is the helper class for compiling reports.

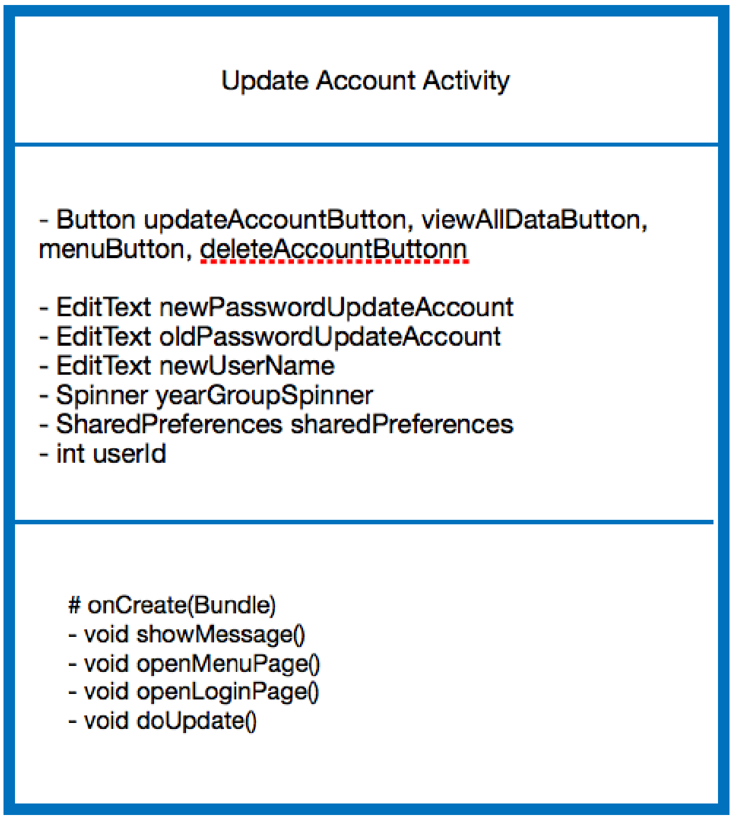
VIEW REPORT ACTIVITY



This is the controller class for viewing reports. The onCreate() function is used to navigate to the pie chart pages.

UPDATE ACCOUNT ACTIVITY

This is the controller class for updating accounts. The relevant information is collected and then sent to the databases using API calls.



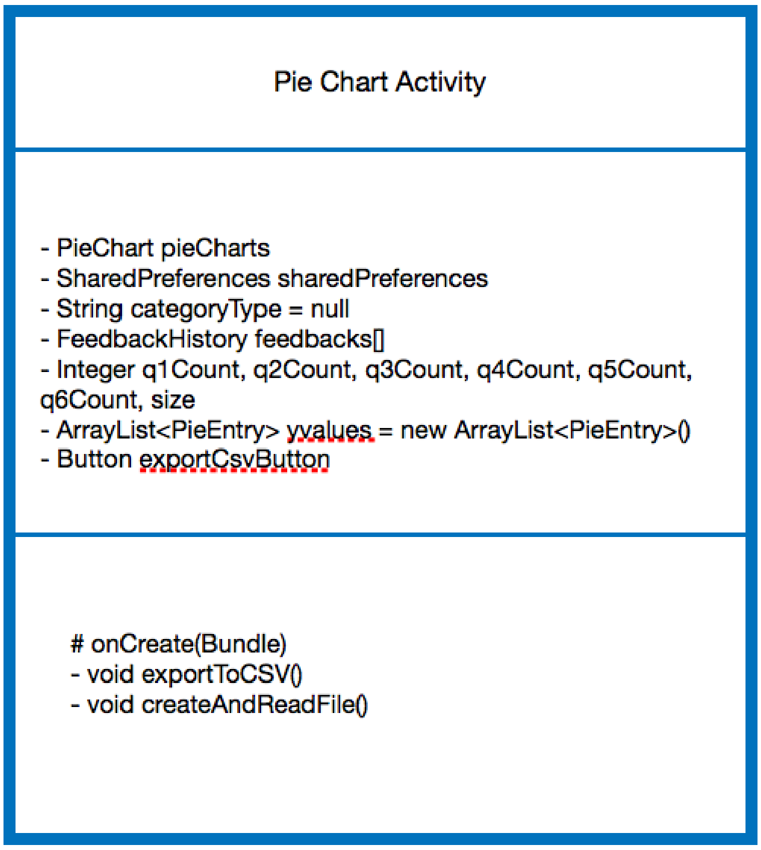
Update Account Activity

- Button updateAccountButton, viewAllDataButton, menuButton, deleteAccountButtonn

- EditText newPasswordUpdateAccount
- EditText oldPasswordUpdateAccount
- EditText newUserName
- Spinner yearGroupSpinner
- SharedPreferences sharedPreferences
- int userId

# onCreate(Bundle)
- void showMessage()
- void openMenuPage()
- void openLoginPage()
- void doUpdate()

PIE CHART ACTIVITY



Pie Chart Activity

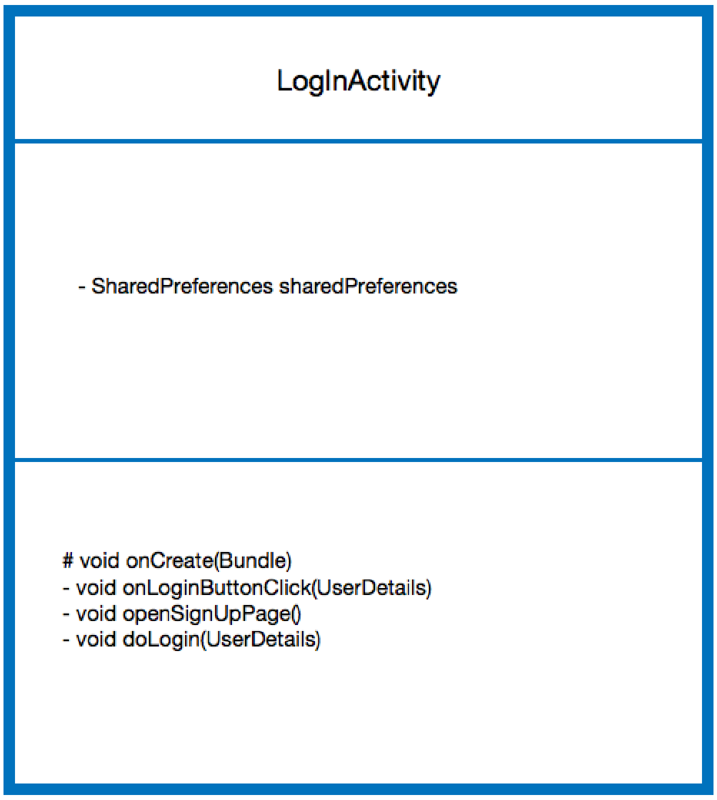
- PieChart pieCharts
- SharedPreferences sharedPreferences
- String categoryType = null
- FeedbackHistory feedbacks[]
- Integer q1Count, q2Count, q3Count, q4Count, q5Count, q6Count, size
- ArrayList<PieEntry> yvalues = new ArrayList<PieEntry>()
- Button exportCsvButton

# onCreate(Bundle)
- void exportToCSV()
- void createAndReadFile()

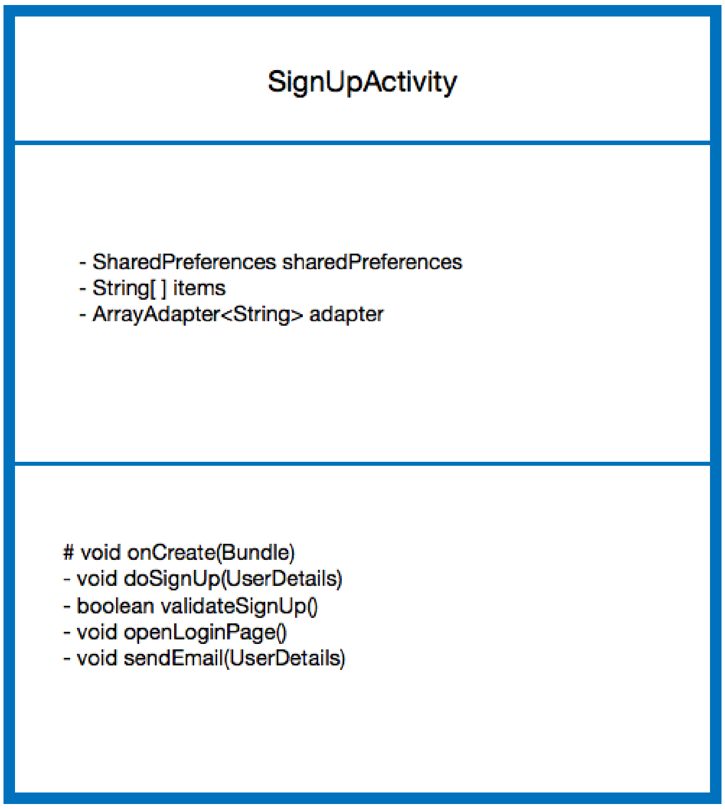
This is the controller class for the Pie Chart

LOGINACTIVITY

This is the controller class for the login page. The use of SharedPreferences stores the user data like username and password directly in the user's phone, saving code later in the program as well because I can directly access such details from their storage. The doLogin() function is where the connection to the database takes place.


LogInActivity
- SharedPreferences sharedPreferences
# void onCreate(Bundle)
- void onLoginButtonClick(UserDetails)
- void openSignUpPage()
- void doLogin(UserDetails)

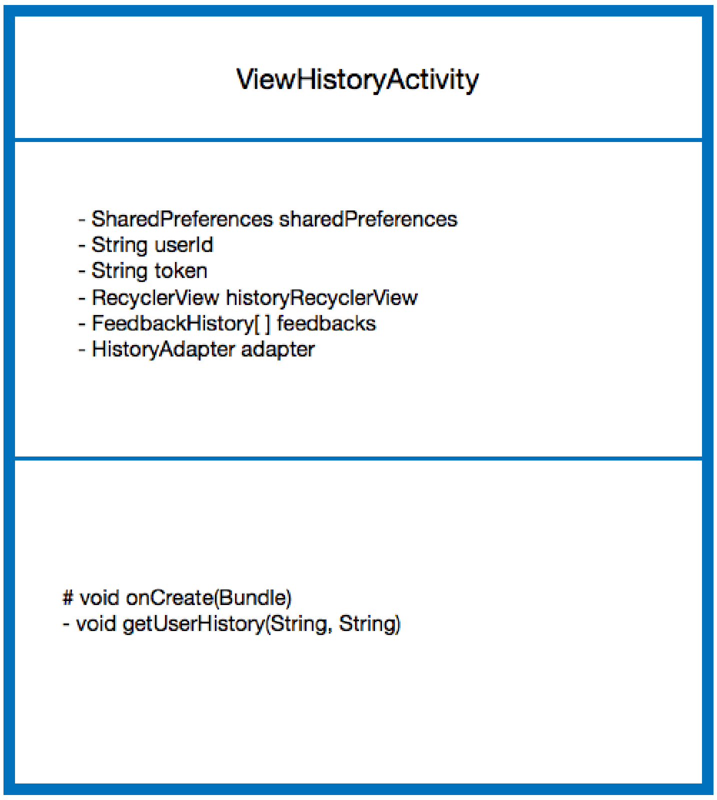
SIGNUPACTIVITY


SignUpActivity
- SharedPreferences sharedPreferences
- String[ ] items
- ArrayAdapter<String> adapter
# void onCreate(Bundle)
- void doSignUp(UserDetails)
- boolean validateSignUp()
- void openLoginPage()
- void sendEmail(UserDetails)

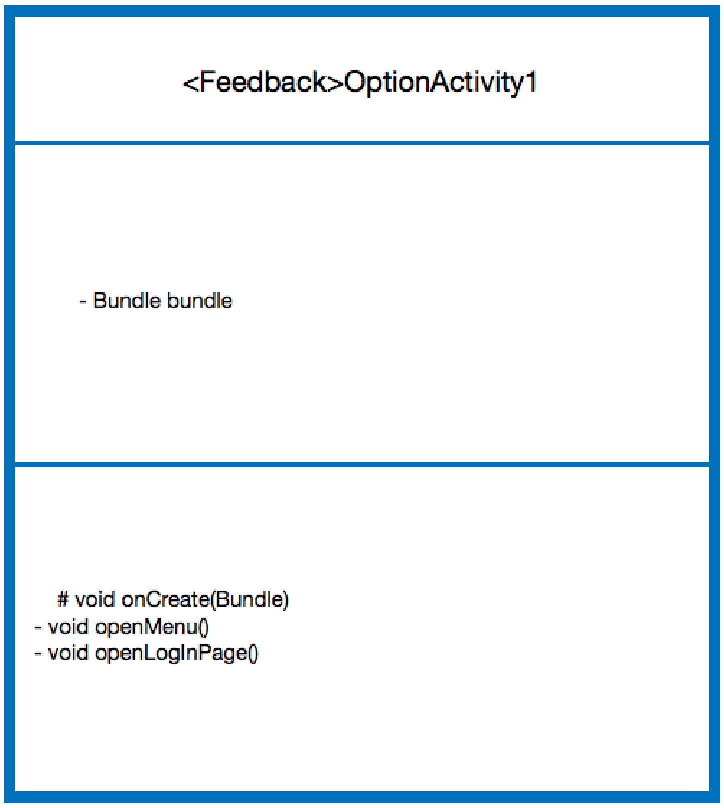
This is the controller class for the Sign-Up page. The doSignUp() function calls the PHP API which then **searches** the database for the username, if none exists then it **creates** a record of that account in tblAccounts .

VIEWHISTORYACTIVITY

This is the controller class for the viewHistory page. The getUserHistory() function **searches** the database for the given userName and then **reads** the database. The API responsible for reading the database then returns an array of FeedbackHistory objects which contain the user's survey answers.


ViewHistoryActivity
- SharedPreferences sharedPreferences
- String userId
- String token
- RecyclerView historyRecyclerView
- FeedbackHistory[ ] feedbacks
- HistoryAdapter adapter
# void onCreate(Bundle)
- void getUserHistory(String, String)

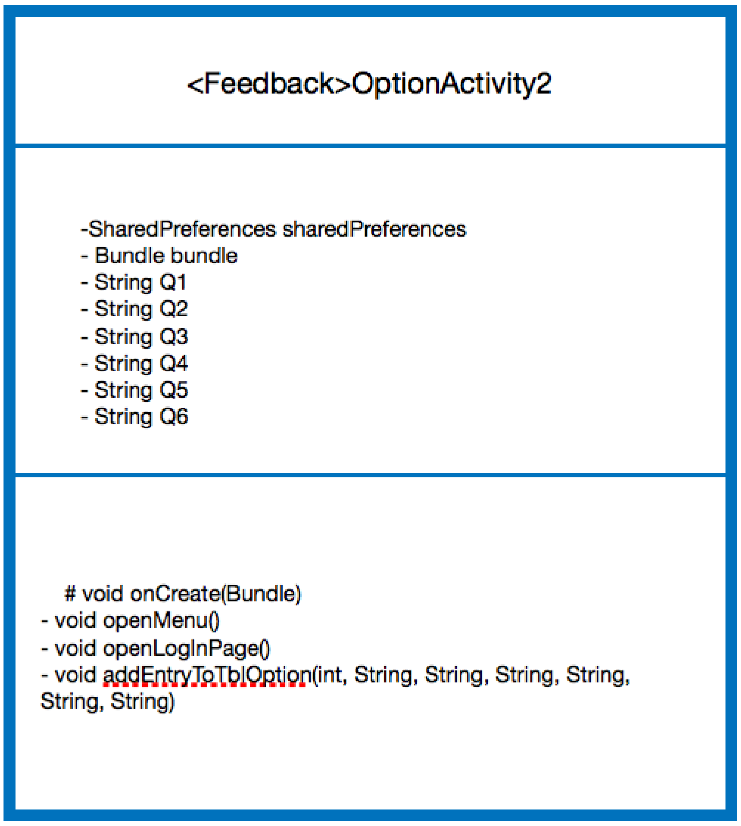
<FEEDBACK>OPTIONACTIVITY1



This is the template of a controller class which applies for the first page of all survey pages. The bundle takes in the data of the current page and is passed on to the next page of the survey.

<FEEDBACK>OPTIONACTIVITY2

This is the template of a controller class which applies for the second page of all survey pages. The bundle passed in from the previous page is deconstructed to get the first page's data. Then all the data is added to the tblFeedback table on the database through the addEntrytoTblOption() function.

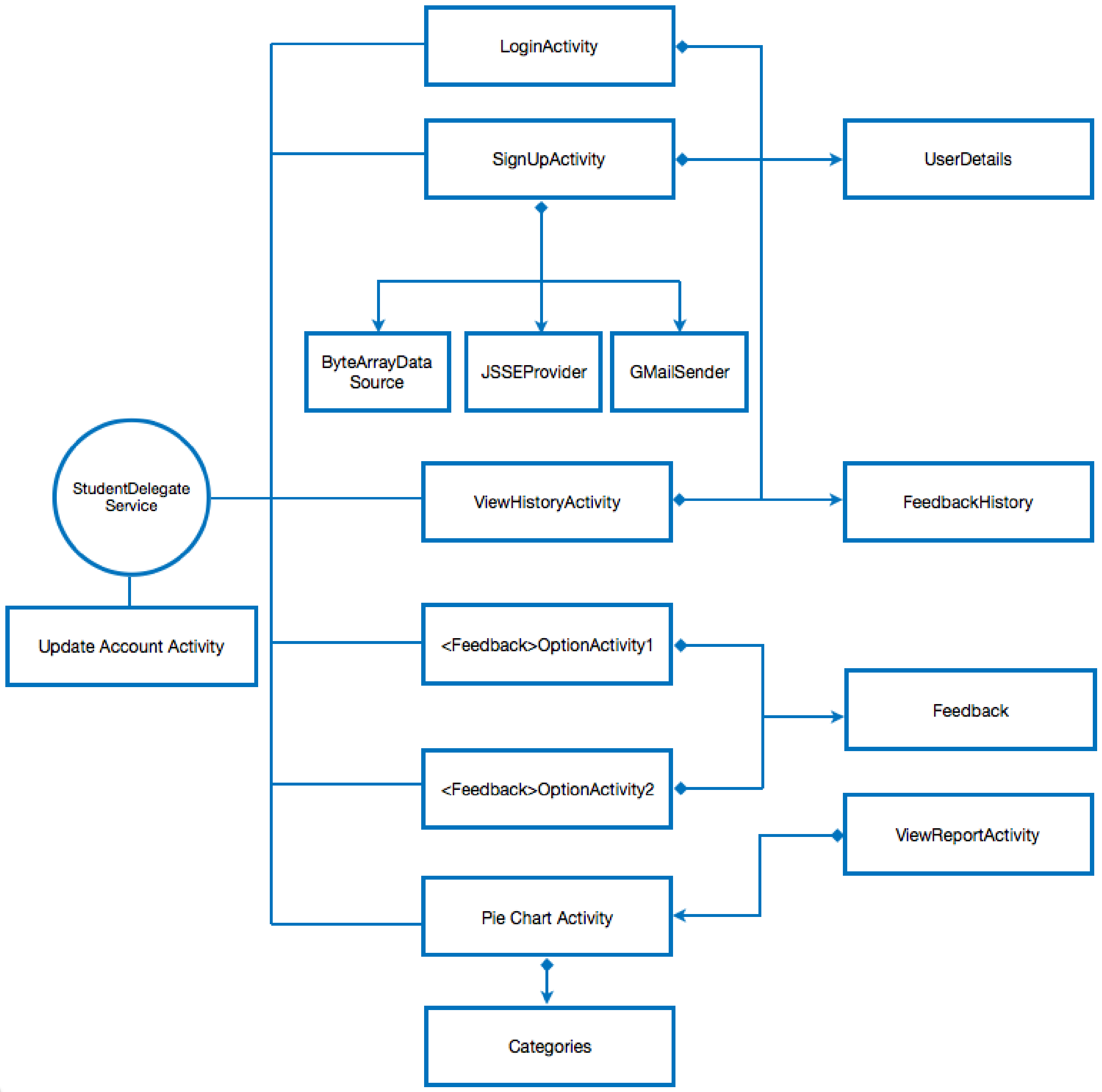


```
<Feedback>OptionActivity2

-SharedPreferences sharedPreferences
- Bundle bundle
- String Q1
- String Q2
- String Q3
- String Q4
- String Q5
- String Q6

# void onCreate(Bundle)
- void openMenu()
- void openLogInPage()
- void addEntryToTblOption(int, String, String, String, String,
String, String)
```

UML DIAGRAM (15/06/18)



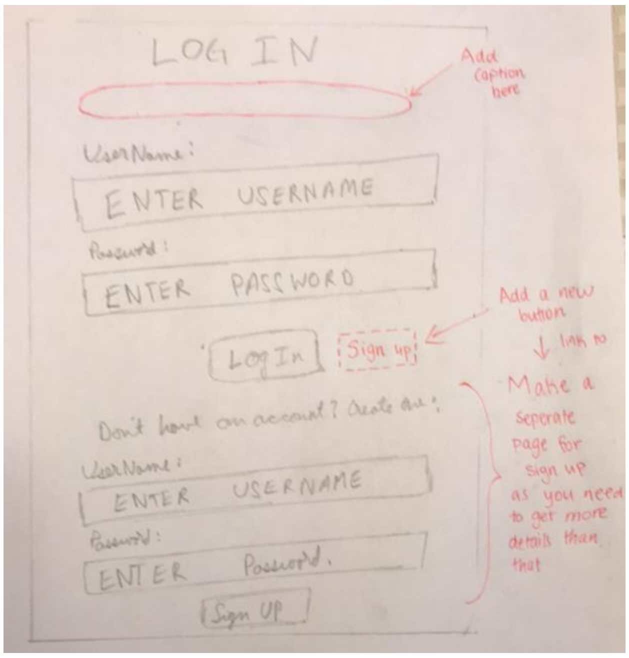
The diagram above shows my UML Diagram which describes how my classes interact with each other. Notice that each “Activity” class above is a controller class. These classes have specific functions which display the designs I made using **FXML files** on the user’s screens. The UML diagram also shows a “StudentDelegatesService” interface. This **interface** contains specific functions which allow me to connect to my **php APIs** which interact with my online database. Hence, this interface was implemented by all my **controller classes** which needed to connect to the **database**.

DESCRIPTION OF SPECIFIC ELEMENTS

FORM PLANS (18/06/18 – 26/08/18)

LOG IN PAGE

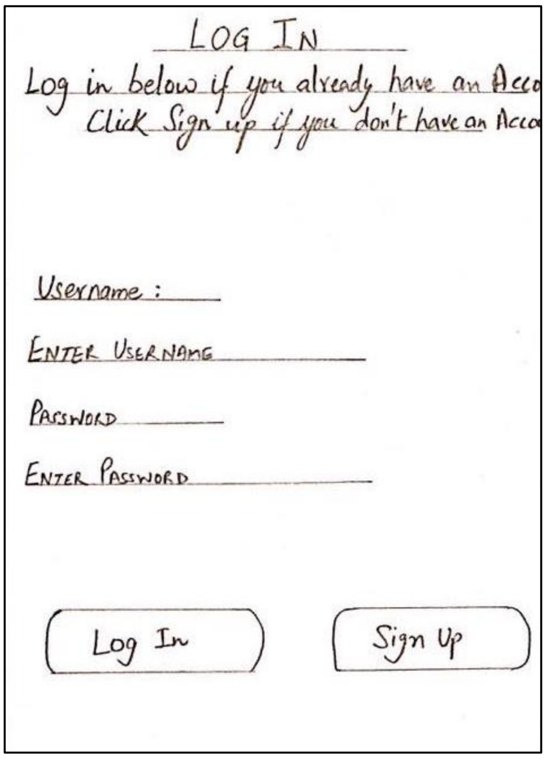
INITIAL DESIGN + CLIENT FEEDBACK



On the left is a picture of my initial "Log In" page with client feedback in red. These changes were made below:

FINAL DESIGN

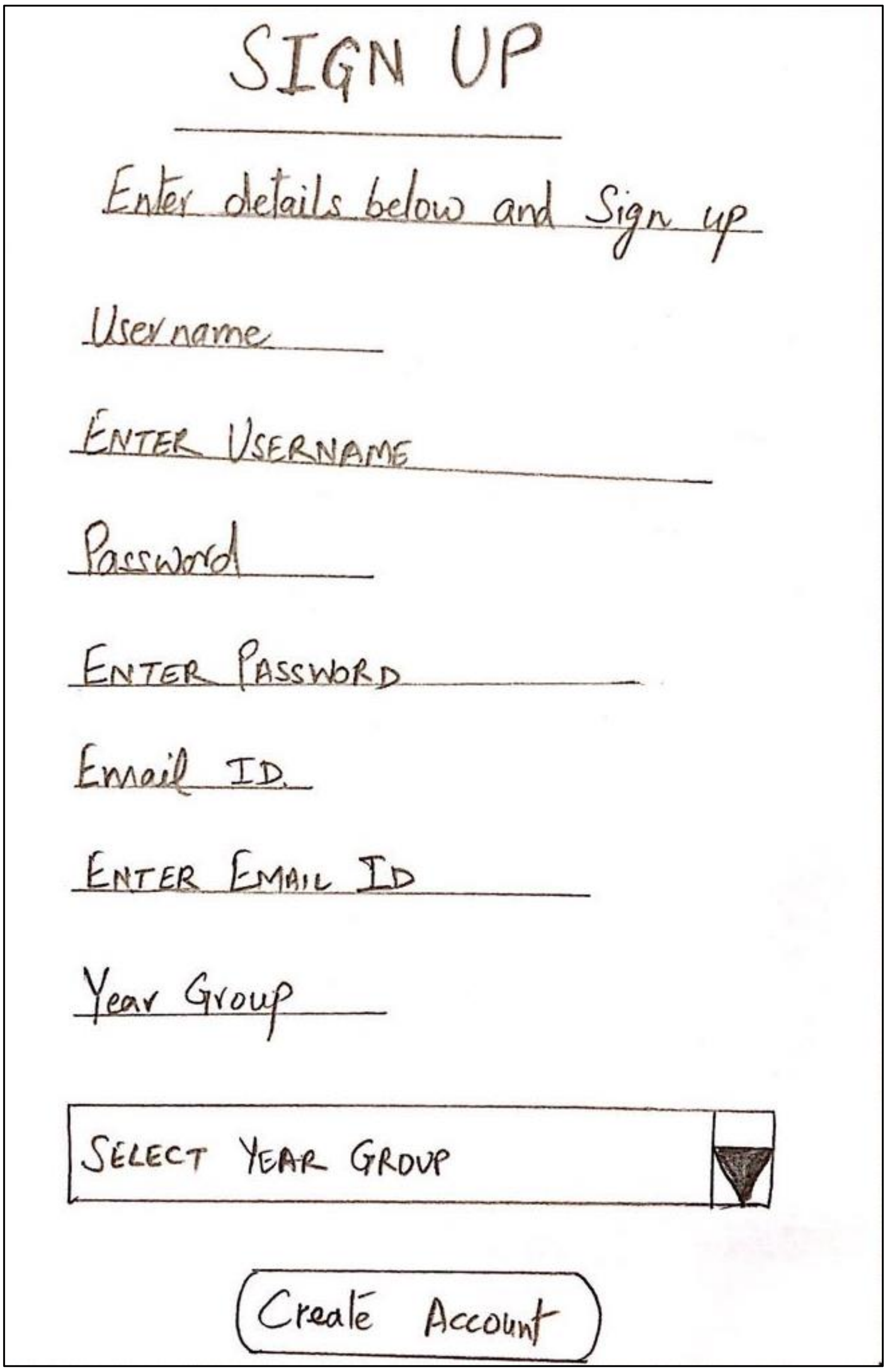
On the right is a picture of my final designs for my Log In Page. All the feedback given to me from my client has been taken into account in this design.



LOG IN
Log in below if you already have an Acco
Click Sign up if you don't have an Acco
Username :
ENTER USERNAME
PASSWORD
ENTER PASSWORD
Log In
Sign Up

SIGN UP PAGE

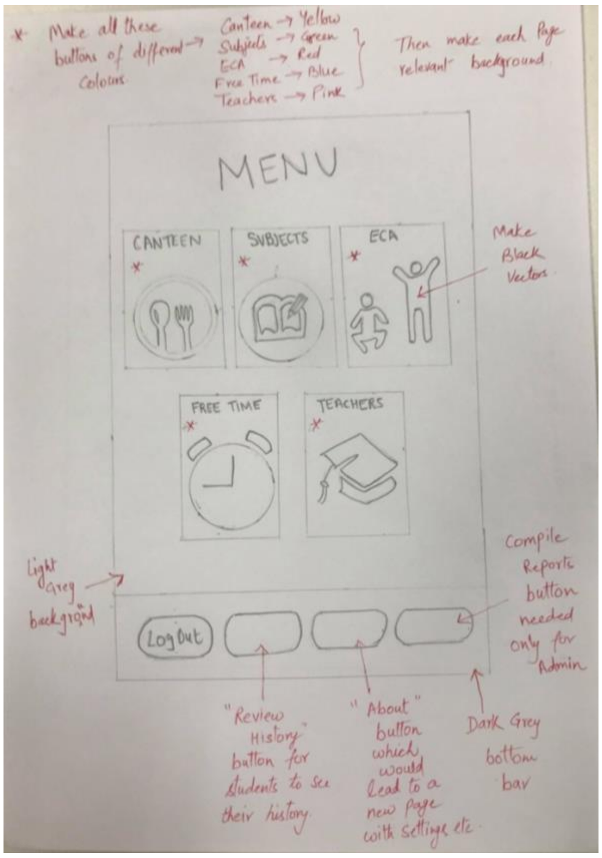
FINAL DESIGNS



The picture above shows my final designs for the sign up page. My client did not give any feedback on the initial designs because he felt it was appropriate without any changes.

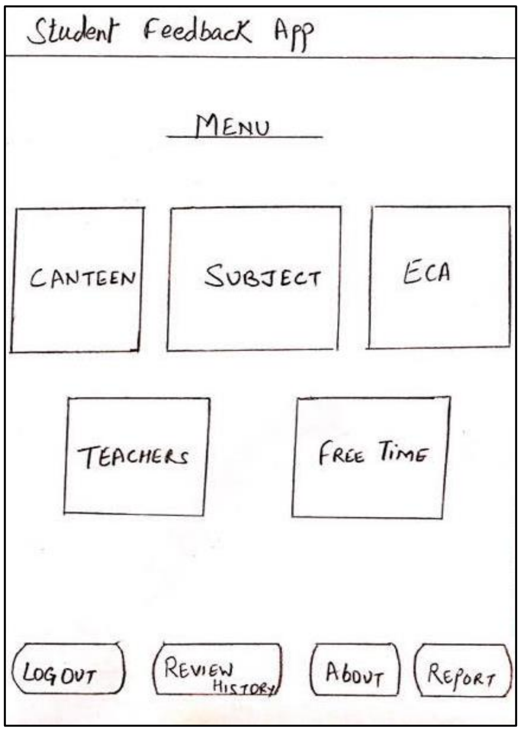
MENU PAGE

INITIAL DESIGN + CLIENT FEEDBACK


* Make all these buttons of different colours
Canteen → Yellow
Subjects → Green
ECA → Red
Free Time → Blue
Teachers → Pink
Then make each Page relevant background
MENU
CANTEEN
SUBJECTS
ECA
FREE TIME
TEACHERS
Make Black Vectors
Light Grey background
Log Out
Compile Reports button needed only for Admin
"Review History" button for students to see their history
"About" button which would lead to a new page with settings etc.
Dark Grey bottom bar

The picture on the left shows the client feedback on my initial designs of the "Menu" page. My client asked me to add in 3 more buttons.

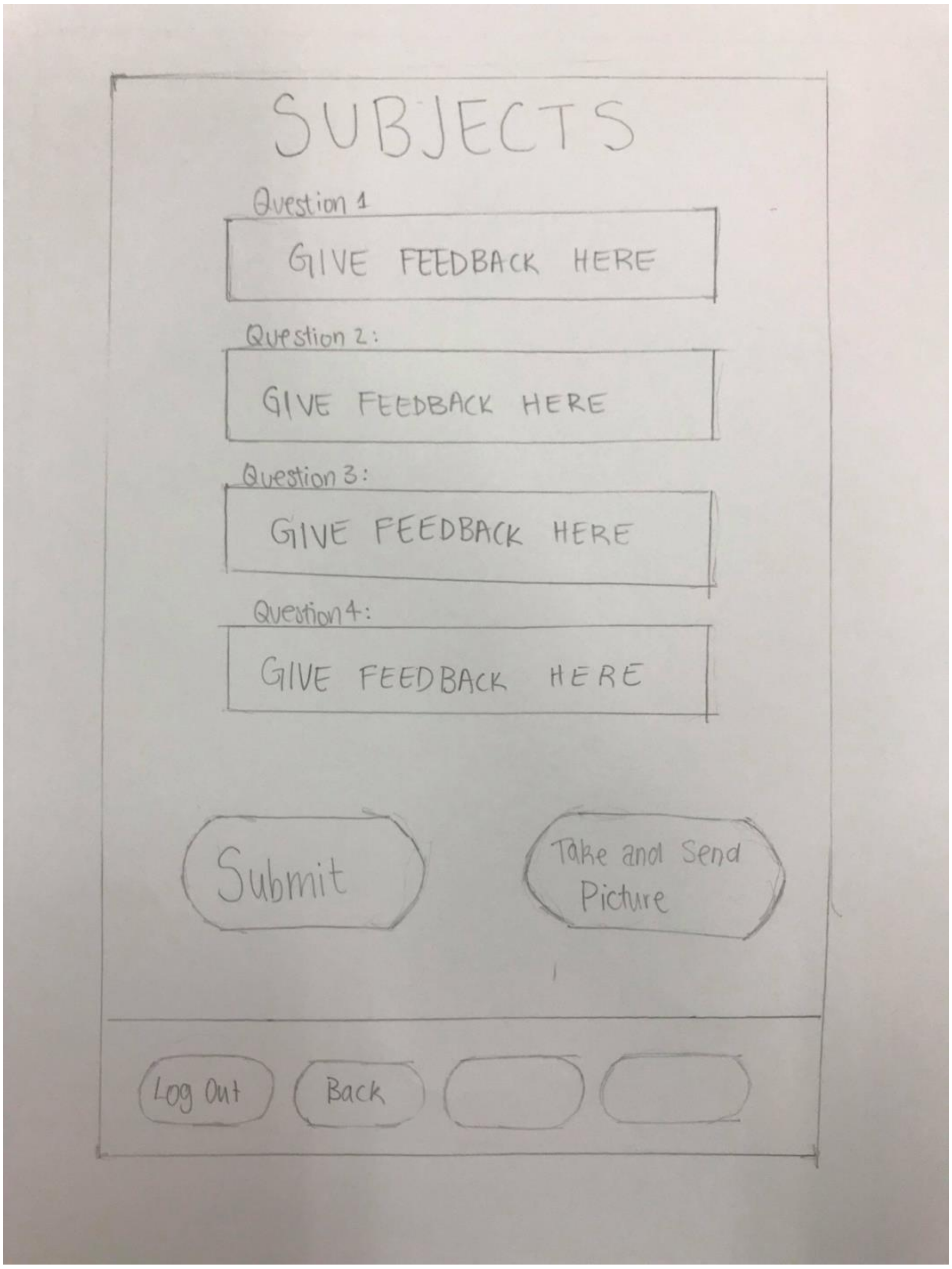
FINAL DESIGN



The screen shot above shows my final designs for the Menu page. For simplicity, I decided that the logos included in the initial designs were not required.

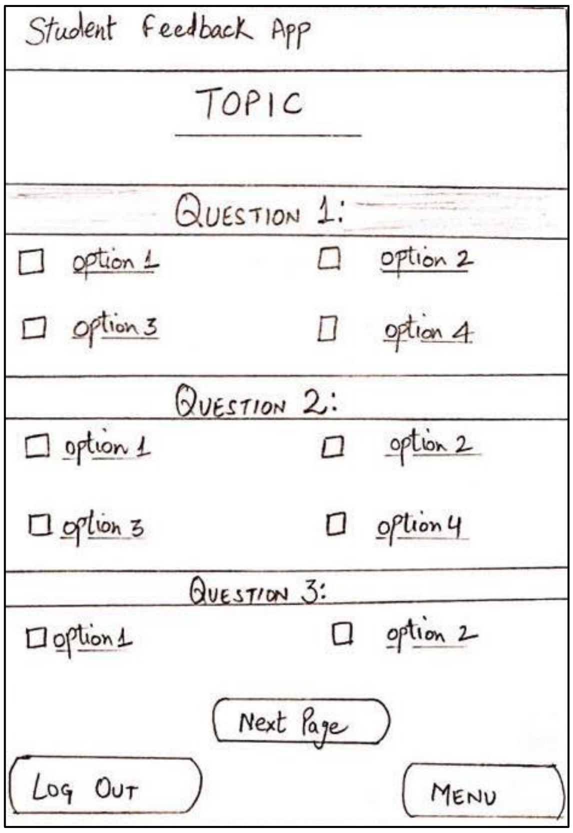
SAMPLE FORM PAGE (THIS APPLIES FOR EACH OPTION PAGE)

INITIAL DESIGN + CLIENT FEEDBACK

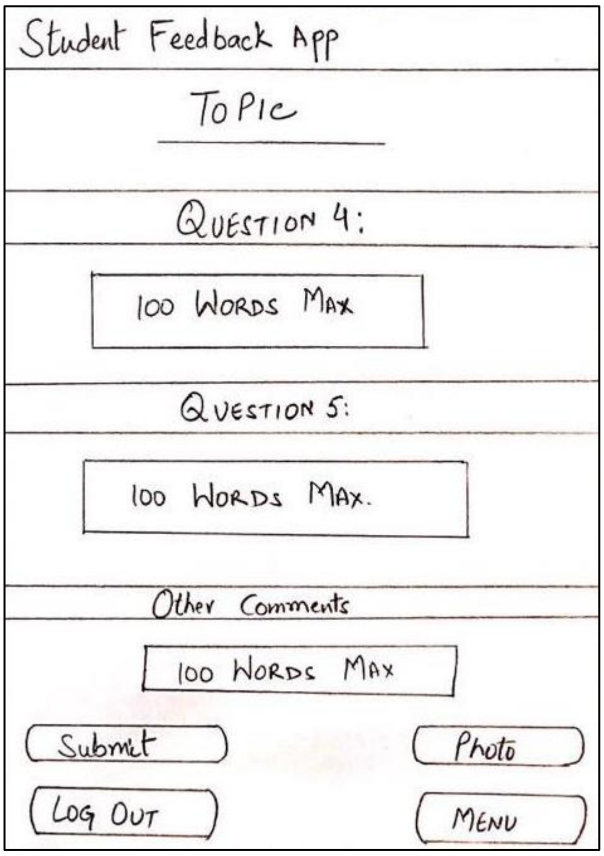


Above is my initial designs for the option forms- the above layout applied to all option forms. However, my client was not happy with only having subjective questions and requested me to change the layout and the questions. This can be seen in (**Appendix B).** I applied this feedback to create the final designs that can be seen below.

FINAL DESIGNS:

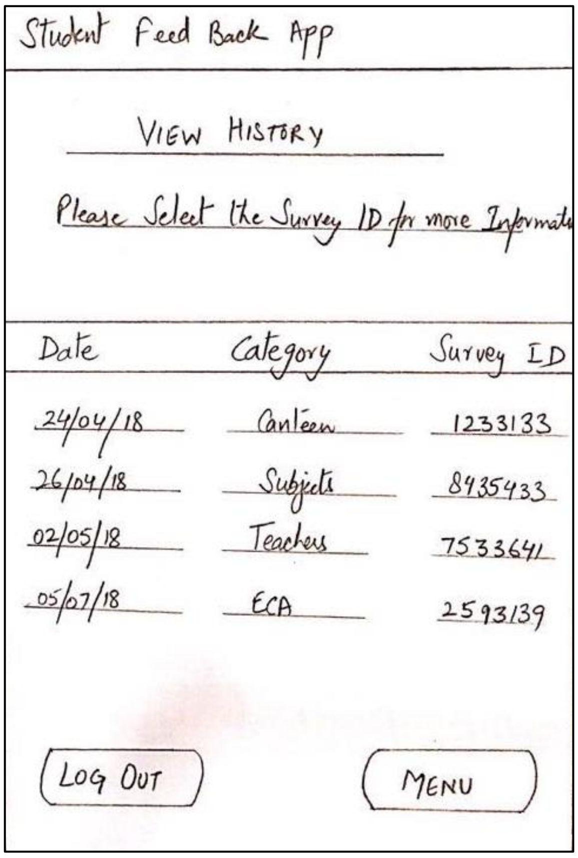


The left is the template for page 1 of each "Option Page". The client asked me to include objective questions to help me compile reports at the end of the process (Appendix B).



On the right is a template for page 2 of each "Option Page". The client asked me to include some subjective questions to help cover the full thoughts of the users (Appendix B).

REVIEW HISTORY PAGE (FINAL DESIGN ONLY)

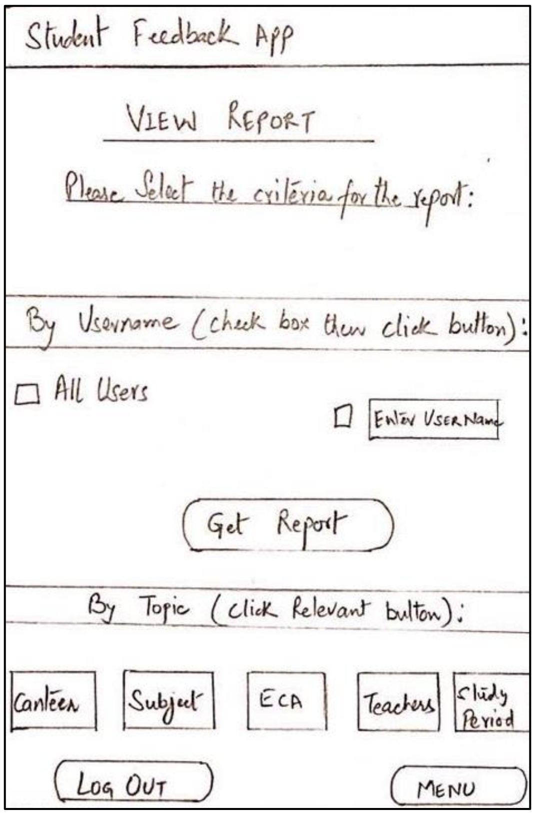


Date	Category	Survey ID
24/04/18	Canteen	1233133
26/04/18	Subjects	8435433
02/05/18	Teachers	7533641
05/07/18	ECA	2593139

Note: as my client did not express any concerns with the design of this page, I did not make any changes from my initial designs

VIEW REPORT PAGE (FINAL DESIGN ONLY)

Note: as my client did not express any concerns with the design of this page, I did not make any changes from my initial designs


Student Feedback App
VIEW REPORT
Please Select the criteria for the report:
By Username (check box then click button):
All Users
Enter UserName
Get Report
By Topic (click Relevant button):
Canteen
Subject
ECA
Teachers
Study Period
LOG OUT
MENU

DESCRIPTION OF NON-STANDARD ALGORITHMS (29/06/18 - 06/07/18)

As I am creating my own android application, I will have to code algorithms which can't be imported from standard libraries:

VALIDATING SIGN UP

Before the user can sign up, my app must check if the username and password are in the correct format:

VALIDATE SIGN UP FUNCTION

```
BOOLEAN validateSignUp()

        INPUT userName
        INPUT password

        IF(userName == “”)
                OUTPUT “Please enter a userName”
        ELSE IF (userName.length != 6)
                OUTPUT “Please ensure username is 6 characters long”
        ELSE
                String d = userName.substring(userName.length() – 2, userName.length())
                String c = userName.substring(0, userName.length() – 2)

                TRY
                        PARSE INT(d) //Checks if d contains integers; if not, error is caught later
                        RETURN True
                CATCH
                        OUTPUT “Ensure the last 2 characters are your graduation date”
                        RETURN False

                IF( c MATCHES “*\\d.*”) //Checks if last 2 characters are not integers
                        OUTPUT “Please enter a valid username
                        RETURN False

        IF( password == “”)
                OUTPUT “Please enter a password”
                RETURN False
        ELSE IF (password.length() > 15)
                OUTPUT “Passwords can only be 15 characters maximum”
                RETURN False
```

The function above is responsible for ensuring that the entered username and password are of the valid format. Note: the valid format can be seen in the “Data Dictionaries” part of Section B above.

ADDING FEEDBACK TO DATABASE

Adding comments to my database will not be a standard task because I am interacting with an online database. Hence, I expect to create connection APIs which can help communicate with my application and database. Moreover, as my input data is very specialised to my application, no standard code would be able to add records to the database for me. I planned the adding process bellow:

SURVEY PAGE ON APP

```
        INPUT question 1
        INPUT question 2
        INPUT question 3
        INPUT question 4
        INPUT question 5
        INPUT question 6
        AddFeedbackToDatabase(question 1, question 2, question 3, question 4, question 5, question 6)
```

The pseudocode above shows how the user inputs will be gathered. The AddFeedbackToDatabase() function will be responsible to connect with the API.

ADDFEEDBACKTODATABASE FUNCTION

```
AddFeedbackToDatabase(question 1, question 2, question 3, question 4, question 5, question 6)
        String Category // Must store category of the survey
        String Tag //To identify which part of the API to access
        String Response = CALL API //The API Call will return a response of “Success” or “Failure”
        IF (API Response == Success)
                OUTPUT “Data added successfully”
        ELSE
                OUTPUT “Something went wrong”
```

The pseudocode above shows where the API is called and how the API response will be handled. The API is planned on the next page.

API ADD FUNCTION

```
        DECODE data passed in to correct format
        OPEN CONNECTION TO DATABASE
        SQL INSERT data into Database
        IF (INSERT == Success)
                Return Response = Success
        ELSE
                RETURN Response = Failure
```

The pseudocode above plans how the “Add function” in the connection AP will be code

VIEW HISTORY

Viewing my history will also not be standard algorithm as I will have to **SEARCH** my database for the given username, then **READ** their respective record, and finally **DISPLAY** their records in a user friendly manner

GETUSERHISTORY FUNCTION

This function will oversee collecting relevant data, connecting with the API, then displaying the result:

```
getUserHistory(String userID, String Token)

        String Tag //To identify which part of the API to access

        String Response = CALL API //The API Call will return a response code of "Success" or "Failure"

        IF (API Response.code == Success)
                ARRAYLIST<FEEDBACKHISTORY> = Response //The returned data will have to be stored in an array
list
                OUTPUT ARRAYLIST<FEEDBACKHISTORY> //Need to display this on the screen
        ELSE
                OUTPUT "Something went wrong"
```

API GET HISTORY FUNCTION

```
        OPEN CONNECTION TO DATABASE

        Verified = SEARCH tblAccounts FOR Token //Searches the Accounts table to see if Token matches
                                        stored token, stores true if given access token
                                        matches stored access token

        IF(Verified == True) //Checks if the request for access is verified
                History = SEARCH tblOptionFeedback FOR USERID // Returns all records from the
                                                        particular UserID
                RETURN History
        ELSE
                RETURN NULL
```

The pseudocode above shows how the API should work. Algorithmically, the function first verifies if the user is allowed to query the database by checking if the given access token matches the stored access token; if it does, then the user has authority to search. Next, the relevant table is searched by the UserID and all relevant records are returned to the mobile application.

LOG IN FUNCTION

The Log In function of the application will also be non-standard as it must be custom fit to the application:

DO LOG IN FUNCTION

This function will be responsible for handling the API calls and responses from within the mobile application:

```
doLogIn(UserDetails user) //I will need to use the UserDetails class I designed above

    String Tag //To identify which part of the API to access

    String Response = CALL API //The API Call will return a response code of “Success” or “Failure”

    IF (API Response.code == Success)
        STORE UserID //This will be stored in the hardware storage of the user’s phone
        STORE UserName //This will be stored in the hardware storage of the user’s phone
        STORE Password //This will be stored in the hardware storage of the user’s phone
        OPEN MENU PAGE //Open the next page of the App
    ELSE
        OUTPUT “Could not log in”
```

API LOG IN FUNCTION

```
    OPEN CONNECTION TO DATABASE

    BOOLEAN Verified = False

    userPasswordCheck = SEARCH tblAccounts FOR UserID //Returns the stored password

    IF( userPasswordCheck == enteredPassword)
        Verified == true

    IF(Verified == True) //Checks if the entered password = stored password
        RETURN “Success”
    ELSE
        RETURN “Failure”
```

The pseudocode above is a plan of how the back-end log in feature will work. Note that first the entered password by the user will have to be checked against the stored password from when the account was created. If they match, then the account can be used to log in the app.

SEND EMAIL

Sending an email on an android application can be quite tricky, in particular as my client requires a specific custom-made email feature. I have planned how this will work below:

SEND EMAIL FUNCTION

This function will generate the email content and then send it using custom made helper classes:

```
sendEmail(UserDetails user) //I will need to use the UserDetails class I designed above

        FINAL String senderEmail //This will have the email I will use to send emails from
        FINAL String INPUT receiverEmail //Ask the user for the email ID they want to receive the email to.
        FINAL String subject
        FINAL String body
        String userName = user.getUserName()
        String userPassword = user.getPassword()

        NEW THREAD //Multithreading should be used to prevent errors from crashing the application
                TRY
                        SEND EMAIL (senderEmail, receiverEmail, subject, body)
                        OUTPUT "EMAIL SENT SUCCESSFULLY"
                CATCH
                        OUTPUT "Email could not be sent"
        START //Start the thread
```

I anticipate the need for ***multithreading*** in my application because I want the sending email process to be separate from my main app workflow, in case any connection errors are caused which may originally crash the app. The email will also have to be specifically generated using the entered fields. The body will also need to be custom generated to include the log in details of each user.

GENERATE REPORTS

Generating reports will prove to be a challenge as I will have to SEARCH the database, READ the records, ANALYSE the results, and COMPILE a report. The functions plan how this will be done.

COMPILE REPORT FUNCTION

```
compileReport()

        String Tag //To identify which part of the API to access
        String category //To identify which category the report is to be compiled for

        String Response = CALL API //The API Call will return a response code of "Success" or "Failure"

        IF (API Response.code == Success)
                ARRAYLIST<FEEDBACKHISTORY> = Response //The returned data will have to be stored in an array
                analyzeData(Response)
        ELSE
                OUTPUT "Could not get data"
```

The function above calls the API and handles the response. Note, the response will be in the form of an arraylist of type FeedbackHistory. Hence, this is stored in the ARRAYLIST<FEEDBACKHISTORY>.

ANALYZEDATA FUNCTION

```
analyzeData(ARRAYLIST<FEEDBACKHISTORY> response)
        INTEGER Q1A, Q1B, Q1C, Q1D
        INTEGER Q2A, Q2B, Q2C, Q2D
        INTEGER Q3A, Q3B, Q3C, Q3D

        ARRAYLIST<FEEDBACKHISTORY> feedbackHistoryArrayList

        FOR (feedbackHistory feedback: response)
                SWITCH (response.getQ1())
                        CASE "a":
                                Q1A ++
                                BREAK
                        CASE "b":
                                Q1B ++
                                BREAK
                        CASE "c":
                                Q1C ++
                                BREAK
                        CASE "d":
                                Q1D ++
                                BREAK
                        DEFAULT
                                OUTPUT "Error in getting feedback"
                                BREAK
                SWITCH (response.getQ2())
```

This function carries on the next page.

```
        SWITCH (response.getQ2())
            CASE “a”:
                Q2A ++
                BREAK
            CASE “b”:
                Q2B ++
                BREAK
            CASE “c”:
                Q2C ++
                BREAK
            CASE “d”:
                Q2D ++
                BREAK
            DEFAULT
                OUTPUT “Error in getting feedback”
                BREAK
        SWITCH (response.getQ3())
            CASE “a”:
                Q3A ++
                BREAK
            CASE “b”:
                Q3B ++
                BREAK
            CASE “c”:
                Q3C ++
                BREAK
            CASE “d”:
                Q3D ++
                BREAK
            DEFAULT
                OUTPUT “Error in getting feedback”
                BREAK
    NEXT

    DISPLAY PIECHART
    EXPORT CSV FILE
    STORE CSV FILE
```

In this function, I will need to READ the API response. Then, I need to break down the response and store the number of times each option is selected for the questions. Using the FOR loop and SWITCH CASE statements, I plan to increment variables which will hold my data. Lastly, using the collected data, I will need to make charts and compile a PDF report.

TEST PLAN (10/07/18 – 11/07/18)

Test Number	Details (Taken from Section A)	Test Case Data	Expected Results
1	**<u>Success Criteria 1:</u>** “A feature that allows users to *register* in the app…” Check if the PHP API can store the user details in the database	**<u>Normal Data:</u>** username: “hoph19” Password: “password123” Email ID: “email@email.com” Year Group: 13	Message: “User registered successfully"
		<u>Abnormal Data:</u> Username: “anme19” Password: “” Email ID: “dhdhd@mails.com” Year Group: 13	Message: "Please enter a valid username”
			Message: “Please enter a password”
		<u>Extreme Data:</u> username: “abab1234” Password: “123456789101213” Email ID: “email@email.com” Year Group: 13	Message: “Username should be 6 characters only”
			Message: “Password should be less than 15 characters”
2	**<u>Success Criteria 2:</u>** “A feature which allows users to *log in* the app…” Check if the PHP API can log in the app using a given username	**<u>Normal Data:</u>** username: “besa19” Password: “12345678”	Open Menu Screen
		<u>Abnormal Data:</u> username: “ab1e3f” Password: “”	Message: “Please check your log in details”
		<u>Extreme Data:</u> username: “abab1234” Password: “123456789101213”	Message: “Please check your log in details”
3	**<u>Success Criteria 3:</u>** “A clear and *structured database*….” Check if the database structure is the most efficient it can be.	**<u>Qualitative data:</u>** This was tested by having numerous conversations with my advisor regarding the database design (Appendix B)	N/A. After the advisor comments on my initial data dictionaries and ERD, I modified my structure. After this, I expect my app to be fully functional.
4	**<u>Success Criteria 4:</u>** “A feature that allows students to add records to the database when they send feedback”	**<u>Normal Data:</u>** <u>Access Token:</u> auto generated <u>userID:</u> auto generated <u>Q4 – Q6:</u> Strings <u>Category Type:</u> String	Message: “Feedback saved successfully”

	Check if the PHP API can add data to the feedback table on the database	**Abnormal Data:** N/A	N/A
		Extreme Data: Q4 – Q6: > 100 characters	Message: “Something went wrong. Please try again”
5	**Success Criteria 5:** “A feature which allows users to view their previous feedback entries” Check if the PHP API can read the database and if the app can display the data properly	**Normal Data:** Access Token: auto generated userID: auto generated	History Page with relevant surveys displayed
		Abnormal Data: N/A	N/A
		Extreme Data: N/A	N/A
6	**Success Criteria 6:** “Ensure a user-friendly user interface is made, custom fit to my client’s needs” Check if the designs suite the client’s needs and allow for full functionality of the app	**Qualitative data:** This was tested by having numerous conversations with my client regarding the database design and getting his feedback on how to improve the designs according to his needs. (Appendix B)	N/A. After getting feedback from my client on the app design, I expect him to be happy with the designs and for the application to be fully functional
7	**Success Criteria 7:** “A feature which allows my client to generate pie charts and bar charts…” Check if the PHP API can read the database, then if the app can collect the queried data and present it as charts	**Normal Data:** Access Token: auto generated userID: auto generated Category: String	Message: Charts created successfully.
		Abnormal Data: N/A	
		Extreme Data: N/A	
8	**Success Criteria 8:** “A feature which sends my users an email…” Check if the JavaMail API can send an email with the respective log in details of the user	**Normal Data:** Receiver: “*@*” Sender: “*@*” Password: “*” Subject: “*” Body: “*”	Message: Email sent successfully
		Abnormal Data: Receiver: “” Sender: “” Password: “” Subject: “” Body: “”	Message: Please input the all the required details before sending the email
		Extreme Data: N/A	N/A

9	<u>**Success Criteria 9:**</u> " A feature that **exports** the **database** to **Excel sheets**…" Check if the PHP API can read the database and pass relevant data to the app, which then needs to export the data to a CSV file to be read using excel.	<u>**Normal Data:**</u> Username: "suko19" Data: bundle	CSV File created and stored on the client's phone
		<u>**Abnormal Data:**</u> Username: "dhmi19" Data: bundle	"Report" button disabled
		<u>**Extreme Data:**</u> N/A	N/A