**Budget Management**

**Design and Planning Document**

Authors: Anna Kim, Ali Zaidi, Na Li, Jack Long, Joon Lim, Andy Boho

*<2020*-*02*-*28>*, version 1.3: Finalized Document

[Document Revision History](#_y9dug76i15t8) **2**

[System Architecture](#_xwk3n6kz4ul8) **2**

[**Overview:**](#_dy7iqicg8o8s) **2**

[System Components:](#_8wysnyu2x8w5) 3

[System Design (MVC Design):](#_fbty74aiekty) 4

[Model + View + Controller Components Summary:](#_83pam4dmb1ok) 4

[Design Risks](#_dn51xbhqczp9) 5

[Design Details](#_pz5jbaryy9yo) **6**

[Activities](#_c99c0vvykm6x) 7

[Descriptions of each class, method, and attribute](#_jk5ump5tez0w) 12

[Connector Class](#_mm214a3byzty) 13

[Login Class](#_ihy90tbvxmj4) 14

[CreateUser Class](#_s7ki8klemf3x) 16

[Home Class](#_kgyzn4yiu8yu) 18

[Settings Class](#_sahzsxykn37k) 20

[Report class](#_9zuun260l5yx) 24

[Calculator Class](#_8nptwutitckf) 26

[Goals Class](#_ye46bjkniy83) 27

[Categories Class](#_2wmahh3x2qjt) 29

[Navigation (Menu bar)](#_x1oo63ucpzfz) 31

[Database, tables, and fields](#_kyrrg6pw7kpn) 31

[ER diagram:](#_fjrdsouo6rga) 31

[Entity sets:](#_wn3v1fv899xi) 32

[Relational Model Schemas:](#_3oy5tbxoc7rn) 32

[Implementation Plan](#_o2s2rlutt8rk) **33**

[Testing Plan](#_msmeq1nldaep) **37**

[Unit testing](#_jg03dl6tf554) 37

[Integration testing](#_wpwqt0sa8w2g) 37

[System testing](#_dzm9picj2go2) 37

[Performance testing](#_q9zk9tsrxlg5) 37

[Compatibility testing](#_wc6vehf210wl) 38

[Beta testing](#_cp2se4wuzhi6) 38

[Regression testing](#_55p7cgy74t56) 38

# Document Revision History

Rev. 1.0 <2020-02-22>: Initial version

Rev. 1.1 <2020-02-25>: Added system architecture and design details

Rev. 1.2 *<2020*-*02*-*27>*: Added implementation plan and testing plan

Rev 1.3 *<2020*-*02*-*28>*: Added UML diagrams, Finalized Document

# 

# System Architecture

## Overview:

We will be using an adapted Model View Controller (MVC) software design for our Android application. Android does not strictly adhere to MVC, as there may exist multiple views per controller, and the controllers may not be aware of what occurs in the views. In a true MVC, the model, view, and controller are separate testable modules. In Android, certain aspects of the software system may play more than one role at any given time, but what is important is that we use abstraction to show that there is a higher level separation that allows us to follow the general principles of MVC.

## System Components:

We will be using a SQLite database to store our applications data. The data will mainly come from the users, and will be organized into different tables depending on the user and type of data. We want to keep tables very specific so that it will be easier to organize and understand where each piece of data will be stored. The SQLite database will be stored locally on a machine running a java request server (alternative design). The server will essentially be listening for requests from our Android application. They will be communicating over a socket with a predetermined port. Our Android application will be running a client that has the sole purpose of sending data requests to the server, as well as sending data to the server that should be added to the database.

The rest of the software system will be an android application that presents the data to the user. We will be using the Android version 4.0, or otherwise known as Ice Cream Sandwich, because it will work on 100% of android devices. Later versions of the Android API do not guarantee this level of compatibility.



## System Design (MVC Design):



## Model + View + Controller Components Summary:

We will discuss the above diagram in a little bit more detail.

Model - The model includes database, and the classes that store the data in a manner that becomes useable to the rest of the application (ie. Transaction class).

View - The views will be the activities which display the graphical representation of the data for the user’s ability to easily decipher and read the data (ie Report Activity).

Controller - The controller has a number of different components. The activities, which also act as views, are also responsible for deciphering a user action into the beginning of a response. The activity will then use the client object to send the request to the server. The server, which is a separate process from the application, will query the database, and return the requested data to the client. The client will then pass the data back to the activity which initialized the request from a user action.

## Design Risks

In this section we highlight the possible risks of our design. To start, compatibility is an issue that is often an issue when developing mobile applications that have to work across a variety of different devices and generations of those devices. We address this by researching which version of the Android API was the most compatible among all Android devices. We found that version 4.0 would be compatible for an estimated 100% of Android devices.

Another risk with mobile development is limited storage. Unlike desktops, mobile devices do not carry the same large degree of storage. To address this, we will be storing the database on the device running the server. This device will have larger storage capabilities so we will not need to worry about taking up too much of the mobile device’s limited storage. Secondly, another risk is connectivity to the server from the application. Unfortunately, connectivity is an advanced issue that we will not be able to address past making efforts to advise users to be in areas with stronger connections.

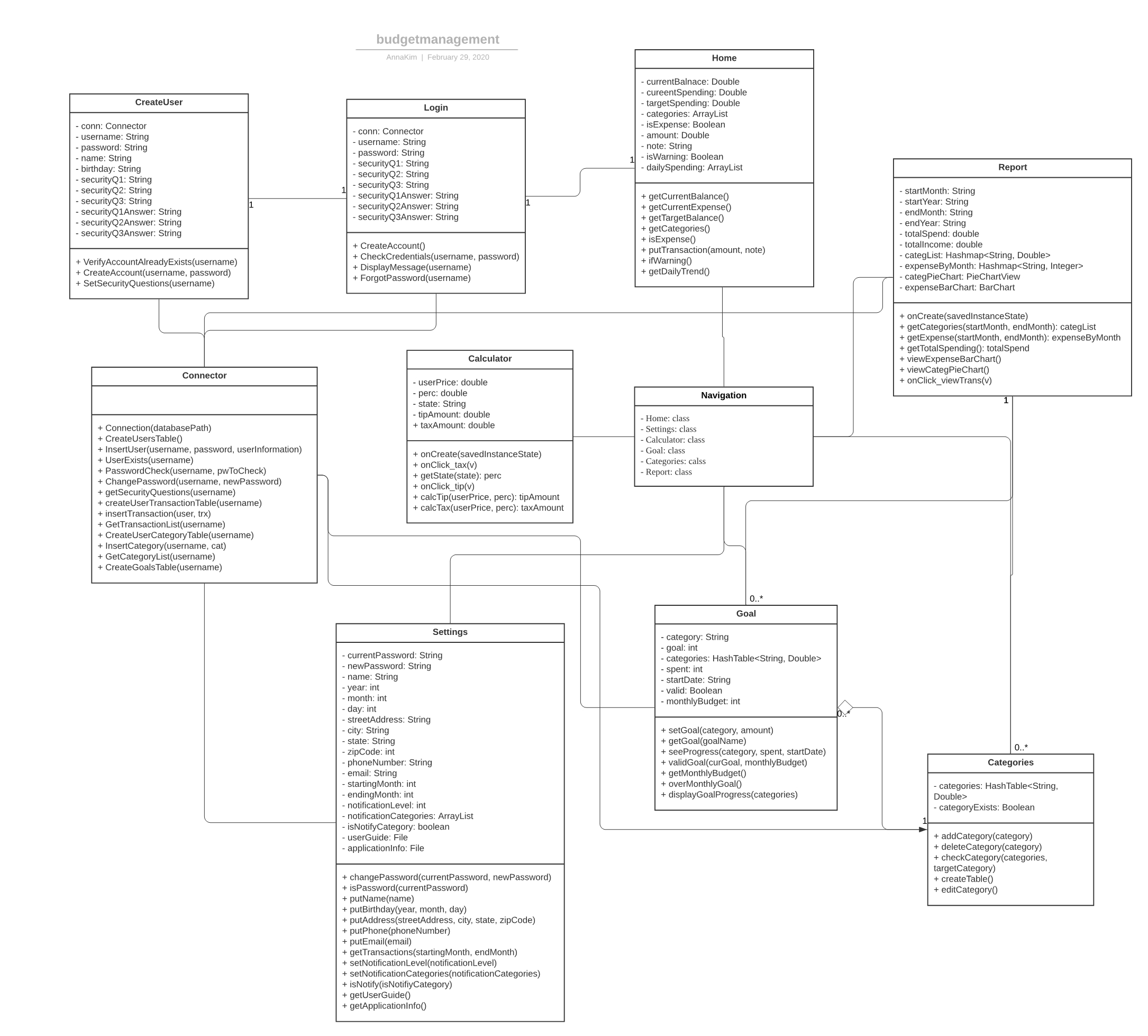
Finally we address the possible security risks with our system design. We will be using embedded SQL calls to the database to insert and query data, some of which will come as a result of user input. This leaves us vulnerable to the possibility of SQL Injection. To address this, we plan on using PreparedStatements, which are common practice to use as they prevent the possibility of SQL Injection. Secondly, we will be storing sensitive user data in our application, including their password. We will be using the conventional Hash With Salt paradigm for storing the password. We will generate a random salt when a password is first set, hash the password with the randomly generated salt, and then store the resulting hash in our database with the salt. This way, we avoid storing plain text passwords in our database, and protecting our user’s information. Upon logging in, we will simply recalculate the hash and compare our stored hash to see if the correct password was used to log in.

# Design Details

In this section go those important facets that are not at the level of “architecture,” such as descriptions of critical algorithms, protocols, and key invariants. Also, wherever possible items should be linked back to your specification. Ideally, you can match up everything in the specification with where it is implemented in the design.

## Class Diagram

The class diagram of the budget management project showing overall classes, fields, and methods. The relationships between classes are also shown in the class diagram.



## Activities

Login activity

|  |  |  |
| --- | --- | --- |
| Content | Component | Description |
| Username | Text box | Where the user fills in their username for an existing account |
| Password | Text box | Where the user fills in their password for an existing account |
| Login | Button | The user will click this button to try and login to an existing account |
| Create account | Button | Opens the Create User page to make a new account |
| Forgot password | Button | Brings up the security questions for the user to answer, provided the account exists |

Create user activity

|  |  |  |
| --- | --- | --- |
| Content | Component | Description |
| Name | Text Field | Where the user fills in their name |
| Username | Text Field | Where the user fills in their desired username |
| Password | Text Field | Where the user fills in their password |
| Birthday | Drop down menu | The user can select their date of birth |
| Security Q1 | Text Field | Where the user fills in their first security question |
| Security Q2 | Text Field | Where the user fills in their second security question |
| Security Q3 | Text Field | Where the user fills in their third security question |
| Security Q1 Answer | Text Field | Where the user fills in their first security question answer |
| Security Q2 Answer | Text Field | Where the user fills in their second security question answer |
| Security Q3 Answer | Text Field | Where the user fills in their third security question answer |
| Create Account | Button | The user presses this button to create an account with the provided information |

Home activity

|  |  |  |
| --- | --- | --- |
| Content | Component | Description |
| Current balance | Text field | Show current balance |
| Current expense | Text field | Show current expense |
| Target expense | Text field | Show target expense |
| Expense or income | Text list | Enable user to choose between expense or income for new transaction |
| Categories | Text list | Enable user to choose category for new transaction |
| Amount | Text field | Enable user to type in amount of new transaction |
| Update | Button | Add new transaction to database and refresh page with updated data |
| Daily trend | Graph | Show user’s daily spending in line graph |

Navigation activity

|  |  |  |
| --- | --- | --- |
| Content | Component | Description |
| Home | Image | Move to home page |
| Category | Image | Move to category page |
| Goal | Image | Move to goal page |
| Report | Image | Move to report page |
| Setting | Image | Move to setting page |
| Calculator | Image | Move to calculator page |

Settings activity

|  |  |  |
| --- | --- | --- |
| Content | Component | Description |
| Account/profile | Text button | Lead to account/profile setting that enable user to edit profile and account |
| Notification | Text button | Leat do notification setting that enable user to set and edit notification |
| Download | Text button | Lead to download page that enable user to download transaction |
| User Guide | Text button | Give brief user guide to the user |
| App Info | Text button | Show user application information file |

Report activity

|  |  |  |
| --- | --- | --- |
| Content | Component | Description |
| MonthStart | Text list | Enable user to choose starting month for report |
| YearStart | Text list | Enable user to choose starting year for report |
| MonthEnd | Text list | Enable user to choose ending month for report |
| YearEnd | Text list | Enable user to choose ending year for report |
| Chart | Graph | Show user monthly report in the form of donut graph |
| Total Spending | Text field | Show user total spending of chosen month |
| Total Income | Text field | Show user total income of chosen month |

Calculator activity

|  |  |  |
| --- | --- | --- |
| Content | Component | Description |
| Tip amount | Text field | Show how much to tip based on calculation |
| Amount (for tip) | Text field | Enable user to enter amount |
| Tip percent | Text field | Enable user to enter tip percentage |
| Tax amount | Text field | Show how much for tax based on calculation |
| Amount (for tax) | Text field | Enable user to enter amount |
| State | Text list | Enable user to select state |

Goals activity

|  |  |  |
| --- | --- | --- |
| Content | Component | Description |
| Goal progress Display | Image(s) | Shows the user’s current goal against how much they’ve spend |
| Month Drop Down Box | Drop Down Box | Lets the user pick the current month being displayed |
| Year Drop Down Box | Drop Down Box | Lets the user pick the current year being displayed |
| Edit Goals | Clickable Text | Allows the user to edit the goal that they previously set. |

Categories activity

|  |  |  |
| --- | --- | --- |
| Content | Component | Description |
| Salary | TextView | Displays the user’s salary that they input. |
| Expense Table | Table | Shows the users categories along with the goals that they set. |
| Delete Category | Clickable Text | Upon clicking, the user will be able to delete a category that they choose. |
| Add Category | Clickable Text | Upon clicking, the user will be able to add a category. Along with inputting the name, the user will also be prompted to enter a monthly goal for the category as well. |

## Descriptions of each class, method, and attribute

The following descriptions are divided by class. The purpose and function of each class is also described. Each class has methods and attributes that are used in the class.

### Connector Class

This class serves the purpose of handling all the calls to the SQLite database. Any operations regarding the database (inserts, create tables, delete entry, queries,...) will go through this class.

**Methods**

1. **Constructor: Connection(String databasePath)**

Creates a connection between the application and the SQLite database

1. **Void CreateUsersTable()**

Creates a table holding all of the users’ login information and personal information

1. **Void InsertUser(String username, String password, userInformation)**

Inserts a new user into the Users table

1. **Boolean UserExists(String username)**

Checks if a username exists in the users table

1. **Boolean passwordCheck(String username, String pwToCheck)**

Checks if a username and password exist in the table. This method will be used for login purposes. We will be storing the hash of the correct password in the table so we will need to repeat the hash process and compare hashes.

1. **Boolean changePassword(String username, String newPassword)**

Will change an existing user’s password, will return fall if the new password’s hash is the same as the existing hash in our database

1. **String[] GetSecurityQuestions(String username)**

Returns a list of the security questions from the user table

1. **Void CreateUserTransactionTable(String username)**

Creates a table for a specific user of all their transactions

1. **InsertTransaction(String User, Transaction trx)**

Inserts a transaction into a specific user’s transaction table

1. **Transaction[] GetTransactionList(String username)**

Returns an array representation of the transactions in a specific user’s transaction table

1. **Void CreateUserCategoryTable(String username)**

Creates a table for a specific user for all the categories that make up their budget

1. **Void InsertCategory(String username, Category cat)**

Inserts a category into a specific user’s category table

1. **Category[] GetCategoryList(String username)**

Gets array representation of the categories in a user’s budget

1. **Void CreateGoalsTable(String username)**

Creates a table for a specific user’s goals

1. **Void InsertGoals(String username, Goal goal)**

Inserts a new goal into the user’s goal table

1. **Goals[] GetGoalsList(String username)**

Returns a list of the goals for a specific user

### Login Class

This page serves as the first page the user sees when they open the application. In this page, the user can log in to their page, or create an account.

**Methods**

1. **CreateAccount(void)**

This function will be linked to a button the user can press. This will take the user to the Create user Page class, which we will describe in detail later.

1. **CheckCredentials(String username, String password)**

This function will check to see if the username and password that the user entered corresponds to a valid account. It will create and establish a connection to our database, then run a sql query to see if such an account with the entered credentials exists in our database. This sql query will be powered by a request to our login server that will return the result of the query. It will return true if such an account exists, and false otherwise.

1. **DisplayMessage(String username)**

This function will display a message to the user that the username that they entered is not associated with any account, and that if they would like they can create a username with this account.

1. **ForgotPassword(String username)**

This function will open up a popup screen that will then prompt the user to enter the answers to each of their security questions. We can use a sql query to our login server to ensure that the entered answers match the correct answers. Upon a correct answering of all three security questions, the user will be allowed to enter the account and then change their password.

**Attributes**

1. **conn: Connector**

This represents a Sqlite connection to the database server that we will maintain throughout the functioning of the application. It will be initialized as soon as the application is opened by the user.

1. **username: String**

This represents the username that the user will try and enter into the field for username when logging in

1. **password: String**

This represents the password that the user will try and enter into the field for password when logging in

1. **SecurityQ1: String**

This represents the security question that the user has already set as their first one.

1. **SecurityQ2: String**

This represents the security question that the user has already set as their second one.

1. **SecurityQ3: String**

This represents the security question that the user has already set as their third one.

1. **SecurityQ1Answer: String**

This represents the security question answer that the user has already set as the answer to their first question.

1. **SecurityQ2Answer: String**

This represents the security question answer that the user has already set as the answer to their second question.

1. **SecurityQ3Answer: String**

This represents the security question answer that the user has already set as the answer to their third question.

### CreateUser Class

This page will be where the user can create a new account that will be registered with our application.

**Methods**

1. **VerifyAccountAlreadyExists(String username)**

This function will take in a username that the user provides, and check to see if that username is already in use by another account in the system. This will be accomplished by running a sql query to the database server with the user input and check to see if the count of accounts with such username is 0. If it is, the function will return false and the user can go ahead and create an account with this function. If not, the function will return true, and a popup message that says “cannot create account, please enter a different username” will show up in front of the user.

1. **CreateAccount(String username, String password)**

This function will create a database entry within our large table of users that will correspond to the username and password that the user provides, assuming that an account with such a username does not already exist. With our database server connection, we will be able to accomplish this by running a sql command to our server. This sql command will take the other information that the user fills out on this page, such as their name and birthday, to get their basic information filled out for this new profile. After the creation of the account, this function will first initialize categories and fake goals based on these fake categories for the user, just so that the other pages linked to their account are somewhat populated. In addition, the user will be told to go to the settings page to read the user guide, which will let them know how to fill out the rest of their account.

1. **SetSecurityQuestions(String username)**

After the account has been created, this function will be called. This function will allow the user to set the security questions and the answers to these security questions in the event that they later forget their password when trying to login. Another tab will be opened where the user can enter three security questions and the answers associated with these questions. This information will then be placed into the users table via a sql query to our database server.

**Attributes**

1. **Database Connection conn:**

This represents a SQLite connection to the database server that we will maintain throughout the functioning of the application. It will be initialized as soon as the application is opened by the user.

1. **username: String**

This represents the username that the user will try and enter into the field for username when logging in

1. **password: String**

This represents the password that the user will try and enter into the field for password when logging in

1. **name: String**

This will represent the name field that the user will fill in when creating their account

1. **birthday: String**

Represents the birthday of the user who is creating their account.

1. **SecurityQ1: String**

This represents the security question that the user will set as their first one.

1. **SecurityQ2: String**

This represents the security question that the user will set as their second one.

1. **SecurityQ3: String**

This represents the security question that the user will set as their third one.

1. **SecurityQ1Answer: String**

This represents the security question answer that the user will set as the answer to their first question.

1. **SecurityQ2Answer: String**

This represents the security question answer that the user will set as the answer to their second question.

1. **SecurityQ3Answer: String**

This represents the security question answer that the user will set as the answer to their third question.

### Home Class

This class acts as the main page of the app. Users are directed to this page after logging in. This class includes data regarding current balance, current spending, and goal of this month. It also enables the user to add new transactions by selecting the type of category and typing in the amount of transaction. The users can add brief notes about the transaction if they desire. Moreover, the app will show daily spending of the past few days.

**Method**

* 1. **getCurrentBalance (void)**

This method will calculate and load the current balance of the user. The balance will be calculated by adding all the income recorded on the transaction data and subtracting that amount with all sum of expenses.

* 1. **getCurrentExpense (void)**

This method will calculate and load the current expense of this month. The current expense will be calculated by adding all the expenses in the current month recorded on the transaction data.

* 1. **getTargetBalence (void)**

This method will get target spending of the current month. The target spending will be calculated by adding all the goals data recorded by the user.

* 1. **getCategories (void)**

This method will get a list of categories. This includes categories that are initially provided by this app and ones created by the user. These categories are used when the user tries to add a new transaction. The user must select a category that fits the new transaction.

* 1. **isExpense (void)**

This method will determine whether the new transaction is expense or income.

* 1. **putTransaction (double amount, String note)**

This method will add new transaction data. New transaction data will include amount of spending or income, brief note if user chooses to write one, and type of category of the transaction.

* 1. **ifWarning (void)**

This method checks whether the result of adding a new transaction exceeds the warning level which is about 80 percent of the goal amount. If it does, the app will display a warning message.

* 1. **getDailyTrend (void)**

This method will get brief data showing the daily expense trend of the current month. The daily spending will be calculated by adding all the expenses of each day recorded on transaction data.

**Attributes**

1. **currentBalance: Double**

A double containing the current balance of the user

1. **currentSpending: Double**

A double containing the current spending of the user this month

1. **targetSpending: Double**

A double containing the target spending of the user this month

1. **categories: ArrayList**

An arraylist containing types of categories for adding new transaction data

1. **isExpense: Boolean**

A boolean that determines whether the new transaction is expense or income

1. **amount: Double**

A double containing the amount of new transaction

1. **note: String**

A string containing brief note about the transaction written by the user

1. **isWarning: Boolean**

A boolean that determines whether the result of adding new transaction lead to exceeding of waring level

1. **dailySpending: ArrayList**

An arraylist containing daily spending of the user for the past few days.

### Settings Class

This class includes the setting of the user. Some of the data in the setting is customized for each user while some data is not. The account/profile setting enables users to manage data of the user and data. Also, the users can download transaction data if they want to. The notification setting enables users to turn on and off notifications of each category. Setting will also include data about user guide, help, application info, and terms of use.

**Method**

* 1. **changePassword(String currentPassword, String newPassword)**

This method enables the user to change password. The user must type both the current password and new password to change the password. The current password must be correct in order to change the password to a new one.

* 1. **isPassword (String currentPassword)**

This method determines whether the password entered is correct.

* 1. **putName (String name)**

This method enables users to add, remove, and edit their names.

* 1. **putBirthday (int year, int month, int day)**

This method enables users to add, remove, and edit their date of birth

* 1. **putAddress (String streetAddress, String city, String state, int zipCode)**

This method enables users to add, remove, and edit their address. This information can be used to provide tax rates to the users which varies on where they leave.

* 1. **putPhone (String phoneNumber)**

This method enables users to add, remove, and edit their phone numbers.

* 1. **putEmail (String email)**

This method enables users to add, remove, and edit their email addresses.

* 1. **getTransactions (int startingMonth, int endingMonth)**

This method gets transaction data of selected months. It will create a list of the selected term when the users wishes to download transaction data.

* 1. **setNotificationLevel (int notificationLevel)**

This method sets notification level. The user can type in percentage of the level for the warning level. For example, if the goal for coffee is 100 dollars and the user sets percentage to 80%, the notification warning will be activated after the user spends 80 dollars.

* 1. **setNotificationCategories (Arraylist notificationCategories)**

This method sets a notification setting. The user can turn on and off notification warnings for each type of category.

* 1. **isNotify (boolean isNotifyCategory)**

This method checks whether notification warning is on or off for a certain type of category.

* 1. **getUserGuide (void)**

This method gets a user guide. User guide includes basic application tutorial for the users that are not familiar with the app. Tutorial covers basic features of the app.

* 1. **getApplicationInfo (void)**

This method gets user application information. Application information includes system requirements, terms of service, and versions of application.

**Attributes**

1. **String currentPassword**

A string containing the current password. This data will be fetched from user accounts table

1. **String newPassword**

A string containing the new password entered by the user. This data will be updated to the user accounts table when the current password entered by the user is valid.

1. **String name**

A string containing the name of the user.

1. **int year**

A string containing the user’s year of birth

1. **month: int**

A string containing the user’s month of birth

1. **day: int**

A string containing the user’s date of birth

1. **streetAddress: String**

A string containing the user’s street address.

1. **city: String**

A string containing the user’s city

1. **state: String**

A string containing the user’s state

1. **zipCode: int**

A string containing the user’s ZIP code

1. **phoneNumber: String**

A string containing the user’s phone number

1. **email: String**

A string containing the user’s email address

1. **startingMonth: int**

An int containing the starting month of the transactions user wishes to download.

* 1. **endingMonth: int**

An int containing the ending month of the transactions user wishes to download.

* 1. **notificationLevel: int**

An int containing the warning level percentage user wishes to get notifications

* 1. **notificationCategories: ArrayList**

An arraylist containing the boolean data of the categories user wishes to warning notifications

* 1. **isNotifyCategory: boolean**

A Boolean that sets notification on and off for a certain category

* 1. **userGuide: File**

A file containing the basic tutorial regarding features of the app

* 1. **applicationInfo: File**

A file containing the system requirements, terms of service, and versions of the application

### Report class

The report class displays a detailed report of budget and spending over a certain period of time in months. A user can select months and years from dropdown lists to bring a report of selected period of time. A range of months is limited for 6 months, since the database holds transactions of 6 months at max. The report page contains a pie chart that is divided by categories and shows spent of each category in both dollar and percentage. Also, a bar chart of expense shows total spending of each month during a period of time that user selected. Total spending and total income for a period of time are displayed. To view full transaction history over a period of time, a user can click the “View Transaction History” button to bring a pop-up window that shows full transaction.

**Methods**

* 1. **onCreate(Bundle savedInstanceState)**

Set starting month:year and ending month:year and call other methods within the report class to display a report

* 1. **getCategories(String startMonth, String endMonth): Hashmap<String, Integer> categList**

Get a list of categories and corresponding expense from database

* 1. **getExpense(String startMonth, String endMonth): Hashmap<String, Integer> expenseByMonth**

Get a list of months (from starting month to end month) and corresponding expense from database

* 1. **getTotalSpending(): double totalSpend**

Calculate total spending for a specified period of time by adding all the monthly spending in expenseByMonth.

* 1. **viewExpenseBarChart()**

Create a bar chart that shows a trend of expense by month. X-axis shows months, and Y-axis shows the amount of expense in dollars. This method uses expenseByMonth to get data.

* 1. **viewCategPieChart()**

Create a pie chart that is divided with categories and shows the amount of expense in dollar and number of percentage in a pie chart. This method uses categList to get data.

* 1. **onClick\_viewTrans(View v)**

Get a pop-up window that shows a transaction history of selected period of time

**Attributes**

**1. startMonth: String**

Use as a starting month of a time range of report

**2. startYear: String**

Use as a starting year of a time range of report

**3. endMonth: String**

Use as an end month of a time range of report

**4. endYear: String**

Use as an end year of a time range of report

**5. totalSpend: double**

Display total spend on report and use for calculating expense proportion of each category

**6. totalIncome: double**

Display total income on report

**7. categList: Hashmap<String, Double>**

A list of pairs of category with corresponding expense in dollar

**8. expenseByMonth: Hashmap<String, Integer>**

A list of pairs of month with corresponding expense in dollar

**9. categPieChart: PieChartView**

A pie chart that shows overall spending per category

**10.expenseBarChart: BarChart**

A bar chart that shows trend of overall spending/expense

### Calculator Class

The calculator class is for calculating amount of tax and tip based on the original price and percentage that user provides.

**Methods**

* 1. **onCreate(Bundle savedInstanceState)**

Get user input of price and percentage for tip or tax

* 1. **onClick\_tax(View v)**

Find matching percentage of tax with user specified state & Call calculateTax method by passing user entered amount and user specified state's percentage of tax and display tax amount.

* 1. **getState(String state): double perc**

Get tax percentage of user specified state

* 1. **onClick\_tip(View v)**

Call calcTax method by passing user entered amount and user specified state's tax rate

* 1. **calcTip(double userPrice, double perc) : double tipAmount**

Return amount of tip (price \* perc)

* 1. **calcTax(double userPrice, double perc); double taxAmount**

Return amount of tax (price \* perc)

**Attributes**

1. **userPrice : double**

The amount of price user wrote as input to calculate corresponding tip or tax

1. **perc : double**

The percent of tip or tax (for tax, it is based on the state)

1. **state: String**

The state in the United States

1. **tipAmount: double**

Amount of tip calculated based on userPrice

1. **taxAmount: double**

Amount of tip calculated based on percent of tax for selected state

### Goals Class

The goals class is responsible for allowing the user to set goals in specific categories for the monthly period. The user will be able to modify the goals if they so choose. It will also have methods to get a particular goal as well as to see a goals progress given the

**Methods**

* 1. **Void setGoal(String category, int amount)**

This method will allow the user to set a goal for monthly spending. The category and amount are fed in and the goal is then set. If the user already has a goal set for a particular category, the goal will be updated to what the user now wants the goal to be.

* 1. **Int getGoal(String goalName)**

This method is for display purposes. It will be used for the front end in order to display the goal on the UI. It will access the hashTable and retrieve the desired goal.

* 1. **Int seeProgress(String category, int spent, String startDate)**

This method calculates the users progress towards the goal that they set for the month. It will calculate and return the value for how much money they’ve spent for that month in the given category

* 1. **Boolean validGoal(int curGoal, int monthlyBudget)**

This method will check to make sure that the goal doesn’t exceed the users overall monthly budget when they set a goal. If they exceed it, they will be alerted the goal isn’t valid.

* 1. **Int getMonthlyBudget()**

This method will get the user’s current set monthly budget. This method will be useful for determining whether or not a goal is valid.

* 1. **Boolean overMonthlyGoal()**

This method will be used to see whether or not a goal is over the monthly limit, returning true if the user has exceeded the monthly spending and false if the user hasn’t.

* 1. **Void displayGoalProgress(HashTable categories)**

This method will handle all of the categories being displayed by the front end using information stored in the database with the help of the connector class.

**Attributes**

1. **category: String**

The name of the goal, so we know which goal the user wants to set.

1. **goal: int**

The dollar amount that the user wants to set for spending for the month.

1. **categories: HashTable**

Key Value pairs will be in the form of <Category, Goal>. The hash table will need to be shared between goals and categories as the two go hand in hand.

1. **spent: int**

The amount a user has spent for the month based on a given category.

1. **startDate: String**

Will be used by the seeProgress method in order to calculate spending for the start of the month to the current date.

1. **valid: Boolean**

Will be used to see if the user’s goal that they set is valid and doesn’t exceed overall budget.

1. **monthlyBudget: int**

This var represents the user’s overall monthly budget.

### Categories Class

The categories activity will allow for users to add a category of their choice as well as set an initial value of spending for the month.

1. **Void addCategory(String category)**

This method will allow a user to add a category that they would like to keep track of in their monthly spending. If a particular category has already been added, it will alert the user that the category already exists and not allow for the user to add a duplicate of that category. The category will be added to a HashTable. In addition, the user will be prompted to set an amount for the category.

1. **Void deleteCategory(String category)**

This method will allow the user to delete a category they have already added. If no such category exists, the user will be alerted that no category is of the name they entered.

1. **Boolean checkCategory(HashTable categories, String targetCategory)**

This method will use the HashTable to check if a category already exists by seeing if it contains the target category. It returns true if the category exists and false if the category doesn’t. This method will be used by addCategory() and deleteCategory() to make sure that they are successful.

1. **HashTable createTable()**

This method will be called to initiate the creation of a Hash Table to store the users categories. The categories will be keys and the goals will be the values. It will only need to be called once to initiate and will be updated in both addCategory() and deleteCategory().

1. **Void editCategory()**

This method will allow the user to edit the name of a category if they so choose. This will only be used for existing categories, so we will need to call checkCategory() here as well.

**Attributes**

1. **categories: HashTable**

Key Value pairs will be in the form of <Category, Goal>

1. **Category: String**

Category will be the key for the HashTable. The key will be the category name.

1. **Goal: Double**

The goal will be used as the value for the HashTable. It will store the user’s goal for that category for the month.

**2. categoryExists: Boolean**

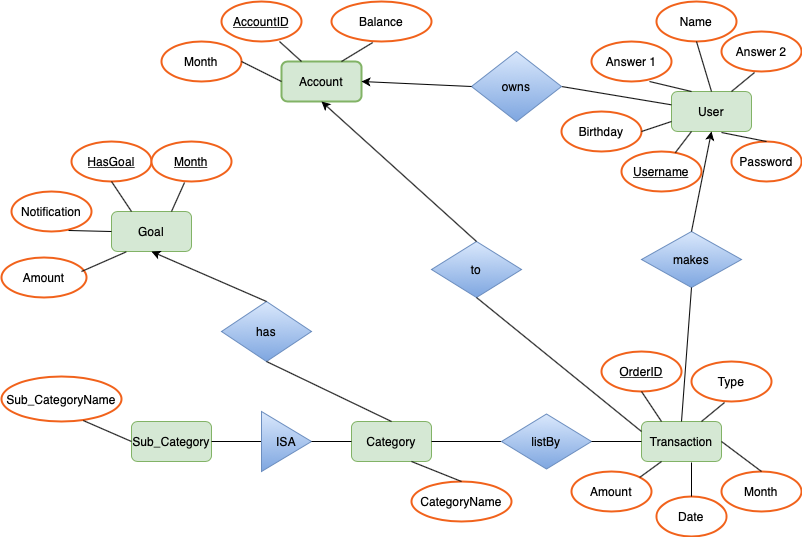
A boolean to check if the category already exists

### Navigation (Menu bar)

Navigation bar will be located at the bottom side of the screen. It will enable users to navigate to different pages including home, category, goal, report, setting, and calculator. Navigation bar is implemented on the following pages: Home, Goal, Categories, Report, Settings, and Calculator. Each fragment is linked with each page. The navigation bar will be set up on a separate xml file and connection will coded in the class pages to enable connection to matching pages.

## Database, tables, and fields

### ER diagram:



### Entity sets:

A***User*** has a unique Username. It contains 6 attributes including Username, Password, Name, Birthday (of the user), and also answers to the registration questions.

An ***Account*** is uniquely identified by AccountID. It contains two other attributes, Month and Balance.

A ***Transaction*** includes 4 attributes: OrderID, Type (expense or income), Amount, Date and Month. OrderID uniquely identifies a transaction.

A***Category*** has an attribute called CategoryName, which also works as the primary key.

A***Sub\_Category*** is a subset of a category. It has a Sub\_CategoryName.

A ***Goal*** is the budget of a user’s account in a given month. It has 4 attributes: hasGoal (whether goal is set or not), Month, Amount (budget amount), and Notification (has or not). Month and HasGoal work uniquely identify the entity.

### Relational Model Schemas:

User(Username, Password, Answer 1, Answer 2, Name, Birthday)

Account(AccountID, Initial\_Balance, Month)

Transaction(OrderID, Type, Amount, Date, Month)

Category(CategoryName)

Sub\_Category(Sub\_CategoryName)

Goal(HasGoal, Month, Amount, Notification)

listBy(CategoryID, OrderID)

# 

# 

# Implementation Plan

**Legend:**

Anna: Project Manager

Ali/Joon: Front-End Developers

Andy/Na: Back-End Developers

Jack: Software Architect

**Iteration 0:**

Pre-programming to setup version control.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Task | Priority (1-5) | Difficulty (1-5) | Time Units (1-5) | Dependency | Person Responsible |
| Setup GitHub remote repository | 1 | 1 | 1 | None | Anna Kim |
| Fork Repositories | 1 | 1 | 1 | None | Andy, Joon, Ali, Na |
| Implement skeleton classes for all the activities | 1 | 1 | 1 | None | Joon/Ali/Jack |

**Iteration 1:**

Iteration one will focus on setting the foundation for our app by setting up things like the database (SQLite and java request server), logging into the app successfully and setting up the home screen. Most of the front-end screens will exist by the end of iteration 1 and the backend will start to be implemented as well. The connector class, the class that is used to fetch information from the back-end and display it on the front-end, will also be implemented by the end of iteration 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Task | Priority (1-5) | Difficulty (1-5) | Time Units (1-5) | Dependency | Person Responsible |
| Implement the database and java request server | 1 | 1 | 3 | None | Andy/Na |
| Implement Front End for “Create User” page, “Login” page, “Home” page, “Settings” page, “Goals” page, and “Categories” Page, “Report” Page so that they’re able to be linked to the back-end | 1 | 1 | 2 | None | Joon/Ali/Anna |
| Implement Back End for “Create User”, “Login”, “Report”, “Goals”, and “Category” pages so ensure that they link with the front-end properly and store the correct data. | 3 | 4 | 2 | Implementing Front End for Pages | Andy/Na |
| Add Handling for invalid usernames and passwords and if profile already exists as well as changing passwords | 1 | 1 | 1 | Implementing Back end of the login page | Andy/Na |
| Add a Handler for a “Create Account” Button | 1 | 1 | 1 | Will need Create Account Button to be developed | Joon/Ali |
| Implement Back end for calculating current expense, current balance, getting categories, adding transaction, getting daily graph. | 1 | 2 | 2 | None | Andy/Na |
| Add Back end Logic for editing a users profile, sending notifications, user guide, getting application file | 1 | 2 | 4 | None | Andy/Na |
| Implement the functionality of the connecting class between the front-end and the back-end so that they properly interact with the rest of the code | 1 | 4 | 3 | Will need back-end data along with front-end data handlers to ensure this is done properly | Jack/Na |
| Implement the functionality of all the SQLite and client/server methods so that they properly interact with the rest of the codebase | 5 | 4 | 3 | None | Jack/Anna |
| Confirm/Test with other developers that the database methods work properly in the code they are writing | 3 | 2 | 2 | Methods will have to be developed | Jack/Anna |

**Iteration 2:**

Iteration 2 will focus on finalizing setting up queries and request handlers for the server as well as adding handlers for buttons and linking the front-end pages to the back-end data. User functionality will come together here, as by the end of the iteration, they will be fully linked to begin testing.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Task | Priority (1-5) | Difficulty (1-5) | Time Units (1-5) | Dependency | Person Responsible |
| Add database server request handler and SQLite queries | 1 | 1 | 2 | Database and server must already exist | Andy/Na/Anna |
| Add handlers to link the front-end and back-end for category and goals page | 1 | 1 | 1 | Both goals and categories front-end and back-end must exist | Joon/Ali |
| Implement handler to add and edit the categories | 1 | 1 | 1 | Edit categories button must already exist | Joon/Ali |
| Implement table using the info from the database to be able to display the categories and the user’s income | 1 | 2 | 2 | Categories page must already exist and data must be already stored in the back-end | Andy/Na |
| Add post account creation information filling, and notification for user to go to user guide | 1 | 2 | 2 | Account must already be created and stored in the back-end along with user data | Ali/Joon |
| Integrate database with logging in backend logic | 3 | 3 | 2 | Database must be defined | Andy/Na |
| Implement Database management code for deleting older data for both SQLite and the server | 3 | 2 | 2 | Data must already be stored in the database to do this. | Jack/Anna |

**Iteration 3:**

Iteration 3 will focus on testing our app to make sure everything has come together as planned. We will need to make sure that the front end of the UI comes together and is able to fetch data from the back-end as needed.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Task | Priority (1-5) | Difficulty (1-5) | Time Units (1-5) | Dependency | Person Responsible |
| Implement Front-End of the Calculator tab if time permits | 1 | 3 | 3 | None | Ali/Joon |
| Implement Back-End of the Calculator tab if time permits | 1 | 3 | 3 | None | Andy/Na |
| Testing to make sure the Front-End is getting the correct data from the back-end | 5 | 5 | 5 | Everything in the front-end and back-end must be implemented | Jack/Anna |
| Test to make sure everything in the front-end works as it is supposed to | 5 | 5 | 5 | Everything in the front-end must be implemented | Ali/Joon/Anna |
| Test to make sure everything in the back-end is both calculated correctly as well as stored correctly | 5 | 5 | 5 | Everything in the back-end must be implemented | Na/Andy/Anna |

## 

# Testing Plan

## Unit testing

Unit tests are to confirm that the individual small features function as we intended. This will make sure that sections of application function properly and meets the requirement. Each unit that will be tested for our app will be each method we create and tables we created in a database. Since many of the pages in our app have separate classes and each of them have multiple methods and linked tables in the database, each of them will be a good bigger unit to test. We will first test each module using JAVA virtual machine and debug functions for accuracy of the algorithms and outcomes. Then we will test on an Android emulator to see what the actual results would look like.

## Integration testing

As we add each activity/page in the application, we manually and automatically test whether the additional activity interacts properly with the already existing pages. We will also write integration tests for the new activities interactions with the database and back end functionality.

## System testing

A group of simulated customers will be used to test our app. This testing will focus on all possible functionalities that can be performed on the different components of the system. We will use Robotium for Android to randomly choose simulated users.

## Performance testing

We will monitor the performance of our application using metrics such as the time it

takes for each page to load. In addition, we will measure how long each of our SQL queries take, as these queries will be the backbone of our application. If we decide to use a server, we will also measure the latency of our server requests. These metrics will allow us to determine how optimized our code is, and we will be conducting these tests at each iteration to get a better idea of how well our application does on these performance tests, and what changes, if any, we should make to our code. These tests will largely be able to be completed automatically, but we can analyze the results over the course of our development to find areas of improvement.

## Compatibility testing

Since we’re using Ice Cream Sandwich, which is compatible with all Android devices, we’ll have to test the functionality over a lot of different Android devices including different screen sizes and processor speeds. Will have to format our app depending on the age of the device.

## Beta testing

We will release our app to a small group of people to test the functionality of the app based on our last iteration. We will choose a group of students having Android phones and ask their using experience, such as whether they have encountered any problems when using the app, and any suggestions to improve the functionalities. We will look into the feedback and make modifications to the app correspondingly.

## Regression testing

After resolving errors or adding new features, we will run the test on the application iteratively to ensure that all the features that worked without error before making new changes still work well on the application. The regression testing will be conducted on both Virtual devices provided in Android Studio and actual android devices. This test can ensure the changes we made to fix errors or implement new features do not affect other functionalities in the application. So this test will be conducted before committing changes to make sure we do not commit or push the application that might contain any unexpected error.