$anity

Assignment 4 - Testing Document

Hitchhikers 2.0

Adam Espinoza	6035082282
David Sealand	9561295475
Aneel Yalamanchili	8556701778
William Wang	7825618977

Table of Contents

1. Preface

1.1 - About/Mission

This document serves to produce the definitions, specifications, architectural/detailed designs, requirements, and implementation for $anity, a mobile finance application that allows users to create a budget across a variety of categories to which users can easily specify when they spend money. As an implementation specification, this document shall include changes of design decisions that are different from Assignment 2, and the reason of the changes as updated from comments from Sarah Cooney and the CSCI-310 document grader.

1.2 - Intended Audience

The intended audience for this implementation document are all parties of the development cycle of $anity: the client of the $anity app, Sarah Cooney, who has commissioned the software and is in charge of clarifications, and the developers of the $anity, thereafter.

1.3 - Version History

This section will be updated as $anity continues to be expanded and new features are implemented as well as a rationale for why these versions were finalized.

- *Version 1.0* - Delivered as of **October 18th, 2017** under conditions specified in Assignment #3. Current application has been written to standards specified by Sarah Cooney with notable exception of certain push notifications which were later clarified with her (in-app notifications will now be allowed) and time frame table which will be implemented in further versions of $anity.

2. Installation/Instructions

In order to run the test cases, it is important as the reader to understand how to run the application. Here are the instructions needed to run $anity on our current platform:

2.1 - Install Ngrok

Ngrok allows for testing mobile apps against a development backend running on your machine. Follow the installation instructions on Ngrok's homepage (https://ngrok.com/) which will guide you in downloading a .zip file and unpacking their files. Once it is completely installed run the following command in the working directory where the bash file was downloaded: `./ngrok http 80`. When you start ngrok, it will display a UI in your terminal with the public URL of your tunnel and other status and metrics information about connections made over your tunnel:

```
ngrok by @inconshreveable

Tunnel Status                 online
Version                       2.0/2.0
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://92832de0.ngrok.io -> localhost:80
Forwarding                    https://92832de0.ngrok.io -> localhost:80

Connnections                  ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

By this point your server should be running that will be listening to a connection from your phone on a specified port.

For more documentation regarding Ngrok, please view their documentation files found here: https://ngrok.com/docs

2.2 - Install Dependencies

Navigate to the $anity project directory using the *cd* Unix command on your terminal. From there, run the command `pod install` in the current working directory. If the

terminal responds with ‘command not found,’ be sure to install pod dependencies via: `sudo gem install cocoapods.` This will download all dependencies and packages required to properly run the application.

2.3 - Running Client on XCode

In order to run the application on XCode, you must open *‘Sanity.xcworkspace’* and run the application with an iOS target device. However, before you do this, you must navigate to the *‘Client.swift’* file and ensure that the WebSocket endpoint is correct which should match the endpoint specified on Ngrok. If you have reached this point without failure you will have access to the application on your local phone or in a virtual phone that XCode provides which can be used for testing purposes.

3. Black Box Tests

The following Black Box tests examine the functionality of an application without peering into its internal structures or workings. These types of tests are applied virtually to every level of software testing: unit, integration, system and acceptance. The tester is aware of *what* the software is supposed to do but is not aware of *how* it does it. The following are black box tests that we tested for quality assurance:

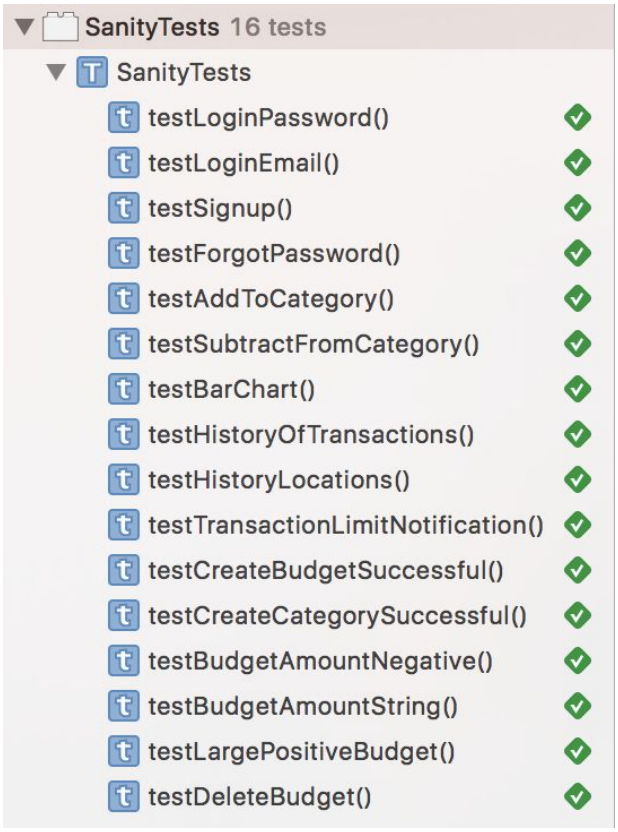


Figure 3.0 - The above screenshot shows the success of all 15 Black Box tests described below. To view them, open the Swift document and see the line for each test.

3.1 - Login with Correct/Incorrect Password Test

Location of Test: SanityTests/SanityTests.swift, Line 29

Description of what the test case does: This is a sort of 'hack test,' or 'penetration test' where the incorrect password is entered at the Login Screen. An error screen should be printed until the correct email/password combination is entered. This test is important because we need to ensure that other users aren't allowed to login to accounts that they don't

have appropriate credentials for and to ensure security. For this test we will be using David Sealand's email: *sealand@usc.edu* whose password is "landofthesea" and we will be entering a dummy password: CSCI310. To get to this screen the user will have started/ran the application. We will add him to the database via sign-up and then go from there.

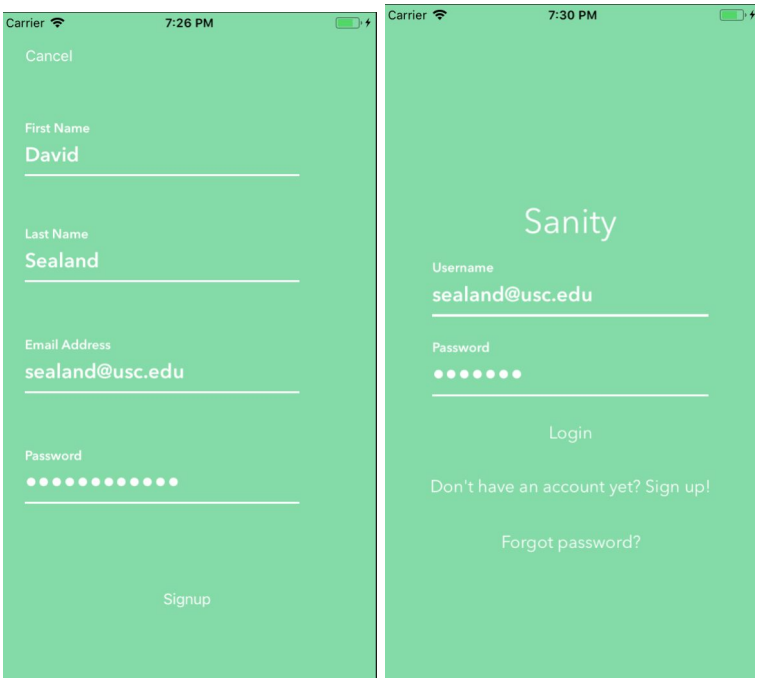
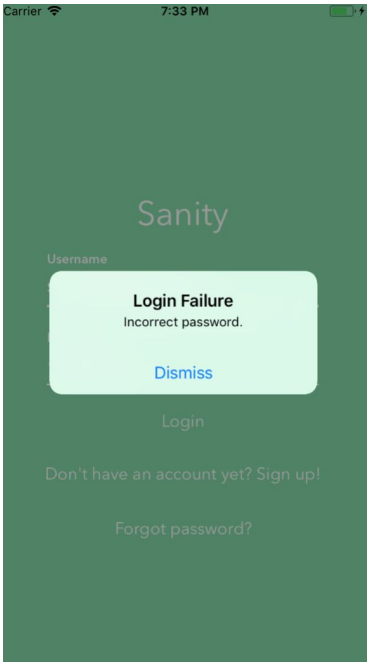


Figure 3.1.1 -Shows David Sealand being signed in and logged in with an Incorrect password

Result of Test Case:



The test showcased that when invalid input/password was entered, a pop-up screen appeared showcasing that an error has occurred which falls in line with what the tests should end with.

Any bugs discovered: For this test we did run into an error where the server stops after this test was conducted. Upon further inspection we found that the Java backend code threw a EOFException error due to threading issues that had occurred. While this type of exception is slightly normal for our tests, it's helpful to remember that when going further in tests.

3.2 - Email Login Test(s)

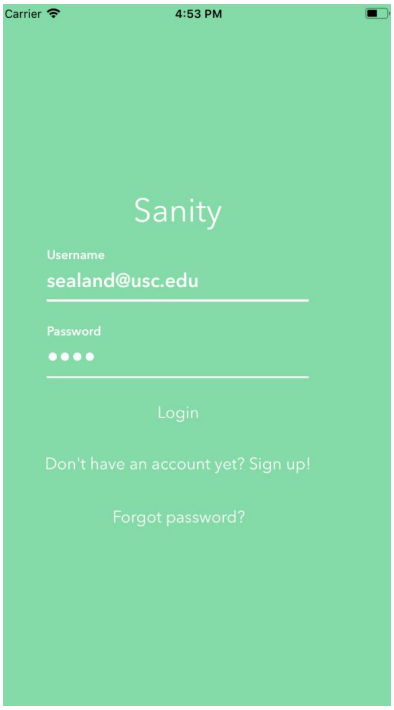
Location of Test: SanityTests/SanityTests.swift, Line 64

Description of what the test case does: This is a two fold test that goes as follows:

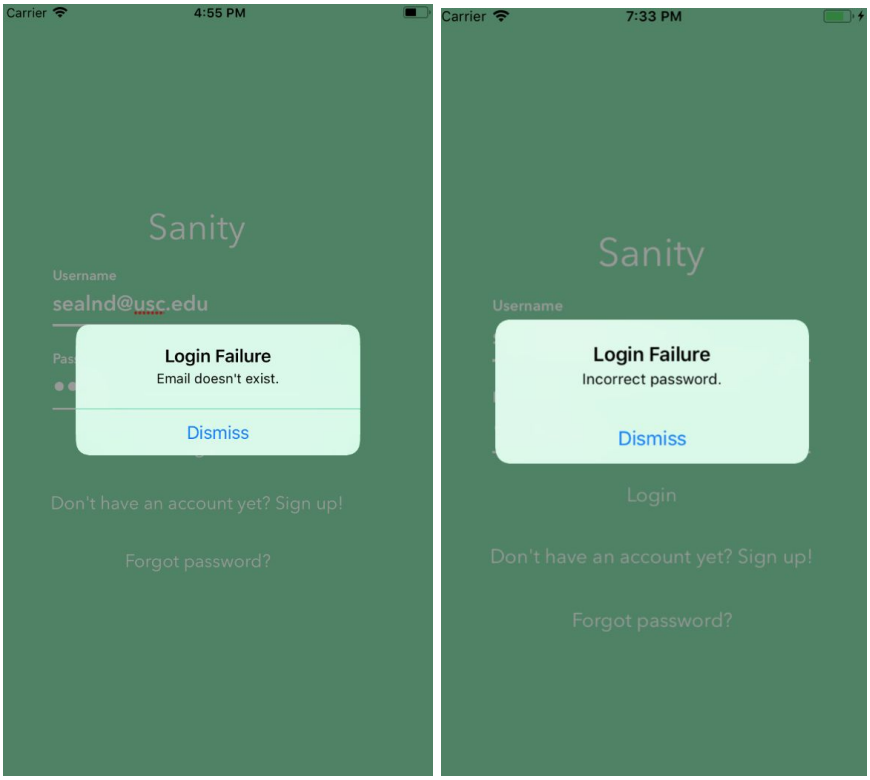
- If the email is not in the database, an error pop-up should also appear for the user (which can be closed out). This test is needed because users should not be allowed to login in with a random email, they should have a registered account with $anity.
- If the email *is* in the database, it needs to be correctly spelled in order to login. This is a security concern in order to prevent users from accessing an account and also

coincides with *Section 3.1 - Login with Correct/Incorrect Password Test* where the correct password should be inputted

For this test we will be using David Sealand's email: *sealand@usc.edu* which will not be in the database at first and then will be in the database (assume that he signed-up correctly). This test is being conduct on the login screen where users are putting in their email credentials.



Result of Test Case:



In the representational view, we imputed the value that would have caused the email not to exist error to appear. 'sealnd@usc.edu' is not in the database and therefore has the appropriate message for the user which they can dismiss. Once Sealand is entered into the database, the user will have to enter the correct password before they can be logged in.
Any bugs discovered: No bugs were found when testing this as far as we know.

3.3 - Sign-Up Test

Location of Test: SanityTests/SanityTests.swift, Line 96
Description of what the test case does: This is a test where we input in an email that is already listed in the database (emails should be unique). We need to ensure that emails are unique so we have an identifying ID for each user. This also prevents other users with similar emails from logging in to an account that they are not registered for and serves as a security prevention measure. For this test we will be signing up as Adam Espinoza's email: adamespi@usc.edu and another Adam Espi with the ***same*** email which is already in the database. An should print out an error pop-up for the user Adam Espi. To navigate to this

page, the user must press the ‘Don’t have an account? Sign up!’ text which is located below the email and password text boxes.

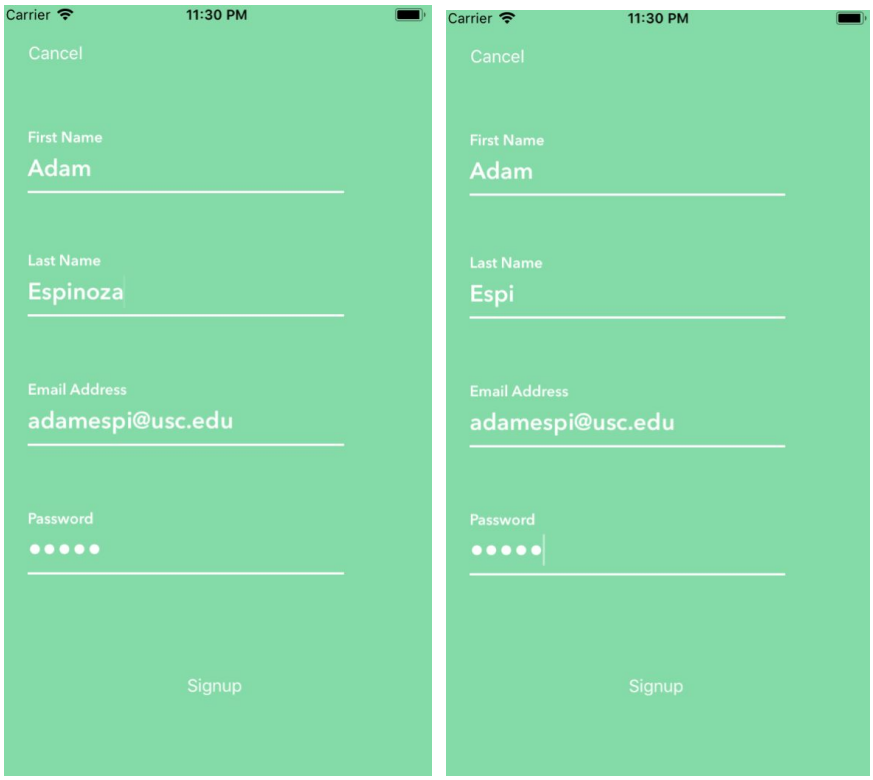
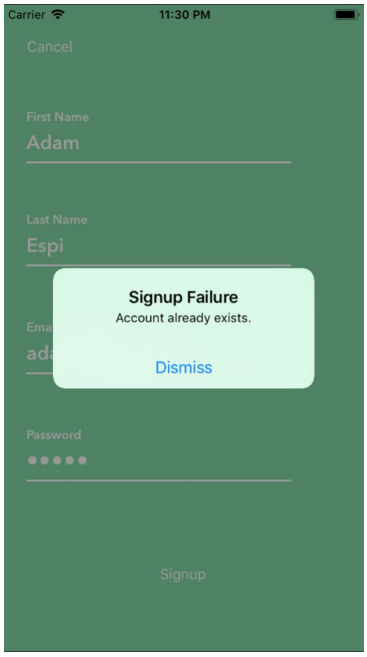


Figure 3.3.1- Two users with the same email are being signed up one after another (left image happens before right)

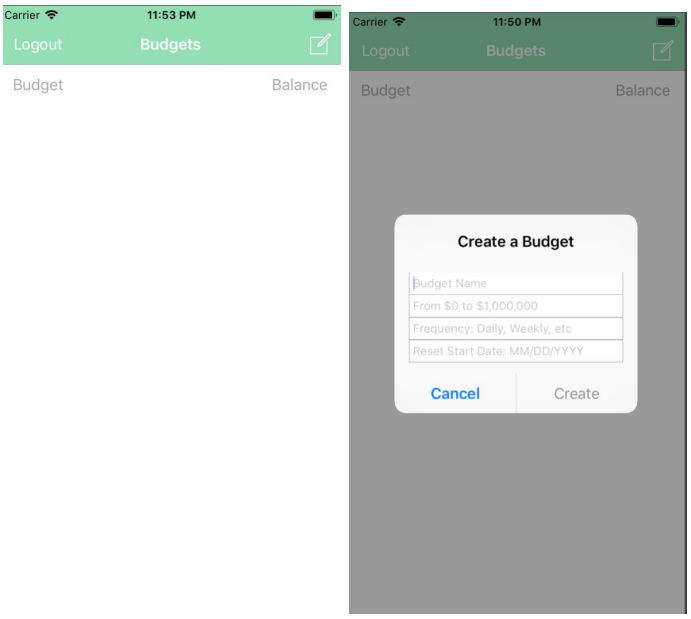
Result of Test Case:



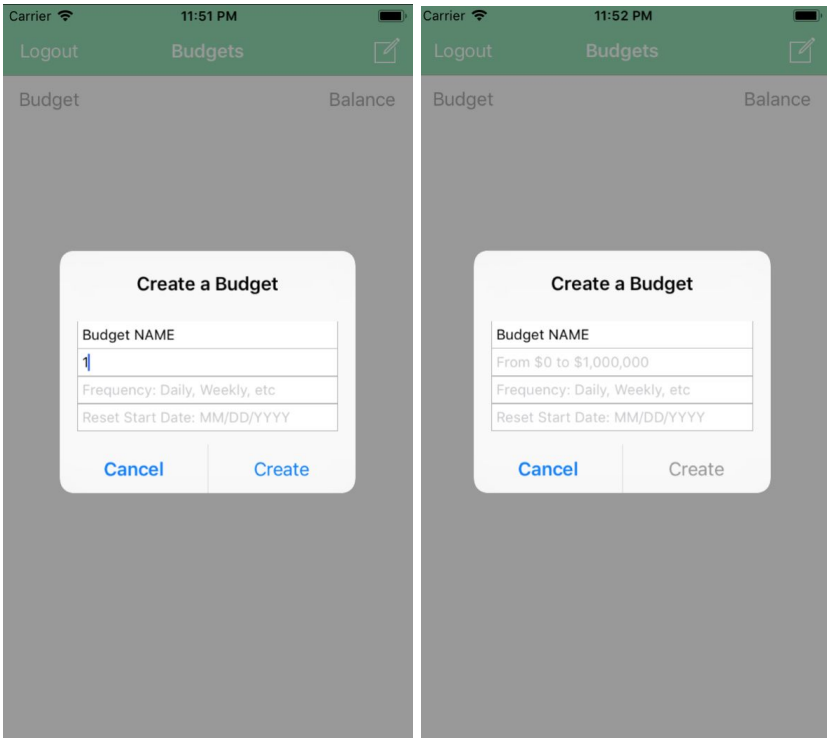
The test fails for the right user Adam Espi because the email is already signed into the account! The user is greeted with a login failure error which can be dismissed. The user will continue receiving the error until they enter an email that suffices.
Any bugs discovered: No bugs were found when testing this as far as we know.

3.4 - Setting Negative Budget

Location of Test: SanityTests/SanityTests.swift, Line 901
Description of what the test case does: This test is for when a user sets a negative value for the overall budget which should not be allowed because the user needs to have to have a positive integer budget criteria. We need to ensure that there is a safeguard for dummy users that attempt to break the program and also to guarantee that the application doesn't crash if the input in provided. Assuming the user has entered correct credentials and logged in, we will press the 'notepad,' icon which will have a pop-up appear that allows users to type the name of the budget and the amount for the budget. For this test we will be inputting the values: '-1'.



Result of Test Case:



For this test the values weren't allowed to be entered on the representational view (for the user). The negative value was ignored and the 1. If a user was able to enter the values,

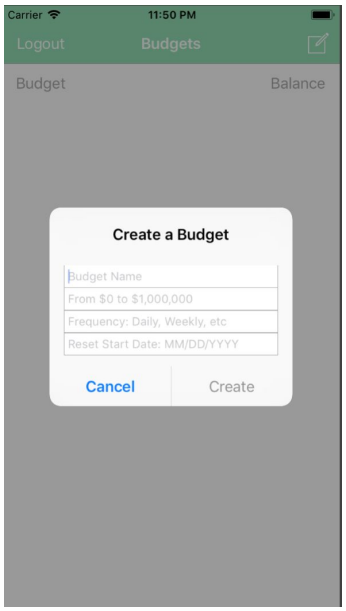
we would have measures to safeguard any malicious values entered that would cause the app to crash.

Any bugs discovered: We found a bug where the 'Daily, Weekly, etc.' and 'Reset Start Date' does not allow the user to enter the appropriate text for each of the text boxes. This was due to an unknown bug in a swift file that prevented the text box from appearing. We are currently investigating the error, but there should be no other problems when this issue is resolved. The main concern deals with keyboard representation on the phone.

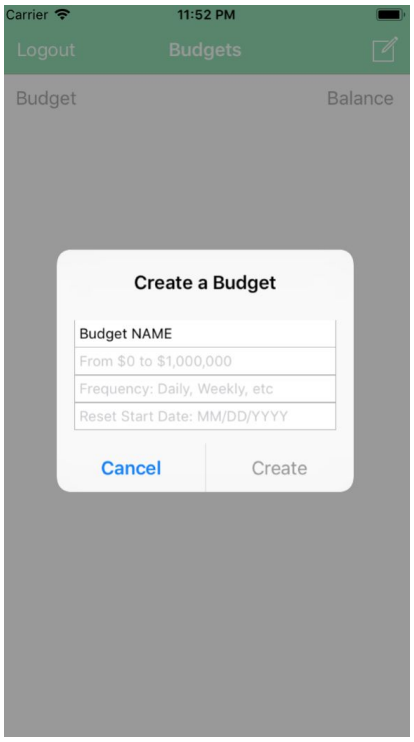
3.5 - String Budget Test

Location of Test: SanityTests/SanityTests.swift, Line 965

This test is for when a user sets a string value for the overall budget which should not be allowed because the user needs to have to have a positive integer budget criteria. We need to ensure that there is a safeguard for dummy users that attempt to break the program and also to guarantee that the application doesn't crash if the input in provided. For this test we will be inputting the value: 'AMOUNT' into the $$ text box. Assuming the user has entered correct credentials and logged in, we will press the 'notepad,' icon which will have a pop-up appear that allows users to type the name of the budget and the amount for the budget.



Result of Test Case:



The result of the test shows that users are not allowed to enter the budget String. The text box has a safeguard that only allows users to enter integer values, which is helpful from preventing users from trying to break the application. You can't see that I tried entering the text, but the application did not allow a string at all.

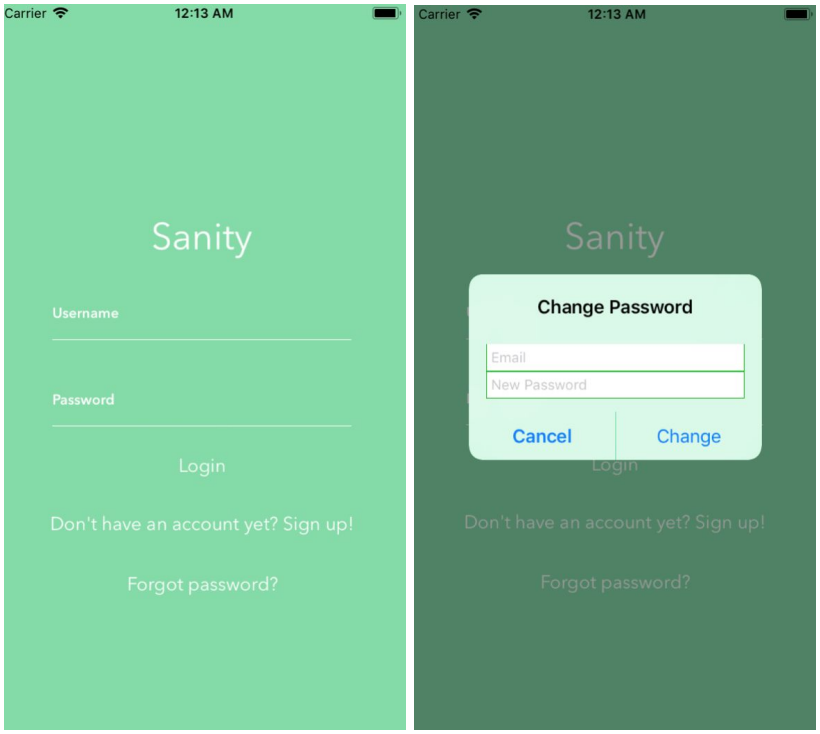
Any bugs discovered: No bugs were found when testing this as far as we know.

3.6 - 'Forgot Password' Test

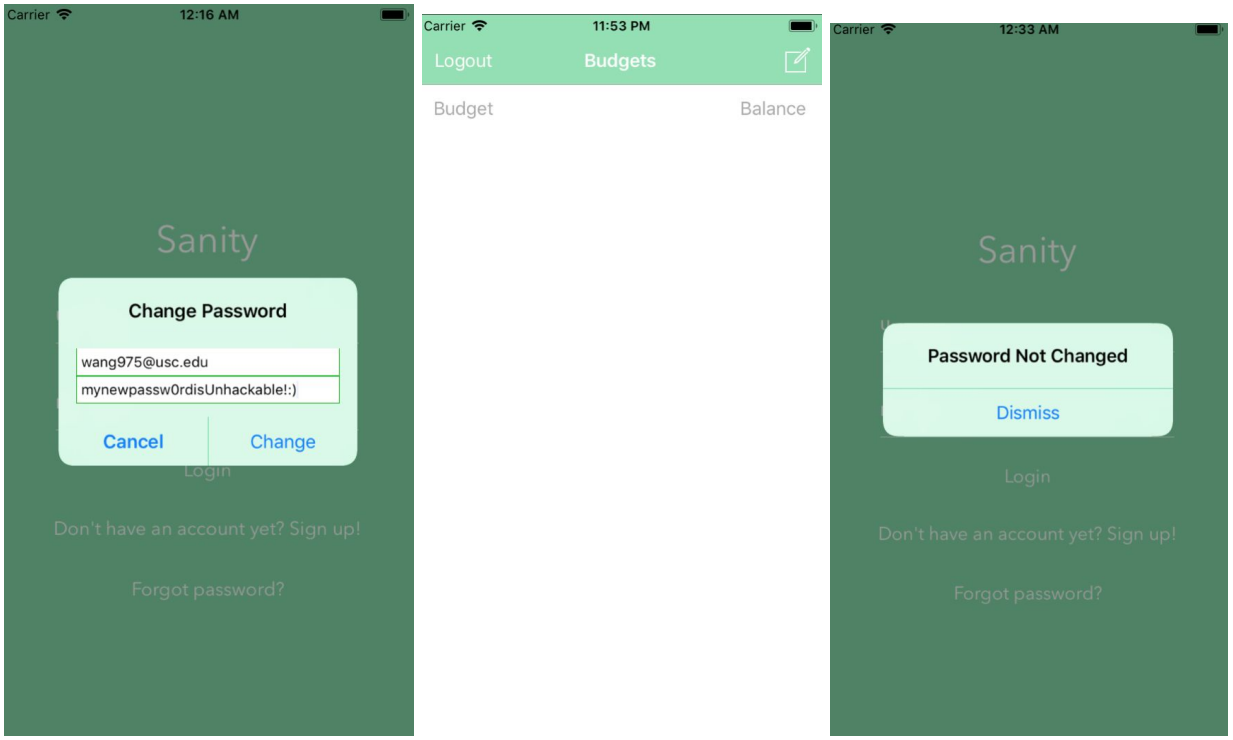
Location of Test: SanityTests/SanityTests.swift, Line 131

Description of what the test case does: This test is in the event the user forgets their password in which they should be prompted to reset their password. We need to conduct this test because we need to have a safeguard for when users forget their password and also is common practice with applications that have login features. The forget password feature is at the bottom of the login screen, when the user presses the white text, they are then prompted to change their password, having the correct email. If the incorrect email is entered they will not be allowed to change their password. For this test we are using

William Wang's email and password which is being changed (see 'Result of Test Case'). If the email is in the database then the password will be changed and the user will be taken to the budget homepage for their respective account.



Result of Test Case:



As you can see on the left William Wang's account email and password are placed into the text boxes. Currently there is no email confirmation features which allows users to 2 Factor Authentication the password change, however, that is a feature we would like to implement in the future. The middle image is of the budget landing page once they enter an email that is in the database. Should the user enter an email that is not in the database when resetting their password, they will receive a 'Password Not Changed' error in the representational view.

Any bugs discovered: This isn't really a bug, rather a security issue that may need to be addressed at some point, but the user is able to change the password and login to another user's account *if* they have their email. For example, I can change Sealand's password by entering his email (sealand@usc.edu) and then changing his password. One other smaller thing to note is that the *new* password should be entered with dots so the person changing their password doesn't have other people looking over their shoulder to see their password.

3.7 - Adding Large Positive Budget Test

Location of Test: SanityTests/SanityTests.swift, Line 1028

Description of what the test case does: This test case is for when users enter a large budget amount when they create a budget. User are initially limited from $0 to $1,000,000 which is showcased in grey text, but this test is to ensure that no more than one million dollars is entered. We need this test to prevent users from entering large values that could cause integer overflow or other complications. Assuming the user has entered correct credentials and logged in, we will press the 'notepad,' icon which will have a pop-up appear that allows users to type the name of the budget and the amount for the budget. For this test case we will be inputting the value: '999,999,999,999,999' which should be prevented.

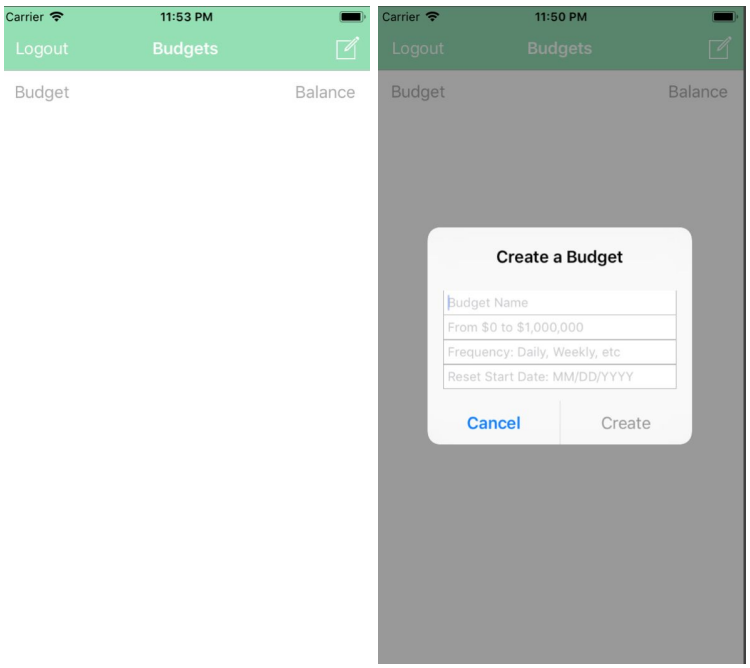
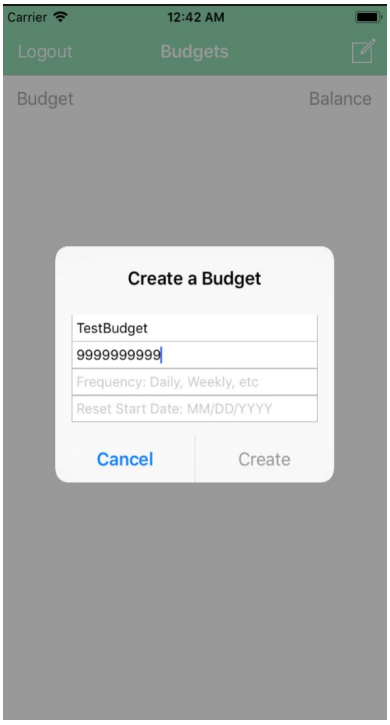


Figure 3.7.1 - The image is illustrates when a user goes to the budget landing page and tries to create a budget (on right).

Result of Test Case:



When the test was conducted on the representational view, the user was allowed to enter 9,999,999,999, however, the user was not allowed to 'create' the budget (the create button is turned gray and the user isn't allowed to press it). The user is only allowed to create the account when it's below 1,000,000, so in this example, the passing value would be 999,999,999.

Any bugs discovered:

3.8 - Deleted a Created Budget Test

Location of Test: SanityTests/SanityTests.swift, Line 1091

Description of what the test case does: This test is to ensure that users are allowed to deleted a budget upon creation. This can be done be sliding the budget to the right, whereby a red button should appear that allows users to press 'delete,' which should cause the budget to disappear from the list of budgets on the screen. Assume that the user has already correctly created budgets which are represented on the budget landing screen. For this test case we will just be sliding the test the last test to the left to delete.

There will be screenshots below in the result of test case which will clarify this description further.

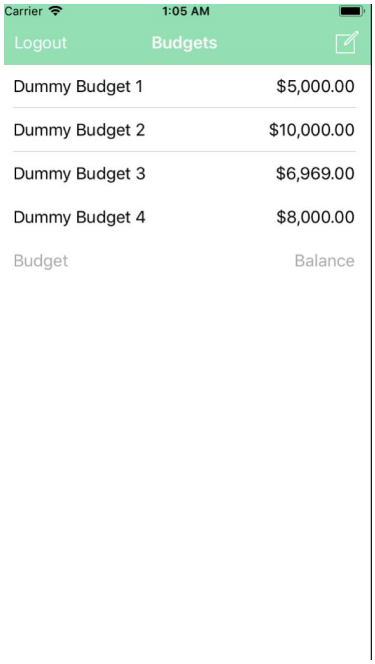
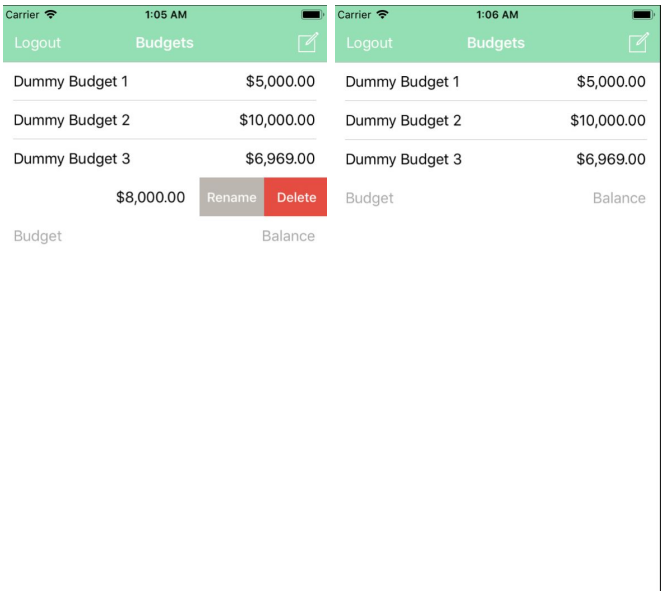


Figure 3.8.1 - The figure above shows 4 different budgets in the stack

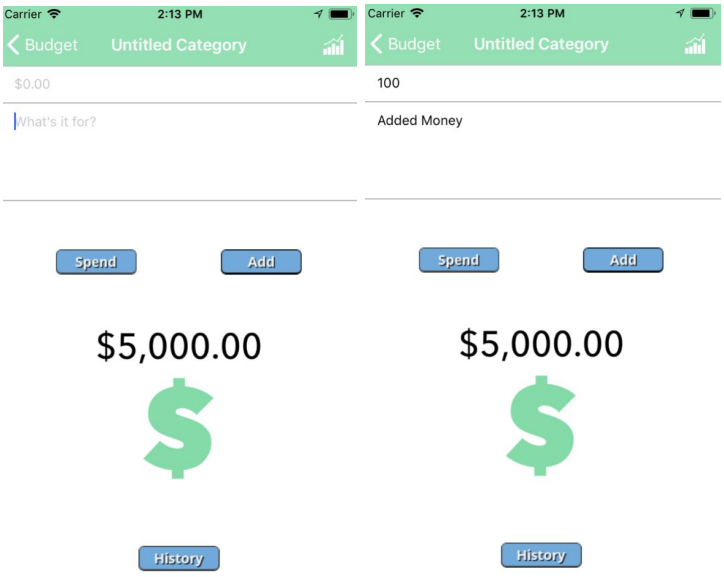
Result of Test Case:



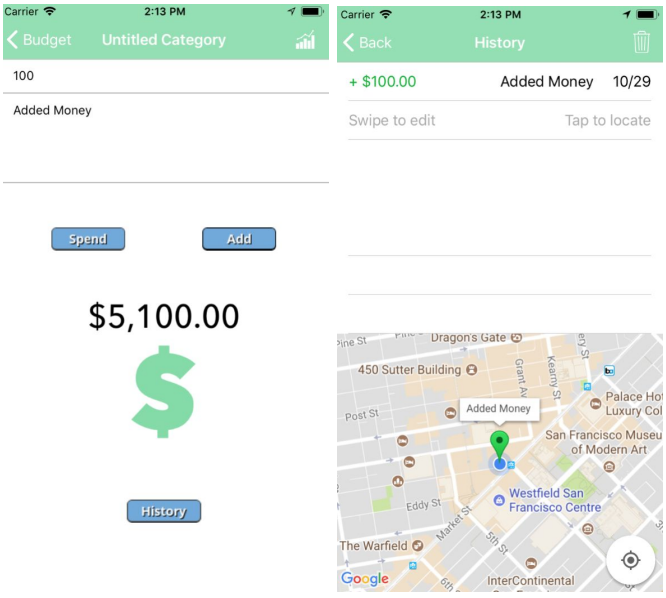
The result of the presentational tests show that Budget 4 is deleted (on the left) with the red button being pressed. The resulting list of budgets (on right) shows that Budget 4 was deleted from the list and only 3 now remain.
Any bugs discovered: The budget was deleted from the application on the iPhone, but the deletion was not reflected in the database (which it should be). This issue was made apparent to backend developer David Sealand who made the correction and tested it in the following white box tests (Section 4).

3.9 - Adding Money to Category Test

Location of Test: SanityTests/SanityTests.swift, Line 158
Description of what the test case does: This test is when a user wants to add money to their budget, meaning that they had a positive transaction into their budget account. This is in the event a user wants to manually transfer from one account to another or they decided to increase their transaction amount. It's important for users to have the flexibility to add financials to their account so that they aren't just limited to subtracting. To navigate to this page, assume that a correct positive budget has been created and that you are on the budget description page. For our test examples we will be adding $100 to our budget with a Memo title: 'Added Money'. Assume that there is already $5,000 allocated in the budget amount.



Result of Test Case:



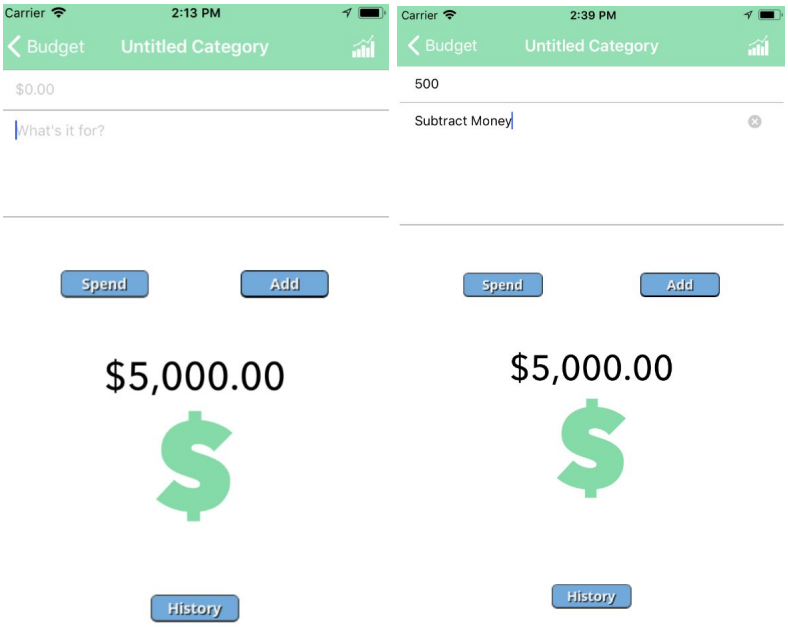
As you can see on the left, after the money was added, the category amount increased by 100. If you want to look further, the amount was also added in the history action with the memo 'Added Money' and date at the top as well as with the positive $100.

Any bugs discovered: No bugs were found when testing this as far as we know.

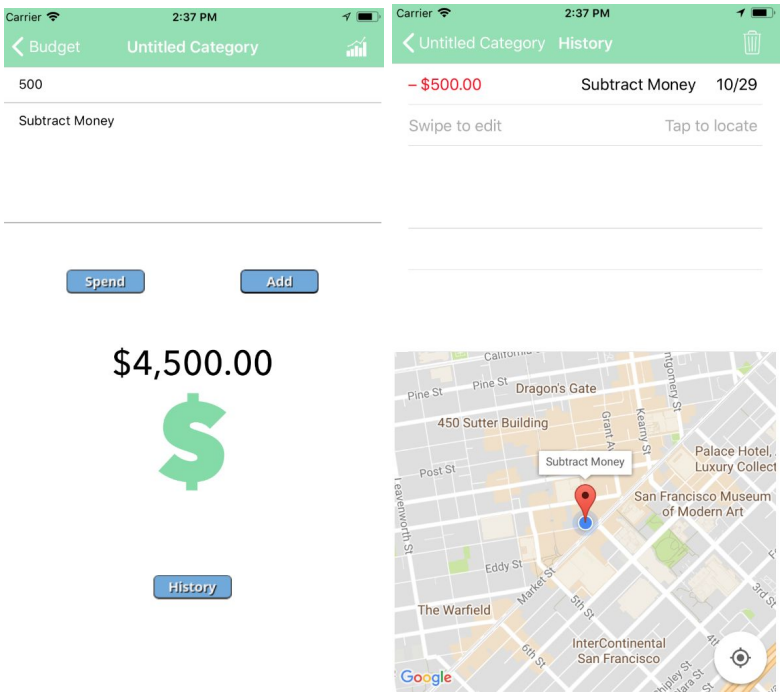
3.10 - Subtracting Money from Category Test

Location of Test: SanityTests/SanityTests.swift, Line 257

Description of what the test case does: This test is when a user is subtracting money from a transaction (from spending or transferring to another account). This event is necessary because it is a required functionality of customer Sarah Cooney and also showcases the main purpose of $anity. Users should have a positive budget in which they subtract money from which is reflected on their overall budget. For our test examples we will be subtracting $500 to our budget with a Memo title: 'Subtracted Money' and description: "This is a test transaction." Assume that there is already $5,000 allocated in the budget amount.



Result of Test Case:



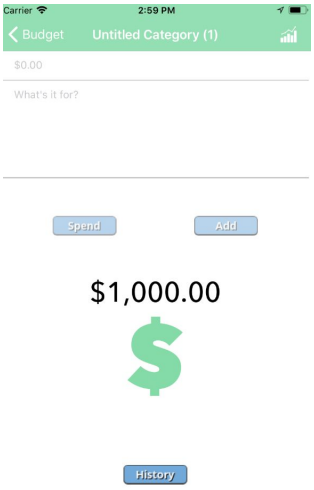
As seen in this representational view, the $500 amount is subtracted from the budget and it is automatically reflected once the button is pressed (on the left image, now showing $4,500). If you press the history button (image on right), also, you can see that the amount is also accurately reflected in the transaction history, showcasing that the information is saved within the backend.

Any bugs discovered: No bugs were found when testing this as far as we know.

3.11 - Bar Chart Budget Test

Location of Test: SanityTests/SanityTests.swift, Line 355

Description of what the test case does: This test to ensure that the bar charts for each individual budget is updated when transactions are imposed. This is done for every *single* category within a budget. To view this feature, assume a category of some amount $1000 is created and you within the Subtract/Add view. To navigate to the bar chart you press the icon at the top right which will show you your spending amount for the previous week/month/year. This feature is helpful for users when helping them keep track of their spending habits over time.



Result of Test Case:

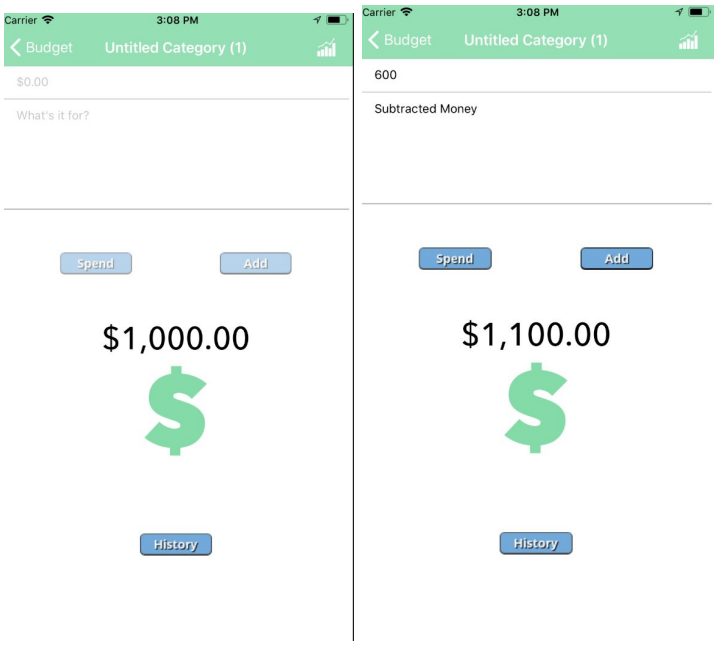
As you can see in the representational view, the value amount was added to the bar chart which is created on the front end for the past week. The average is then calculated from the amount.

Any bugs discovered: The bar chart is dependent on current static values, it is something we are testing via manually inputted values (which in fact work) but it's important to point that out as an ongoing issue/bug for the program. These feature will be updated as the project develops as it is not a core functionality that Sarah Cooney specified in her original requirements document.

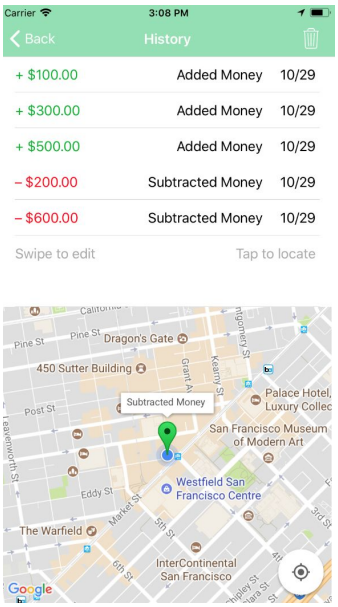
3.12- History of Transactions in Category Test

Location of Test: SanityTests/SanityTests.swift, Line 459

Description of what the test case does: This test to ensure that when transactions are created/deleted, the user has the ability to track where the flow of money is going. This is an added feature not overtly specified by Sarah Cooney, but our team felt was necessary to include as users are going to need to see which categories their money is going to. This also ensures that budgets are consistent with the transactions associated with them. To navigate to this page, assume that the user has created a budget and has pressed 'history' which is found when a user decides to add or subtract from a **category**. For this test case, assume there are transactions, and we are adding: $100, $300, $500, and subtracting: $200 and $600. The resulting history section should list out all of the aforementioned transactions.



Result of Test Case:



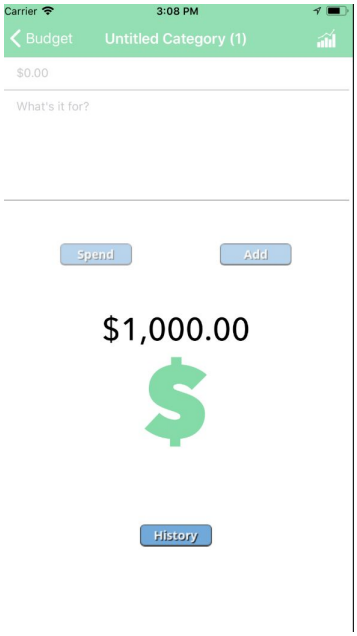
As you can see in the representational view,

Any bugs discovered: No bugs were found when testing this as far as we know.

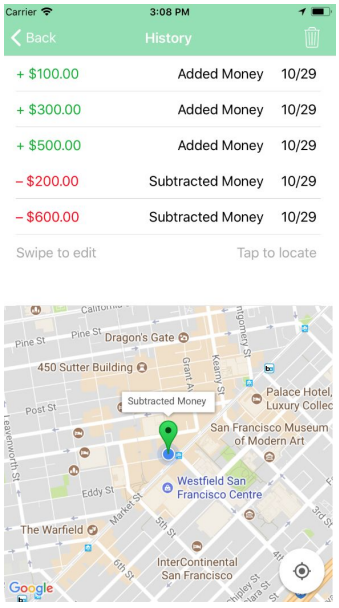
3.13 - Transaction Location Map Test

Location of Test: SanityTests/SanityTests.swift, Line 557

Description of what the test case does: This is to test that when transactions are made, the location is represented on a map where the history of transaction is located. This is an added feature not overtly specified by Sarah Cooney, but our team felt was necessary to include as users are going to need to see *where* their money is spent in case they are traveling around the world. To navigate to this page, assume that the user has created a budget and has pressed 'history' which is found when a user decides to add or subtract a budget. For our test example, we will be adding amounts at various locations to our budget using a feature in XCode that allows you to change your location on your iPhone. Assume we are using the values from Section 3.12- History of Transactions in Category Test.



Result of Test Case:



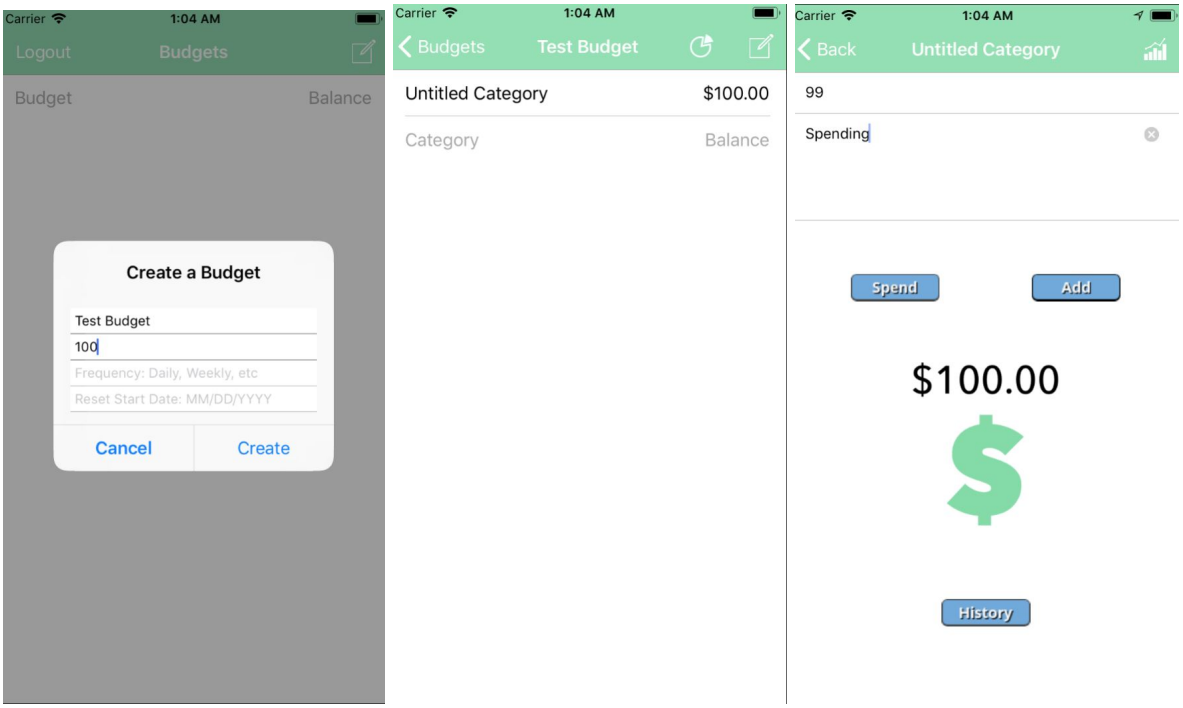
This is the representational view which is the same values we inputted in Section 3.12-History of Transactions in Category Test. The above tests are saved to San Francisco, but the test worked for money in other place such as Mexico City. This can be done with creating a virtual location which is a feature that XCode that allows you to use.

Any bugs discovered: No bugs were found when testing this as far as we know.

3.14 - Transaction Limit Notification Test

Location of Test: SanityTests/SanityTests.swift, Line 658

Description of what the test case does: This test to ensure that when a user is close to their transaction limit, an in app notification is popped up in their application. This is to prevent users from going over their allocated budget amount and to help the customer better track their budget amount. To navigate to this page, assume that a budget is created, and that the user pressed the budget icon. For this test, assume that a user only has a $100 in their account and $99 is spent from their budget.



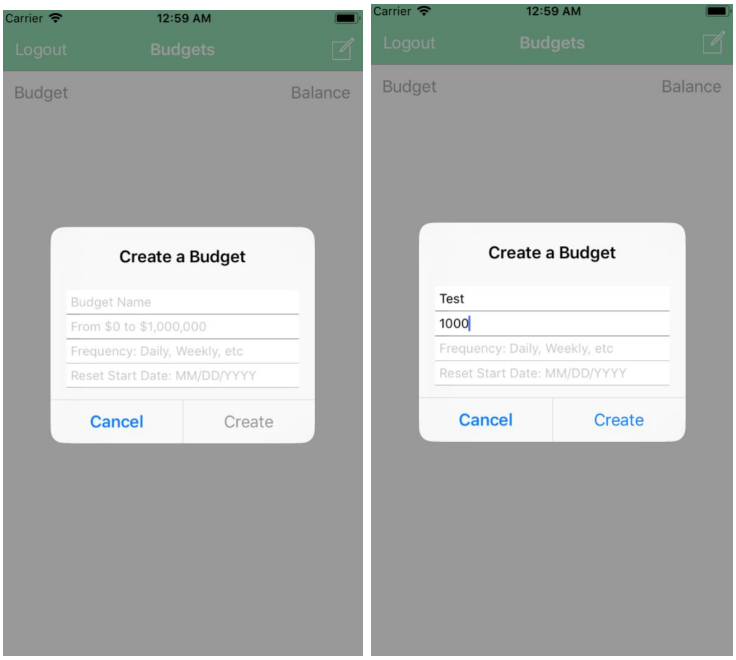
Result of Test Case: The representation pop-up does not show up on the application which isn't what we're looking for but that's okay! The error was actually printed out in the backend, though since this is black box testing we don't know that. In theory the pop-up will be connected in the future.

Any bugs discovered: No bugs were found when testing this as far as we know, aside from the comments made up above in the 'result of test case'.

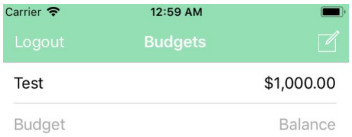
3.15 - Create Budget Successfully

Location of Test: SanityTests/SanityTests.swift, Line 756

Description of what the test case does: This test is is to ensure that a budget is created successfully when the user inputs the fields necessary in the alert that pops up when clicking the “Create” button on the top right of the Budget View. This will guarantee that the new budget will be appear in the view when a user wishes to create a new one. For this test, assume that the budget name is “Test”, the budget amount is $1000, and the reset frequency and reset date are left empty.



Result of Test Case:



The test worked as functioned, creating a budget which was added to the list of budgets on the budget landing page.

Any bugs discovered: No bugs were found when testing this as far as we know.

4. White Box Tests

White box tests refer to tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). Test cases are built around knowledge of the internal system and how it interacts with other components of the applications. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test.

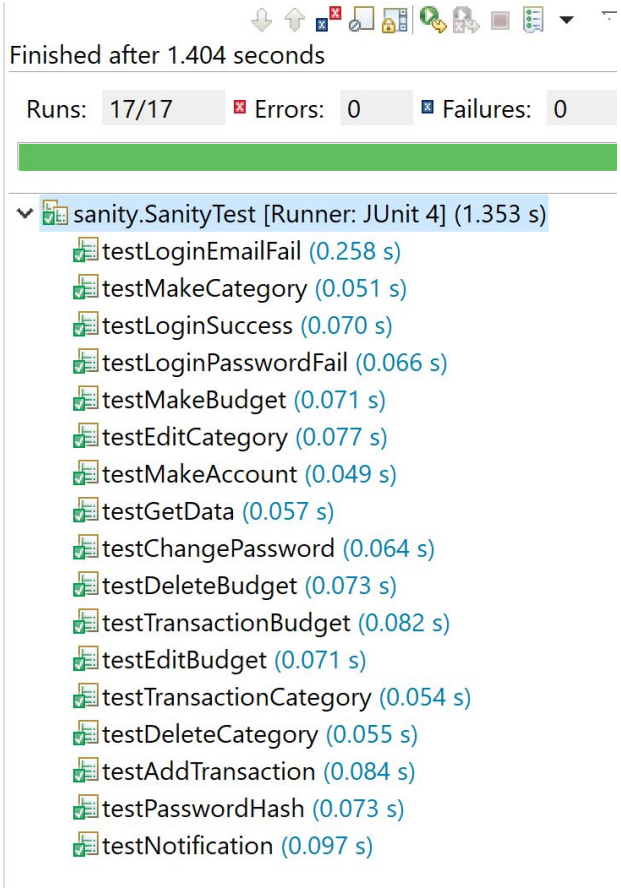


Figure 4.0 - The above screenshot shows the success of all 17 White Box tests described below, the line number refers to the line in the document below.

4.1 - Database Updates with New User

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 39

Coverage Area & Objective: A new user is signed up and then it is confirmed the profile is added to the database for a successful test.

Description Of Test Case & Rationale: A user account is created and added to the database and then a query to TotalUsers is executed to check if one exists. The rationale of having this test is to ensure that users can make an account if they aren't already a part of the application, which is a core functionality that was specified by Sarah Cooney on **October 23rd, 2017**.

```
38    @Test
39    public void testMakeAccount() {
40        sanity.deleteAll(conn);
41        JSONObject t = new JSONObject();
42        try {
43            t.put("message", "signup");
44            t.put("firstname", "Will");
45            t.put("lastname", "Wang");
46            t.put("email", "will@usc.edu");
47            t.put("password", "will");
48            JSONObject r = sanity.signUp(t, conn);
49            Statement st = conn.createStatement();
50            ResultSet rs = st.executeQuery("SELECT * FROM TotalUsers WHERE Email = 'will@usc.edu';");
51            boolean next = rs.next();
52            assertEquals(true, next);
53        } catch (JSONException | SQLException | NullPointerException e) {
54            // TODO Auto-generated catch block
55            e.printStackTrace();
56        }
57    }
```

Result of test case(s): **Success**. Profiles were created and added to the database once the SQL statement added the user to the backend.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.2 - Edit Password Updates SQL

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 59

Coverage Area & Objective: A user changes their password and then it is confirmed that it is changed in the database for a successful test.

Description Of Test Case & Rationale: An account is made and then the password is edited and sent to the database. The profile is then queried from the database and the password is checked against the new password. The rationale for including this test is that we need to ensure that users are capable of changing their passwords for security reasons and also in the event that they don't remember their password. We used William Wang and his information for this test and switched his password from 'will' (line 68) to 'test' (line 72)

```
58      @Test
59      public void testChangePassword() {
60          sanity.deleteAll(conn);
61          JSONObject t = new JSONObject();
62          JSONObject t1 = new JSONObject();
63          try {
64              t.put("message", "signup");
65              t.put("firstname", "Will");
66              t.put("lastname", "Wang");
67              t.put("email", "will@usc.edu");
68              t.put("password", "will");
69              JSONObject r = sanity.signUp(t, conn);
70              t1.put("message", "changePassword");
71              t1.put("email", "will@usc.edu");
72              t1.put("newPassword", "test");
73              JSONObject r1 = sanity.editProfile(t1, null, conn);
74              Statement st = conn.createStatement();
75              ResultSet rs = st.executeQuery("SELECT * FROM TotalUsers WHERE Email = 'will@usc.edu';");
76              if (rs.next()) {
77                  assertEquals(sanity.hash("test"), rs.getInt("Password"));
78              }
79          } catch (JSONException | SQLException | NullPointerException e) {
80              // TODO Auto-generated catch block
81              e.printStackTrace();
82          }
83      }
```

Result of test case(s): **Success**. The password changed successfully in the database and the test returns successfully when run.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.3 - Budget Added to SQL Database

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 85

Coverage Area & Objective: A user adds a budget and then it is confirmed that it is added in the database for a successful test.

Description Of Test Case & Ratinoale: A user is created and then a budget is created. The budget is then queried from the database to check if it exists. The rationale for including this test is that we need to sure that users are able to enter budgets and that they are saved in the database so when they go back to their budget list, we can easily provide updated information to the user. In this test, we used William Wang and his information for this test, having him create a budget on Line 96 and then sending a JSON to be sent to the backend which will signal the SQL statement to save the information to the database.

```
84    @Test
85    public void testMakeBudget() {
86        sanity.deleteAll(conn);
87        JSONObject t = new JSONObject();
88        JSONObject t1 = new JSONObject();
89        try {
90            t.put("message", "signup");
91            t.put("firstname", "Will");
92            t.put("lastname", "Wang");
93            t.put("email", "will@usc.edu");
94            t.put("password", "will");
95            JSONObject r = sanity.signUp(t, conn);
96            t1.put("message", "createBigBudget");
97            t1.put("userID", 1);
98            t1.put("bigBudgetAmount", 100);
99            t1.put("bigBudgetName", "Fun");
100           JSONObject r1 = sanity.createBigBudget(t1, null, conn);
101           Statement st = conn.createStatement();
102           ResultSet rs = st.executeQuery("SELECT * FROM BigBudgets WHERE userID = 1;");
103           assertEquals(true, rs.next());
104       } catch (JSONException | SQLException | NullPointerException e) {
105           // TODO Auto-generated catch block
106           e.printStackTrace();
107       }
108   }
109   @Test
```

Result of test case(s): **Success**. The budget is added successfully to the database after line 102 is completed.

Bugs Uncovered: SQL syntax was wrong but then fixed for the correct key constraints once the error was made known. No regression test was necessary as the SQL statement is only making one call to the database (the Query statement on line 102) which should not be causing issues with any other part of the code.

4.4 - Category Added to SQL Database

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 110

Coverage Area & Objective: A user adds a category and then it is confirmed that it is added in the database for a successful test.

Description Of Test Case & Rationale: A user is created, a budget is created, and then a category is created. The category is then queried from the database to check if it exists. We need to ensure that the core functionality of adding a category to the SQL database is appropriately saved. This was specified by Sarah Cooney in a meeting on **October 23rd 2017** and such serves as as something that needs to be tested on the Java backend. For

this test we will adding a test category under William Wang's account. We have to make sure that a budget is created (line 122) before a category is created (line 127).

```
109     @Test
110     public void testMakeCategory() {
111         sanity.deleteAll(conn);
112         JSONObject t = new JSONObject();
113         JSONObject t1 = new JSONObject();
114         JSONObject t2 = new JSONObject();
115         try {
116             t.put("message", "signup");
117             t.put("firstname", "Will");
118             t.put("lastname", "Wang");
119             t.put("email", "will@usc.edu");
120             t.put("password", "will");
121             JSONObject r = sanity.signUp(t, conn);
122             t1.put("message", "createBigBudget");
123             t1.put("userID", 1);
124             t1.put("bigBudgetAmount", 100);
125             t1.put("bigBudgetName", "Fun");
126             JSONObject r1 = sanity.createBigBudget(t1, null, conn);
127             t2.put("message", "createBudget");
128             t2.put("bigBudgetID", 1);
129             t2.put("userID", 1);
130             t2.put("budgetAmount", 50);
131             t2.put("budgetName", "More fun");
132             JSONObject r2 = sanity.createBudget(t2, null, conn);
133             Statement st = conn.createStatement();
134             ResultSet rs = st.executeQuery("SELECT * FROM Budgets WHERE bigBudgetID = 1;");
135             assertEquals(true, rs.next());
136         } catch (JSONException | SQLException | NullPointerException e) {
137             // TODO Auto-generated catch block
138             e.printStackTrace();
139         }
140     }
```

Result of test case(s): **Success**. The category is added and connected to the budget successfully on the database. This test is reflected in the SQL database after the test is run.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.5 - Budget Removed from SQL

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 142

Coverage Area & Objective: A user deletes a budget and then it is confirmed that it and its related categories are removed from the database for a successful test.

Description Of Test Case & Rationale: A user is created, a budget is created, and then the budget is deleted. The database is then queried to check if any budgets still exist in the database. This rationale for including this test is to ensure that when a budget is created, it

can be appropriately deleted from the front end user and saved in the backend when a user logs back into the application. For this test, we will be signing William Wang up, and they creating a budget only for it to be deleted (line 159).

```
141     @Test
142     public void testDeleteBudget() {
143         sanity.deleteAll(conn);
144         JSONObject t = new JSONObject();
145         JSONObject t1 = new JSONObject();
146         JSONObject t2 = new JSONObject();
147         try {
148             t.put("message", "signup");
149             t.put("firstname", "Will");
150             t.put("lastname", "Wang");
151             t.put("email", "will@usc.edu");
152             t.put("password", "will");
153             JSONObject r = sanity.signUp(t, conn);
154             t1.put("message", "createBigBudget");
155             t1.put("userID", 1);
156             t1.put("bigBudgetAmount", 100);
157             t1.put("bigBudgetName", "Fun");
158             JSONObject r1 = sanity.createBigBudget(t1, null, conn);
159             t2.put("message", "deleteBigBudget");
160             t2.put("userID", 1);
161             t2.put("bigBudgetID", 1);
162             JSONObject r2 = sanity.deleteBigBudget(t2, null, conn);
163             Statement st = conn.createStatement();
164             ResultSet rs = st.executeQuery("SELECT * FROM BigBudgets;");
165             assertEquals(false, rs.next());
166         } catch (JSONException | SQLException | NullPointerException e) {
167             // TODO Auto-generated catch block
168             e.printStackTrace();
169         }
170     }
```

Result of test case(s): **Success**. The budget and its categories are removed successfully and are reflected in the database.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.6 - Category Removed from SQL Database

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 172

Coverage Area & Objective: A user removes a category and then it is confirmed that it is removed from the database for a successful test.

Description Of Test Case & Rationale: A user is created, a budget is created, a category is created, and then the category is deleted. The database is then queried to confirm that the

category does not exist. Similar to the test in Section 4.5 this rationale for including this test is to ensure that when a category is created, it can be appropriately deleted from the front end user and saved in the backend when a user logs back into the application. For this test, we will be signing William Wang up, and they creating a budget (line 185) and a category (line 190) only for it to be deleted on line 196.

```
171    @Test
172    public void testDeleteCategory() {
173        sanity.deleteAll(conn);
174        JSONObject t = new JSONObject();
175        JSONObject t1 = new JSONObject();
176        JSONObject t2 = new JSONObject();
177        JSONObject t3 = new JSONObject();
178        try {
179            t.put("message", "signup");
180            t.put("firstname", "Will");
181            t.put("lastname", "Wang");
182            t.put("email", "will@usc.edu");
183            t.put("password", "will");
184            JSONObject r = sanity.signUp(t, conn);
185            t1.put("message", "createBigBudget");
186            t1.put("userID", 1);
187            t1.put("bigBudgetAmount", 100);
188            t1.put("bigBudgetName", "Fun");
189            JSONObject r1 = sanity.createBigBudget(t1, null, conn);
190            t2.put("message", "createBudget");
191            t2.put("bigBudgetID", 1);
192            t2.put("userID", 1);
193            t2.put("budgetAmount", 50);
194            t2.put("budgetName", "More fun");
195            JSONObject r2 = sanity.createBudget(t2, null, conn);
196            t3.put("message", "deleteBudget");
197            t3.put("budgetID", 1);
198            t3.put("userID", 1);
199            JSONObject r3 = sanity.deleteBudget(t3, null, conn);
200            Statement st = conn.createStatement();
201            ResultSet rs = st.executeQuery("SELECT * FROM Budgets");
202            assertEquals(false, rs.next());
203        } catch (JSONException | SQLException | NullPointerException e) {
204            // TODO Auto-generated catch block
205            e.printStackTrace();
206        }
207    }
```

Result of test case(s): **Success**. The category is removed successfully and updated in the database.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.7 - Transaction Added to SQL Database

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 209

Coverage Area & Objective: A user adds a transaction and then it is confirmed that it is added in the database for a successful test.

Description Of Test Case & Rationale: A user is created, a budget is created, a category is created, and then a transaction is created. The database is then queried to check that the transaction is added to the database. The rationale for this test is to ensure that when a user conducts a transaction, that it is properly saved to the database and can be pulled up when the user logs into the history section of the their transactions. For this test, we will be logging in as William Wang and then creating a budget (line 224) and category (line 227), only for the budget to have a value of 40 on line 236.

```
208     @Test
209     public void testAddTransaction() {
210         sanity.deleteAll(conn);
211         JSONObject t = new JSONObject();
212         JSONObject t1 = new JSONObject();
213         JSONObject t2 = new JSONObject();
214         JSONObject t3 = new JSONObject();
215         try {
216             t.put("message", "signup");
217             t.put("firstname", "Will");
218             t.put("lastname", "Wang");
219             t.put("email", "will@usc.edu");
220             t.put("password", "will");
221             JSONObject r = sanity.signUp(t, conn);
222             t1.put("message", "createBigBudget");
223             t1.put("userID", 1);
224             t1.put("bigBudgetAmount", 100);
225             t1.put("bigBudgetName", "Fun");
226             JSONObject r1 = sanity.createBigBudget(t1, null, conn);
227             t2.put("message", "createBudget");
228             t2.put("bigBudgetID", 1);
229             t2.put("userID", 1);
230             t2.put("budgetAmount", 50);
231             t2.put("budgetName", "More fun");
232             JSONObject r2 = sanity.createBudget(t2, null, conn);
233             t3.put("message", "addTransaction");
234             t3.put("budgetID", 1);
235             t3.put("userID", 1);
236             t3.put("amountToAdd", 40);
237             JSONObject r3 = sanity.addTransaction(t3, null, conn);
238             Statement st = conn.createStatement();
239             ResultSet rs = st.executeQuery("SELECT * FROM Transactions;");
240             assertEquals(true, rs.next());
241         } catch (JSONException | SQLException | NullPointerException e) {
242             // TODO Auto-generated catch block
243             e.printStackTrace();
244         }
245     }
```

Result of test case(s): **Success**. The amount of $40 is added to the database successfully and passes the test cases.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.8 - Password Properly Hashed

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 247

Coverage Area & Objective: A user enters a password and it must be stored correctly in its hashed form for a successful test.

Description Of Test Case & Rationale: A user is created and we log in with a password. The database is then queried and the password is selected from the user profile in the database to confirm that the entered hashed password is the same as the stored hashed password. The rationale of including this test is to ensure that there is security when passwords are being sent over the websocket and prevents issues regarding 'Man-in-the-middle' cyber security attacks. For this test we will be logging in as William Wang and retrieving his hashed password, hashing it, and checking to see if they match.

```
246     @Test
247     public void testPasswordHash() {
248         sanity.deleteAll(conn);
249         JSONObject t = new JSONObject();
250         try {
251             t.put("message", "signup");
252             t.put("firstname", "Will");
253             t.put("lastname", "Wang");
254             t.put("email", "will@usc.edu");
255             t.put("password", "will");
256             JSONObject r = sanity.signUp(t, conn);
257             Statement st = conn.createStatement();
258             ResultSet rs = st.executeQuery("SELECT * FROM TotalUsers");
259             if (rs.next()) {
260                 assertEquals(sanity.hash("will"), rs.getInt("Password"));
261             }
262         } catch (JSONException | SQLException | NullPointerException e) {
263             // TODO Auto-generated catch block
264             e.printStackTrace();
265         }
266     }
```

Result of test case(s): **Success**. The hashed password is added to the database properly.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.9 - Login Password Fail

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 268

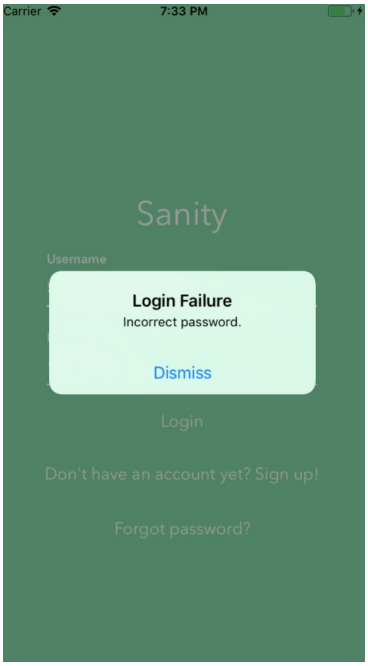
Coverage Area & Objective: A user enters a wrong password and a failed login message must be returned for a successful test.

Description Of Test Case & Rationale: A user is created and then tries to login with a different password than what it was created with. The returned JSONObject message is checked to confirm that a login fail was returned. The rationale for including this test is to ensure that when a user logs in with an incorrect password, a message is sent to the front

end XCode document to print out the login failure message. For this test we will be signing William Wang up as a user with password **will** (line 277), and then trying to sign in with password: **wang** (line 281) which should cause a login failure error.

```
267     @Test
268     public void testLoginPasswordFail() {
269         sanity.deleteAll(conn);
270         JSONObject t = new JSONObject();
271         JSONObject t1 = new JSONObject();
272         try {
273             t.put("message", "signup");
274             t.put("firstname", "Will");
275             t.put("lastname", "Wang");
276             t.put("email", "will@usc.edu");
277             t.put("password", "will");
278             JSONObject r = sanity.signUp(t, conn);
279             t1.put("message", "login");
280             t1.put("email", "will@usc.edu");
281             t1.put("password", "wang");
282             JSONObject r2 = sanity.signIn(t1, null, conn);
283             assertEquals("loginfail", r2.getString("message"));
284         } catch (JSONException | NullPointerException e) {
285             // TODO Auto-generated catch block
286             e.printStackTrace();
287         }
288     }
```

Result of test case(s): Success. A wrong password returns a login failure on the application.



Bugs Uncovered: No bugs were found when testing this as far as we know.

4.10 - Login Email Fail

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 290

Coverage Area & Objective: A user enters a wrong email and a login failure message must be returned for a successful test.

Description Of Test Case & Rationale: A user is created and then tries to login with a different email than what it was created with. The returned JSONObject message is checked to confirm that a login fail was returned. Similar to the test in Section 4.11, we need to ensure that users have to enter the correct email that was specified in the database, otherwise they won't be able to login. This is a key feature in preventing other users from hacking into someone's account if they don't have the appropriate login credentials.

```
289     @Test
290     public void testLoginEmailFail() {
291         sanity.deleteAll(conn);
292         JSONObject t = new JSONObject();
293         JSONObject t1 = new JSONObject();
294         try {
295             t.put("message", "signup");
296             t.put("firstname", "Will");
297             t.put("lastname", "Wang");
298             t.put("email", "will@usc.edu");
299             t.put("password", "will");
300             JSONObject r = sanity.signUp(t, conn);
301             t1.put("message", "login");
302             t1.put("email", "wang@usc.edu");
303             t1.put("password", "will");
304             JSONObject r2 = sanity.signIn(t1, null, conn);
305             assertEquals("loginfail", r2.getString("message"));
306         } catch (JSONException | NullPointerException e) {
307             // TODO Auto-generated catch block
308             e.printStackTrace();
309         }
310     }
```

Result of test case(s): **Success**. A login fail message is returned for a successful test run on line 305.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.11- Correct Login Test

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 312

Coverage Area & Objective: A user logs in correctly and a login success message must be returned for a successful test.

Description Of Test Case & Rationale: A user is created and then a correct login is executed. The returned JSONObject message is checked to see if it is a correct login message. The rationale is to ensure that users are able to correctly login with their correct credentials. This is a core requirement specified by Sarah Cooney on **October 23rd, 2017** and shall be tested as such. For this test we will be signing up William Wang as a user and signing him up on Line 326. The login success message is then printed out thereafter.

```
311    @Test
312    public void testLoginSuccess() {
313        sanity.deleteAll(conn);
314        JSONObject t = new JSONObject();
315        JSONObject t1 = new JSONObject();
316        try {
317            t.put("message", "signup");
318            t.put("firstname", "Will");
319            t.put("lastname", "Wang");
320            t.put("email", "will@usc.edu");
321            t.put("password", "will");
322            JSONObject r = sanity.signUp(t, conn);
323            t1.put("message", "login");
324            t1.put("email", "will@usc.edu");
325            t1.put("password", "will");
326            JSONObject r2 = sanity.signIn(t1, null, conn);
327            assertEquals("loginsuccess", r2.getString("message"));
328        } catch (JSONException | NullPointerException e) {
329            // TODO Auto-generated catch block
330            e.printStackTrace();
331        }
332    }
```

Result of test case(s): **Success**. Logging in as an existing user returns a login success message for a successful test.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.12 - Budget Notification From Adding Transaction Test

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 334

Coverage Area & Objective: A user adds two transactions that sum to more than 80% of the budget's limit and a notification message must be sent for a successful test.

Description Of Test Case: A user is created, a budget is created, a category is created, and two transactions are created. The sum of the two transaction amounts is greater than 80% of the budget amount. The returned JSONObject message from the last transaction is checked to see that a notification message is added. The rationale for including this test is to be sure that users are able to add transactions to budgets and are notified that the sum of more than 80% of the budget is spent.

```
333     @Test
334     public void testNotification() {
335         sanity.deleteAll(conn);
336         JSONObject t = new JSONObject();
337         JSONObject t1 = new JSONObject();
338         JSONObject t2 = new JSONObject();
339         JSONObject t3 = new JSONObject();
340         JSONObject t4 = new JSONObject();
341         JSONObject t5 = new JSONObject();
342         try {
343             t.put("message", "signup");
344             t.put("firstname", "Will");
345             t.put("lastname", "Wang");
346             t.put("email", "will@usc.edu");
347             t.put("password", "will");
348             JSONObject r = sanity.signUp(t, conn);
349             t1.put("message", "createBigBudget");
350             t1.put("userID", 1);
351             t1.put("bigBudgetAmount", 100);
352             t1.put("bigBudgetName", "Fun");
353             JSONObject r1 = sanity.createBigBudget(t1, null, conn);
354             t2.put("message", "createBudget");
355             t2.put("bigBudgetID", 1);
356             t2.put("userID", 1);
357             t2.put("budgetAmount", 50);
358             t2.put("budgetName", "More fun");
359             JSONObject r2 = sanity.createBudget(t2, null, conn);
360             t3.put("message", "addTransaction");
361             t3.put("budgetID", 1);
362             t3.put("userID", 1);
363             t3.put("amountToAdd", 40);
364             JSONObject r3 = sanity.addTransaction(t3, null, conn);
365             t4.put("message", "createBudget");
366             t4.put("bigBudgetID", 1);
367             t4.put("userID", 1);
368             t4.put("budgetAmount", 50);
369             t4.put("budgetName", "More fun");
370             JSONObject r4 = sanity.createBudget(t4, null, conn);
371             t5.put("message", "addTransaction");
372             t5.put("budgetID", 1);
373             t5.put("userID", 1);
374             t5.put("amountToAdd", 41);
375             JSONObject r5 = sanity.addTransaction(t5, null, conn);
376             assertEquals("yes", r5.getString("notification"));
377         } catch (JSONException | NullPointerException e) {
378             // TODO Auto-generated catch block
379             e.printStackTrace();
380         }
381     }
```

Result of test case(s): **Success**. A notification message is sent when more than 80% of the budget is used to the user on the iPhone application front-end.

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.13 - Budget Edited in SQL Database

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 383

Coverage Area & Objective: A user edits a budget and must be edited in the database for a successful test.

Description Of Test Case: A user is created, a budget is created, and then the budget is edited. The database is then queried and the budget is checked to see if the fields are updated in the database. The rationale for conducting this test is to ensure that the budget can be edited on the database and can be dynamically called forward for the user whenever they log in and out.

```
382     @Test
383     public void testEditBudget() {
384         sanity.deleteAll(conn);
385         JSONObject t = new JSONObject();
386         JSONObject t1 = new JSONObject();
387         JSONObject t2 = new JSONObject();
388         try {
389             t.put("message", "signup");
390             t.put("firstname", "Will");
391             t.put("lastname", "Wang");
392             t.put("email", "will@usc.edu");
393             t.put("password", "will");
394             JSONObject r = sanity.signUp(t, conn);
395             t1.put("message", "createBigBudget");
396             t1.put("userID", 1);
397             t1.put("bigBudgetAmount", 100);
398             t1.put("bigBudgetName", "Fun");
399             JSONObject r1 = sanity.createBigBudget(t1, null, conn);
400             t2.put("message", "editBigBudget");
401             t2.put("budgetID", 1);
402             t2.put("budgetName", "Edited");
403             t2.put("budgetAmount", 300);
404             JSONObject r2 = sanity.editBigBudget(t2, null, conn);
405             Statement st = conn.createStatement();
406             ResultSet rs = st.executeQuery("SELECT * FROM BigBudgets;");
407             if (rs.next()) {
408                 assertEquals("Edited", rs.getString("BigBudgetName"));
409             }
410         } catch (JSONException | SQLException | NullPointerException e) {
411             // TODO Auto-generated catch block
412             e.printStackTrace();
413         }
414     }
```

Result of test case(s): **Success**. The budget is edited in the database and a successful response is showcased on the testing screen.
Bugs Uncovered: SQL syntax needed to be fixed for an Update statement. No regression test was necessary as the SQL statement is only making one call to the database which should not be causing issues with any other part of the code.

4.14 - Category Edited in SQL Database

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 416
Coverage Area & Objective: A user edits a category and it must be edited in the database for a successful test.
*Description Of Test Case:*A user is created, a budget is created, a category is created, and then the category is edited. The database is then queried to check that the category fields are the same as the edited fields that were passed in. The rationale for including this test is to make sure that, similarly to budgets, that categories can be edited and dynamically updated on the database in case a user needs to logout and in. For this test we will be logging William wang as a user and having him edit a category in the database.

```
415     @Test
416     public void testEditCategory() {
417         sanity.deleteAll(conn);
418         JSONObject t = new JSONObject();
419         JSONObject t1 = new JSONObject();
420         JSONObject t2 = new JSONObject();
421         JSONObject t3 = new JSONObject();
422         try {
423             t.put("message", "signup");
424             t.put("firstname", "Will");
425             t.put("lastname", "Wang");
426             t.put("email", "will@usc.edu");
427             t.put("password", "will");
428             JSONObject r = sanity.signUp(t, conn);
429             t1.put("message", "createBigBudget");
430             t1.put("userID", 1);
431             t1.put("bigBudgetAmount", 100);
432             t1.put("bigBudgetName", "Fun");
433             JSONObject r1 = sanity.createBigBudget(t1, null, conn);
434             t2.put("message", "createBudget");
435             t2.put("bigBudgetID", 1);
436             t2.put("userID", 1);
437             t2.put("budgetAmount", 50);
438             t2.put("budgetName", "More fun");
439             JSONObject r2 = sanity.createBudget(t2, null, conn);
440             t3.put("message", "editBudget");
441             t3.put("userID", 1);
442             t3.put("budgetID", 1);
443             t3.put("budgetName", "Less Fun");
444             t3.put("budgetAmount", 100);
445             JSONObject r3 = sanity.editBudget(t3, null, conn);
446             Statement st = conn.createStatement();
447             ResultSet rs = st.executeQuery("SELECT * FROM Budgets;");
448             if (rs.next()) {
449                 assertEquals("Less Fun", rs.getString("BudgetName"));
450             }
451         } catch (JSONException | SQLException | NullPointerException e) {
452             // TODO Auto-generated catch block
453             e.printStackTrace();
454         }
455     }
```

Result of test case(s): Success. The category is edited in the database and a successful test is printed on the Java test file.

Bugs Uncovered: SQL syntax needed to be fixed for an Update statement. No regression test was necessary as the SQL statement is only making one call to the database which should not be causing issues with any other part of the code.

4.15 - Transaction Added Changes Budget Test

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 457

Coverage Area & Objective: A user adds a transaction and the budget must change in the database for a successful test.

Description Of Test Case: A user is created, a budget is created, a category is created, and a transaction is created. The database is then queried for the corresponding budget to check that the TotalAmountSpent is increased based on the transaction. This test is important because we need to ensure that when a transaction is added to the total budget amount, that it is accurately updated on the SQL database.

```
456     @Test
457     public void testTransactionBudget() {
458         sanity.deleteAll(conn);
459         JSONObject t = new JSONObject();
460         JSONObject t1 = new JSONObject();
461         JSONObject t2 = new JSONObject();
462         JSONObject t3 = new JSONObject();
463         try {
464             t.put("message", "signup");
465             t.put("firstname", "Will");
466             t.put("lastname", "Wang");
467             t.put("email", "will@usc.edu");
468             t.put("password", "will");
469             JSONObject r = sanity.signUp(t, conn);
470             t1.put("message", "createBigBudget");
471             t1.put("userID", 1);
472             t1.put("bigBudgetAmount", 100);
473             t1.put("bigBudgetName", "Fun");
474             JSONObject r1 = sanity.createBigBudget(t1, null, conn);
475             t2.put("message", "createBudget");
476             t2.put("bigBudgetID", 1);
477             t2.put("userID", 1);
478             t2.put("budgetAmount", 50);
479             t2.put("budgetName", "More fun");
480             JSONObject r2 = sanity.createBudget(t2, null, conn);
481             t3.put("message", "addTransaction");
482             t3.put("budgetID", 1);
483             t3.put("userID", 1);
484             t3.put("amountToAdd", 40);
485             JSONObject r3 = sanity.addTransaction(t3, null, conn);
486             Statement st = conn.createStatement();
487             ResultSet rs = st.executeQuery("SELECT * FROM BigBudgets;");
488             if (rs.next()) {
489                 assertEquals(Double.toString(40), Double.toString(rs.getDouble("TotalAmountSpent")));
490             }
491         } catch (JSONException | SQLException | NullPointerException e) {
492             // TODO Auto-generated catch block
493             e.printStackTrace();
494         }
495     }
```

Result of test case(s): **Success**. The budget's TotalAmountSpent is increased when a transaction is added!

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.16 - Transaction Added Changes Category Test

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 497

Coverage Area & Objective: A user adds a transaction and the category's TotalAmountSpent must be increased in the database for a successful test.

Description Of Test Case: A user is created, a budget is created, a category is created, and a transaction is created. The database is then queried for the corresponding category to check that the TotalAmountSpent is increased based on the transaction. This test is necessary because we need to ensure that categories can have transactions dynamically added to them.

```
496     @Test
497     public void testTransactionCategory() {
498         sanity.deleteAll(conn);
499         JSONObject t = new JSONObject();
500         JSONObject t1 = new JSONObject();
501         JSONObject t2 = new JSONObject();
502         JSONObject t3 = new JSONObject();
503         try {
504             t.put("message", "signup");
505             t.put("firstname", "Will");
506             t.put("lastname", "Wang");
507             t.put("email", "will@usc.edu");
508             t.put("password", "will");
509             JSONObject r = sanity.signUp(t, conn);
510             t1.put("message", "createBigBudget");
511             t1.put("userID", 1);
512             t1.put("bigBudgetAmount", 100);
513             t1.put("bigBudgetName", "Fun");
514             JSONObject r1 = sanity.createBigBudget(t1, null, conn);
515             t2.put("message", "createBudget");
516             t2.put("bigBudgetID", 1);
517             t2.put("userID", 1);
518             t2.put("budgetAmount", 50);
519             t2.put("budgetName", "More fun");
520             JSONObject r2 = sanity.createBudget(t2, null, conn);
521             t3.put("message", "addTransaction");
522             t3.put("budgetID", 1);
523             t3.put("userID", 1);
524             t3.put("amountToAdd", 40);
525             JSONObject r3 = sanity.addTransaction(t3, null, conn);
526             Statement st = conn.createStatement();
527             ResultSet rs = st.executeQuery("SELECT * FROM Budgets;");
528             if (rs.next()) {
529                 assertEquals(Double.toString(40), Double.toString(rs.getDouble("TotalAmountSpent")));
530             }
531         } catch (JSONException | SQLException | NullPointerException e) {
532             // TODO Auto-generated catch block
533             e.printStackTrace();
534         }
535     }
```

Result of test case(s): **Success**. The TotalAmountSpent is updated in the database!

Bugs Uncovered: No bugs were found when testing this as far as we know.

4.17 - Get User Data Test

Location of Test: SanityBackend1/src/sanity/SanityTest.java, Line 537

Coverage Area & Objective: Populating the user's application with their budget/category/transaction data from the database results in a successful test.

Description Of Test Case: A user is created, a budget is created, a category is created, and a transaction is created. The getData function is then called for the created user and the returned JSONObject is then checked against the expected JSONObject to be returned.

```
536     @Test
537     public void testGetData() {
538         sanity.deleteAll(conn);
539         JSONObject t = new JSONObject();
540         JSONObject t1 = new JSONObject();
541         JSONObject t2 = new JSONObject();
542         JSONObject t3 = new JSONObject();
543         try {
544             t.put("message", "signup");
545             t.put("firstname", "Will");
546             t.put("lastname", "Wang");
547             t.put("email", "will@usc.edu");
548             t.put("password", "will");
549             JSONObject r = sanity.signUp(t, conn);
550             t1.put("message", "createBigBudget");
551             t1.put("userID", 1);
552             t1.put("bigBudgetAmount", 100);
553             t1.put("bigBudgetName", "Fun");
554             JSONObject r1 = sanity.createBigBudget(t1, null, conn);
555             t2.put("message", "createBudget");
556             t2.put("bigBudgetID", 1);
557             t2.put("userID", 1);
558             t2.put("budgetAmount", 50);
559             t2.put("budgetName", "More fun");
560             JSONObject r2 = sanity.createBudget(t2, null, conn);
561             t3.put("message", "addTransaction");
562             t3.put("budgetID", 1);
563             t3.put("userID", 1);
564             t3.put("amountToAdd", 40);
565             JSONObject r3 = sanity.addTransaction(t3, null, conn);
566             JSONObject result = sanity.getData(conn, 1);
567             if (result.getInt("numBudgets") > 0) {
568                 JSONObject budget = new JSONObject();
569                 JSONObject category = new JSONObject();
570                 category.put("categoryAmount", 50);
571                 category.put("categoryName", "More fun");
572                 budget.put("category1", category);
573                 budget.put("size", 1);
574                 budget.put("budgetAmount", 100);
575                 budget.put("budgetName", "Fun");
576                 System.out.println(budget.toString() + result.getJSONObject("budget1").toString());
577                 assertEquals(budget.toString(), result.getJSONObject("budget1").toString());
578             }
579         } catch (JSONException | NullPointerException e) {
580             // TODO Auto-generated catch block
581             e.printStackTrace();
582         }
583     }
```

Result of test case(s): **Success**. The data is returned for the specific user's ID and is reflected on the testing document.

Bugs Uncovered: No bugs were found when testing this as far as we know.