Testing Document

CSCI-310 Software Engineering

Professor: **Nenad Medvidović**

TA: **Sarah Cooney**

Fall 2017

$a₦i₹¥

Ba$iL (TEAM 15)

Tri Nguyen 1410884747

Ang Li 3340862395

Utsav Thapa 9717428024

Andre Takhmazyan 8764629970

Kevin Nguyen 215820474

2. Preface

This testing document is intended for technical readers. It outlines our tests for the Android app “Sanity.” We have over 15 black box tests and over 15 white box tests, all with screenshots of the tests passing. We also outline the intention of each test and what it does.

3. Instruction

The Black-box testing must be done in Android Studio v3.0. UI Black-box tests are kept in the folder Sanity\app\src\androidTest\java\basil\sanity. These tests are grouped by the UI screen that they primarily test. For example, tests for login functionality are in LoginScreenTest.java. Some tests will involve multiple UI screens, such as adding a user and then logging in. Their naming convention, A_, B_, C_, is because Espresso will run the tests in alphabetical order. There are “tests” whose only job is to clear the database. Since they do not test UI, they are omitted in the Black-box tests section. The White-box testing are kept in the same folder. These tests are named have UnitTest as suffix to indicate that they are white box tests. Some of the tests are ordered and start with ordered letters to make them perform in turn. This is necessary as these tests deal with database and build on top of each other.

4. Black-box Tests

1. **A_nonexistentUsername():**

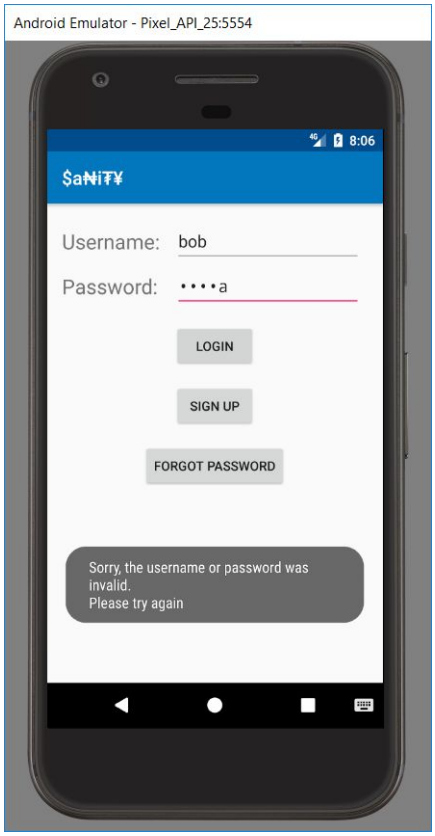
   i.
   Sanity\app\src\androidTest\java\basil\sanity\LoginScreenTest.java

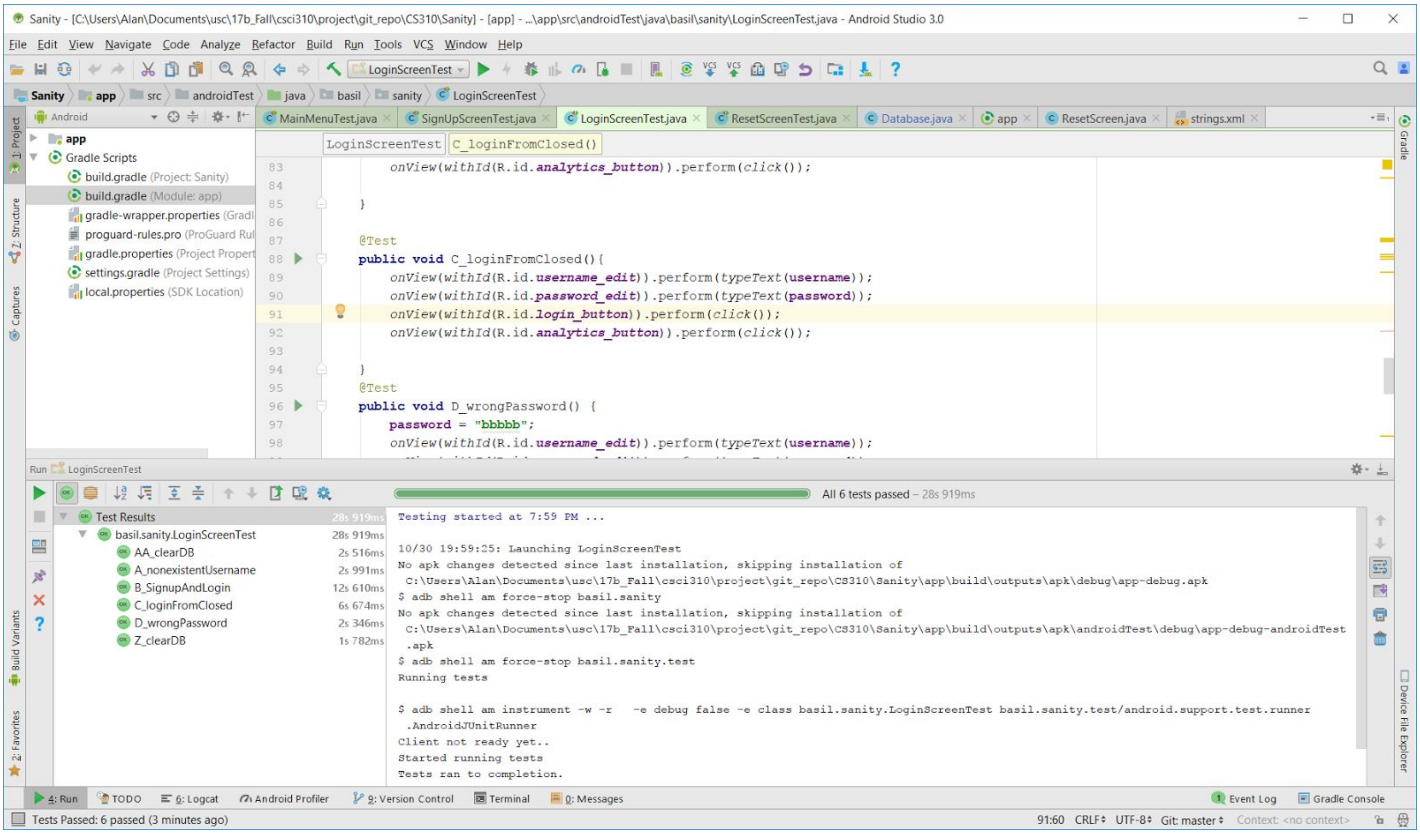
   ii.
   This test tries to login to the app before anyone has created an account. Previously, the database was cleared. The user should receive an error message notifying them that their username or password was invalid. This shows that the app will not allow users to log in without signing up first. To run this test, right click LoginScreenTest in Project and choose Run ‘LoginScreenTest.’

   iii.
   The inputs could have been any non-whitespace string for the username. The database is reset before the tests so the app should not permit anyone to log in. This test shows that users must sign up before logging in.

iv.

The test passes because the Toast message matches the correct message for invalid username or password when “Login” is clicked.





This screen shows that all tests in the LoginScreenTest class (1-4) passed.

2. B_SignupAndLogin():

i.

Sanity\app\src\androidTest\java\basil\sanity\LoginScreenTest.java

ii.

This test tries sign up a new user, logout, and log back in with the correct credentials. To run this test, right click LoginScreenTest in Project and choose Run 'LoginScreenTest.'

iii.

To signup, the username could have been any non-empty string since the database had no other users. The password must be at least 5 characters and entered twice. The security answer must be a non-empty string. To login again,

the username and password must be the same as the ones used to signup. This shows the sign up, log out, and log in functionality work.

iv.
The test passes because the username was able to login successfully and click the analytics button in the main menu.

3. **C_loginFromClosed():**

i.
Sanity\app\src\androidTest\java\basil\sanity\LoginScreenTest.java

ii.
This test tries to login to the app after opening the app again. The user had signed up in the previous test. To run this test, right click LoginScreenTest in Project and choose Run 'LoginScreenTest.'

iii.

The inputs could have been any non-whitespace string for username and password. The database is reset before the tests so the app should not permit anyone to log in. This shows that the database saved the user's information.

iv.
The test passes because the username was able to login successfully and click the analytics button in the main menu.

4. **D_wrongPassword():**

i.
Sanity\app\src\androidTest\java\basil\sanity\LoginScreenTest.java

ii.
This test tries to login with the correct username but wrong password. Therefore, the user should receive an error message notifying them that their username or password was invalid. To run this test, right click LoginScreenTest in Project and choose Run 'LoginScreenTest.'

iii.
The inputs must be a valid username but the incorrect password. The username that was used to sign up was used again. This test shows that the password verification is working.

iv.
The test passes because the Toast message matches the correct message for invalid username or password when "Login" is clicked.

5. **A_addUser():**

i.
Sanity\app\src\androidTest\java\basil\sanity\SignUpScreenTest.java

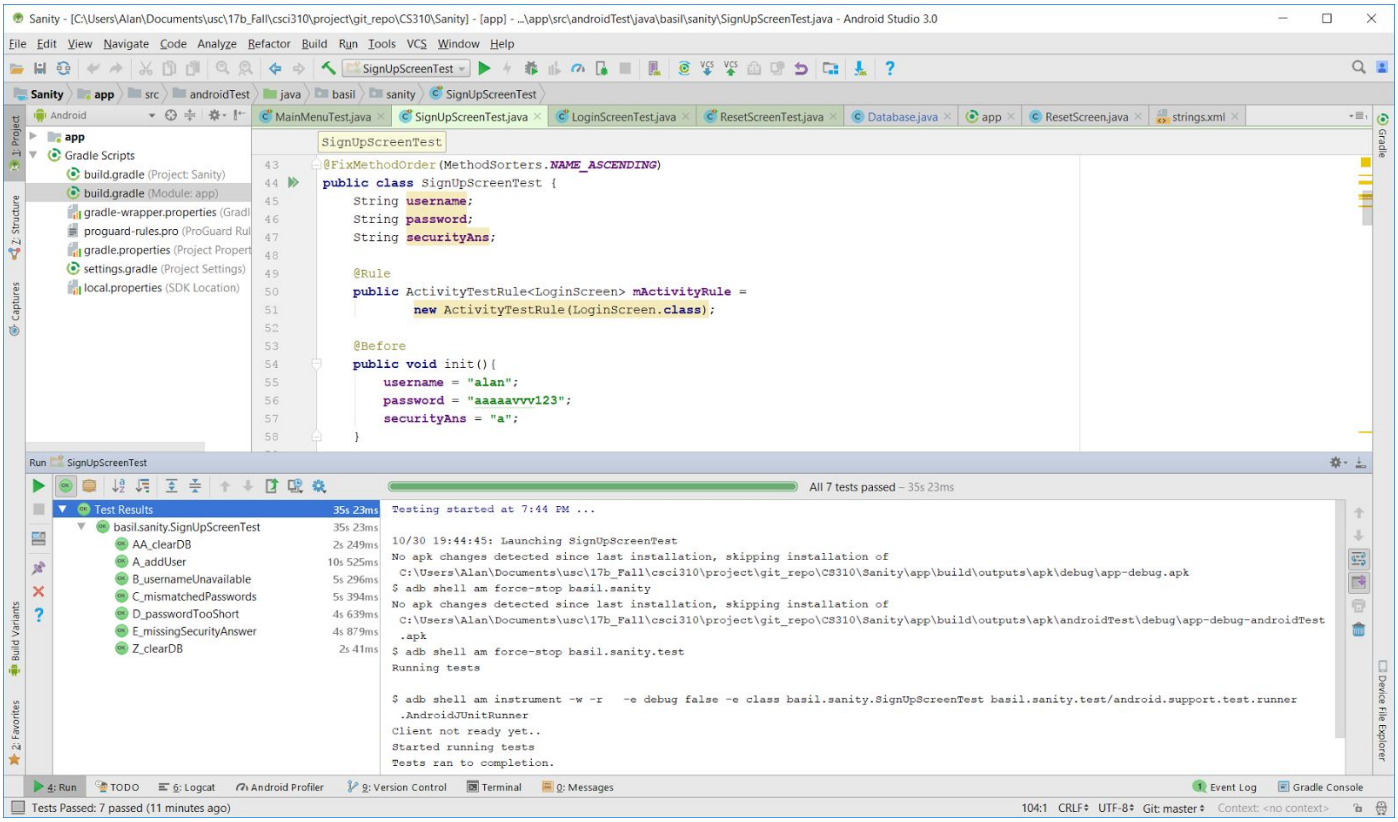
ii.
This test tries to sign up a new user. The security question is chosen to be the second from the drop-down menu, "Who is your favorite teacher?". The user should have signed up successfully and gone to the main menu. Then, the logout button is clicked and the forgot password button is clicked. The test checks that the security question that is being asked in the reset password screen is the correct one. To run this test, right click SignUpScreenTest in Project and choose Run 'SignUpScreenTest .'

iii.

The inputs must be a valid username and the password must be at least 5 characters. The security answer must be a non-empty string. This test shows that the security question was saved in sign up and reappears in reset screen.

iv.

The test passes because the security question in the reset password screen is “Who is your favorite teacher?”.



This screen shows that all tests in the SignUpScreenTest class (5-9) passed.

6. **B_usernameUnavailable():**

i.

Sanity\app\src\androidTest\java\basil\sanity\SignUpScreenTest.java

ii.

This test tries to sign up a new user. However, the username has already been taken. This should not be allowed because two users cannot have the same username. Therefore, the user should receive an error message notifying them

to pick a different username. To run this test, right click SignUpScreenTest in Project and choose Run ‘SignUpScreenTest .’

iii.
The input username must be a username that is already in the database. To accomplish this, the test uses the same username in the test A_addUser(). This test shows that the user cannot sign up with a username that is already taken.

iv.
The test passes because the Toast message matches the correct message for unavailable username.

7. **C_mismatchedPasswords():**
   i.
   Sanity\app\src\androidTest\java\basil\sanity\SignUpScreenTest.java

   ii.
   This test tries to sign up a new user. The signup screen asks the user to enter their desired password twice. In the test case, a different password is entered the second time. Therefore, the user should receive an error message notifying them that their passwords do not match. To run this test, right click SignUpScreenTest in Project and choose Run ‘SignUpScreenTest .’

   iii.
   The input username must be a username that is in the database. Otherwise, the test will instead get an error that the username is unavailable because that is checked first. Additionally, the password must be entered differently the first and second time. This test shows that users cannot sign up if their passwords do not match.

   iv.
   The test passes because the Toast message matches the correct message for mismatched password.

8. **D_passwordTooShort():**
   i.
   Sanity\app\src\androidTest\java\basil\sanity\SignUpScreenTest.java

ii.
This test tries to sign up a new user. However, the desired password is 4 characters in length. This should not be allowed because the minimum password length is 5. Therefore, the user should receive an error message notifying them to pick a different password. To run this test, right click SignUpScreenTest in Project and choose Run ‘SignUpScreenTest .’

iii.
The input username must be a username that is in the database. Otherwise, the test will instead get an error that the username is unavailable because that is checked first. Additionally, the same password must be entered twice. Otherwise, the test will get an error that the passwords are mismatched. Therefore, a password of length 4 is entered twice. This test shows that the user cannot sign up with a password under 5 characters even if other fields are correct.

iv.
The test passes because the Toast message matches the correct message for unavailable username.

9. E_missingSecurityAnswer():

i.
Sanity\app\src\androidTest\java\basil\sanity\SignUpScreenTest.java

ii.
This test tries to sign up a new user. However, the user does not supply a security answer. Therefore, the user should receive an error message notifying them enter a security answer. To run this test, right click SignUpScreenTest in Project and choose Run ‘SignUpScreenTest .’

iii.
The input username must be a username that is in the database. Otherwise, the test will instead get an error that the username is unavailable because that is checked first. Additionally, the same password of 5 or more characters must be entered twice. Otherwise, the test will get an error that the passwords are mismatched or it is too short. Finally, the security answer must be left blank. This test shows a user cannot sign up without a security answer even if other fields are correct.

iv.

The test passes because the Toast message matches the correct message for missing security answer.

10. A_nonexistentUser():

i.

Sanity\app\src\androidTest\java\basil\sanity\ResetScreenTest.java

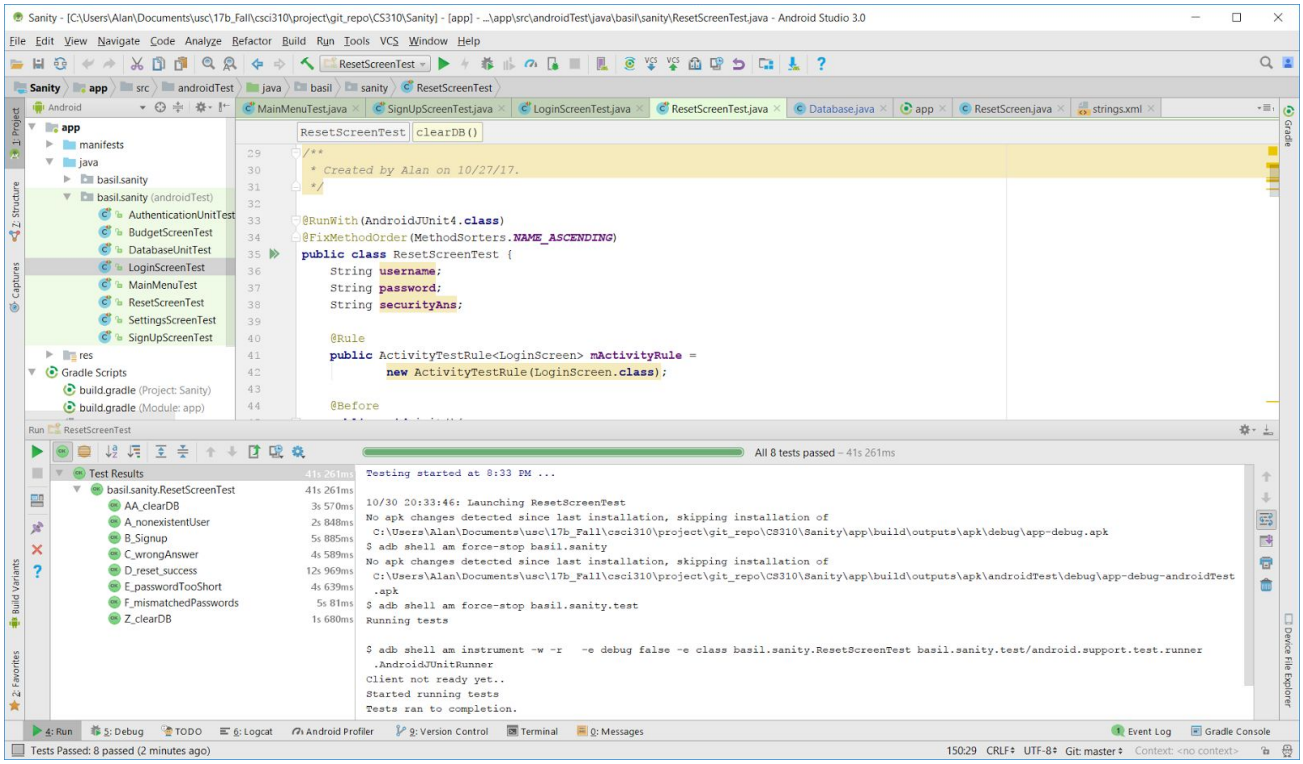
ii.

This test tries to reset the password for a username that does not exist. The app should tell the user to enter a valid username. To run this test, right click ResetScreenTesting Project and choose Run 'ResetScreenTest.'

iii.

The input username must be a username that is not in the database. The database was cleared beforehand so the username can by any non-empty string. This test shows that only existing users can reset their password.

iv.



The test passes because the Toast message matches the correct message for nonexistent user. This screenshot shows that all tests in ResetScreenTest (10-14) passed.

11. C_wrongAnswer():

i.
Sanity\app\src\androidTest\java\basil\sanity\ResetScreenTest.java

ii.
This test tries to reset the password for but an incorrect security answer is given. The app should tell the user to enter the correct security answer. To run this test, right click ResetScreenTesting Project and choose Run 'ResetScreenTest.'

iii.
The input username must be a username that is in the database. The security answer must be non-empty but also incorrect. This test shows that the password cannot be reset if the security answer is wrong.

iv.
The test passes because the Toast message matches the correct message for incorrect security answer.

12. D_reset_success():

i.
Sanity\app\src\androidTest\java\basil\sanity\ResetScreenTest.java

ii.
This test tries to reset the password correctly. After resetting, the user logs out and logs in with the updated passoword. To run this test, right click ResetScreenTesting Project and choose Run 'ResetScreenTest.'

iii.
The input username must be a username that is in the database. The security answer must be correct for that user. The new password must be at least 5 characters and be entered twice. This test shows the reset functionality is working..

iv.
The test passes because the user is able to log in with the new password and click on the analytics button..

13. E_passwordTooShort():

i.
Sanity\app\src\androidTest\java\basil\sanity\ResetScreenTest.java

ii.
This test tries to reset the password for a valid user but the new desired password is under 5 characters. The app should inform the user of this error when they click reset password. To run this test, right click ResetScreenTesting Project and choose Run 'ResetScreenTest.'

iii.
The input username must be a username that is in the database. The new password should be 4 or fewer characters. The password must be entered twice correctly. This test shows that the password cannot be reset if the user enters passwords that are too short.

iv.
The test passes because the Toast message matches the correct message for password too short.

14. F_mismatchedPasswords():

i.
Sanity\app\src\androidTest\java\basil\sanity\ResetScreenTest.java

ii.
This test tries to reset the password for a valid user but mismatched passwords are given. The app should inform the user of this error. To run this test, right click ResetScreenTesting Project and choose Run 'ResetScreenTest.'

iii.
The input username must be a username that is in the database. Different passwords should be entered. This test shows that the password cannot be reset if the user enters mismatched passwords.

iv.

The test passes because the Toast message matches the correct message for mismatched passwords.

15. aDeleteAllBudgetsTest():

i.
Sanity\app\src\androidTest\java\basil\sanity\MainMenuTest.java

ii.
This test deletes all the current budgets, then makes sure the overall budget goes down to $0/$0. To run this test, right click MainMenuTest Project and choose Run 'MainMenuTest.'

iii.
The purpose of this test case is to test the delete budget functionality since it is a major requirement and make sure that it doesn't crash the program. The test case was generated using Espresso.

iv.
The test case passes if the program doesn't crash and when all the the budgets have been deleted, the overall budget is at $0/$0.

v.
A bug was uncovered where if you deleted a budget, the overall budget limit was not getting updated in the database correctly. This was found by deleting all the budgets, the signing out and signing back in and failing an assertion where the overall budget was not $0/$0.

16. bAddBudgetTest():

i.
Sanity\app\src\androidTest\java\basil\sanity\MainMenuTest.java

ii.
This test adds numerous randomly generated budgets to the budget list. It acts as a stress test for the adding budget functionality since it is a major requirement. To run this test, right click MainMenuTest Project and choose Run 'MainMenuTest.'

iii.

The purpose of this test case is to test the add budget functionality since it is a major requirement and make sure that it doesn't crash the program. The test case was generated using Espresso.

iv.
The test case passes if the program doesn't crash and when all the the budgets have been added and the overall budget has the correct overall amount.

v.
Fixed a bug in the layout of overall budget where the contents were not matching the parent which made it show up as a smaller box than intended. This wasn’t a direct result of the test but was seen by the tester when the test was running.

17. cAddTransactionTest():

i.
Sanity\app\src\androidTest\java\basil\sanity\MainMenuTest.java

ii.
This test adds numerous randomly generated transactions to random budgets in the budget list. It acts as a stress test for the adding transaction functionality since it is a major requirement. To run this test, right click MainMenuTest Project and choose Run ‘MainMenuTest.’

iii.
The purpose of this test case is to test the add transaction functionality since it is a major requirement and make sure that it doesn't crash the program. The test case was generated using Espresso.

iv.
The test case passes if the program doesn't crash and when all the the transactions have been added and the overall budget has the correct overall spent and amount.

18. dDeleteAllBudgetsTest():

i.
Sanity\app\src\androidTest\java\basil\sanity\MainMenuTest.java

ii.
This is another test that deletes all the current budgets, then makes sure the overall budget goes down to $0/$0. To run this test, right click MainMenuTest Project and choose Run ‘MainMenuTest.’

iii.
The purpose of this test case is to test the delete budget functionality again since it is a major requirement and make sure that it doesn't crash the program. We also make sure that the delete functionality works after adding multiple transactions. The test case was generated using Espresso.

iv.
The test case passes if the program doesn't crash and when all the the budgets have been deleted, the overall budget is at $0/$0.

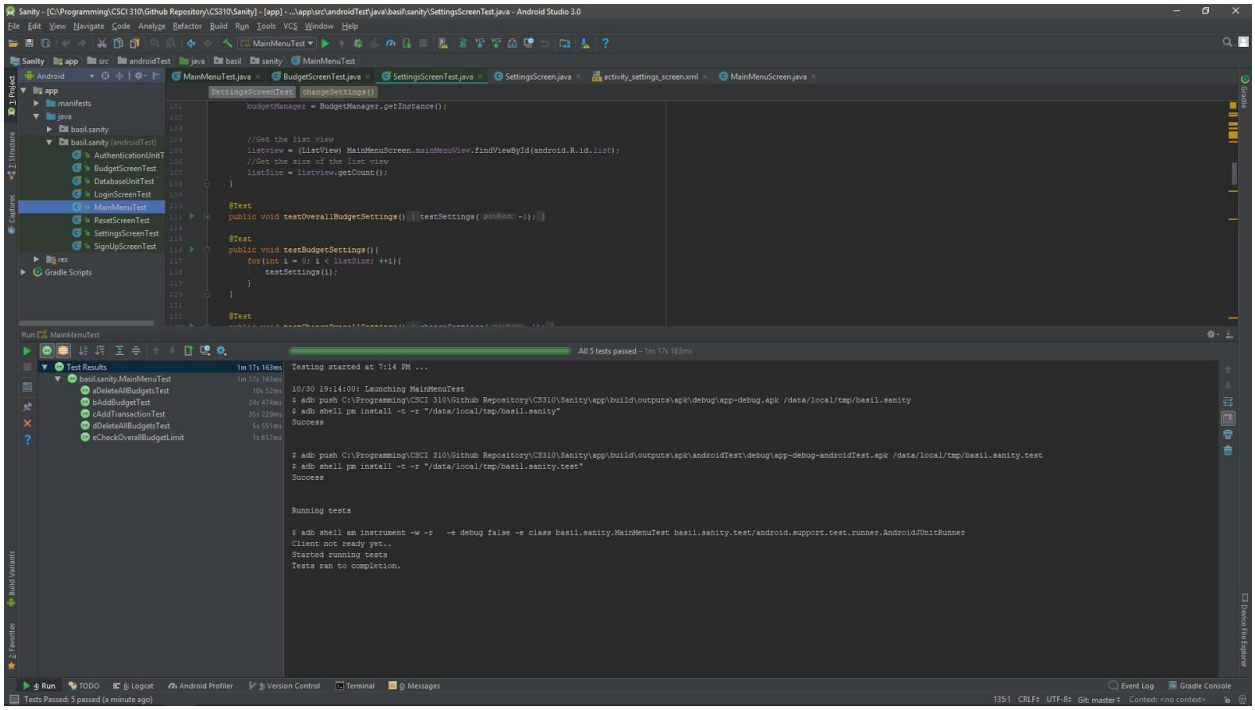
19. eCheckOverallBudgetLimit():

i.
Sanity\app\src\androidTest\java\basil\sanity\MainMenuTest.java

ii.
This test logs back in to make sure that the database correctly updated the overall budget after all the budgets were deleted. To run this test, right click MainMenuTest Project and choose Run ‘MainMenuTest.’

iii.
The purpose of this test case is to test the delete budget functionality in the database again since it is a major requirement and make sure that it updated the values properly. The test case was generated using Espresso.

iv.
The test case passes if the program doesn't crash and when all the the budgets have been deleted, the overall budget is at $0/$0.



MainMenuTest test cases

20. checkTransactionList():

i.

Sanity\app\src\androidTest\java\basil\sanity\BudgetScreenTest.java

ii.

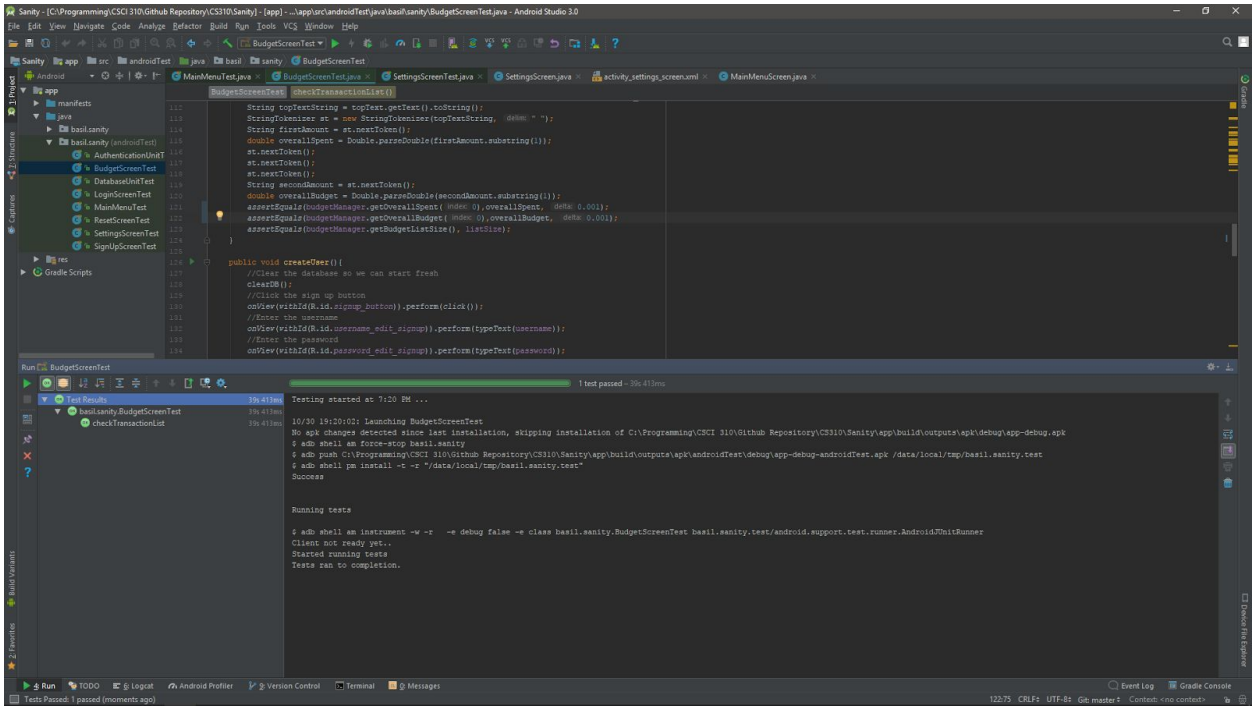
This test logs in and makes sure that the screen that shows all the transactions is formatted correctly and is indeed showing all the transactions of the correct budget. To run this test, right click BudgetScreenTest Project and choose Run 'BudgetScreenTest.'

iii.

The purpose of this test case is to test the transaction screen functionality since it is a major requirement and make sure that it is showing the correct values of the correct budget. The test case was generated using Espresso.

iv.

The test case passes if the program doesn't crash and all the transaction have been checked to make sure they are valid.



BudgetScreenTest test cases

21. testOverallBudgetSettings():

i.

Sanity\app\src\androidTest\java\basil\sanity\SettingsScreenTest.java

ii.

This test logs in and makes sure that the screen that shows all the settings and makes sure that they are showing the correct settings for the overall budget. To run this test, right click SettingsScreenTest Project and choose Run 'SettingsScreenTest.'

iii.

The purpose of this test case is to test the settings screen functionality since it is a major requirement and make sure that it is showing the correct values of the overall budget. The test case was generated using Espresso.

iv.

The test case passes if the program doesn't crash and all the settings are correctly shown.

22. testBudgetSettings():

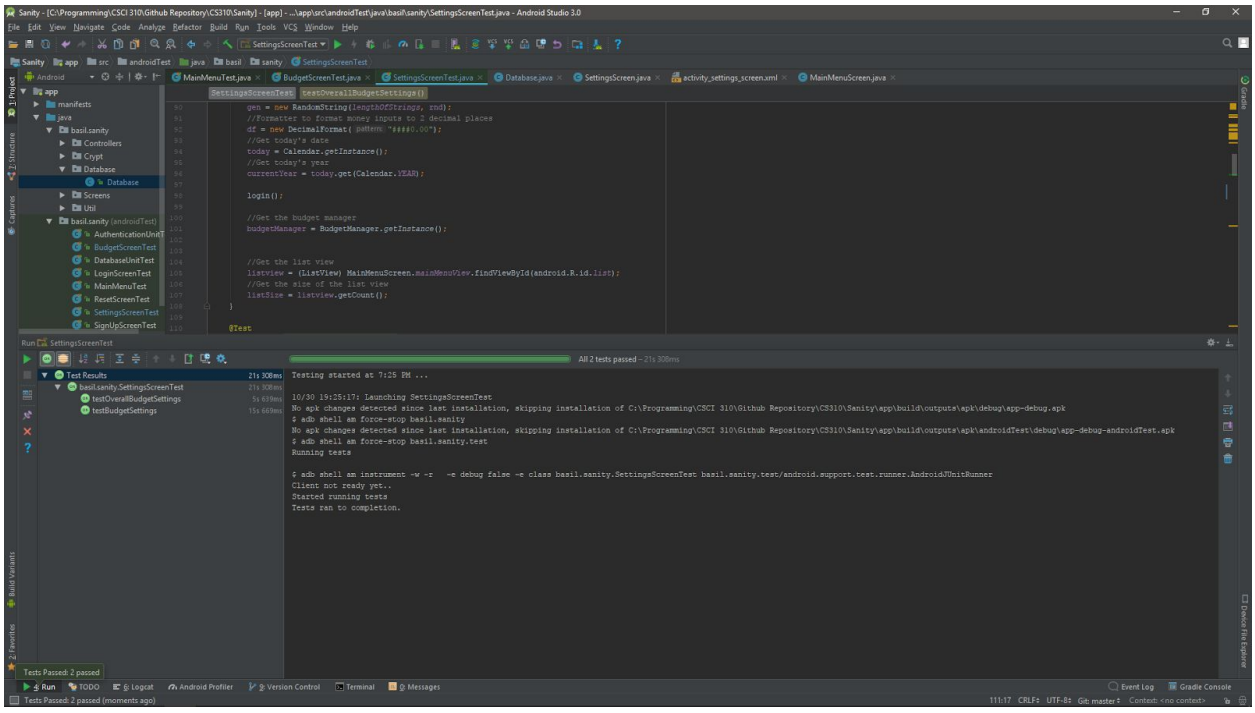
i.

Sanity\app\src\androidTest\java\basil\sanity\SettingsScreenTest.java

ii.
This test logs in and makes sure that the screen that shows all the settings and makes sure that they are showing the correct settings for the specific budget clicked. To run this test, right click SettingsScreenTest Project and choose Run 'SettingsScreenTest.'

iii.
The purpose of this test case is to test the settings screen functionality since it is a major requirement and make sure that it is showing the correct values of the specific budget. The test case was generated using Espresso.

iv.
The test case passes if the program doesn't crash and all the settings are correctly shown for all the budgets specifically.



SettingsScreenTest test cases

5. White-box Tests

1. **testAddBudget()**
   i.
   Sanity\app\src\androidTest\java\basil\sanity\BugetManagerTest.java

   ii.
   The test case calls a BudgetManager set up before with 0 budgets and adds a single budget to it, then retrieves the budget containing $100 named “Budget” and checks to make sure that the two are equal.

   iii.
   This shows that users can add custom budgets and set the threshold amounts, and they will be stored and reflected in the app correctly. The rationale is that whenever a user chooses to add their first custom budget with fields such as a name and threshold, this test case is called.

   iv.
   The result is that BudgetManager now has 1 budget named “Budget” with a threshold for $100.

2. **testAddMultipleBudgets():**
   I.
   Sanity\app\src\androidTest\java\basil\sanity\BugetManagerTest.java.

   ii.
   The test case calls a BudgetManager set up before with 0 budgets and adds two budgets to it, one with $100 called “Budget” and one with $200 called “Budget 2”. Both of these are then added to the BudgetManager and retrieved and checked with the original budgets to make sure they are equal

   iii.
   The rationale is to ensure that the user can add multiple budgets and make sure that they are all represented accurately in the BudgetManager. Whenver a user adds budgets in addition to the initial budget, this test case will be called.

   Iv.
   The result is that BudgetManager now has 2 budgets, the first named “Budget” with a threshold for $100 and the second named “Budget2” with a threshold for 200.

3. **changeBudgetSettings()**
   i.
   Sanity\app\src\androidTest\java\basil\sanity\BugetManagerTest.java.

   ii.
   The test creates two budgets, one named “Budget” and one named “Budget2” with the default settings for a threshold for reminders at 75% and for notifications to appear once each day afterwards. The test changes the settings for Budget2 to have a threshold for reminders at 60% instead and to repeat once a week.

   Iii.
   The rationale behind this test is to test the requirements that users can change the settings for each budget including the notification, and that each budget can have individual settings that can be adjusted separately.

   Iv.
   The result is that there will be two budgets called “Budget” and “Budget2”, and the second one will have its settings changed for a threshold at 60% and reminders once a week afterward.

4. **addRefund()**
   i. Sanity\app\src\androidTest\java\basil\sanity\BugetManagerTest.java.

   Ii. The test creates a budget called Budget with a threshold for $100 and with a spending of $75, then adds a refund for $50 to the budget.

   iii. The rationale is that whenever a user tries to refund a transaction for a budget, the refund should work properly and lower the amount spent by the same amount as the refund.

   iv. The result is that there is a budget called Budget with a budget for $100, except now the budget has a spending of $25 rather than $75, indicating a successful refund.

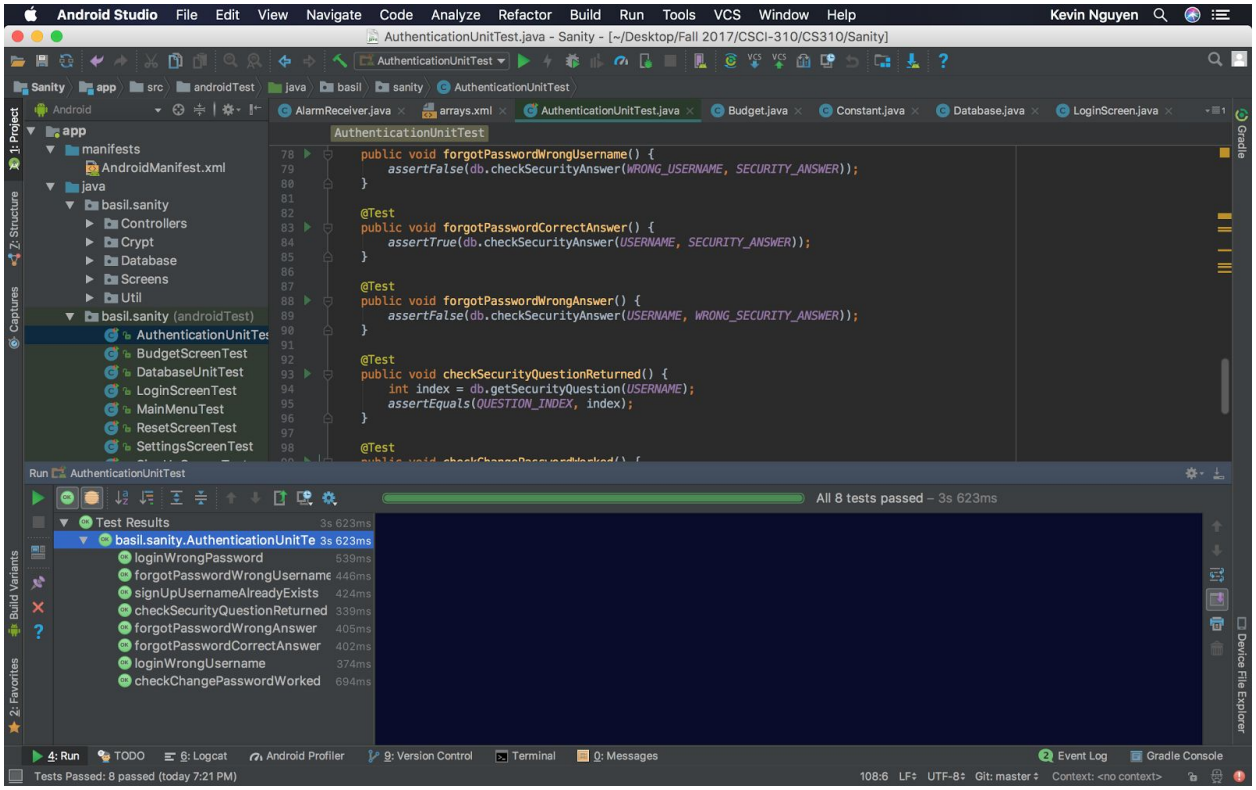
5. **aCheckAddUser():**
    i. Sanity\app\src\androidTest\java\basil\sanity\DatabaseUnitTest.java
    ii. This test tries to add a user directly to the database.It adds a user that has not been created yet. To run this test, right click DatabaseUnitTest in Project and choose Run DatabaseUnitTest.
    iii. The database is one of the major tier within the program, and we have to make sure that the data that is returned from the database is the one that was sent in. To verify this, we add a user to the database and compare it to a same instance which has not been added. The username has to be a username that hasn't yet been created, but since we reset the database at the beginning and this test is ordered for database purposes , the username can be arbitrary.
    iv. The test passes as the user returned from the database is the same as the one that was added to the database.
6. **bCheckAddBudget():**
    i. Sanity\app\src\androidTest\java\basil\sanity\DatabaseUnitTest.java
    ii. This test tries to add a budget directly to the database.It adds a user that has not been created yet.To run this test, right click DatabaseUnitTest in Project and choose Run DatabaseUnitTest.
    iii. This test verifies that the budget returned from the database is the same one that was put into the database. The name of the budget name has to be a budget name that hasn't yet been created, but since we reset the database at the beginning and this test is ordered for database purposes , the budget name can be arbitrary.
    iv. The test passes as the budget returned from the database is the same as the one that was added to the database.
7. **cCheckAddTransaction():**
    i. Sanity\app\src\androidTest\java\basil\sanity\DatabaseUnitTest.java
    ii. This test tries to add a transaction directly to the budget in the database.It adds a transaction that has not been created yet.To run this test, right click DatabaseUnitTest in Project and choose Run DatabaseUnitTest.
    iii. This test verifies that the transaction returned from the database is the same one that was put into the database. The transaction name can be arbitrary, however the name of the budget has to be of one that is precreated as we are checking the data sent is same as the data returned and transaction has to be matched with already created database.

iv. The test passes as the transaction returned from the database is the same as the one that was not added to the database.

8. **dCheckGetOverallTransactionList():**
   i. Sanity\app\src\androidTest\java\basil\sanity\DatabaseUnitTest.java
   ii. This test tries to add multiple transaction directly to the database.It adds a transaction that has not been created yet. To run this test, right click DatabaseUnitTest in Project and choose Run DatabaseUnitTest .
   iii. This test verifies that the transactionlist returned from the database is the same one that was put into the database. The transaction names can be arbitrary, however the name of the budget has to be of one that is pre created as we are checking the data sent is same as the data returned and transaction has to be matched with already created database.
   iv. The test passes as the transactionlist returned from the database is the same as the one that was not added to the database.
9. **eCheckWrongLogin():**
   i. Sanity\app\src\androidTest\java\basil\sanity\DatabaseUnitTest.java
   ii. This test tries to check directly in the database that incorrect username and login will or will not be allowed to login. To run this test, right click DatabaseUnitTest in Project and choose Run DatabaseUnitTest .
   iii. This verifies that incorrect username and password will not be allowed to login, and will return false. This test is essential as we don't allow users with wrong username and password to login. The test will work as long as the username or the password is wrong.
   iv. The test passes as it returns false, as there is no user with the matching username and password.
10. **fCheckGetAllBudget():**
   i. Sanity\app\src\androidTest\java\basil\sanity\DatabaseUnitTest.java
   ii. This test tries to gets all budget directly from the database.To run this test, right click DatabaseUnitTest in Project and choose Run DatabaseUnitTest.
   iii. This test will verify that all budgets that are returned are the same as the ones put in. It also checks and verifies that pre made budgets also match the one you get by getBudgets(). This compares already created database so there is no actual input.
   iv. The test passes as the budgetList returned from the database is the same as the one that was added to the database including the ones created by the app.
11. **gCheckDeleteBudget():**

i. Sanity\app\src\androidTest\java\basil\sanity\DatabaseUnitTest.java
ii. This test tries to delete a budget directly from the database.It removes a budget that has already been created. .To run this test, right click DatabaseUnitTest in Project and choose Run DatabaseUnitTest.
iii. This test verifies that removing a budget, will fully remove the budget from the database. This is important as user might want to delete a budget, and we have to make sure that the budget is fully removed. The budgetname has to be one that has already been created, so the budget created at checkAddBudget() is removed.
iv. The test passes as we try to find the deleted database and that returns false meaning no sign of database is present.

12. **fCheckRefund():**
    i. Sanity\app\src\androidTest\java\basil\sanity\DatabaseUnitTest.java
    ii. This test tries to add a refund directly to the database.To run this test, right click SignUpScreenTest in Project and choose Run DatabaseUnitTest.
    iii. This test verifies that the refund returned from the database is the same one that was put into the database. The refund's name can be arbitrary, however the name of the budget has to be of one that is pre created as we are checking the data sent is same as the data returned.
    iv. The test passes as the refund returned from the database is the same as the one that was not added to the database.



AuthenticationUnitTest.java results - 8/8 test cases pass

13. **loginWrongPassword():**
    i. Sanity/app/src/androidTest/java/basil/sanity/AuthenticationUnitTest.java.
    ii. This particular test case simulates a method call to verify a user in the database using an incorrect password. The test can be run by running it either individually in AuthenticationUnitTest.java or by running the AuthenticationUnitTest suite.
    iii. The input username must be a username that is already registered in the database. The input password must be an incorrect password for the given user. These parameters are provided in the @Before annotation in the unit test class.
    iv. The test passes because the database cannot match the username and password combination to an existing user.
14. **forgotPasswordWrongUsername():**
    i. Sanity/app/src/androidTest/java/basil/sanity/AuthenticationUnitTest.java.
    ii. This test attempts to signal the database to check the answer to a security answer provided with an incorrect username. This situation would arise when a user has forgotten his/her password and is trying to reset it in the changePassword screen.

iii. The input username must be an invalid username that does not exist within the database. The input answer to the security question can be an arbitrary string because the method, checkSecurityAnswer(), will terminate upon failing to find the invalid username in the database.
iv. This test passes because the database cannot match the username to an existing user in the database.

15. signUpUsernameAlreadyExists():

i. Sanity/app/src/androidTest/java/basil/sanity/AuthenticationUnitTest.java.
ii. This test attempts to add a new user to the database that already has a user registered with the same username. The test case can be run individually in AuthenticationUnitTest.java. Furthermore
iii. This test case is important because usernames are unique in our system and this concept is critical to our app's functionality. Thus this constraint should be upheld at all costs. The test case takes a password and hashes it securely with salting. Then, a new user object is instantiated with the hash and the test case username. Lastly, we verify that the database will not create a new user when addUser() is called on with the new user as a parameter.
iv. This test passes because the database finds an existing user in the database that is already registered with the desired username.

16. checkSecurityQuestionReturned():

i. Sanity/app/src/androidTest/java/basil/sanity/AuthenticationUnitTest.java.
ii. This unit test verifies that the security question returned for a given user is the correct question via the question index. In other words, the question the user answered when creating an account should be the same question that is returned when calling checkSecurityQuestion() with the username as an input.
iii. The rationale behind this unit test is that the security question is very important to the functionality of change password. If a different question is returned to the user, he/she may not be able to answer it and will, as a result, be locked out of the app. The only input to this unit test is the username.
iv. This test passes because the question index returned from the database matches the index that we prepared before the test case. We knew which question index we registered the user with and the call to checkSecurityQuestion returned the same index.

17. forgotPasswordWrongAnswer():

i. Sanity/app/src/androidTest/java/basil/sanity/AuthenticationUnitTest.java.
ii. This unit test simulates a user trying to reset his/her password but with an incorrect answer to his/her security question.
iii. If a user inputs the wrong answer to his/her security question, he/she must not be able to reset the password, otherwise an unfriendly person could tamper with and reset the account credentials on the user's device. The inputs to this unit test are the username to so the database can match to a user and an answer to the user's security question. These parameters are passed to the checkSecurityAnswer() function in the database.
iv. This unit test passes because the database recognizes that the answer provided by the user is incorrect by checking the answer stored in the database for the given username.

18. forgotPasswordCorrectAnswer():

i. Sanity/app/src/androidTest/java/basil/sanity/AuthenticationUnitTest.java.
ii. This unit test simulates a user trying reset his/her password by providing the correct answer to his/her security question.
iii. This functionality is important to test because, when a user gets the question right, we don't want the database to lock the user out. Many times in real systems, this happens much to the frustration of the user. The inputs to this unit test are the username and the answer to the security question.
iv. This test passes because the database recognizes that the answer provided by the user is the correct answer by checking the answer stored in the database for the given username.

19. loginWrongUsername():

i. Sanity/app/src/androidTest/java/basil/sanity/AuthenticationUnitTest.java.
ii. This unit test simulates a user trying to log in using an incorrect username. That is, the username the user provides is not registered in the database.
iii. This unit test is important because we don't want the user to be able to log in to the application if they provide an incorrect username. If that were the case, anyone who had the user's device could also log in with an incorrect username, accessing financial data. The inputs to this unit test are the username and password.

iv. This unit test passes because the database correctly doesn't verify the user's attempt to log in with the invalid username.

20. checkChangePasswordWorked():

i. Sanity/app/src/androidTest/java/basil/sanity/AuthenticationUnitTest.java.

ii. This unit test checks to see if the update password functionality works when a user changes his/her password. First, we check to see if the user exists in the database with the current password. Then we update the password via updatePassword() and then check to see if the user's password has been updated.

iii. This unit test is important because we want to ensure that when a user changes his/her password, the result of the update password is reflected in the database. The inputs to this unit test are the username and old password as well as the new password.

iv. This unit test passes because the database finds the user with the old password, successfully updates the user account with the new password, and then returns the user with the new password when queried.