***Scrumptious Finance***

***Testing and Refactoring***

Financial Web Application

March 28th, 2018

Version 1.1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[Project Lead Signature Block]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[Project Manager Signature Block]

**Revision Sheet**

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Brief Summary of Changes | Signed |
| Version 1.1 | March 28th 2018 | Created Unit Testing for Transactions | *Jordan E.* |
| Version 1.1 | March 29th 2018 | Added Unit Testing for Account Functions | *Jordan E.* |
| Version 1.1 | April 1st 2018 | Updated Testing Document | *Jordan E.* |
| Version 1.1 | April 1st 2018 | Added Testing for Bank Statement Uploads | *Levi M.* |

**Table of Contents**

[1 Unit Testing Transactions 4](#_Toc510364717)

[1.1 Summary 4](#_Toc510364718)

[1.2 Refactoring 4](#_Toc510364719)

[1.3 Testing 4](#_Toc510364720)

[1.4 Console of Unit Tests 5](#_Toc510364721)

[2 Unit Testing Financial Account’s Related Functions 6](#_Toc510364722)

[2.1 Summary 6](#_Toc510364723)

[2.2 Testing 6](#_Toc510364724)

[2.3 Refactoring 7](#_Toc510364725)

[2.4 Console of Unit Tests 7](#_Toc510364726)

[3 Functional Testing Uploading a Bank Statement 8](#_Toc510364727)

[3.1 Summary 8](#_Toc510364728)

[3.2 Refactoring 8](#_Toc510364729)

[3.3 Testing 8](#_Toc510364730)

# 1 Unit Testing Transactions

## 1.1 Summary

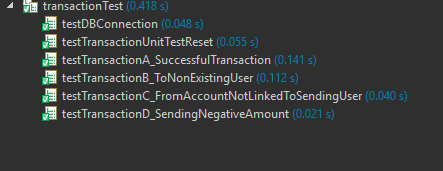
The goal of testing transactions was to have extremely secure transactions while keeping fast transaction speeds. In order to complete these two main points, we developed a system where the data was received from the client, sanitized of any malicious characters, checked if it was sending to an existing user, checked if the withdrawing account is linked with the logged in user, and then the transaction was either confirmed or denied based on these security points. If the transaction was successful, then we would remove the balance from the sending account and add the balance to the receiving account. However, we changed this concept in the refactoring process.

## 1.2 Refactoring

While testing we came across a few small bugs that we have since fixed. But, one larger problem we had in testing was that if someone were to interrupt a transaction or the database crashed in the middle of a transaction then the database would have inconsistent data with incorrect results. So, we researched and found that there is something called SQL Transactions that allows you to disable auto-committing when dealing with databases. What this does is if there are any type of interruptions during a section of code, then it will automatically fall back to the previous version of the database. This completely solved the issue that we were having, and it also increased the speeds of our transactions by about 5%.

## 1.3 Testing

The first task we completed when testing the Transaction was to make sure that the database connection was working properly during the tests. Next, we created a test for verifying that the data reset was working because we don’t want to leave test data on the database. After this, we tested a transaction with valid input to see how long it would take (~ 0.15s). Finally, we created a few tests to check that the proper error codes were returned for the matching errors.



## 1.4 Console of Unit Tests

PHPUnit 7.0.1 by Sebastian Bergmann and contributors.

...... 6 / 6 (100%)

-----Transaction A-----

Sent Transaction('jordems', 12.12, "Money Owed", 36) in format(tousername,amount,reason,fromaccountid)

\*\*\*Expecting Successful Transaction\*\*\*

Transaction A's Request is Successful

--Verifying with Database's Account Data--

FromAccount Balance Before: $5000

FromAccount Balance After: $4987.88

Amount withdrew: $12.12

ToAccount Balance Before: $416.38

ToAccount Balance After: $428.5

Amount deposited: $12.12

-----Transaction A's Test is SUCCESSFUL!-----

-----Transaction B-----

Sent Transaction('NOtExistentUser', 12.12, "Money Owed", 36) in format(tousername,amount,reason,fromaccountid)

\*\*\*Expecting Failed Transaction as Username Doesn't Exist\*\*\*

Transaction B Failed, Receiving Username Doesn't exist or Doesn't have a main Account

FromAccount Balance Before: $5000

FromAccount Balance After: $5000

Amount withdrew: $0

-----Transaction B's Test is SUCCESSFUL!-----

-----Transaction C-----

Sent Transaction('jordems', 12.12, "Money Owed", -1) in format(tousername,amount,reason,fromaccountid)

\*\*\*Expecting Failed Transaction as withdrawing Financial Account Doesn't link to this user\*\*\*

Transaction C Failed, User Not Linked to Account

FromAccount Balance Before: $0

FromAccount Balance After: $0

Amount withdrew: $0

-----Transaction C's Test is SUCCESSFUL!-----

-----Transaction D-----

Sent Transaction('jordems', -12.12, "Money Owed", 36) in format(tousername,amount,reason,fromaccountid)

\*\*\*Expecting Faild Transaction, Error Message: Insufficient funds\*\*\*

Transaction D Failed, Insufficient Funds

-----Transaction D's Test is SUCCESSFUL!-----

Time: 544 ms, Memory: 8.00MB

OK (6 tests, 6 assertions)

# 2 Unit Testing Financial Account’s Related Functions

## 2.1 Summary

When creating the financial account’s related functions, we wanted to create an easy to use UI to allow the users not have to think when adding/editing/viewing their accounts. So, we came up with a two-column layout that gives the user more access, allowing most functions on one page instead of creating separate pages for each function (Figure 2.1).

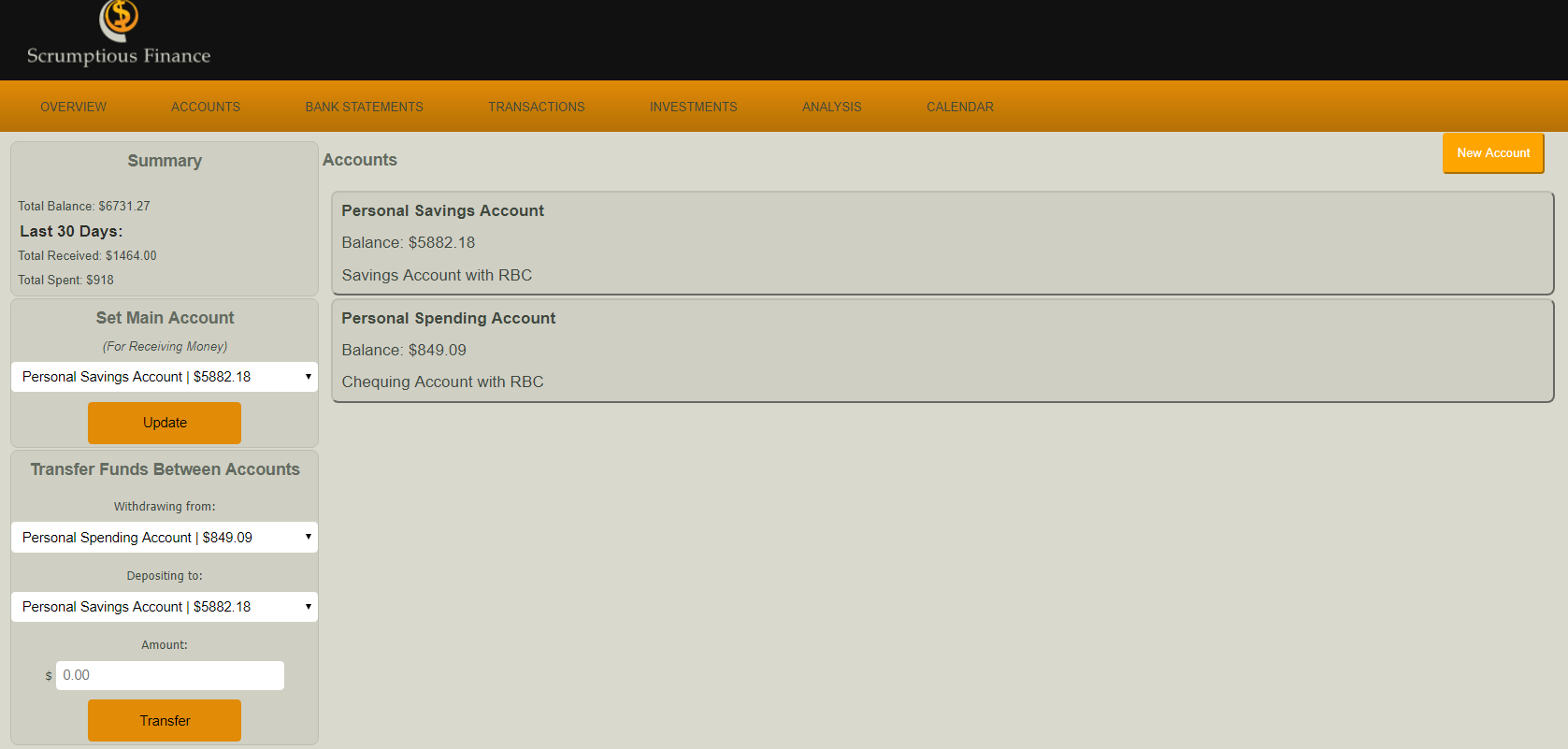


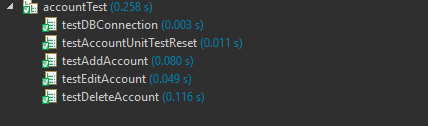
Figure 2.1

## 2.2 Testing

When testing the account related functions, the main result we wanted from this is to have a consistent and fast functionality. We first added a test to make sure that the database connection was successful. We then added a function that would remove the current test scenario data of the database. As, we don’t want to keep test data on the server. We then added a test to check if the add account was working properly. Then, we added a test to edit the just created account to make sure that it is working correctly.

## 2.3 Refactoring

Since the original release (Version 1.0). We realized that there was currently no way to transfer funds from one of your accounts to another easily. Also, we have implemented the main account feature for sending money directly into someone’s account. So, since then we have added these features, as well as fixing some small bugs that people have found in the debugging session.



## 2.4 Console of Unit Tests

PHPUnit 7.0.1 by Sebastian Bergmann and contributors.

...

-----testAddAccount-----

Using data addAccount(Lifesd Savings, CIBC,Savings Account,12323) in format addAccount(title, financialinstitution,type,balance)

addAccount Request Successful

-----Checking Database to Confirm addAccount worked Properly-----

Expected: (Lifesd Savings, CIBC,Savings Account,12323)

Actual: (Lifesd Savings, CIBC, Savings Account, 12323.00)

-----testAddAccount SUCCESSFUL-----.

-----testEditAccount-----

Using data editAccount(Life Savings, RBC,Savings Account,123) in format addAccount(title, financialinstitution,type,balance)

Expecting the Account to change all of its old values to these new values

editAccount Request Successful

-----Checking Database to Confirm editAccount worked Properly-----

Expected: (Life Savings, RBC,Savings Account,123)

Actual: (Life Savings, RBC, Savings Account, 123.00)

-----testEditAccount SUCCESSFUL-----. 5 / 5 (100%)

-----testDeleteAccount-----

Using function deleteAccount(64) in format addAccount(accountID)

Expecting the Account to nolonger exist

deleteAccount Request Successful

-----Checking Database to Confirm deleteAccount worked Properly-----

Expected: Nothing Returned from Query

Actual: Nothing Returned from Query

-----testDeleteAccount SUCCESSFUL-----

Time: 393 ms, Memory: 8.00MB

OK (5 tests, 5 assertions)

# 3 Functional Testing Uploading a Bank Statement

## 3.1 Summary

The goal of testing the procedure of uploading a bank statement was to predict all the possible errors that could arise in the process of the user formatting and submitting their financial information. To prepare for these cases we created code that handles the errors and displays a descriptive message to the user so they know what went wrong.

## 3.2 Refactoring

Since the original release (Version 1.0) we have vastly improved the error handling that occurs during the upload process. We now filter and sanitize all the data that comes in through the bank statement to ensure that it is formatted properly and contains no empty fields. If there is invalid data contained within the bank statement it will not be uploaded to the database and the user will get a message indicating the exact column and row that the problem occurred so that they can locate and fix it before attempting to upload again. We also now check to make sure their file doesn’t contain any extra data beyond the three columns it is supposed to have.

## 3.3 Testing

To test the functionality of the importing a bank statement process we devised a series of cases that would likely cause problems in the system and created a plan on how to handle them. These cases included handling invalid file data (anything other than numeric values in the amounts column, date formatted incorrectly, and prohibited symbols used in the description column), handling an invalid file type (not a CSV extension), and handling a file size that is too large. Once we put the error handling code in place we tested the functionality by creating a series of invalid CSV files like the one below (Figure 3.1) and attempting to upload them to the website. As desired, these invalid files were not uploaded to the database and the user received a helpful message indicating why and where the problem occurred (Figure 3.2).

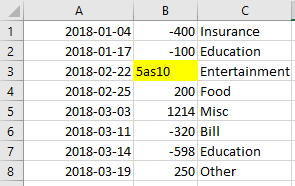


Figure 3.1

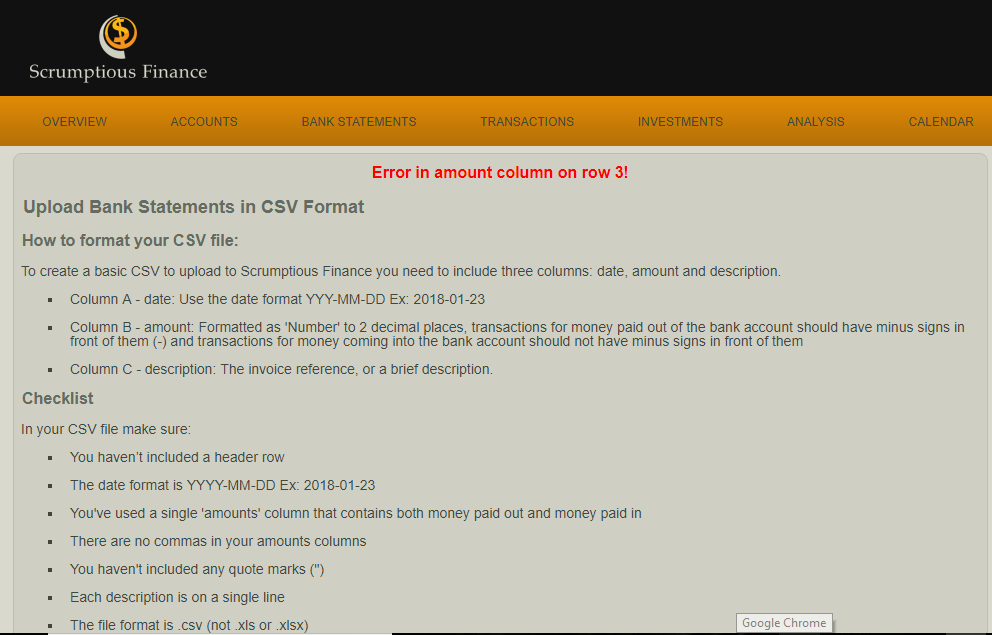


Figure 3.2