System and Unit Test Report

Product Name: Atheneum
Team Name: Atheneum
Members:

Edwin Ramirez
Nikola Panayotov
Seth Little
Ian Hardy
Cole Boyer

6/6/2017

Sprint 1:

A. User Story 1 from Sprint 1: As a user I want to be able to upload and download documents through a web interface to be able to more readily access information

Scenario (assuming screen prompts are followed-- the user already has metamask, and their Ether is already mined):

1. Start Atheneum app, select "sign up" and for the given fields type
    a. Email = "ihardy@ucsc.edu"
    b. Password = "passwordispassword"
    c. MM ID= "0x----..."
    d. MM PW= *******
    e. Press sign up
2. User can now access upload and download functions
3. To Upload- hit the drawer in the top left, hit "upload," and the link will direct to the upload card. Select the appropriate file with the upload button and type:
    a. Subject = "Other"
    b. Title = "test file"
    c. Description = "a sample file"
    d. Press "Send it!" and your file will be uploaded
4. To Download- hit the drawer in the top left, hit "download," and the link will direct to the download page. Select the appropriate tab if necessary, find the file you wish to download, and simply click the hyperlinked filename. This will prompt metamask to display a popup window for the transaction. Confirm your purchase, and the file will download!

Sprint 2:

A.User Story 1: As a user I want to be able to have my own account so that I can upload my files, keep a list of files downloaded, and link my Ethereum wallet to my account to earn Ether.

1. Start app.
2. Proceed to the log in process. The user will be asked to create an account. During account creation, the user will enter their Wallet ID and Password. These are the same as the MetaMask id and password.
3. Now that the user is logged in, they proceed to the upload page.
4. The user can also download items by going to the download page.
    a. From here, the user finds the document they want to download and press download.
    b. A popup will confirm that the user wishes to make the transaction. Assuming they do, a final prompt confirms with the user the amount of Ether they are spending and what their future Ether balance will be.
    c. Once the download is complete, the user is presented with both a link to download the document, as well as a receipt link for the transaction.

B. User Story 2: As a user I want to be able to have an Ethereum contract be deployed when someone downloads my file(s) so that I can earn Ether.

1. Start app.
2. Log in and navigate to "Upload". (note: must have an account with your Ethereum wallet linked already)
3. Select a file, input its Title and Description fields, as well as selecting a category.
4. Press the "Send it!" button to upload a file. (Known bug causes upload to fail on the first try, repeat steps 3 and 4 again to successfully upload the file)
5. Your file is now available with your Ethereum wallet tied to it!
6. When someone else downloads your file, they will be presented with a confirmation notice for the contract. Upon confirming the transaction, they will be able to download the file, and you will earn Ether.

Sprint 3:

A. User Story 2 of Sprint 3: As a user, I want to be able to sign into my account so that I can upload files, and download them in exchange for ether. (This test will pertain to all functionality of the website that involves accounts in any form)

Scenario

1. Load the index page of the website.
    i. Assume that, as per the instructions on the index, the user has created a Metamask account, is logged into it, is connected to the rinkeby testnet, and owns some ether.(Follow the tutorial on the homepage to learn how to obtain Testnet Ether)
    ii. Click the 'Sign-up' tab on the top bar of the index page. A window will load. Enter in each of the fields as following.

1. Email "test@gmail.com"
2. Password: Test1test
3. Ethereum Wallet Address: 0x39da84129C72CC8E2b7d94D9742B0c44b17955aa
4. Ethereum Wallet Password: Hidden for security reasons.
5. Click 'Create' on the pop-up window.
6. In the Firebase Console, under the authentication tab, confirm that there is an account under the given email.
7. In the Firebase Console, under the database tab, confirm the existence of a "Users" table, and that there is an entry in this table for the recently created account, containing in each of its three fields, the correct value as typed into the sign-up fields.

2. Upload a file, following the instructions in the Sprint 1 Testing.
    i. In the firebase console, in the 'files' table of the database, confirm that the 'owner' field of the file is '0x39da84129C72CC8E2b7d94D9742B0c44b17955aa'
3. Download the file, following the instructions in the Sprint 1 Testing.
    i. Confirm that, in the Metamask popup which appears upon clicking 'download', the Buyer address is '0x39da84129C72CC8E2b7d94D9742B0c44b17955aa'
        1. This is fetched from the 'wallet_addr' field of the 'users' table in the firebase database.
    ii. Confirm that, in the Metamask popup, the Receiver address is '0x39da84129C72CC8E2b7d94D9742B0c44b17955aa'
        1. This is fetched from the 'owner' field of the file in question in the 'files' table in the firebase database.

Unit testing:

Edwin

Module: Integrating transactions of Ether with download functionality.

To verify that user wallets were linking with the download process, I first utilized the web3.js console injected by MetaMask. First I had to verify that user accounts were unlocked when the user was logged in, and I had to check the balance of each account to make sure they were being updated. I utilized the following commands within the to obtain this info on the console:

-Web3.eth.accounts[0] - to obtain the wallet address and print it to the console

-web3.eth.getBalance(<wallet address here>, <callback function here>)

If the user is not logged in to MetaMask, then both commands should fail. Once these 2 factors are confirmed and pass, I then had to verify that a transaction could be prompted once a user tried to access a document. To do this I utilized the following commands from web3 to initiate a simple transaction from clicking any of the files stored in our database:

```
web3.eth.sendTransaction({from: buyer_address, to: seller_address, value: web3.toWei(0.05, "ether") },
        function(err,res){
           if(!err){
              // Transaction has finished successfuly,
              // allow user to download the file.
             //Print addresses and receipt to the console
             console.log(“Buyer Address: “ + buyer_address)
             console.log(“Seller Address: “ + seller_address)
             console.log("Confirmation Receipt: " + res)
             transaction_receipt = res;
             provide_download();
           }else{
              console.log("Transaction Failed: " + err)
           }
        });
```

What this block of code does is attempt to test a transactions, and uses a callback function to print the result. If it succeeds then the console should print the address of the buyer’s wallet

address, the seller's wallet address, and the hex transaction receipt that can be searched on the Ethereum blockchain. Additionally I had to test that the transaction would fail if the user tried to complete a transaction while their MetaMask settings were configured to anything other than the Rixeby testnet. To do this I made sure that we were only testing the receipts against the Rixeby testnet blockchain. If a user tried to complete a transaction on the Ropsten testnet, then the test should print a transaction failure. This also means that if a user creates an account, and their account is tied to a Ropsten testnet address, then all transactions will fail. Thus to prevent this, I wrote a step-by-step setup guide on the home page on what steps and configuration settings the user should enable on their MetaMask account.

Ian

A. Modular- For the upload and file browsing UI (specifically for the automated card fill), modular tests were completed locally with different numbers of cards to test the display structure. Tests were also completed with access to firebase to
B. Functional- Upload tests were completed with equivalence classes based on subject to test both the subject selection functionality as well as the tab-browser functionality. Files' subjects match the user's selection, and files are appropriately filter when a specific tab is selected

Nikola

Module: Firebase integration with list display and downloading

To ensure that Firebase was correctly storing our files in both the database and cloud storage, I first checked on the Firebase console that there were full writing permissions to both components, meaning that anyone could upload to the Firebase project. I polled the 'files' section of our database in the code to create a reference to the database, and iterated through each item in the section, logging its "name" child value to the console to verify that the correct names were being returned.

With the names in the database correctly verified, I next used the "name" field from the database to search the cloud storage component of Firebase to find a file with a matching name. When found, I logged to the console "match found", and used the Firebase API call getDownloadURL() to obtain the direct download link for the file. I then downloaded the file, and confirmed that it was indeed the correct one.

Seth

Module: Voting System

To test the voting system, I first upload a new file from the upload page. I then click the upvote and downvote buttons next to the file on the browse files page; nothing should happen, because I do not "own" that file and should not have privileges to decide whether it was good or bad. I then purchase/download the file. In that files' 'download_list' field in the firebase "files"

database, I confirm that its contents are just my email, followed by the number zero, separated by a semicolon. Every email in the 'download_list' field for a given file is followed by an integer which represents that user's vote on the file. A zero represents that the user preceding the zero in 'download_list' has not yet voted on the file. Thus, when downloading a file, the number is initialized to zero. I also check the 'upvotes' field in the 'files' table, for that file, which should also be initialized to zero.

Next, I upvote the file. I check in the firebase console that the number following 'sealittl@ucsc.edu' in the 'download_list' field of that file is now a 1, representing the upvote. The 'upvotes' field for that file should now be -1, because upvotes are represented in that field as negative numbers. I then try to upvote (using the '+' button next to the file's name) the file a second time. The number should not change from a 1, because users are only permitted to vote once per file. I confirm that this is the case. Then, I try to downvote the file, and confirm again that the number has not changed in the database.

Lastly, I upload a second file, and repeat the above testing, this time downvoting the file, and checking that the number representing my vote in 'download_list' is now '2', which means I have downvoted it. The 'upvotes' field for that file should be '1', meaning it has been downvoted once. Again, further clicks on either of the vote buttons on either file should now do nothing. In the list of files on the "browse files" page, the file which I have downvoted should now appear below the file which has been upvoted, as files are sorted in decreasing order of the 'upvotes' field for that file. This completes the testing of the voting module.

Cole

Module: Accounts

To test accounts, we need to understand the two sides user view accounts from. First, users must create an account, then users must sign into their account. This distinction is important to make, since it allows us to split up unit testing.

First, testing users creating an account. First, for the sake of a repeatable test, remove the account you are making from the database. Second, set the input fields for email and password to the test account. Then press submit. From here, the program pulls the values from the input fields, and passes them into the create account function. From there, two things can happen. Either the account is not created, or the account is. When the account is not created, the program logs the error. When the account is created, the program logs the email registered, confirming that the account used the correct email. Then, log-out and sign-in. This ensures that the password is correctly stored when the account is created.

Second, testing users logging into their account. This test is performed directly after the sign-up test, so it can use the account already created. When the user logs in, it dumps the user's current information to the logs, confirming that the proper information is being tied to the user's account.