Team: HvZ App

Members: Kyra Clark, Santiago Rodriguez, Alex Hadley, Matthew Waddell
Project 3E.1 — Final Test Plan

Github Link:
https://github.com/alexhad6/HvZ-app/blob/master/Administrative/Phase%203/test_3e_sr.pdf
Primary author: Santiago

Component Test Plan: File Storage

Goals:
Test that uploading and downloading from file storage works correctly.

Testing Framework:
The testing frameworks available for this component are flutter unit testing (https://flutter.dev/docs/testing#unit-tests). Flutter unit testing allows users to test their functions, methods, or classes within an easy to use automated testing framework. It allows for automated testing and gives clear pass/fail notifications for each test. It can also emulate necessary equipment such as a phone.

Tests:
Upload tests:

- Make sure the file storage has stored the URL of an uploaded file in the database
  - Reason: must do this so that the file can be found again later for downloading
  - Set up: Create file to upload
  - Execution: Upload a file to file storage and check if it is in the database
  - Correctness recognition: the URL passed to the addFile() function is the same as a predetermined correct URL.
  - Clean up: Delete file that was uploaded and clear the database.
- Make sure an uploaded txt file shows up in the file storage
  - Reason: must be able to upload files to file storage
  - Set up: create a txt file and an easy pathing to pass for it to be uploaded
  - Execution: upload file and list files that have been uploaded
  - Correctness recognition: list the items in the file storage and make sure the list is the same as a predetermined corect list.
  - Clean up: clear file storage
- Make sure an uploaded image file shows up in the file storage
  - Reason: must be able to upload files to file storage
  - Set up: create an image file and an easy pathing to pass for it to be uploaded
  - Execution: upload file and list files that have been uploaded
  - Correctness recognition: list the items in the file storage and make sure the list is the same as a predetermined corect list.
  - Clean up: clear file storage

- Make sure an uploaded audio file shows up in the file storage
    - Reason: must be able to upload files to file storage
    - Set up: create an audio file and an easy pathing to pass for it to be uploaded
    - Execution: upload file and list files that have been uploaded
    - Correctness recognition: list the items in the file storage and make sure the list is the same as a predetermined corect list.
    - Clean up: clear file storage
- Make sure an uploaded video file shows up in the file storage
    - Reason: must be able to upload files to file storage
    - Set up: create an video file and an easy pathing to pass for it to be uploaded
    - Execution: upload file and list files that have been uploaded
    - Correctness recognition: list the items in the file storage and make sure the list is the same as a predetermined corect list.
    - Clean up: clear file storage

Download Tests:

- Make sure a downloaded file can be accessed by the phone
    - Reason: must be able to successfully download a file from file storage.
    - Set up: put text file with one letter in file storage
    - Execution: call download function and check to see if file has been downloaded (match the letter downloaded with test letter)
    - Correctness: see that data from text file matches predetermined correct data
    - Clean up: clear file storage
- Try to download something that doesn't exist
    - Reason: handle unexpected behavior
    - Set up: None
    - Execution: call download function for a file whose url does not exist
    - Correctness: Error message matches with predetermined correct error message

Complex Behavior Tests:

- Upload 3 images to the same mission and make sure they are correctly named
    - Reason: uploading a file of the same type in a given mission folder causes each one to be enumerated based on the order they were uploaded in. This tests if they were correctly enumerated in their names in the file storage.
    - Set up: Create 3 images to be uploaded to file storage
    - Execution: Call the upload image function 3 times. Use Firebase's list directory function to make sure they have the appropriate names.
    - Correctness: compare the listed file names with predetermined correct file names.
    - Clean up: clear file storage
- Upload 3 images to the same mission and make sure they have the correct name when downloaded
    - Reason: the correct names could have been given to the wrong files when uploading, so this checks if they are matched correctly. Tests the download for all files in a mission function as well.
    - Set up: Create 3 images to be uploaded to file storage

    - Execution: Call upload image function 3 times, one for each image. Download all 3 images. Make sure the files that were downloaded match the images that were uploaded and have the correct name.
    - Correctness: The image file should match up with an image file on the device for the predetermined correct name for each image.
    - Clean up: clear file storage, clear downloads from emulated phone
- Upload 3 images to a different mission folder each and download files from one folder
    - Reason: test that the download only takes from one mission folder rather than all the mission folders
    - Set up: create 3 images and 3 folders
    - Execution: upload each image to a different mission folder. Download from one of the folders. Make sure that only one file was downloaded and that it is the desired one
    - Correctness: The downloaded file must match a predetermined correct file and must come alone (not in a list of files)
    - Clean up: clear file storage, clear downloads on emulated phone

Component Test Plan: Zombie Lineage

Goals:

Test if the zombie lineage component can successfully track the lineage of the zombies across multiple players.

Framework:

Flutter unit tests as described above.

Tests:

- Successfully create a LineageUser object
    - Reason: The LineageUser object is the basis of how information is represented in the zombie lineage component. It must be able to be created for any of the component to work.
    - Set up: fill in the database with a user who is an original zombie who has tagged no one
    - Execution: call the LineageUser constructor for the user in the database. Use the getter methods on that object for name and userID to see if it has been stored
    - Correctness: the name and userID return strings that match with predetermined correct name and ID strings
    - Clean up: clear database

- Check that the LineageUser object is stored as the current user in the ZombieLineage object
    - Reason: The ZombieLineage instance must be able to store the current user it is looking at when it is created. Otherwise it cannot relay any information to the application
    - Setup: fill in the database with a user who is an original zombie who has tagged no one
    - Execution: call the ZombieLineage constructor for the user in the database. Use the getter methods on the currentUser LineageUser object for its name and ID string.
    - Correctness: the name and ID can be retrieved from the currentUser LineageUser object and match predetermined correct strings for each
    - Clean up: clear database
- Check that the representation of a user is accurate for an original zombie who has tagged no one
    - Reason: This is the simplest representation that can be displayed, so it must be accurate.
    - Setup: fill in the database with a user who is an original zombie who has tagged no one
    - Execution: call the ZombieLineage constructor for the user in the database. Call lineageDepthOne() to get the representation of the user. Compare the representation with what was expected.
    - Correctness: The representation of the user has a list where the first item is null (no person tagged this original zombie), the second item is the LineageUser being examined, and the third item is null (the user has not tagged anyone)
    - Clean up: clear database
- Check that the representation of a user is accurate for an original zombie who has tagged someone
    - Reason: it shows that the hierarchy part of the lineage is working, that you can trace lineage in the tagged direction.
    - Setup: fill in the database with a user who is an original zombie who has tagged one person. Make sure the person who was tagged is also in the database.
    - Execution: call the ZombieLineage constructor for the user in the database. Call lineageDepthOne() to get the representation of the user. Compare the representation with what was expected.
    - Correctness: The representation of the user has a list where the first item is null (no person tagged this original zombie), the second item is the LineageUser being examined, and the third item is another LineageUser which represents a person who has been tagged.
    - Clean up: clear database
- Check that the representation of a user is accurate for the first person that the original zombie tagged
    - Reason: it shows that the hierarchy part of the lineage is working, that you can trace lineage in the tagged by direction.

    - Setup: fill in the database with a user who is an original zombie who has tagged one person. Make sure the person who was tagged is also in the database.
    - Execution: call the ZombieLineage constructor for the user who was tagged in the database. Call lineageDepthOne() to get the representation of the user. Compare the representation with what was expected.
    - Correctness: The representation of the user has a list where the first item is the LineageUser object representing the original zombie, the second item is the LineageUser being examined (the one who was tagged), and the third item is null because this person has not tagged anyone
    - Clean up: clear database
- Check that lineage can be viewed at a depth of two
    - Reason: this functionality is important to give players an expanded way of viewing zombie lineage
    - Setup: add players A,B,C,D,E to the database such that A is an original zombie who tagged B, B tagged C, C tagged D, and D tagged E
    - Execution: call the ZombieLineage constructor for player C. Call lineageDepthTwo() to get the representation of the user at 2 levels of depth. Compare with what was expected.
    - Correctness: The representation of the user at this depth is a list of lists. The first element in the list should be the list that is player B's representation. For simplicity, consider the capital letters to be each player's corresponding LineageUser object. Player B's representation is [A,B,C]. Next in the list would be a list form of the player we are observing: [C]. Finally, there would be the representation of player D, the player that player C tagged: [C,D,E]. The final product should look like this: [[A,B,C],[C],[C,D,E]]
    - Clean up: clear database
- Change the user that is currently being observed by the ZombieLIneage object
    - Reason: this is the way that the players can traverse through the zombie lineage. Once they access the representation of the lineage, they can select one of the players to go through their lineage. Thus it is imperative that they are able to traverse through different users.
    - Setup: fill in the database with a user who is an original zombie (A)who has tagged one person. Make sure the person who was tagged (B) is also in the database.
    - Execution: call the ZombieLineage constructor on the original zombie. Call the changeUser function on the ZombieLineage object with the parameter of the tagged player's LineageUser object. Call the lineageDepthOne() function to view the tagged player's representation. Compare with what is expected.
    - Correctness: For simplicity, consider the capital letters to be each player's corresponding LineageUser object. The representation should look like this: [A,B,null].

Addressing Feedback:

- Feedback: Using the actual Firebase services seems better for testing than using mockito to emulate every aspect of them.
    - Source: Professor Kampe
    - **Change Accepted**: Using mockito for everything would mean that I am testing my mockito functions as much as I am testing my own functions. Plus, building mockito functions that emulate firebase functionality would be extremely time consuming and complex. Mockito is no longer used in the test plan.
- Feedback: It is not entirely clear that the Firebase unit tests are automated and give clear pass/fail notifications.
    - Source: Professor Kampe
    - **Change Accepted**: It has now been plainly stated that Firebase unit testing does provide this functionality. This is important because testing without a good framework will not yield the desired results. It must be made clear that the framework is sufficient in its capabilities before moving on.
- Feedback: The testing suite is largely incomplete and must be completed to test the functionality of the components.
    - Source: Professor Kampe
    - **Change Accepted**: The testing suite has been filled in. It must be complete to actually verify that the code is working properly. Without a complete test suite, there is no way to guarantee that the code functions okay, at the very least.
- Feedback: The File storage does not have enough complexity for many white box tests
    - Source: Professor Kampe
    - **Change Accepted**: The zombie lineage component has been added and will provide white box tests with its complexity.