Team: HvZ App

Members: Kyra Clark, Santiago Rodriguez, Alex Hadley, Matthew Waddell
Project 3E — Final Specifications

Plan:
https://github.com/alexhad6/HvZ-app/blob/master/Administrative/Phase%203/spec_3e_sr.pdf
Primary author: Santiago

Slip days: 3 By Santiago

File Storage Component Specifications

Functionality of the component to be designed:

The file storage will be responsible for storing and retrieving large files. These include files such as images, files containing large amounts of text, or audio files. This component will have all of the functions related to storing these files in a way that can be easily accessed as well as functions for retrieving those files. For the current version of the app, all files will be uploaded to or downloaded from file storage by the app. Moderators will be able to tell the file storage to reorganize itself for a new game.

The file storage system itself will be implemented with Cloud Storage from the Firebase suite of app development tools. The code to access the storage system will be using Flutter, a Google software development tool. The library FlutterFire has functions that allow Flutter to interface with the Firebase tools.

Overall role:

The file storage will be organized into folders for each game hosted and for each mission within each game. Its main purpose is to supply the images, audio files, and mission texts for each mission that the moderators create. The moderators will be able to upload these files from their apps, and the players' apps will automatically download them and format them for viewing.

Meeting Requirements:

The moderators will be able to run distinct games by being able to set different folders for each game in the file storage. The players will be able to access mission information such as images and the associated text through the file storage component. The app will download the required files and display it to the users when appropriate. The moderators will be able to update this mission information or add on to it through the app.

Architecture Connections and Interfaces:

Application ←→ File Storage:

The application will be able to call a function to tell the file storage system to reorganize its folders for a new game. The application will be able to upload files such as images, text, video, and audio to the file storage. The application can also download these files from the file storage. Finally, the application will be able to tell the file storage to create a new folder for a new mission within a game. The images, text files, videos, and audio files would then go into the mission folder. Only moderators would be able to upload to file storage (as dictated by the security component).

Database ←→ File Storage:

The file storage will send a string of the path that an uploaded file can be reached at to the database. For example, an image put in a mission 1 folder will have a path that somewhat resembles this string: “storage/game2021/mission1/image.jpg.” The path string is then received by the file storage from the database when the application requires the specified files. This is important for being able to locate the necessary files in the file storage at a later point in time. Otherwise the application would have no way of knowing how to navigate the file storage to get the correct files.

Public Methods for Interfaces:

- Application ←→ File Storage:
  - ```
    Future<int> uploadImage(int missionNumber, String filePath)
    async
    ```
    - Function: uploads an image to the file storage in the correct mission folder.
    - Caller: application
    - Input: mission number (int), image file path on mobile device (string)
    - Output: success/failure indicator (int)
  - ```
    Future<int> uploadTxt(int missionNumber, String filePath)
    async
    ```
    - Function: uploads a text file to the file storage in the correct mission folder.
    - Caller: application
    - Input: mission number (int), text file path on mobile device (string)
    - Output: success/failure indicator (int)
  - ```
    Future<int> uploadVideo(int missionNumber, String filePath)
    async
    ```
    - Function: uploads a video to the file storage in the correct mission folder.
    ```

        - Caller: application
        - Input: mission number (int), video file path on mobile device (string)
        - Output: success/failure indicator (int)
    - ```
      Future<int> uploadAudio(int missionNumber, String filePath) async
      ```
    -
        - Function: uploads an audio file to the file storage in the correct mission folder.
        - Caller: application
        - Input: mission number (int), audio file path on mobile device (string)
        - Output: success/failure indicator (int)
    - ```
      File downloadFile(String downloadURL)
      ```
    -
        - Function: downloads a file from file storage onto mobile device
        - Caller: application
        - Input: image path (string)
        - Output: File (or file not found exception)
    - ```
      Future<int> newGame(String gameName) async
      ```
    -
        - Function: create a new game folder to be used for the new game's files
        - Caller: application
        - Input: name of the game (string)
        - Output: success/failure indicator (int)
    - ```
      Future<int> newMission(int missionNumber) async
      ```
    -
        - Function: create a new mission folder in the current game's directory
        - Caller: application
        - Input: mission number (int)
        - Output: success/failure indicator (int)
    - ```
      List<File> downloadMissionFiles(int missionNumber)
      ```
    -
        - Function: download all the files for a given mission
        - Caller: application
        - Input: mission number (int)
        - Output: list of the files (file list)
- Database ←→ File Storage:
    - `Future<void> addFile(String missionNum, String fileID, String url)`
        - The Database will store the URLs as strings of all the objects in file storage. This function will write the file URL and create a new object in the database.
      ```

        - Called By: File Storage
        - Input: File URL as a String, Mission Number, File ID
        - Output: NA (the file URL will appear under a new entry in the database associated with the mission number.)
    - `List<String> getMissionFiles(String missionNum)`
        - Function: database retrieves paths of files to be located from the file storage.
        - Caller: file storage
        - Input: mission number (int)
        - Output: file paths (string array)

Component Suitability:

This component fits the size requirement almost immediately as each upload or download function takes about 10 lines of code each. All the public methods put together easily bring the component over 100 lines of code.

The complexity in this component was supposed to come from properly caching the files. However, this is made much less complex by the use of an already provided library (https://pub.dev/packages/flutter_cache_manager_firebase).

This component is easily testable using Firebase's testing suite. There is an emulator that allows automatic testing (https://firebase.google.com/docs/rules/unit-tests). This would allow actions such as testing if something was properly downloaded from file storage from the app. The testing suite allows testing the file storage code in relative isolation without the rest of the application code being complete.

Zombie Lineage Component Specifications

Functionality and Overall Role of the component to be designed:

The zombie lineage component allows players to track the lineage of each zombie. The player is able to view a specific player and see who tagged that player and who that player has tagged. From there, the player can investigate these other players to see who they were tagged by and who they've tagged. This allows the player to traverse through the entire relationship between the zombies (tagged and tagged by). The player is able to view this at one or two levels of depth in a single view.

Architecture Connections and Interfaces:

Application ←→ Zombie Lineage:

The application instantiates an instance of the ZombieLineage class to allow the user to traverse through the zombie relationships. The zombie lineage component gives the information to the application, and the app takes care of displaying the information to the user. The zombie lineage class represents the users in a way that is easy for the application to read from and display to the users.

Database ←→ Zombie Lineage:

The zombie lineage component uses database functions to get information on the users being inspected. These functions provide information about which users tagged which other users and by whom those users were tagged by. The database holds all the information necessary to construct the relationships between the zombies. The zombie lineage component takes this information and transforms it into something meaningful and accessible to the users.

Public Methods for Interfaces:

- Application ←→ Zombie Lineage:
  - ```
    ZombieLineage(String userID, Database database)
    ```
  -
    - Function: Creates a ZombieLineage object (constructor). Should be called every time a user wants to start a new session of using this class
    - Caller: Application
    - Input: user ID (string), database instance (Database)
    - Output: None
  - ```
    List<LineageUser> lineageDepthOne()
    ```
  -

        - Function: Passes on the representation of the zombie lineage at one level of depth to the application
        - Caller: Application
        - Input: None
        - Output: List of lineage users representing the zombie lineage of the current user being examined (List<LineageUser>)
    - `List<List> lineageDepthTwo()`
    -
        - Function: Passes on the representation of the zombie lineage at two levels of depth to the application
        - Caller: Application
        - Input: None
        - Output: List of lists of lineage users representing the zombie lineage of the current user and the zombie lineage of the users that the current user is connected to (List<List>)
    - `void changeUser(LineageUser newUser)`
    -
        - Function: Changes the user that the ZombieLineage object is focusing on to a different user passed in by the application. This is how a player can traverse through the zombie lineage.
        - Caller: Application
        - Input: the user to center the lineage observations on (LineageUser)
        - Output: None (the current user being inspected is changed to this user)
- LineageUser Class
    - `String get userName`
    -
        - Function: Getter function for the name of a LineageUser object. This should be the real name of the player that the object represents. Useful for the UI of this entire component.
        - Caller: Application
        - Input: None
        - Output: player's name (string)
    - `String get userID`
    -
        - Function: Getter function for the user ID in the database of the LineageUser object. This is useful for accessing more information about the player that the LineageUser object represents.
        - Caller: Application
        - Input: None
        - Output: player's database user ID (string)
- Database ←→ Zombie Lineage
    - `String getName(String userID)`
    - `String getTaggedBy(String userID)`

    - `List<String> getTagged(String userID)`
        - All of these database functions are called by the LineageUser class of the Zombie Lineage component to get the name of the user being examined, the person who tagged the user being examined, and the people the user being examined has tagged.

Addressing Feedback:

- Feedback: The specification needs to include the function names so that other people can write code that calls these functions.
    - Source: Professor Kampe
    - **Change Accepted**: This is a basic feature of any specifications document, so it is accepted and each function has a name to it now.
- Feedback: The download functions were inconsistent with the design. There were download functions for each type of file in the specifications, but there was only one function for all files in the design.
    - Source: Professor Kampe
    - **Change Accepted**: The inconsistency was removed by consolidating all the download functions into one in the specifications. Further research since writing the specifications showed that the 4 different download functions were redundant.
- Feedback: There should be a function that downloads all the files for a mission rather than having to download each one individually.
    - Source: Alex Hadley
    - **Change Accepted**: This is the primary use of file storage: to download files for missions. This has been modified in the specifications as well as the design.
- Feedback: There should be a sentence describing the reason there is a connection between file storage and the database.
    - Source: Alex Hadley
    - **Change Accepted**: It may distract from the specifications if there is no observable reason why the file storage functions are included. The description previously stated the functionality, but now it includes the reason as well.
- Feedback: File storage component is not complex.
    - Source: Professor Kampe/Santiago
    - **Change Accepted**: The file storage does not do enough complex behavior on its own. The cache component was supposed to add complexity, but there was a library that already took care of it. A new component, the Zombie Lineage component, has been added to supplement the file storage component with complexity. This component must keep track of the relationships between zombies like a data structure. Cycling through the relationships between the different zombies adds the complexity.