Accesa Internship Project Documentation

Application name: Accesa Quest

Coding language used: Java

Building tool: Maven

Storage: Azure Cosmos DB

Demo Link: https://youtu.be/VnU_WdmD5to

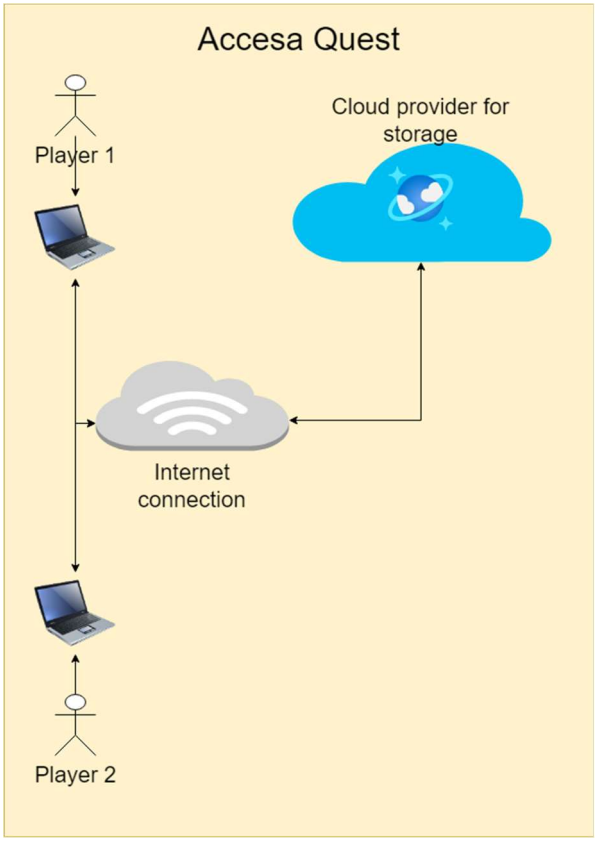

Fig. 1: Layout of the user interaction

For the application to work as we can see from Fig. 1 players need to be connected to the Internet. The players just need a stable connection and to have JVM installed on their desktop/laptop.

The application is packaged in a .jar. When they open the application, they will be prompted with a screen where they choose to log in or to register.

After they are logged in, they will be redirected to the “Profile” page, where they can see something like shown in Fig. 2. On the Profile screen they can see the quests that they are involved in (the quests they created, the quests that they took) and the badges that they earned using the application.

Fig. 2: Profile scene

They will be able to take quests from the “Home” page (Fig. 3) which are made by other people (The page will display only 10 random quests from the database that the player can take, if they want to see other quests, they can press the “Home” button again to get other results, this was done to make the application more scalable). They can also create quests from this page by pressing the “Create quest” button.

Fig. 3: Home Screen

On the Leaderboard screen (Fig. 4) they can see a leaderboard of the highest ranked players (top 10) of all time based on their rank, you can also see the badges that they earned.

Fig. 4: Leaderboard

Technical Aspects:

For login and register, there are regexes in place for the email and the password. Email needs to be a valid email (not necessarily accessible, because it won't send a confirmation link). The password needs to be between 4 to 8 characters containing at least one of each: lowercase letter, uppercase letter, number.

All the queries that go to the database are sanitized to prevent SQL Injection, by preparing the statements with parameterized queries.

The passwords are salted and hashed, using PBKDF2 (Password-Based Key Derivation Function) before being added to the database, as a security measure.

The input for tokens, in the create quest area is checked for valid input (an integer that is higher or equal to 0).

Technologies used:

Java 20 – Used the latest version of Java as the Internship is for a Java position.

Java FX – Used to have access to the development of interfaces as it provides a lot of features that can be set up easily for display.

FXML + Scene Builder – For the files that deal with the UI as they can easily be added to Scene Builder where you can have most of the code regarding the layout of the scene easily and it's easy to see how the elements look together. It was also easy to add CSS styling to the elements using these 2 methods.

IntelliJ Idea – IDE – Used because of the functionality that it provides regarding creating of the project and the variated options of build methods that can be used to build the project automatically through it. Also, the code completion it provides and the extensions, such as support for Azure, by having an explorer where you can easily see the databases through the IDE. It provided an easy way to commit to the GitHub repository too.

Azure Cosmos – Storage DB – Used to store all the data about the players and the quests, the NoSQL providing an easy way of storing the input in a JSON format and allows fast prototyping. There's also no need to worry about the database in this stage of the project as it will be handled by the cloud provider.

Testing – Junit5 – Used to have a Test-Driven Development methodology and to easily test the functions that are in the backend. Making the tests in the early stages allowed for Regression Testing which could be performed after new features were added, doing this before pushing to Git to make sure that the code works.

Maven – Building Tool – Used as it provided an easy way to maintain all the dependencies into a pom.xml file and all the build requirements later used for packaging the application too. I used this over Gradle as I was more experienced with it.

MVC (Model View Controller) – Pattern used to split the code to make it maintainable and easy to add more features to it (Fig. 5)

Lucid Chart – Drawing the class diagram (Fig. 6) and the Sequence diagram (Fig. 7) which were done in the beginning to make the development of the code structured and later edited to follow the structure of the code as during development some of the things changed (e.g., I added the Security class to Util so that I can hash the passwords before putting them in the database)

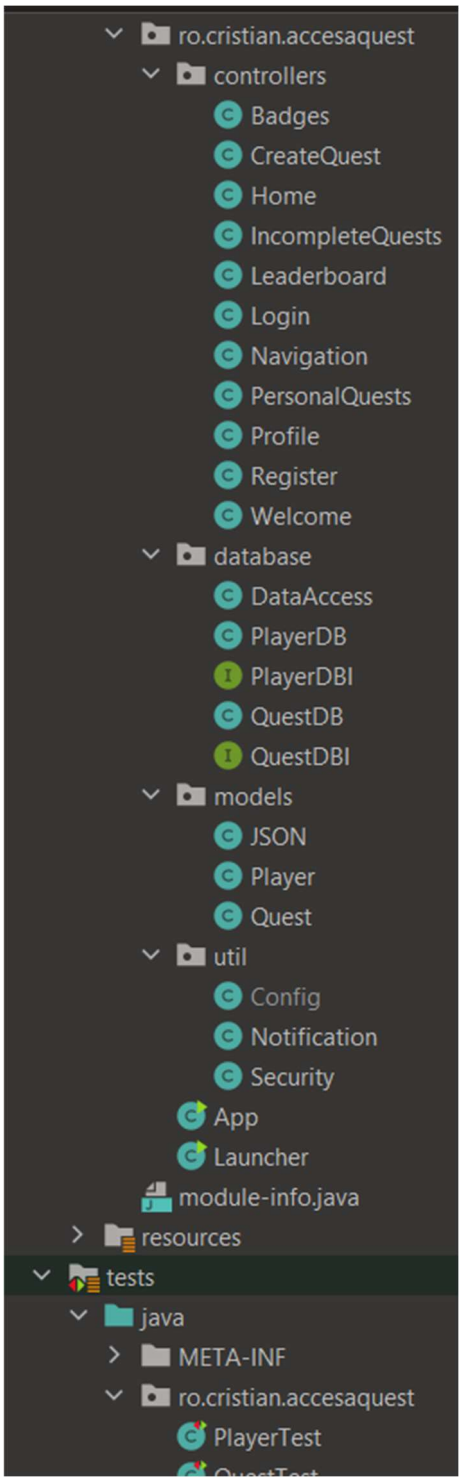

Fig. 5: The layout of the files

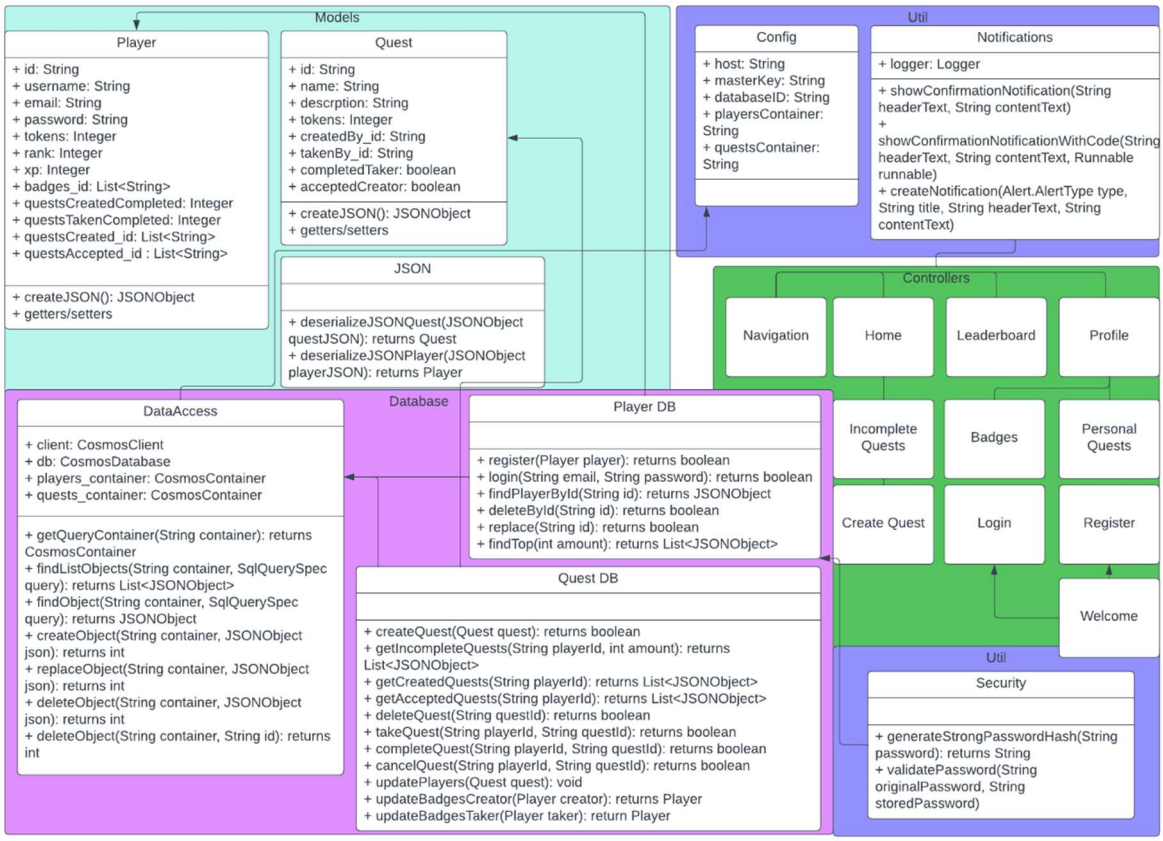

Fig. 6: Class Diagram

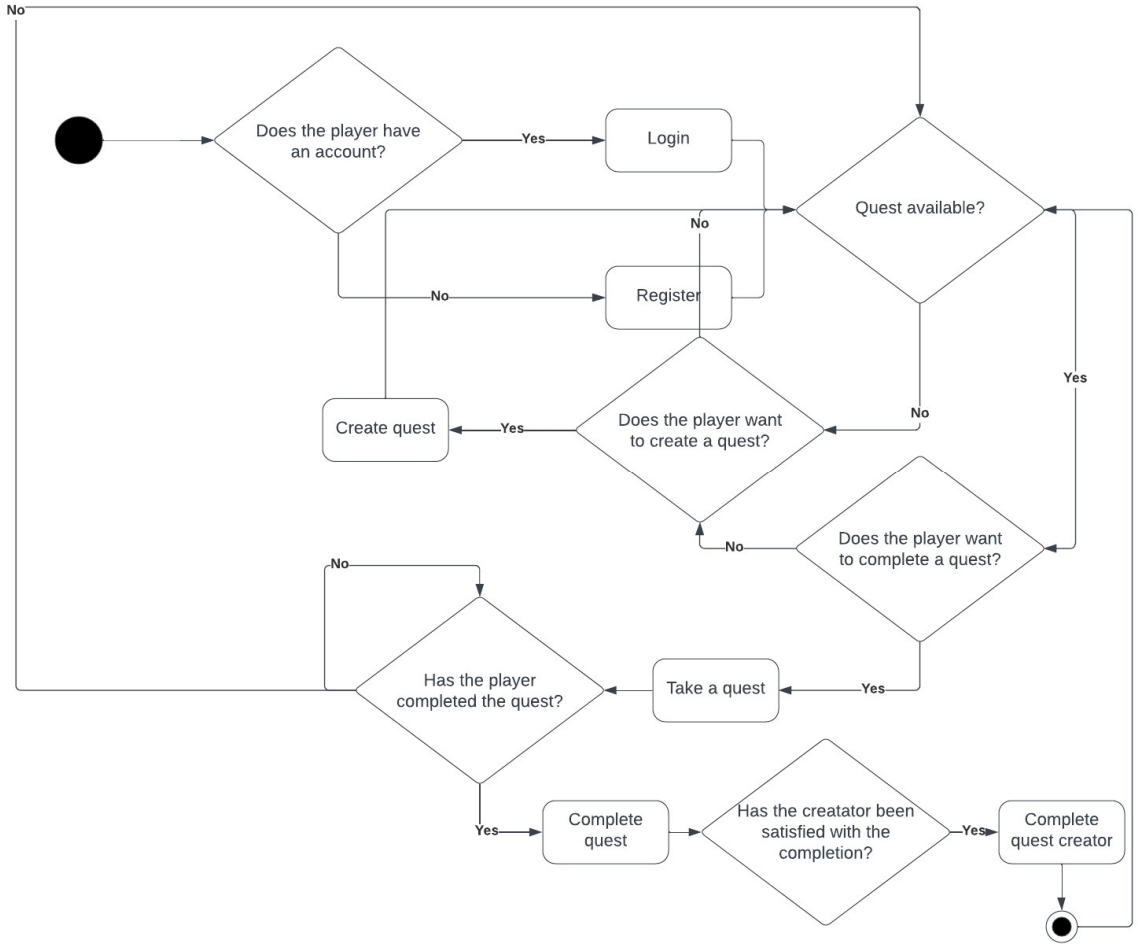

Fig. 7: Sequence diagram

Limitations:

Dynamic reload - Currently the application is not dynamically reloaded, this can be achieved by having a new container in the database which stores the connections in a format {String: player_id, Socket: connection, Integer: expiryTime}. This way whenever an operation is completed (someone accepts a quest that the player has created) there is a notification for update to the socket that corresponds to the other's player_id and it can trigger an update of the view.

Badges stored on the database – This can be done when there are more badges and it would make the application too big, but for the purposes of this project the approach of having the images stored locally was to show that this can also be done.

Admins – This is a feature that can be added later because currently the players might reach a conflict where the quest is stuck in an endless loop one of the players not accepting the completion (typically the creator). The scam technique from the creator side was tried to be taken care of by having the tokens removed from his account as soon as he creates the quest, but this makes the taker of the quest to accept mindlessly the quests and mark them as complete, without completing

them, which is a bad thing. Given these scenarios there's a need for a middleman between the two parties to make sure that the quest has been completed.

Functions on cloud – Currently there's no deployment of the query functions to a cloud infrastructure as I considered the functions are not requiring heavy computation, but if a feature that requires big computation power would be implemented then the functions can be moved remotely, this should be an easy process as there's a package "database" which handles all the database interactions, so these can be moved to the cloud by having functions for each of the methods such as "login", "createQuest" etc.

Thank you for reading through my documentation. Any feedback is welcome.