# Final Project Report

**Team Members**

- Drishti Pareshbhai Sabhaya
- Mili Parikh
- Vaishnavi Rakeshbhai Shah

**Project Objectives**

We have implemented a Distributed Password Storage System project. We are trying to create a system where the users can store, retrieve and delete passwords once they are signed up or logged in. This system ensures that the operations performed on the password storage are end-to-end encrypted. The password will be only visible to the user and then encrypted and stored in the database. Similarly at the time of retrieval of password, it is decrypted first and then displayed to the user. The user must be logged in order to perform get, put and delete operations on the datastore.

This project uses multiple servers to perform all these operations. The servers are considered to be geographically separated and replicas of the information about the user will be stored in each of the servers. There will be a centralized server called the coordinator which will manage the communication between all the servers. The coordinator will be responsible for creating new servers, dropping servers and maintaining the sharing of information between servers. The server contains two types of datastore at their end: one is used to store password and other is used to store users login/signup information. All the servers will communicate through coordinator. First the coordinator will start and register the server, after which if the new server comes then the coordinator will send the datastore from the existing servers to the new server. This way every server is up to date with the users information.

The client will be able to perform three operations: GET, PUT and DELETE. The client needs to signup first if they don't have an account and then they can store passwords by performing the operations mentioned above. The client will connect to the server that is nearest to them. Now if the client moves to a different region and connects to a different server then retrieves the data then the coordinator must transfer the client's information from the old server to the new server.

Overall, the client will have a secure system of distributed storage of passwords wherein they can login anytime and retrieve the information they want.

**Overview**
**Key Algorithms:**
**1. Distributed Concurrency Control:**
Concurrency controlling techniques ensure that multiple transactions are executed simultaneously while maintaining the ACID properties of the transactions and serializability in the schedules. We have two important resources that can have transactions happening concurrently on them. One is the user credentials map using which a user is signed in or signed up and other is the password storage which contains the name and value of the password. We initially planned on using token based mutual exclusion to ensure the concurrency control in user map, but found mutual exclusive locking mechanism to be better suited given our use case. In this method, each transaction locks an item before use and releases the lock as soon as it has finished using it. This locking method provides for maximum concurrency but does not always enforce serializability.

**2. Distributed Transactions:**
A distributed transaction is a set of operations on data that is performed across two or more data repositories (especially databases). It is typically coordinated across separate nodes connected by a network, but may also span multiple databases on a single server.

We used distributed transactions, which we learnt about in the course, to accomplish consistency. We can respond atomically to client requests thanks to transactions. When dealing with atomicity, there are a number of important considerations to keep in mind, including the All or Nothing and Isolation factors. The All or Nothing factor states that all operations must either be successfully completed or have no impact in the event that the server crashes (for example, a get request that does not modify any data).

We utilized the Two Phase Commit (2PC) atomic commit mechanism to make sure that our transactions adhered to the ACID properties. In 2PC, a server plays the part of a coordinator whose responsibility it is to organize the transaction between all servers. Every time a request is made, the coordinator notifies all servers to be ready to commit. The servers give a response of either commit or abort and keep the transactional data in local storage. Following then, the coordinator starts the second phase of 2PC and instructs the servers to either commit the transaction if and only if all servers answered with commit, or to abort the transaction, if not. When a server recovers from a phase 1 failure, it simply looks up the transaction data on the local server before committing it pending instructions. This helps in making our system fault-tolerant and robust.

**3. Group Communication:**

Communication between two processes in a distributed system is required to exchange various data, such as code or a file, between the processes. When one source process tries to communicate with multiple processes at once, it is called Group Communication.

For this project we have used two different approaches for group communication:

**a) Interprocess Communication:**

A distributed, scalable and robust password storage system must be deployed and synchronized on multiple servers to achieve horizontal scaling because there can be instances where our server encounters a large number of concurrent requests. To address this issue we used interprocess communication between the client and the server. We discussed a number of communication methods that we had studied throughout the course, including Remote Procedure Calls (RPC), Remote Method Invocation, and transmitting data across sockets using the TCP or UDP protocol (RMI). However, we decided to go ahead with RMI for multiple reasons.

RMI stands for Remote Method Invocation. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM. RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java.rmi. RMI, which adheres to Java's object-oriented paradigm, makes it simple to transfer objects from the client to the server. This makes it extremely simple to specify behavior by designing and putting in place interfaces that provide a particular function. We can utilize Java's built-in security features with RMI as well. The fact that RMI is multi-threaded, which enables us to efficiently use threads to handle concurrent client requests, is a highly significant feature.

**b) Indirect Communication:**

For communication between the server and the coordinator, we used Indirect Communication because we wanted to decouple the functionality of the coordinator and the server and both should communicate without necessarily knowing anything about each other. With indirect communication, senders and receivers can be highly decoupled from one another because it is conducted through a third party. Receivers do not need to know who they are receiving from, and senders do not need to know who they are sending to in space uncoupling. Senders and receivers do not have to be online at the same time in time uncoupling. Multiple techniques, such as group communication, publish-subscribe systems, distributed shared memory, tuple spaces, and message queues, might have been used to accomplish indirect communication.

We decided to leverage Message queues for this purpose. We built up a method of communication between our Coordinator and Server using a message queue so they are not need to be aware of one another. The Coordinator needs to know how many servers are in the cluster, thus this is helpful. All a new Server needs to do to join the cluster is send a message to a designated topic notifying the Coordinator that it has done so. When a Server departs the cluster, the same is done.

## 4. Fault Tolerance:

Fault-tolerant distributed computing refers to the algorithmic controlling of the distributed system's components to provide the desired service despite the presence of certain failures in the system by exploiting redundancy in space and time.

We made the decision to implement replication enabled by distributed transactions in order to achieve scalability and high speed. We get better performance, high availability, and fault tolerance via replication. All of our server instances duplicate the user-stored passwords.

The system is partially fault tolerant since the user may still view the most current versions of the passwords they submitted even if a server goes down. Consistency is a crucial criterion for duplicated data. All instances must be subject to the same operations that are done on one instance which as we discussed earlier is ensured by 2PC.

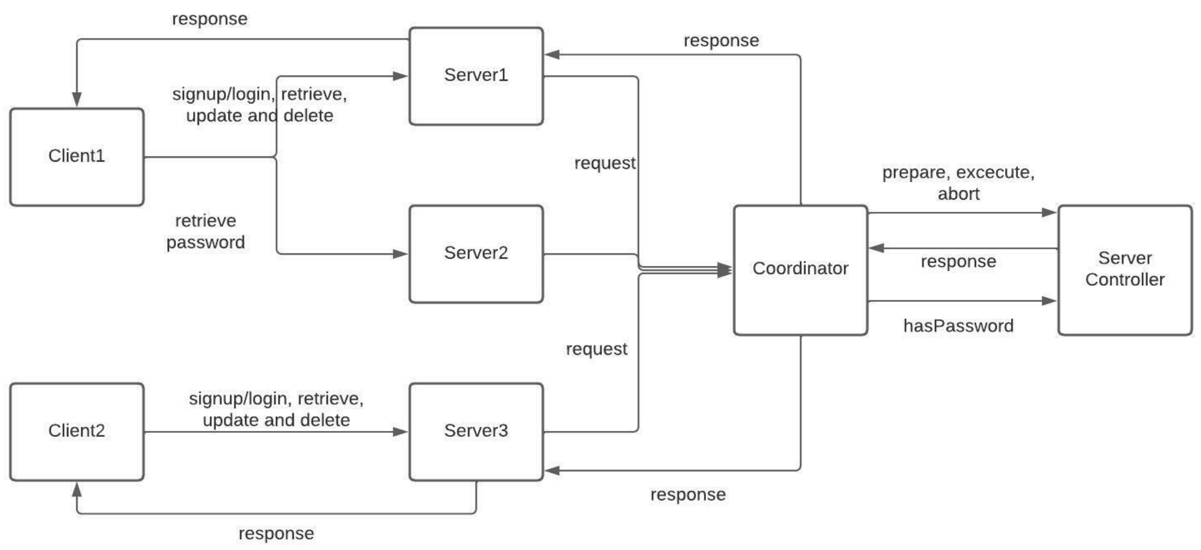
## 5. Encryption and Decryption:

Encryption is the process of converting normal messages (plaintext) into meaningless messages (Ciphertext). Decryption is the process of converting meaningless messages (Ciphertext) into its original form (Plaintext). For this project we used the AES algorithm to encrypt and decrypt the password that the user wants to store in our system to ensure that the password cannot be leaked from the server under any attacks.

**Technical Impressions**
Our strategy involves storing the passwords along with their key names on the server they were uploaded to and then retrieving them as needed. It also entails replicating user credentials across each replica.
The password manager has four main components : client, server, coordinator and server controller.

- The client performs operations like sign-up, login or retrieving / updating passwords from the server. It uses RMI Registry to connect to a server and make remote calls to server methods in order to execute operations.
- Each server consists of 2 types of datastore : one for storing the signup / login information of the client and another datastore for storing the credentials of the user for different passwords.
- The main features of our program, such as password storage and retrieval, are exposed to the client by the client-facing server, also known as the “ClientServer”.
- In order to establish a connection with a server and use remote server methods, the client leverages Java's RMI Registry.
- The “CoordinatorServer”, a coordinator-facing server that exposes the server's internal functionality to the coordinator, enables the coordinator to change the server's status and conduct polling activities. These consist of operations such as prepare, execute, abort, hasPassword, and others.
- Each server keeps two separate datastores: one for user credentials and the other for password storage.
- We are enabling the locking mechanism for accessing the resources when multiple processes/operations are trying to access the same resources. The coordinator is responsible to ensure the data reliability across all the replicas in the servers. If a resource is locked, no changes can be made until it is unlocked. If many processes attempt to alter the same resource, this will aid in maintaining database consistency.
- All of the servers that are active at any given time are accessible to the coordinator. The coordinator is in charge of making sure that all server replicas are duplicated using 2PC for any changes that can affect shared data.

Below is the mentioned architecture diagram of our project :



The order of operations for starting the server module is as follows:

1. The coordinator first starts up. The coordinator exports a remote object that is bound to a name in an RMI registry.
2. Next, the server starts running:
   - ➔ The server first develops its own data stores.
   - ➔ After that, it searches the registry for the coordinator's remote object.
   - ➔ The coordinator is located, the coordinator server starts all the server modules, exports the object to the registry, and then the servers are ready for any client to connect.
   - ➔ It then initializes the remote server object that the coordinator sees and requests that the coordinator add this server to its list of servers.
   - ➔ The coordinator locates the remote object for the coordinator-facing server after receiving the request and includes it in the list of servers it keeps.
   - ➔ The server now requests the coordinate to synchronize the data so that it is consistent between the data on the newly established server and the data on any existing servers.
3. Requests from clients can now be accepted by the server.

For the events involving executing operations:

1. The client first makes a request to the server with a specific request such as retrieve (GET), update(PUT) and delete(DELETE) password.
2. If the server has data regarding the password queried, then it returns it or else it forwards the request to the coordinator.

3. The coordinator then requests the server coordinator to retrieve data from the specified server and forward the response to the first server.
4. For operations like signup/login, the client first calls the specified method on the server. The server forwards the request to the coordinator and then to the server controller.
5. The server controller prepares all the servers and asks whether they are ready to perform that operation.
6. If all the servers are ready, then signup/login is executed and the response of this operation is conveyed to the client.
7. If any of these servers are not ready, then the server controller aborts the operation and the local memory is cleared in the servers.

**Running the Project**

The project consists of 3 jar files: coordinator.jar, server.jar and client.jar.

We need to start the coordinator first using the following command:

```
java -jar Coordinator.jar -jp <jmsport> -p <rmiport>
```

After starting the coordinator, we can start the server using:

```
java -jar Server.jar -p <selfport> -cp <coordinatorjmsport> -ch
<coordinatorhost> -n <servername>
```

We can start multiple servers by using the coordinator JMS port.

After starting the coordinator and server, we can start the client by using:

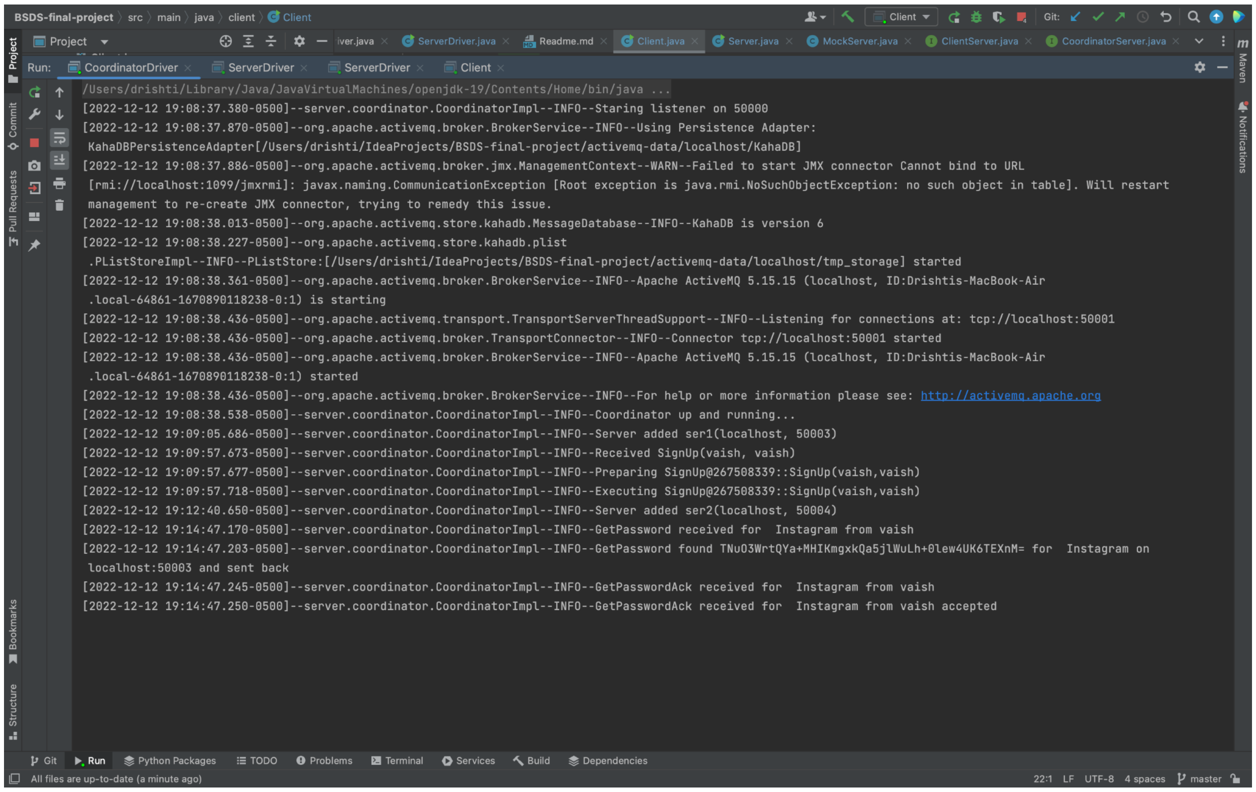
```
java -jar Client.jar <port> <hostname>
```

**Demo of the project**

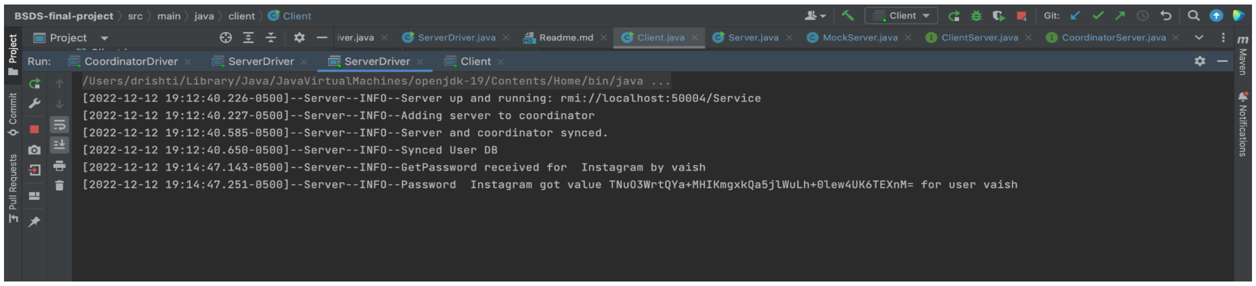
https://drive.google.com/file/d/1RTpHbLI3lIh8IR8HmW98mGwn0N1R9eXD/view?usp=sharing

## Results

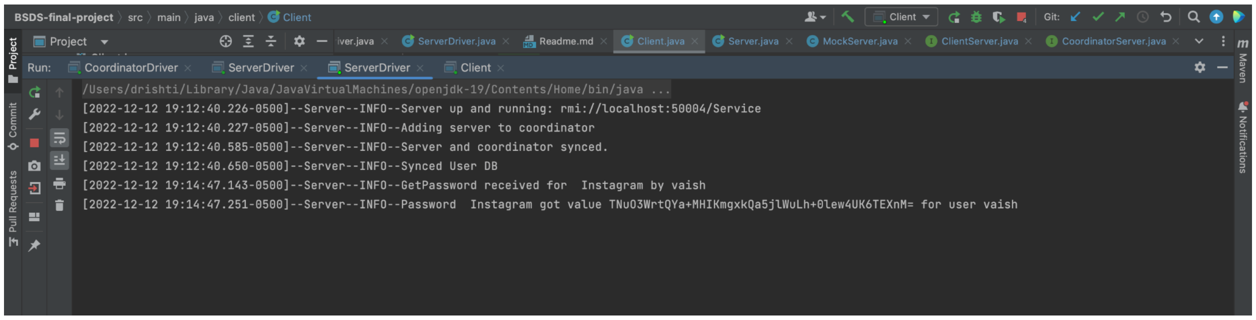
### Coordinator logs:



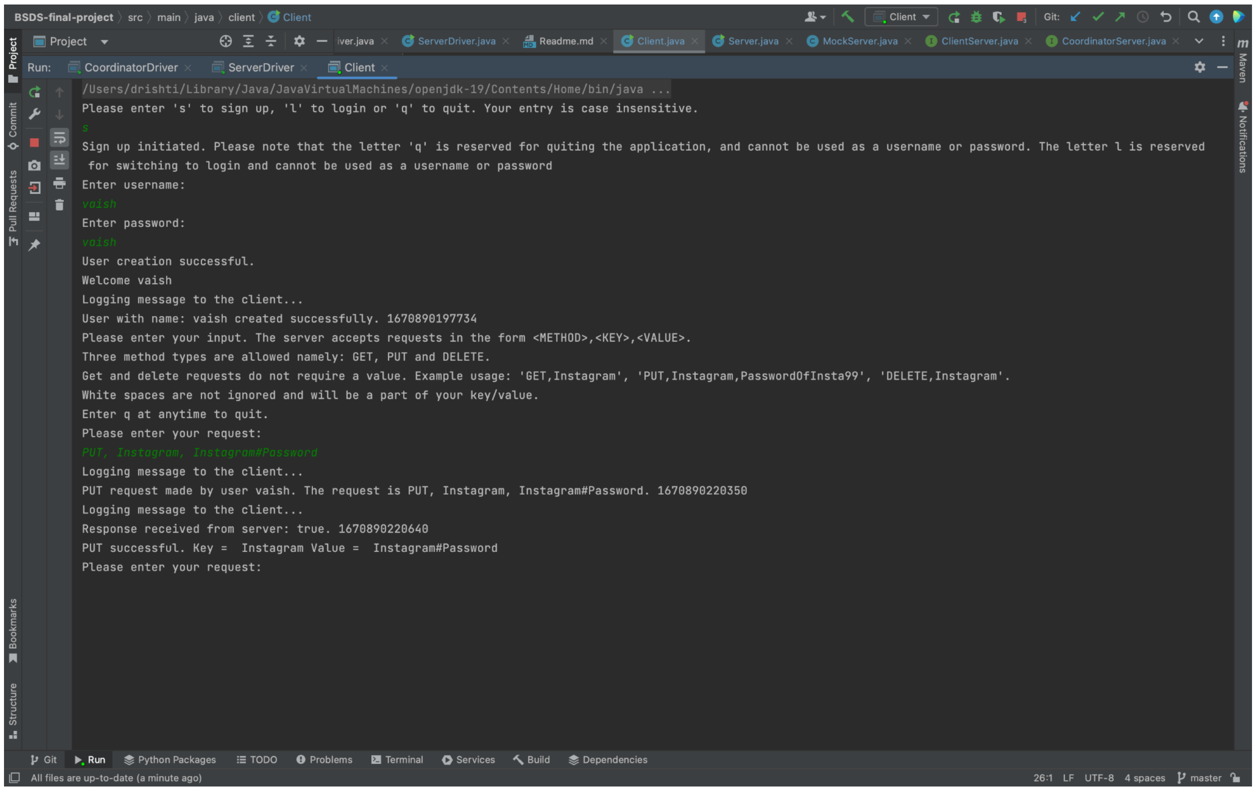
```
/Users/drishti/Library/Java/JavaVirtualMachines/openjdk-19/Contents/Home/bin/java ...
[2022-12-12 19:08:37.380-0500]--server.coordinator.CoordinatorImpl--INFO--Staring listener on 50000
[2022-12-12 19:08:37.870-0500]--org.apache.activemq.broker.BrokerService--INFO--Using Persistence Adapter:
 KahaDBPersistenceAdapter[/Users/drishti/IdeaProjects/BSDS-final-project/activemq-data/localhost/KahaDB]
[2022-12-12 19:08:37.886-0500]--org.apache.activemq.broker.jmx.ManagementContext--WARN--Failed to start JMX connector Cannot bind to URL
 [rmi://localhost:1099/jmxrmi]: javax.naming.CommunicationException [Root exception is java.rmi.NoSuchObjectException: no such object in table]. Will restart
 management to re-create JMX connector, trying to remedy this issue.
[2022-12-12 19:08:38.013-0500]--org.apache.activemq.store.kahadb.MessageDatabase--INFO--KahaDB is version 6
[2022-12-12 19:08:38.227-0500]--org.apache.activemq.store.kahadb.plist
 .PListStoreImpl--INFO--PListStore:[/Users/drishti/IdeaProjects/BSDS-final-project/activemq-data/localhost/tmp_storage] started
[2022-12-12 19:08:38.361-0500]--org.apache.activemq.broker.BrokerService--INFO--Apache ActiveMQ 5.15.15 (localhost, ID:Drishtis-MacBook-Air
 .local-64861-1670890118238-0:1) is starting
[2022-12-12 19:08:38.436-0500]--org.apache.activemq.transport.TransportServerThreadSupport--INFO--Listening for connections at: tcp://localhost:50001
[2022-12-12 19:08:38.436-0500]--org.apache.activemq.broker.TransportConnector--INFO--Connector tcp://localhost:50001 started
[2022-12-12 19:08:38.436-0500]--org.apache.activemq.broker.BrokerService--INFO--Apache ActiveMQ 5.15.15 (localhost, ID:Drishtis-MacBook-Air
 .local-64861-1670890118238-0:1) started
[2022-12-12 19:08:38.436-0500]--org.apache.activemq.broker.BrokerService--INFO--For help or more information please see: http://activemq.apache.org
[2022-12-12 19:08:38.538-0500]--server.coordinator.CoordinatorImpl--INFO--Coordinator up and running...
[2022-12-12 19:09:05.686-0500]--server.coordinator.CoordinatorImpl--INFO--Server added ser1(localhost, 50003)
[2022-12-12 19:09:57.673-0500]--server.coordinator.CoordinatorImpl--INFO--Received SignUp(vaish, vaish)
[2022-12-12 19:09:57.677-0500]--server.coordinator.CoordinatorImpl--INFO--Preparing SignUp@267508339::SignUp(vaish,vaish)
[2022-12-12 19:09:57.718-0500]--server.coordinator.CoordinatorImpl--INFO--Executing SignUp@267508339::SignUp(vaish,vaish)
[2022-12-12 19:12:40.650-0500]--server.coordinator.CoordinatorImpl--INFO--Server added ser2(localhost, 50004)
[2022-12-12 19:14:47.170-0500]--server.coordinator.CoordinatorImpl--INFO--GetPassword received for  Instagram from vaish
[2022-12-12 19:14:47.203-0500]--server.coordinator.CoordinatorImpl--INFO--GetPassword found TNuO3WrtQYa+MHIKmgxkQa5jlWuLh+0lew4UK6TEXnM= for  Instagram on
 localhost:50003 and sent back
[2022-12-12 19:14:47.245-0500]--server.coordinator.CoordinatorImpl--INFO--GetPasswordAck received for  Instagram from vaish
[2022-12-12 19:14:47.250-0500]--server.coordinator.CoordinatorImpl--INFO--GetPasswordAck received for  Instagram from vaish accepted
```



```
/Users/drishti/Library/Java/JavaVirtualMachines/openjdk-19/Contents/Home/bin/java ...
[2022-12-12 19:12:40.226-0500]--Server--INFO--Server up and running: rmi://localhost:50004/Service
[2022-12-12 19:12:40.227-0500]--Server--INFO--Adding server to coordinator
[2022-12-12 19:12:40.585-0500]--Server--INFO--Server and coordinator synced.
[2022-12-12 19:12:40.650-0500]--Server--INFO--Synced User DB
[2022-12-12 19:14:47.143-0500]--Server--INFO--GetPassword received for  Instagram by vaish
[2022-12-12 19:14:47.251-0500]--Server--INFO--Password  Instagram got value TNuO3WrtQYa+MHIKmgxkQa5jlWuLh+0lew4UK6TEXnM= for user vaish
```



```
/Users/drishti/Library/Java/JavaVirtualMachines/openjdk-19/Contents/Home/bin/java ...
[2022-12-12 19:12:40.226-0500]--Server--INFO--Server up and running: rmi://localhost:50004/Service
[2022-12-12 19:12:40.227-0500]--Server--INFO--Adding server to coordinator
[2022-12-12 19:12:40.585-0500]--Server--INFO--Server and coordinator synced.
[2022-12-12 19:12:40.650-0500]--Server--INFO--Synced User DB
[2022-12-12 19:14:47.143-0500]--Server--INFO--GetPassword received for  Instagram by vaish
[2022-12-12 19:14:47.251-0500]--Server--INFO--Password  Instagram got value TNuO3WrtQYa+MHIKmgxkQa5jlWuLh+0lew4UK6TEXnM= for user vaish
```


```
/Users/drishti/Library/Java/JavaVirtualMachines/openjdk-19/Contents/Home/bin/java ...
Please enter 's' to sign up, 'l' to login or 'q' to quit. Your entry is case insensitive.
s
Sign up initiated. Please note that the letter 'q' is reserved for quiting the application, and cannot be used as a username or password. The letter l is reserved
 for switching to login and cannot be used as a username or password
Enter username:
vaish
Enter password:
vaish
User creation successful.
Welcome vaish
Logging message to the client...
User with name: vaish created successfully. 1670890197734
Please enter your input. The server accepts requests in the form <METHOD>,<KEY>,<VALUE>.
Three method types are allowed namely: GET, PUT and DELETE.
Get and delete requests do not require a value. Example usage: 'GET,Instagram', 'PUT,Instagram,PasswordOfInsta99', 'DELETE,Instagram'.
White spaces are not ignored and will be a part of your key/value.
Enter q at anytime to quit.
Please enter your request:
PUT, Instagram, Instagram#Password
Logging message to the client...
PUT request made by user vaish. The request is PUT, Instagram, Instagram#Password. 1670890220350
Logging message to the client...
Response received from server: true. 1670890220640
PUT successful. Key =  Instagram Value =  Instagram#Password
Please enter your request:
```