ExBanking test assignment

1. Could you please provide functional and non-functional test cases?

    1. **Functional Test Cases**

Test ID	Test Area	Test Name	Data (Sample Request)	Expected Result (Response code)
T001	Test CreateUser Functionality	Verify that a user is successfully created with valid input. Example: Valid username with valid initial balance.	`CreateUser( { user_id: 'user5', balance: 132 }`	`OK (200)`
T002		Verify creating a user with missing or invalid input. Example: Empty username and valid initial balance.	`CreateUser( { user_id: '', balance: 132 }`	`INVALID_ ARGUMENT (400)`
T003		Verify creating user with existing username(Unique identifier)	`CreateUser( { user_id: 'user5', balance: 132 }`	`ALREADY_ EXISTS (409`
T004	Test Deposit Functionality	Verify depositing a valid amount and update the balance correctly.	`client.deposit({ user_id: 'user1', amount: 100}`	`OK (200)`
T005		Verify depositing a negative or zero amount is rejected.	`client.deposit({ user_id: 'user1', amount: -100}`	`INVALID_ ARGUMENT (400)`
T006		Verify invalid data for deposit amount.(Example: ‘Eur 100’)	`client.deposit({ user_id: 'user1', amount: 'Eur 100'}`	`INVALID_ ARGUMENT (400)`
T007	Test Withdraw Functionality	Verify withdrawing an amount less than or equal to the current balance.	`client.withdraw({ user_id: 'user1', amount: 50}`	`OK (200)`
T008		Verify withdrawing more than the balance fails.	`client.withdraw({ user_id: 'user1', amount: 5000}`	`FAILED_P RECONDIT ION (400)`
T009		Verify withdrawing money from invalid user.	`client.withdraw({ user_id: 'user999', amount: '20'}`	`NOT_FOUN D (404)`
T010	Test GetBalance Functionality	Verify retrieving the balance for an existing user returns the correct amount.	`client.GetBalance({ user_id: 'user1' }`	`OK (200)`

T011		Verify querying a non-existent user returns an error.	`client.GetBalance({ user_id: 'user999' }`	`NOT_FOUND (404)`
T012	Test Send Functionality	Test transferring money between valid users adjusts balances correctly.	`client.Send({ from_username: 'user1', to_username: 'user2', amount: 25 }`	`OK (200)`
T013		Test transferring money when the sender has insufficient funds fails.	`client.Send({ from_username: 'user1', to_username: 'user2',amount: 5000 }`	`FAILED_PRECONDITION (400)`
T014		Test transferring negative amount of money.	`client.Send({ from_username: 'user1', to_username: 'user2', amount: -25 }`	`INVALID_ARGUMENT (400)`
T015		Test sending and receiving money to same user.	`client.Send({ from_username: 'user1', to_username: 'user1', amount: 25 }`	`INVALID_ARGUMENT (400)`

Non-Functional Test Cases

Perf Test ID	Performance Test Type	Description	Example
P001	Single User Test	Measure the response times taken to complete a single operation or transaction for a single user. Compare the response times with baseline results.	Measure the time between request and its response for ● User creation ● deposit money ● Withdraw money ● getBalance ● Send operations.
P002	Load Test	Measure the application performance under the expected user load.	Measure response times, throughput, and system stability under a defined user load as the banking application handles user creation, deposit, withdrawal, balance inquiry, and transfer functions, with concurrent users logging in and performing these operations.
P003	Endurance Test	Measure application's stability and performance over an extended period under a sustained load. Measure	Measure the response times, throughput, RAM, processor utilization for 10000 concurrent users performing user creation,

		Memory leaks, response time consistency, and resource utilization.	deposit, withdrawal, balance inquiry, and transfer functions during continuous 48 hours.
P004	Scalability Test	Analyse how well the application scales with increasing user loads.Identify response times and resource utilization varying with userload	● Increase user loads gradually from 0 to 10000 users for the first 2 hours performing user creation, deposit, withdrawal, balance inquiry, and transfer functions. ● Keep 10000 users for 4 hours performing user creation, deposit, withdrawal, balance inquiry, and transfer functions. ● Decrease user loads gradually from 10000 to 0 users for the next 2 hours performing user creation, deposit, withdrawal, balance inquiry, and transfer functions.
P005	Stress Testing	Identify application's breaking point by subjecting it to extreme or beyond normal load conditions increasing user loads.Identify response times and resource utilization at the breaking point.	Increase user load gradually from 0 to large number until the banking application breaks and identify user loads, memory utilization, response times, processor utilization and other properties.

Apart from these we can do **Security testing** and **Accessibility testing** according to the world standard guidelines.

2. Could you please provide automation for a non-functional test case?

Instruction for Performance Test

Step 1: Select the Service for Testing

For this performance test, the “Deposit” service was chosen.

Step 2: Select the Testing Tool

Initially, the Artillery tool was considered for load testing. However, since it does not support gRPC responses, the ghz tool was used instead.

Step 3: Install the ghz Tool

Download the ghz tool from the official website or repository (Download link).

Move the executable to a directory included in your system’s PATH, such as C:\Windows\System32.

Step 4: Start the gRPC Server

Ensure the gRPC server is running before executing the test.

Step 5: Execute the Performance Test

Open PowerShell or a command prompt.

Run the following command to initiate the test. I used 100 concurrent users for 30 seconds.

```
C:\Windows\System32>ghz.exe --insecure
--proto C:\Users\Eranga\Desktop\BankingSystem1\proto\banking.proto
--call banking.BankingService.Deposit
--data "{\"user_id\": \"user1\", \"amount\": 10}"
--duration 30s
--concurrency 100
--output
C:\Users\Eranga\Desktop\BankingSystem1\Tests\Performance_Test\result.json
0.0.0.0:50051
```

Parameters

--proto : Path to proto file
--call : Service.method
--data: JSON-data
--duration: Duration for the performance test
--concurrency: Number of concurrent requests to send to the server during the test.
--output: Path to save the performance test results in JSON format

Step 6: View the Results

The test results will be saved in the following file: C:\Users\Eranga\Desktop\BankingSystem1\Tests\Performance_Test\result.json. You can open this file to analyze the performance metrics.

3. Could you please provide following items for 10% of functional test cases;

a. Mock / service virtualization of ExBanking

1. Open a terminal and run mock Server. [Powershell command: node **mockServer.js**]
2. Open another terminal and run mock tests. [Powershell command: node **mockTests.js**]

b. Test automations

1. Open a terminal and run Server. [Powershell command: node **server.js**]
2. Open another terminal and run Client. [Powershell command: **npm test**]
3. Results can be seen in server and client terminals