Gaming Platform

gitlab repository: https://gitlab.com/skobnikov.p/pm-student-project-2

Project's maintainers: Group #4

Group members: Danylo Rusakov, Valeriia Luniakina, Tai Bui Minh, Pavel Skobnikov

How to start the program:

Steps to set up our program:

1. Build project:
    - Open Project Structure
    - Create *Web Application: Archive*
    - Add all available elements (the required libraries for the app in the *Project Settings>Artifacts*)
    - Add resources folder (‘src/main/java/resources) to the project structure in the *Project Settings>Modules*
    - Click ‘*Apply*’
    - Build the program
2. Setup the TomCat
    - Run the TomCat
    - Set Connector port to ‘9995’
    - Put new (in our case it was ‘*project2.war*’) file in *webapps* in TomCat
3. Create *postgres_container* (we have used the following resource to create this https://github.com/khezen/compose-postgres)
    - Create at port: 5432
    - Set user_name = postgres
    - Set password = change me
    - Run container through the docker

Steps to run our program:

1. Open the Postman
    - Write requests which are listed below to set up and initialize a database through the servlet:
        - *http://localhost:9995/project2/start?command=createDb* - to create a Database
        - *http://localhost:9995/project2/start?command=createTables* - to create a Tables inside the Database
        - *http://localhost:9995/project2/start?command=addTrigger* - to add the trigger
        - *http://localhost:9995/project2/start?command=insertData* - to fill Tables with some data

    - *http://localhost:9995/project2/start?command=dropTables* - to remove all tables from the database
    - *http://localhost:9995/project2/start?command=dropDb* - to drop the current database
    - ***IMPORTANT NOTE***: do not run the requests too quickly, as the connections used for these critical operations may not close in time for the next one!
- Then you can start to send a request in *GameIntegrationService*
    - To successfully send to the *GameIntegrationService* a request, you have to, first of all, generate the correct MD5 hash value and add it to the HTTP request as a header called 'request-hash-sign.'
    - The way our application generates the MD5 hash value is that we take all values of a deserialized JSON as strings (for *Bet* and *Win*) or simply retrieve the Player ID parameter (for *getBalance*), sort the string values, and concatenate a SECRET_KEY value from properties. Then, we run the resulting string through an MD5 hasher and return the result.
    - Of course, the *GameIntegrationService* expects a proper request for each of the respective servlets, as seen in the *Group Project* HTML task examples. For example, we expect invariant requests from clients, so, our service doesn't encompass any checks for JSON body request or request parameters "correctness" in any way.
    - Some errors not mentioned in the

Structure of the program

Our program consists of two main directories (*GameIntegrationService* - has all logic for receiving requests from game providers, proprietary integration API to internal wallet API and game flow validation; *SimpleWallet* - integrates with the database and adds some specific logic;). Also, there is a *"startapp"* directory that is responsible for starting our application.

Structure of "gameintegrationservice":

- communicationalmodels - enables communication between Client <-> GameIntegrationService <-> SimpleWallet <-> DB Package via convenient (de-)serialization of various objects through Jackson.
- exceptions - contains all custom-defined exceptions, making error handling easy and understandable both in GameIntegrationService and SimpleWallet
- utilproccessors - additional utilitarian classes with static methods, providing clear logic for GameIntegrationService.

    - gameflow - contains models for rounds and bet to track which ones actually belong to the round.

Structure of "SimpleWallet":

- communicationalmodels
- DB - contains all logic with database
    - DAO - contains DAO interface(1st level of abstraction) and classes for every DTO model which implements all methods from the interface (2nd level of abstraction)
    - DTO - all DTOs
    - services - contains all services for every model which wrap all DAO classes with additional business logic (3rd level of abstraction)
- exceptions - there are all custom exceptions for SimpleWallet
- utilprocessors - additional classes for our logic

Tests:

We have provided tests for Gameintegration service and simple wallet processor

Custom logic for round closing:

As well as the task does not have any information about conditions when the round should be closed. We have decided to implement this logic in the simple wallet which closes the round in 1 minute after the bet. If the round is closed and the server gets a request for a win - you will get an error response with the code 801 and information that the round is over and we cannot accept any win.

API for communication between SimpleWallet and GameIntegrationService:

- POST db/credit + body - to withdraw money from the balance, create a transaction and a round;
- POST db/debit + body - to deposit money on player's balance, create a transaction and close round if necessary;
- GET db/balance?playerId= - to get a player's balance.