Coding Challenge Manual

This manual contains three parts: design, assumption and How-to execute the solution.

Design

The solution uses Model-View-Controller (MVC) design patterns. In the model part, there is an entity called *BetEntity* which consists of all fields data. This entity will map all fields between Java and SQL. A collection of *BetEntity* instances are stored in a backend database which can be accessed via an interface called *BetRepository* (extends from *CrudRepository* interface). *CrudRepository* is a part of *Spring Framework* data interface; in order to use it, we need to create our interface by extending *CrudRepository* for a specific type. Spring provides *CrudRepository* implementation class automatically at runtime. It contains methods such as *save, findById, delete,* etc. If we want to add extra methods, we need to declare it within our interface. The *BetRepository* declares three additional methods called *findByBetType*, *findByCustomerId*, and *findAllByDateTimeBetween*. These methods would return a list of *BetEntity* instances.

In the controller part, we define all required endpoints and their implementations. The endpoint is the URL that is focused on creating a request. In our case, we specify 7 endpoints: create bet, retrieve all bets, clean database and 4 endpoints to generate reports. The endpoints are defined in *BetController* class and simply call their associated business logic which are located in *BusinessService* class. The endpoints are secured by entering a username and password.

In the *BusinessService* class, we put some restrictions for creating a bet (such as the maximum investment value must not exceed $20,000), querying the database and creating an object to be passed to the UI part. For example, in creating a bet, we receive a *Bet* instance from the user through the controller. The *Bet* class contains all fields defined in the UI. All field values are validated based on two criteria: the time shouldn't be in the past and its investment value should be less than or equal to 20,000. An exception will be thrown to the controller and http status code 400 with its error message will be returned to user if any of the fields does not contain a valid value. Otherwise, the value will be stored to the database by calling the *save* method of the repository interface. The http status 200 will be returned to the user if the value is successfully stored to the database.

For generating a report, a parameter is passed as a part of URL to the controller. The controller passes the parameter to the *BusinessService*. The *BusinessService* will then query the database and a list of *BetEntity* instances will be returned. A *Report* instance would be generated and their values are the returned database result. The *Report* contains a *list of Bet instances* found in database, and also computed *totalBets* and *totalInvestment*. It would be empty, otherwise. If total investment report is being generated, the *totalInvestment* field would contain an investment value and the *totalBets* will contain -1. On the other hand, if total bets report is being generated, the *totalBets* would contain a number of bets found in database and *totalInvestment* would be -1. For example, to generate a report for total investment per bet type, a bet type (eg. WIN) is specified as a part of the URL (e.g. "/report/totalInvestmentPerBetType/WIN"). The report returned would contain a list of bet instances found in database, *totalInvestment* field is a number (eg. 300.0) and *totalBets* field is -1.

Unit Test

Three unit test classes are created to test the implementation. In each class, some instances of *Bet* are created by using mock up values and these instances are stored in database. Then, the database is queried based on the specified value and the database returned result is validated against the *Bet* instance created. An *open source in-memory database* called *H2* database is used. Each time a unit test is run, a table will be created and dropped once finished.

The three unit test classes are *BetRepositoryTest*, *BetControllerCreateTest*, and *BetControllerGetTest* classes. The first class, *BetRepositoryTest,* is created to test our custom repository function inside the *BetRepository* class. This class is annotated with *DataJpaTest* and *SpringRunner* class. The annotations mean this class is a DataJPA test class and would be run with *SpringRunner* class. A *TestEntityManager* instance is created to persist the data to database. Three tests are developed to test each method in the *BetRepository* interface.

The other two unit test classes*, BetControllerCreateTest* and *BetControllerGetTest,* contain unit tests for creating bets and generating reports through mock up web application. These classes are inherited from one abstract class. The abstract class has @*RunWith*, @*SpringBootTest* and @*WebAppConfiguration* annotations. The first annotation denotes that the unit test is run with the *SpringRunner* class. The @*SpringBootTest* annotation tells Spring Boot to go and look for a main configuration class (one with @*SpringBootApplication* for instance), and use that to start a Spring application context. @*WebAppConfiguration* annotation to test the web controllers. The *WebApplicationContext* is loaded using the @*WebAppConfiguration* annotation and inject their instances using @*Autowired*.

In both BetController tests, one or more bets are created and stored in database. To test create bet, we create a bet instance, serialize it to json object and pass it to create bet end point. To test generating a report, some bets are created before the test started. An endpoint with a parameter attached to the end of endpoint is sent to the web application. The parameter can be bet type, customer id or time. Then, the web application endpoint returns a json object and we deserialize it into a Bet instance. As for the *BetControllerGet* class, four records to be used for the tests are inserted and a number of tests are specified which would be used to test each endpoint defined in the Controller class.

How to run the solution

1. Clone the source code using git from https://github.com/jmswn89/codingChallenge.git
2. Install postgresql.
3. Create database *tabcorp* and username *demo* with password *demo123.*
4. Run *mvn clean spring-boot:run* in command line.
5. To test in browser: http://localhost:8080/bets/, you need supply username (test) and password (test1). This will give an empty json string as there is no data in database yet..

Below is the available endpoint (for all endpoints, username is *test* and password *test123*):

- `/bet, Method POST`
    - `To create a bet with specific payload.`

    - payload: {"dateTime":"18/03/2019 20:36:12","betType":"DOUBLE","propNumber":12,"customerId":2,"investmentAmount":300.0}
- /bets, Method GET
    - To retrieve all bets from database.
- /report/totalInvestmentPerBetType/{betType}, Method GET
    - To generate a report based on a specific bet type and total investment is calculated.
    - The Bet types are WIN, PLACE, TRIFECTA, DOUBLE, QUADDIE
- /report/totalBetSoldPerBetType/{betType}
    - To generate a report how many bets have been sold based on a specific bet type.
- /report/totalInvestmentPerCustomerId/{customerId}
    - To generate a report how much investment have been made for a particular customer id.
- /report/totalBetSoldPerHour
    - To generate a report how many bets have been sold for last one hour.

Assumptions

- The date time input from user should not be more than one hour in the past from current time.
- Total bets sold per hour is calculated as the number of bets sold in the past one hour.
- The default username and password for URL is *test* and *test123* respectively. They are defined in src/main/resources/application.properties.
- The default database name, database username and database password are defined in src/main/resources/application.properties.
- The endpoints are only secured when the application is being run, i.e. not secured when unit test is being run.