Lodbrok Endpoint Project

Revision History

Revision	Date	Name	Comments
1.0	15-Feb-2016	Fabio Riberto	Initial version

Contents

1. Description

See attached document *"Java back-end developer test rev5.pdf"*

2. Overview

Lodbrok HTTP-based game endpoint provides a system to serve the external requests for Lodbrok's World.

The great King Lodbrok is able to front to almost all the external requests (but not the entire set) to let its world game working in a concurrent environment like the VIII century's one was.

3. Scope

The world of the Vikings' King Lodbrok needs our help to let the very entire set of tasks ready to work properly in the north European environment: high concurrence environment.

The scope of this project is to help basically for 2 main areas of activities:

- Login management
- Scores management

4. Design

4.1. General purpose

Kattegat town needs to be organized to receive several and different requests.
Lodbrok divide the inhabitants in two parts:

- ✓ People that manage an interface for the external world able to receive several requests and route them to the proper service to be processed (the server itself)
- ✓ At the same time, other people that manage the data (the control mechanism itself)

To do that Lodbrok buys the *java.util.concurrent* package of tool from the nearest Count for delegating the schedule and the thread life-cycle activities.

4.1.1. Server

The system has to be supported by a server that manage the HTTP communication (in a high concurrent environment): only *com.sun.net.httpserver* package will be used for this purpose.
This way, it needs to have a HTTP server somewhere between the application and the user: this will be done through a Java class named *LodbrokServer*:

```
public class LodbrokServer {
    public static void main(String[] args) {
    //Server tasks
    }
```

}

where the Lodbrok endpoint switches on.

This class is responsible for the HTTP communication through a handler that will take care to process the requests

public class LodbrokHandler implements com.sun.net.httpserver.HttpHandler

“Ad-hoc” filter will be used for discriminating between the three different kinds of requests:

public class LodbrokFilter extends com.sun.net.httpserver.Filter

For the threads management a *java.util.concurrent.ExecutorService* will be used since it offers golden tools for a concurrent environment like threads reusing, automatic schedule and mostly use a thread pool that reduces response time by avoiding the thread creation during task processing.

4.1.2. Control mechanism

Since the concurrence is the key, we need a unique control that also satisfy the thread-safety requirement. The *LodbrokController* for rescue:

public class LodbrokController

Since lot of threads will take control of it shall be implemented to be a unique instance and the threads shall be able to update its own cache on it to speed up the handshake.
Also, this class will delegate the login and score activities to two specialized classes to optimize the resources:

- ✓ A controller that manages the score operations and checks:
  public class ScoreController
- ✓ A controller that manages the login procedures and checks:
  public class LoginController

ScoreController will take care to filter a score with a level detection mechanism to ensure the proper storage of a new score for a particular level and retrieve the high score list for a particular level while requested.

LoginController will take care to create new session for a particular user, verify and remove sessions, determine while a session has expired based on a required timeout. This class will be called by *ScoreController* to check login credentials (sessionkey and start/expiration time)

4.2. Scores

The *score storing* procedure is

- ✓ Recognize a valid login URI using regular expressions. The score URI includes:
  - the <*levelid*> that is every sequence (not *null*, not blank) of decimal digits: *(\d+)*
  - the literal *score*
  - the literal *sessionkey*
  - the <*sessionkey*> that is every sequence (not *null*, not blank) of digits: *(.+)*
- ✓ Identify the score URI in a collection of three possible URI patterns following the recognized score expression
- ✓ Set the *score, levelid* and *sessionkey* parameters
- ✓ Extract the score number from the score request body
- ✓ Pass the login filtered parameters to the *LodbrokController* class for processing by calling a proper method to execute the login
- ✓ Pass the *sessionkey* to *LoginController* to verify that is valid and not expired
- ✓ Pass the *score, levelid and sessionkey* parameters to *ScoreController* for score storing
- ✓ Include the score to the high score pool, listing it while required; Check the stored scores to let a maximum of 15 scores in the list

The *high score list retrieving* procedure is

- ✓ Recognize a valid high score list URI using regular expressions. The high score list URI includes:
  - the <*levelid*> that is every sequence (not *null*, not blank) of decimal digits: *(\d+)*
  - the literal *highscorelist*
- ✓ Identify the high score list URI in a collection of three possible URI patterns following the recognized high score list expression
- ✓ Set the *levelid* parameter
- ✓ Pass the *levelid* parameter to *ScoreController* to get the high score list. If the level id doesn't exist, a blank list has returned
- ✓ Print the high score list to the response body

4.3. Login

The *login procedure* is

- ✓ Recognize a valid login URI using regular expressions. The login URI includes:
  - the <*userid*> that is every sequence (not *null*, not blank) of decimal digits: *(\d+)*
  - the literal *login*
- ✓ Identify the login URI in a collection of three possible URI patterns following the recognized login expression
- ✓ Set the *userid* parameter
- ✓ Pass the login filtered parameter to the *LodbrokController* class for processing by calling a proper method to execute the login

- ✓ Pass the *userid* parameter to *LoginController* to execute the login
- ✓ Open a new session
- ✓ Store the open time to start the timeout
- ✓ Return the session number to the response body

5. Implementation

The better way to organize the implementation is MVC architectural pattern.

First of all the communication shall be supported by a server that is in charge to start the application, open a port, let the handlers, filters, executors and controllers able to:

- ✓ Receive inputs from the View
- ✓ Send outputs to the View

This server is represented by the class:

```
public class LodbrokServer {
        public static void main(String[] args) {
        //Server tasks
        // Configure JMX
        MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
                                                                    // Instance of MBean
                                                                    //server
        Ragnarok mBeanClient = new Ragnarok(DEFAULT_NUMBER_OF_THREADS
                                                                );
        ObjectName objName = new ObjectName("com.lodbrok.jmx:type=Ragnarok");
        mBeanServer.registerMBean(mBeanClient, objName); // MBean client
                                                                    // registration
        //More JMX tasks
        }
}
```

Model Entities:

- ✓ *Login* class

  Need to have an object representing the session itself (valid during 10 minutes):

  Since the request include a used id, this object shall have at least two attribute representing the user id and the session key.

  A user shall interact with the application keeping a session key active for a limited time (10 minutes): after that the session expires: for this reason a mechanism to throw away unused old expired sessions is needed. This means that the Login objects shall be collected in a pool of alive logins and they shall be removed while they expire.

  Since the session shall be reasonably unique it will include custom equals() and hashCode() methods.

```
public class Login implements Serializable {
   private long bornTime;
   private Integer userId;
   private String sessionKey;
   @Override
   public boolean equals(Object o) {
   }
   @Override
   public int hashCode() {
   }
   //More logic here
}
```

- ✓ *Score* class

  During the high score list request *couples (userid, score)* shall be returned to be listed in the response body.
  An entity that describe the single couple is needed: it will include at least two attributes, the user id and the relative score.
  Because of the unique presence of a user in the list and duplicated score for the same user (at the same level), this entity shall be able to compare similar objects.

  Comparable interface shall be implemented, equals() and hashCode() methods shall be customized.

```
public class Score implements Comparable<Score>, Serializable {
   private int userId;
   private int score;
   @Override
   public int compareTo(UserScore o) {
   }
   @Override
   public boolean equals(Object o) {
   }
   @Override
   public int hashCode() {
   }
   //more logic here
}
```

- ✓ *HighScoreList* class

  This is the most complicated logic: because of the number of levels is potentially high and all the high score lists shall be potentially present forever (since a persistence mechanism is not required, during the JVM instance life);
  Note that also a number of users is potentially very high but they shall interact with the application, keeping a session key active, for a limited time (10 minutes).
  This object shall encapsulate a collection that stores the high score list itself to be retrieved while requested.

Package *java.util.concurrent* provide the ConcurrentMap interface that help in a high concurrence environment with its atomic methods to insert, remove and manage a component into the map.

```
public class HighScoreList implements Serializable {
   private ..."some kind Of concurrent map"<Score> highScoreLists;
   //more logic here
}
```

Controllers:

- ✓ *LodbrokController* class

```
public class LodbrokController {
        private final LoginController loginCtrl;
        private final ScoreController scoreCtrl;
        public LodbrokController() {
                //Initialization
        }
public volatile static LodbrokController controlInstance;
public static LodbrokController getInstance() {
//Out-of-order writes + Double-checked locking
return controlInstance;
}
        @Override
        protected Object clone() throws CloneNotSupportedException {
                throw new CloneNotSupportedException();
        }
        //More logic here
}
```

This class is the main controller. It is a unique instance and its scope is to route the request to the proper service (login service, score service)
Since the concurrence is high its constructor has to take in count all the possible side-effects: *out of order writes* and *double-checked locking* techniques shall be implemented. Volatile modifier is present to force thread caches updating.
Also this class shall avoid cloning: the *clone()* method has to be override to throw a *CloneNotSupportedException()* front to a tentative of cloning.

- ✓ *ScoreController* class

```
public class ScoreController {
/**
* Thread-safe java.util.Map to store all the High Score Lists for the
* existing levels
*/
private ConcurrentMap<Integer, HighScoreList> allLevelsHighScoreListMap;

public synchronized void storeAScore(final Integer level, final Score score) {
        //More Logic here
        }
}
public HighScoreList extractHighScoreList(final int level) {
        //More Logic here
}
//More logic here
}
```

allLevelsHighScoreListMap is the key: when an element is inserted in the map, *java.util.concurret* framework take cares to the threads management to resolve *lock* and *wait* states.

6. Test

Looking at the requirements in the attached document *"Java back-end developer test rev5.pdf"* they should be implemented driven by tests.

A reasonable choice is to use:

- ✓ *Junit-4.1.2.jar*
- ✓ *Hamcrest-core-1.3.jar*
- ✓ *Mockito-all-1.10.19.jar*

libraries that shall be included in the project.

The below Lodbrok Endpoint tasks shall be fully tested for having a robust test coverage:

- ✓ Switch the server on
- ✓ Accept/reject valid/erroneous URIs
- ✓ Open a session
- ✓ Accept/reject alive/dead sessions
  Java.util.concurrent.TimeUnit enumeration will help in doing this timing operation.
  The main steps are:
  - ➢ Open a new session
  - ➢ Verify that the new session is valid
  - ➢ Move time forward > 10 minutes
  - ➢ Verify that the new session is not valid

The key point is moving time forward 10 minutes. This shall be too much expensive in time so *Mockito Framework* should help here. The idea is mocking the call for the session life time to a new shortest time (says, few seconds).

- ✓ Show opened session key
- ✓ Store a score in a proper high score list
- ✓ Create high score lists
- ✓ Maintain high score lists with a max established length
- ✓ Maintain high score lists with the higher scores only
- ✓ Show high score lists
- ✓ Stress the system to be able to front to an enormous number of requests
  A specific test should be designed to test a big number of requests

```
public class LotOfRequestsStressTest {

        List<String> userIds = new ArrayList<>();
        List<String> sessionKeys = new ArrayList<>();
        List<String> levels = new ArrayList<>();
        List<String> scores = new ArrayList<>();
        public static final int HUGE_NUM_OF_LOGINS = 201;
        public static final int HUGE_NUM_OF_LEVELS = 201;
        public static final int HUGE_NUM_OF_SCORES =
                                HUGE_NUM_OF_LEVELS*2;

        @Before

        public void init() throws Exception {

                LodbrokServer.main(null);
                for (int i = 1; i < HUGE_NUM_OF_LOGINS; i++) {
                        if (i % 2 != 0)
                                userIds.add(String.valueOf(i));
                }
                for (int i = 1; i < HUGE_NUM_OF_LEVELS; i++) {
                        levels.add(String.valueOf(i));
                }
                for (int i = 1; i < HUGE_NUM_OF_SCORES; i++) {
                        scores.add(String.valueOf(i * 25));
                }
                rng = new Random();
        }
        @Test
        public void testEnormousNumberOfRequests() throws IOException {
        // Test enormous number of login, score and high score list requests
        }
}
```

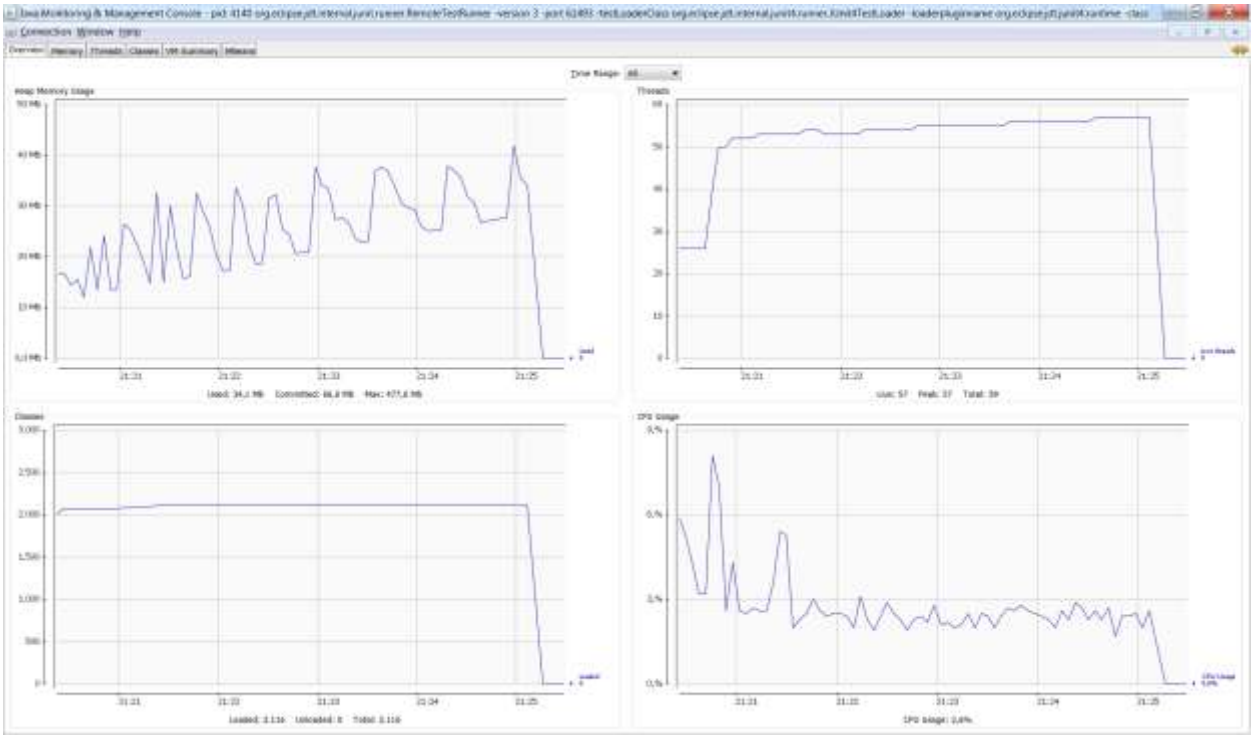
Where *HUGE_NUM_OF_LOGINS*, *HUGE_NUM_OF_LEVELS* and *HUGE_NUM_OF_SCORES* can be increased or decreased.

If *Out Of Memory Exception* is reached there shall be a chance to increase the memory size with VM parameters: (i.e.: *-Xms64m -Xmx512m*)

Finally, in order to test enormous number of requests *jconsole* shall be used; the VM used parameters are to let the application ready to be monitored are:

-Dcom.sun.management.jmxremote.port=8085
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
-Djava.rmi.server.hostname=localhost

For example, this below is what happened while *HUGE_NUM_OF_LOGINS = 100*, *HUGE_NUM_OF_LEVELS = 200* and *HUGE_NUM_OF_SCORES = 400* have been set:



Where *number of total requests = 20300* and *execution time = 5min 03sec*, with:

- *Windows 7.6.1 8-core amd64* machine
- *Java HotSpot(TM) 64-Bit Server VM version 25.66-b18*
- Heap max size = *512MB*

This result has been collected in "*LotOfRequestsStressTest_user100_level200_score400.pdf*" and "*LotOfRequestsStressTest_user100_level200_score400.png*" in attachment

Also, JMX can be used for custom management and administration purposes: the details have been exposed in the implementation section.

7. Considerations

Below considerations can be done:

- ✓ In order to speed up the test process a little shrewdness has been done at code level.
  The session life time is stored as a *public static final integer* and initialize to *10 minutes* to be immutable forever during the JVM life.
  Anyway it has been used by calling a no private, final end static method to be mocked to a most flexible time (2 seconds) during the unit test phase

```
@Mock
private LoginController lconMock;
…
int timeWarp = 2000 // 2 seconds
…
when(lconMock.getSessionLife()).thenReturn(timeWarp);
                                    // Wait only 2 seconds instead 10 minutes!
```

- ✓ The system has been designed to be modular for future additional functionalities: if a new functionality will be add, a new specialized controller will be create and instantiate in *LodbrokController*
- ✓ The result of the stress test depend on the kind of machine that hosts the VM and the kind of VM used (32bits, 64bits, etc…). The test machine has currently 8 cores so, since only one thread can be mapped at the same time to one core, the maximum capacity is to use 8 thread at the same time.

8. Installation procedures

8.1. Zip content

The package "***LodbrokEndpointProject.zip***" archive contains:

- ✓ *src* folder: it contains a sub-folder *main* that include the Lodbrook Endpoint source code and *test* sub-folder that include the test source code
- ✓ *lib* folder: it contains all the libraries used for unit test:
  - ➢ *hamcrest-core-1.3.jar*
  - ➢ *junit-4.12.jar*
  - ➢ *mockito-all-1.10.19.jar*
- ✓ *classes* folder: it contains java binary files (.class)
- ✓ *doc* folder: it contains the Javadoc for the Lodbrock Endpoint:
  - ➢ double click on *index.html* to open the Lodbrok Endpoint Javadoc
- ✓ *CompileMain* file: file containing the java class (.java) names for compiling
- ✓ *LodbrokManifest* manifest file: custom manifest file to be included into the jar package
- ✓ *LotOfRequestsStressTest_user100_level200_score400.pdf* file: test evidence for 100 users, 200 levels and 400 scores
- ✓ *LotOfRequestsStressTest_user100_level200_score400.png* file: test evidence for 100 users, 200 levels and 400 scores
- ✓ *LodbrokEndpoint.jar* archive: jar archive containing the Endpoint to be executed.
- ✓ *Java back-end developer test rev5.pdf* file: requirements document

- ✓ *LodbrokEndpointProject.pdf* file: design document
- ✓ *readme.txt*: readme file to get start to the *HTTP-based mini game back-end (Lodbrok Endpoint)*

8.2. Run Lodbrok Endpoint

jdk1.8.0_66 Java Development Kit has been used to compile and run the application.

Lodbrock Endpoint can be run by executing the below prompt command:

C:> java –jar LodbrokEndpoint.jar

The follow line means that the Endpoint started properly:

\m/ >_< \m/ LodbrokServer works for the Kattegat inhabitants @ http://localhost:8081/ \m/ >_< \m/

Exit the Endpoint by typing:

<Ctrl>+c

8.3. Contingency and Roll out

Case of some kind of class files corruption or jar file corruption, a new jar file can be generated starting by source code contained in the source folder following the below steps:

- ✓ Enter the *LodbrokEndpointProject* folder (the one created by unzipping the *LodlockEndpointProject.zip* archive
- ✓ *C:> javac -d classes @CompileMain*
- ✓ *C:> jar cmf LodbrokManifest LodbrokEndpoint.jar -C classes .*
- ✓ *C:> java -jar LodbrokEndpoint.jar*

The follow line means that the Endpoint started properly:

\m/ >_< \m/ LodbrokServer works for the Kattegat inhabitants @ http://localhost:8081/ \m/ >_< \m/

Exit the Endpoint by typing:

<Ctrl>+c