Beyond Spock Endpoint Project

Revision History

Revision	Date	Name	Comments
1.0	01-Mar-2016	Fabio Riberto	Initial version

Contents

1. Description

See attached document *“[Java][BackEnd]Test.pdf”*

2. Overview

Beyond Spock HTTP-based game endpoint provides a system to login, store and retrieve information.

The environment is high concurrent and the HTTP communication shall take in count this requirement.

3. Scope

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

- Login management: where a user shall be always logged in for doing more actions
- Bids management: where a logged user can store a new bid and retrieve lists of bids

4. Design

4.1. General purpose

The HTTP-based mini bids back-end is design to have mainly 2 separated blocks:

- ✓ An interface that provide filtering and handling of the in/out messages (the server itself)
- ✓ A business unit that provide to manage the encapsulate data (the control mechanism itself)

java.util.concurrent JAVA package has been chosen to front high concurrency because it contains tools that help in delegating the schedule and managing the threads life-cycle.

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 business unit and the user: this will be done through a Java class named *BeyondSpockServer*:

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

where the Beyond Spock endpoint switches on.

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

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

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

public class BeyondSpockFilter 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 concurrency is the key, we need a unique control that also satisfy the thread-safety requirement. The *BeyondSpockController* for rescue:

public class BeyondSpockController

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

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

BidController will take care to filter a bid with an item detection mechanism to ensure the proper storage of a new bid for a particular item and retrieve the top bid list for a particular item 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 *BidController* to check login credentials (sessionkey and start/expiration time)

4.2. Bids

The *bid storing* procedure is

- ✓ Recognize a valid login URI using regular expressions. The bid URI includes:
  - the *<itemID>* that is every sequence (not *null*, not blank) of decimal digits: *(\d+)*
  - the literal *bid*

        - the literal *sessionkey*
        - the <*sessionkey*> that is every sequence (not *null*, not blank) of digits: *(.+)*
- ✓ Identify the bid URI in a collection of three possible URI patterns following the recognized bid expression
- ✓ Set the *bid, itemID* and *sessionkey* parameters
- ✓ Extract the bid number from the bid request body
- ✓ Pass the login filtered parameters to the *BeyondSpockController* 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 *bid, itemID and sessionkey* parameters to *BidController* for bid storing
- ✓ Include the bid to the top bid pool, listing it while required; Check the stored bids to let a maximum of 15 bids in the list

The *top bid list retrieving* procedure is

- ✓ Recognize a valid top bid list URI using regular expressions. The top bid list URI includes:
    - the <*itemID*> that is every sequence (not *null*, not blank) of decimal digits: *(\d+)*
    - the literal *topBidList*
- ✓ Identify the top bid list URI in a collection of three possible URI patterns following the recognized top bid list expression
- ✓ Set the *itemID* parameter
- ✓ Pass the *itemID* parameter to *BidController* to get the top bid list. If the item ID doesn't exist, a blank list has returned
- ✓ Print the top bid 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 *BeyondSpockController* 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

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

<u>Model Entities</u>:

- ✓ *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 attributes 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
}
```

- ✓ *Bid* class

  During the top bid list request *couples (userID, bid)* 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 bid.
  Because of the unique presence of a user in the list and duplicated bid for the same user (for the same item), this entity shall be able to compare similar objects.

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

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

- ✓ *TopBidList* class

  This is the most complicated logic: because of the number of items is potentially high and all the top bid 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 top bid 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 TopBidList implements Serializable {
    private ..."some kind Of concurrent map"<Bid> topBidLists;
    //more logic here
}
```

Controllers:

- ✓ *BeyondSpockController* class

```
public class BeyondSpockController {
        private final LoginController loginCtrl;
        private final BidController bidCtrl;
        public BeyondSpockController() {
                //Initialization
        }
public volatile static BeyondSpockController controlInstance;
public static BeyondSpockController 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, bid 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.

- ✓ *BidController* class

```
public class BidController {
/**
* Thread-safe java.util.Map to store all the Top Bid Lists for the
* existing items
*/
private ConcurrentMap<Integer, TopBidList> allItemsTopBidListMap;

public synchronized void storeABid(final Integer item, final Bid bid) {
        //More Logic here
        }
}
public TopBidList extractTopBidList(final int item) {
        //More Logic here
}
//More logic here
}
```

  allItemsTopBidListMap 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][BackEnd]Test.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 Beyond Spock 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 bid in a proper top bid list
- ✓ Create top bid lists
- ✓ Maintain top bid lists with a max established length
- ✓ Maintain top bid lists with the higher bids only
- ✓ Show top bid 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 ManyRequestsStressTest {

        List<String> userIds = new ArrayList<>();
        List<String> sessionKeys = new ArrayList<>();
        List<String> items = new ArrayList<>();
        List<String> bids = new ArrayList<>();
        public static final int HUGE_NUM_OF_LOGINS = 201;
        public static final int HUGE_NUM_OF_ITEMS = 201;
        public static final int HUGE_NUM_OF_BIDS =
                                HUGE_NUM_OF_ITEMS*2;

        @Before

        public void init() throws Exception {

                BeyondSpockServer.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_ITEMS; i++) {
                        items.add(String.valueOf(i));
                }
                for (int i = 1; i < HUGE_NUM_OF_BIDS; i++) {
                        bids.add(String.valueOf(i * 25));
                }
                rng = new Random();
        }
        @Test
        public void testEnormousNumberOfRequests() throws IOException {
        // Test enormous number of login, bid and top bid list requests
        }
}
```

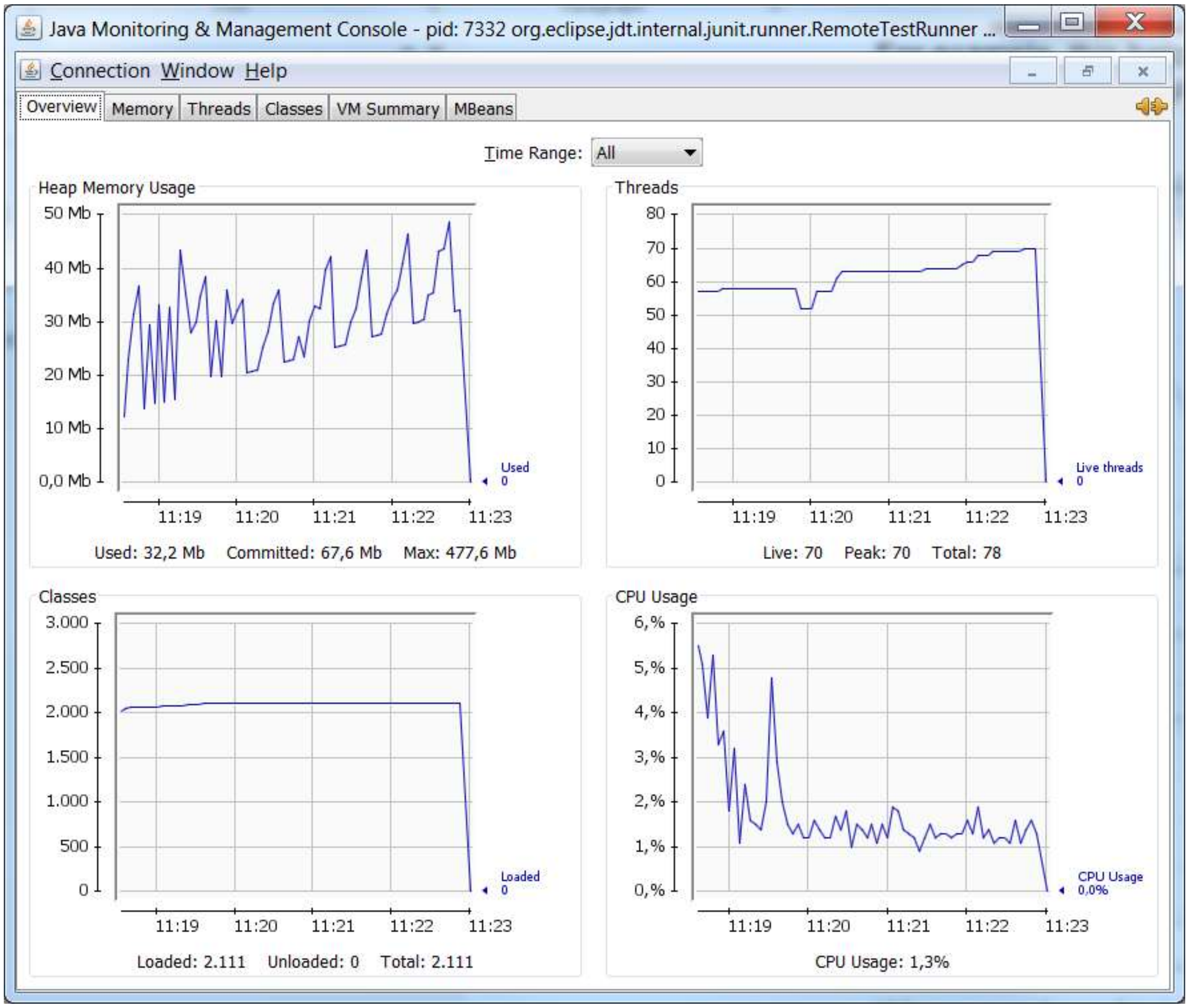
Where *HUGE_NUM_OF_LOGINS*, *HUGE_NUM_OF_ITEMS* and *HUGE_NUM_OF_BIDS* can be increased or decreased.

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_ITEMS = 200* and *HUGE_NUM_OF_BIDS = 400* have been set:



Used: 32,2 Mb   Committed: 67,6 Mb   Max: 477,6 Mb

Live: 70   Peak: 70   Total: 78

Loaded: 2.111   Unloaded: 0   Total: 2.111

CPU Usage: 1,3%

Where <u>*number of total requests = 20300*</u> and <u>*execution time = 4min 32sec*</u>, 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 "*ManyRequestsStressTest_user100_item200_bid400.pdf*" and "*ManyRequestsStressTest_user100_item200_bid400.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 *BeyondSpockController*
- ✓ 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. GitHub repository content

The GitHub repository "***BeyondSpockEndpointProject***" contains:

- ✓ *src* folder: it contains a sub-folder *main* that include the Beyond Spock 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 Beyond Spock Endpoint:
  - ➢ double click on *index.html* to open the Beyond Spock Endpoint Javadoc

- ✓ *CompileMain* file: file containing the java class (.java) names for compiling
- ✓ *BeyondSpockManifest* manifest file: custom manifest file to be included into the jar package
- ✓ *ManyRequestsStressTest_user100_item200_bid400.pdf* file: test evidence for 100 users, 200 items and 400 bids
- ✓ *ManyRequestsStressTest_user100_item200_bid400.png* file: test evidence for 100 users, 200 items and 400 bids
- ✓ *BeyondSpockEndpoint.jar* archive: jar archive containing the Endpoint to be executed.
- ✓ *[Java][BackEnd]Test.pdf* file: requirements document
- ✓ *BeyondSpockEndpointProject.pdf* file: design document
- ✓ *readme.txt*: readme file to get start to the *HTTP-based mini game back-end (Beyond Spock Endpoint)*

8.2. Run Beyond Spock Endpoint

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

Beyond Spock Endpoint can be run by executing the below prompt command:

C:> java –jar BeyondSpockEndpoint.jar

The follow line means that the Endpoint started properly:

\m/ >_< \m/   BeyondSpockServer works for the Founders @ 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:

- ✓ Synchronize *BeyondSpockEndpointProject GitHub repository*
- ✓ *C:> javac -d classes @CompileMain*
- ✓ *C:> jar cmf BeyondSpockManifest BeyondSpockEndpoint.jar -C classes .*
- ✓ *C:> java -jar BeyondSpockEndpoint.jar*

The follow line means that the Endpoint started properly:

\m/ >_< \m/   BeyondSpockServer works for the Founders @ http://localhost:8081/   \m/ >_< \m/

Exit the Endpoint by typing:

<Ctrl>+c