**Description**

WTTS is a simple ticket service that facilitates the discovery, temporary hold, and final reservation of seats within a high-demand performance venue.

**Setup (TEST, BUILD, AND RUN)**

* Download or clone the project from github.
* In the command line, navigate into the project folder. [same level as the pom.xml]
* To run the tests, run the command “mvn test”. [Note that test will take approximately 40 seconds to complete]
* To build the application run Maven command “mvn clean install ” [Note that this command will run tests again]
* When you successfully build the application, a target folder will be created.
* To run the application, enter to target folder [“cd target”]. Then run the command “java –war wtts.war”.
* Open the browser, and visit the link [http://localhost:23000](http://localhost:23000/) where the application should be up and running.

**Assumptions**

This application was created with keeping in mind some main requirements. The most important of them is that the application manages a high-demand resource. Therefore, performance is a major factor while designing the solution.

Before explaining the application design flow, let’s make some assumptions:

1. Because customers do not have the ability to choose seats when holding them, the best available seats in this application will be the first available found, starting from the right side in the closest line to the stage.
2. This application will run on one mono-processor server.
3. Venue size is 100 seats.
4. Time to hold seats before expiring is: 10 seconds.

**TESTS**

There are 8 Unit Tests in WTTS, documented in details in the source code.

**Tools**

* This is a SpringBoot application, so it is self contained and can run locally.
* Uses Spring MVC to simplify data communication from front\_end, and manage session.
* Using JQuery library to send form requests. It could be without it, by just binding the form with spring tags.

**High Level Design**

Please access the High Level Design Diagram WTTS.pdf

The Idea around this design is to separate the three main tasks from each other, Search for available seats, hold the best seats, and reserve the held seats. For that, the application uses 3 data structures and 1 primitive variable.

1. int primitive : numberSeatsAvailable.
2. ConcurrentHashMap: temporary SeatHoldMap
3. HashMap: reservedHashMap
4. ArrayList: Actual list of seats to hold and reserve.

Let’s go thru a test case to illustrate better the idea:

* Starting by holding 5 seats:

The system will check if 5 <= numberSeatsAvailable (100), which is the case. So it processes to hold the seats directly by creating a new SeatHold in the temporary seatHoldMap. Then in the Seats list, it will directly set the time stamp, because the value of their seatHoldId will be 0, which means that the seat has never been held before, and it is eligible to be held immediately. Seats which will be held are [1,2,3,4,5]

* Second action will be to reserve those 5 seats:

Within 10 seconds or less, the customer pushes the button reserve; the application removes the seathold from the temporary map, adds it into reserveSeatHoldMap, and set the confirmation code.

* 3rd action is to hold 3 other seats;

Same thing, the application will check 3<=numberSeatsAvailable(95), so it will hold it same as first action. Note that, while holding the seats this time, the value of seatHoldId of the first 5 Seats is not 0 because they had already been reserved. So for each of them, it checks if their seatHold is reserved in reservedSeatMap. This last operation looks like an additional task that can slowdown the application. Well, it is not, because it will set isReserved attribute of those 5 seats as true, so next time when holding seats again, no need to check again within reserveSeatMap, since the flag isReserved is set to true. In addition, accessing the concurrent hash map with a specific key is a very fast operation, according to java documentation. Seats which will be held are [6,7,8]

* 4th action, leave the last seatHold expires;
* 5th action, hold 94 seats.

Now the 94 seats are not <= numberSeatsAvailable(92), in this case, instead of iterating the list of actual seats to know if we have available seats, the service will iterate the temporary seatHoldMap. It finds the first seatHold node, check for it is expired (which is the case), it adds its number of seats to numberSeatsAvailable, and check again (94<= 92+3(expired)). True, then go directly and hold the best available seats for the current customer.

**Basically, the application is simply uses the primitive numberSeatsAvailable as Proxy against the temporary map, and also uses the temporary map seatHoldMap as proxy against the actual list of Seats.**

**Note**

This solution uses multi-threading and allows concurrent access to some resources in order to achieve a high performance. This will give some complexity when maintaining and testing the application.

**Beyond**

We are assuming that the best available seats to hold are the first available found in the closest line to the stage. If this assumption changes, like for example if we imagine the venue a an movie amphitheater, I personally will prefer a seat in between the middle and the last line. Which makes requirements more complex. But, the Ideal solution will be to give the state, and location of each seat to users in the front\_end. And with some observable pasterns and reactive programming, the application can maintain the state of each seat in real time for users, so they can choose any available seats, according to their own preferences. However, this solution requires a reach client user interface, which is not mandatory in the requirements.