DCU School of Computing

Assignment Submission

Student Name(s):	**Shaun Kinsella, Shane Creedon**
Student Number(s):	**14740175, 15337356**
Programme:	**BSc in Computer Applications**
Project Title:	**Distributed Computing Assignment 2**
Module code:	**CA4006**
Lecturer:	**Dr Martin Crane, Dr Rob Brennan**
Project Due Date:	**12/04/2019**

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying is a grave and serious offence in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references.

I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.

This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. I have read and understood the referencing guidelines found at http://www.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.

I understand that I may be required to discuss with the module lecturer/s the contents of this submission.
I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.

Signed: Shaun Kinsella, Shane Creedon

How to run

Pre-steps:

1. Sudo apt install maven

Server:

1. CD into the server directory
2. mvn package
3. cd into target dir
4. java -jar {jarname}.jar

Client:

1. CD into the client directory
2. mvn package
3. cd into target dir
4. java -jar {jarname}.jar
5. optionally provide ip and port in the form of {127.0.0.1:8080} (it will default to localhost)

Note: This is written to run on a Linux distribution.
Note: If running on Windows, one must copy(cp) the rooms.json file from the src directory of the server to the target directory of the server before running.

Architecture

The Server

Our booking system records are written as a json structure which contains the particulars of each available room, the days, time-slots and capacity left for each day. We utilise a jackson **ObjectMapper** in the class **RoomsMapper** to read in this file and it then converts this into the java classes; **Days**, **Room, Rooms** which we use to manipulate for the booking system. This is done initially on server startup by providing a **@Bean** for both the **RoomMapper** and the instantiation of the **Rooms** object, in **AppConfig** class, and then declaring them both **@Autowired** in our **RoomsController** class via Java's Spring framework.

The **RoomsController** class consists of our rest endpoints for the booking system, *returnAllRooms()*, *returnRoom()*, *returnDay()*, *bookDay()* and *roomAvailableAtSpecificTime()* all of which bar the last are declared asynchronous to prevent requests from blocking further client requests. We explicitly declare the rest verb for each method, all of which are *GET*, except for bookday() which is declared *PUT* instead of *POST* to avoid multiple records of bookings and to maintain **Idempotency**. We use the **@RequestMapping** annotation to do this, as well as declaring the URI to each method. This allows us to address our arguments to the methods as **@PathVariables.**

Eg: "http://172.16.204.2:8080/rooms/L221/0/12-1**"** with the *PUT* verb attempts to book the room "L221", Monday, at slot 12-1.

As mentioned, our endpoint methods are declared as asynchronous, returning a completedFuture version of the object to avoid blocking. Each time one of these methods are called, the **ThreadPoolTaskExecutor** in our **AppConfig** class gives this task to one of the pooled threads when it becomes free. This threadpool contains a

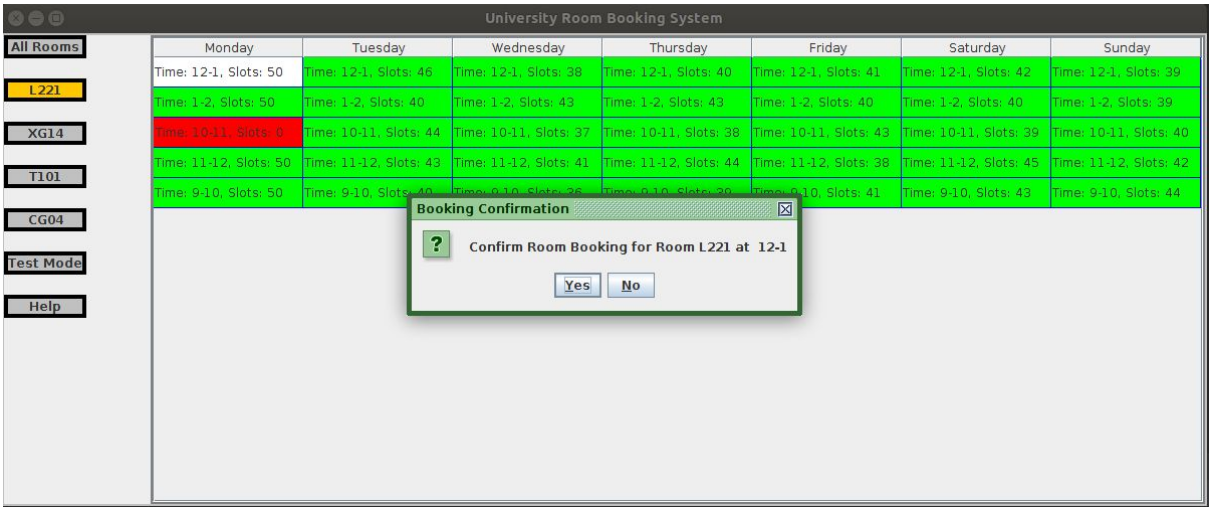
queue for jobs awaiting for a thread to become available. It can be provisioned accordingly for the servers need/hardware available in terms of starting threadpool size, its max size the thread pool can grow to and then the size of the queue of waiting tasks. If the queue becomes fully saturated it will throw a RejectedExecution exception so it is important to provision correctly.

The capacity available for a given room, day and time-slot is held in a **concurrentHashmap** belonging to the **Days** class. Any request which checks the availability of a room must query this concurrentHashmap, which it does using the **AtomicInteger** operation **updateAndGet()**. The concurrentHashmap allows multiple threads to access the booking records concurrently, not locking for any reads, and only locking the specific entry that is being written to at that time, guaranteeing mutual exclusion.

```
public ConcurrentHashMap<String, AtomicInteger> updateTimeSlotCapacity(String timeslot)
{
    ConcurrentHashMap<String, AtomicInteger> newTimeslots = timeslotCapacity;
    newTimeslots.get(timeslot).updateAndGet(capacity -> capacity > 0 ? capacity - 1 : capacity);
    return newTimeslots;
}
```

The Client

Our client consists of a GUI written in **JSwing**, allowing a user to view the available rooms and their max capacity, the timetable of a room for a full week and the slots left free to book by clicking the relevant room button. Upon clicking a time-slot to book it, the user will be prompted for confirmation to attempt to book it. Clicking yes will cause the client to fire off a *GET* request to ensure that there is a slot available since the client last updated the time-slot listings. If there are no available spaces left the user will be prompted to select another time. If there is a free slot a *PUT* request will be fired off to the server. The above does not guarantee that another client won't steal the last available slot, the only thing that is guaranteed is that the room will not be overbooked. Whomever's request arrives first and processed first will be given the spot. This can come down to network and thread execution order.



The requests are processed and sent off by the **RestClient** class. It is here that the server ip can be set. We utilise **Spring restTemplates** to build our requests. **getForObject()** allows us to pass it a URI string built up from the clients request and returns a **responseEntity** which we then store as a string. In order to allow the client to manipulate this, we declare a TypeRef of type **HashMap<String, Object>** and pass that back to the method that requested it, whether it be the client table booking button or the testing mode functions.

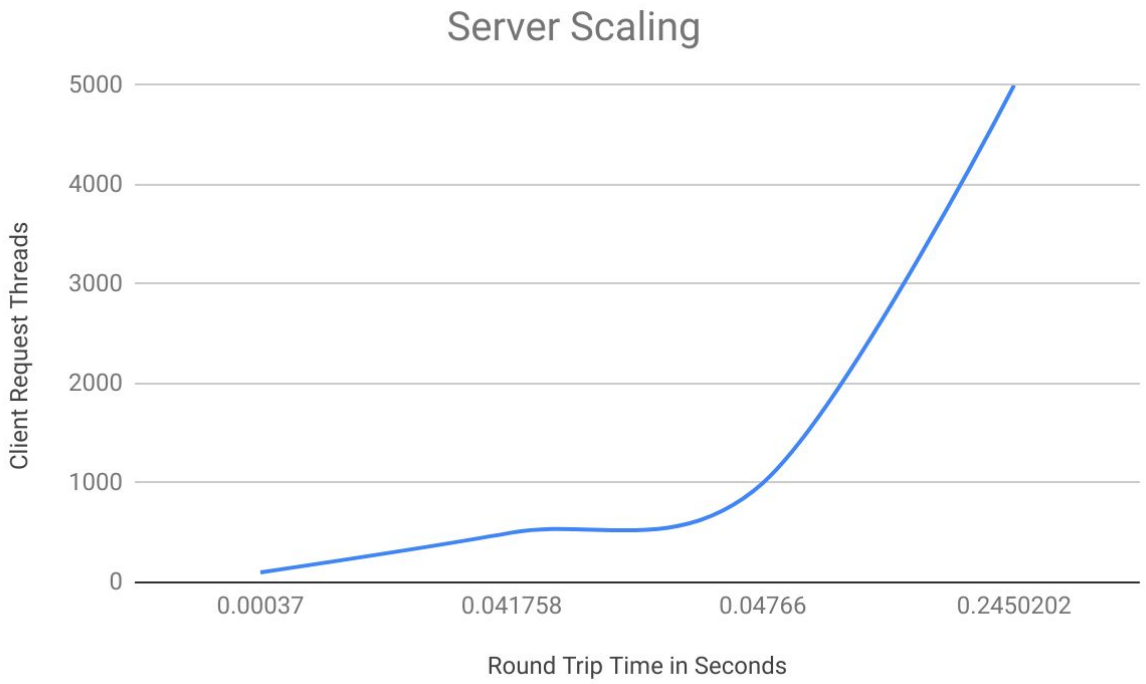
In the case of the **RestClients bookRoom()** method, we pass **pathvariables** as before but also create **HttpHeaders** in order to declare the encoding of the object we are passing in the request body. In this case it is the name of the current thread and a concatenation of the requested room, day and time-slot, which is then extracted server side and logged by our **serverLogWriter** class for debugging purposes.

To handle server downtime and inform the client of such a downtime, we would catch any exception that was potentially thrown via the **restClient** and subsequently spin up a special 'ServerCrashed' thread using the Executor Service framework. This thread would disable the functionality of the UI temporarily while the server remains down. Additionally, the thread would enter into a 'pingful' state. What we mean by this is, we would try to poll/connect the server's IP address to check for a response. We set a limit of this polling mechanism to every 3 seconds such that we don't send too many requests too quickly. We wrote a method namely **hostAvailabilityCheck** which would ping the web server for availability. To do this we used the notable library of java.net.Socket. Once the server came back online, the client would detect that change within 3-seconds and would react appropriately, by re-enabling the UI capabilities and giving access back to the user in relation to the booking system.

Additionally, if the client has connected to the server and the server goes down shortly after, the application will also react appropriately, either by means of GUI textual display or via logging to the terminal. If the server is reconnected, the GUI will inform the user subsequently when the system has returned back online.

For the testing mode, we created both the **ClientRequests** and **ClientRequestFactory** classes. **ClientRequestFactory** is responsible for generating the supplied amount of ClientRequest threads, which will call *PUT* requests on random rooms, days, times and also a random "request time" simulating a booking system under stress. It is also responsible for scheduling these threads using a **scheduledExecutorService** and **scheduledThreadPool.** The threads are generated at startup time but will not be ran until test mode is selected. This test mode can be initiated by clicking it on the main GUI menu, the scheduler will then proceed to issue requests. The **ClientRequests** class will log these requests to clientOutput.dat using the **ClientLogWriter**.

Scaling Testing:



What we would do next time:

- ~~Not use json…~~
- Implement a reverse proxy to sit in-between the client and server. This would allow us to give greater horizontal scaling by allowing the proxy server to act as a job dispatcher. All job processing logic would remain on the server but as to how jobs are distributed would be up to the proxy server, providing a degree of loose coupling. The proxy server could maintain a **blockingQueue** to hold incoming client requests and dispatch them to the available servers which could be held in a simple **arrayList**. The dispatching method could be swapped to match the requirements of the server, a simple round robin mechanism, or a dumb master/slave system. The current "master" server would be unaware of its authority so if it were to crash, the proxy server would designate the slave as the new main. This avoids the split brain problem if the old "master" server would suddenly come back, or if it was just a network connection error between the master and slave. Introducing the proxy server would however keep a single point of failure so other contingencies would have to be introduced. However, mercifully this report is limited in length.

- Related to the above, we would have to reconsider how/where we would store our "database", and share state between multiple servers, if sticking with a file we would most likely host it on a dedicated machine to share the data between servers. One possible easy way out would be to split the db between the servers when under load. One server could handle requests dealing with the first x rooms, the rest to the 2nd server. However if one server went down we would have to forward on the any requests that the proxy did not receive back.