Computer Networks
Socket Programming – HTTP-Based Room Reservation

Mehmet Alıcı 150118060
Emrullah Sevmiş 150119016
Abdül Samet Yılmaz 150118059

Due Date: 03.01.2023

1) Overall Description

The main goal of this project is to implement three HTTP servers that can be used to manage room reservations and activities. The servers should be able to handle various types of GET requests, as described in the original question, in order to add and remove rooms and activities, check the availability of rooms and activities, and reserve rooms for specific days and times.

2) Requirement Specifications

The Room Server should listen for incoming HTTP requests on a specific port number, which is specified as a command-line argument when the server is started.

The Room Server should support the following types of GET requests:

/add: This request should add a new room to the server's database, if the room does not already exist.

/remove: This request should remove a room from the server's database, if the room exists.

/reserve: This request should reserve a room for a specific day and time, if the room is available.

/checkavailability: This request should check the availability of a room for a specific day, and return a list of available times.

The Activity Server should listen for incoming HTTP requests on a specific port number, which is specified as a command-line argument when the server is started.

The Activity Server should support the following types of GET requests:

/add: This request should add a new activity to the server's database, if the activity does not already exist.

/remove: This request should remove an activity from the server's database, if the activity exists.

/check: This request should check whether an activity exists in the server's database.

The Reservation Server should listen for incoming HTTP requests on a specific port number, which is specified as a command-line argument when the server is started.

The Reservation Server should handle requests to reserve rooms for activities by communicating with both the Room Server and the Activity Server to check the availability of the requested room and activity.

The servers should return appropriate HTTP response codes and messages for each type of request.

3) Implementation Details

Before going into the implementation details, we would like to point out that we used Java as the programming language and txt files for data storage.

While implementing this project, since there are 3 servers that needed to be programmed, we have collected each server in packages under their own name. And each packet, apart from some differences, consists of 1 main class that is used to manage client connections and sending messages to the server, and 2 helper classes, one that listens for incoming connections and the other that specifies the request types required for that server.

Now let's examine in detail how we implement each server.

Room Server

To implement this server we used 4 clases:

RequestType: It is a minor helper class which consists of only 4 fields representing different types of requests that can be made to a room reservation server add, remove, checkavailability and reserve. It has no methods.

RoomServer: It is an another minor class that represents a server for managing room reservations. It has a main method, which is the entry point for the program when it is run, and a static final field called ROOMS_FILE that represents the file where the server stores information about the rooms it manages.

The main method creates a ServerSocket object on port 8080 and enters a loop where it listens for incoming connections. When a connection is accepted, it creates a new SocketHandler (the main class in the package that handles connection and requests) object and starts a new thread to handle the connection. This allows the server to handle multiple requests concurrently.

SocketHandler:

The SocketHandler class is used to handle client connections to the server. It implements the Runnable interface, so it can be run in a separate thread. This is useful because it allows the server to handle multiple client connections concurrently.

The SocketHandler class has a single field, client, which is a Socket object representing the client connection.

The run method of the SocketHandler class is the entry point for the thread. It reads the first line of the client request, which indicates the type of request, and then processes the request accordingly.

There are four types of requests that the server can handle:

add: Add a new room to the database.

remove: Remove an existing room from the database.

checkavailability: Check the availability of a room on a particular day of the week.

reserve: Reserve a room for a particular time period on a particular day.

To process the request, the run method calls one of several helper methods:

getRequestType(String request): This method parses the request string and returns the request type as an enumeration. The request string has the following format: "<request type> <room name>". The request type is one of "add", "remove", "checkavailability", or "reserve". This method extracts the request type from the request string and returns it.

getNameOfRoom(String request): This method parses the request string and returns the name of the room specified in the request. The request string has the following format: "<request type> <room name>". This method extracts the room name from the request string and returns it.

RoomExistInDB(String roomName): This method checks if a room with the given name exists in the database. It returns true if the room exists, and false otherwise.

AddRoomToDB(String roomName): This method adds a new room with the given name to the database. It stores the room information in a file on the server's file system.

RemoveRoomFromDB(String roomName): This method removes the room with the given name from the database. It removes the room information from the file on the server's file system.

getCheckAvailabilityParameters(String request): This method parses the request string and returns a list of parameters for the checkavailability request, including the name of the room and the day of the week. The request string has the following format: "checkavailability <room name> <day>". This method extracts the room name and day from the request string and returns them as a list.

getAvailableHours(String roomName, int day): This method returns a list of available hours for the given room on the given day of the week. It reads the room information from the database and determines which hours are available.

SendAvaibleHours(Socket client, List<Integer> availableHours, String roomName, int day): This method sends a response to the client with the list of available hours for the given room on the given day of the week. The response is a JSON object with the following format: {"roomName": <room name>, "day": <day>, "availableHours": <available hours>}.

getReserveParameters(String request): This method parses the request string and returns a list of parameters for the reserve request, including the name of the room, the day of the week, the start time, and the end time. The request string has the following format: "reserve <room name> <day> <start time> <end time>". This method extracts the room name, day, start time, and end time from the request string and returns them as a list.

MakeReservation(List<String> reserveParameters): This method makes a reservation for the given room on the given day at the given start and end times. It updates the room information in the database to reflect the new reservation.

Finally, the SocketHandler class has several methods for sending response messages to the client:

Send200Message(Socket client, String message): This method sends a 200 OK response to the client with the given message.

Send403Message(Socket client, String message): This method sends a 403 Forbidden response to the client with the given message.

Send404Message(Socket client, String roomName): This method sends a 404 Not Found response to the client with a message indicating that the room with the given name was not found.

RoomInfo: Another helper class which is used to store information about the rooms, including their name and a list of reservations.

Here is a description of the fields and methods of the RoomInfo class:

String roomName: This is a string representing the name of the room.

List<Reservation> reservations: This is a list of Reservation objects representing the reservations for the room.

RoomInfo(String roomName): This is the constructor for the RoomInfo class. It takes a string representing the room name and assigns it to the roomName field. It also initializes the reservations list.

void addReservation(Reservation reservation): This method adds a Reservation object to the reservations list.

List<Integer> getAvailableHours(int day): This method returns a list of available hours for the room on the given day of the week. It does this by iterating through the reservations list and checking which hours are not already reserved.

Activity Server

To implement this server we used 3 clases:

RequestType: It is a minor helper class which consists of only 3 fields representing different types of requests that can be made to a room reservation server; add, remove, check. It has no methods.

ActivityServer: It is an another minor class that listens for connections on a specified port and handles requests made to it via socket connections. When the server is started, it creates a ServerSocket object and binds it to the specified port(8082). It then enters an infinite loop that listens for connections and creates a new SocketHandler object to handle each connection. The SocketHandler object is passed to a new Thread object, which allows multiple connections to be handled concurrently. The ACTIVITY_FILE field is a File object that represents the database file that is used to store the activities.

SocketHandler:

The SocketHandler class is a class that handles requests made to a server via socket connections. It has several methods that are used to add, remove, and check for the existence of activities in a database. The database is represented by the file ActivityServer.ACTIVITY_FILE.

Here is a detailed explanation of each method in the class :

run() method: The run() method is the main method of the SocketHandler class. It is called when a new SocketHandler object is created and passed to a Thread object.

The run() method reads a line of input from the input stream of the client socket, which represents a connection to a client. It then determines the type of request made using the getRequestType() method (which is not shown in the code snippet). Based on the type of request, it either adds, removes, or checks for the existence of an activity in the database. The run() method also includes a try-catch block to handle any exceptions that may occur. If an exception occurs, the error message is printed to the console and is sent to the client server.

addActivityToDB() method: The addActivityToDB() method adds an activity to the database by appending the activity's name to the end of the ActivityServer.ACTIVITY_FILE file.

To do this, the method first synchronizes access to the ActivityServer.ACTIVITY_FILE file to avoid race conditions. It then opens the file in append mode using a BufferedWriter object and writes the activity's name to the file. Finally, it closes the BufferedWriter object.

activityExistInDB() method: The activityExistInDB() method checks whether an activity with a given name exists in the database by reading the ActivityServer.ACTIVITY_FILE file and searching for a line that matches the given name.

To do this, the method first synchronizes access to the ActivityServer.ACTIVITY_FILE file to avoid race conditions. It then opens the file using a BufferedReader object and reads each line of the file until a match is found or the end of the file is reached. If a match is found, the method returns true. If no match is found, the method returns false. Finally, the method closes the BufferedReader object.

removeActivityFromDB() method: The removeActivityFromDB() method removes an activity from the database by reading the ActivityServer.ACTIVITY_FILE file and creating a new list of lines that does not contain the name of the activity to be removed. It then writes the new list of lines back to the file, effectively overwriting the original file and removing the activity from the database.

To do this, the method first synchronizes access to the ActivityServer.ACTIVITY_FILE file to avoid race conditions. It then reads the lines of the file using the Files.lines() method and filters the lines using the Collectors.toList() method to create a list of lines that do not contain the name of the activity to be removed. Finally, the method writes the list of lines back to the file using the Files.write() method.

getNameOfActivity() method: The getNameOfActivity() method extracts the name of the activity from the request string.

To do this, the method first splits the request string into two parts using the split() method and the space character as the delimiter. It then splits the second part of the request string into two parts using the split() method and the forward slash character as the delimiter, taking the second part of the resulting array as the request endpoint.

Next, the method splits the request endpoint into two parts using the split() method and the question mark character as the delimiter, taking the second part of the

resulting array and splitting it again using the split() method and the ampersand character as the delimiter. It then takes the first part of the resulting array and splits it again using the split() method and the equal sign character as the delimiter, taking the second part of the resulting array as the name of the activity.

Rerservation Server:

To implement this server we used 5 clases:

RequestType: It is a minor helper class which consists of only 4 fields representing different types of requests that can be made to a room reservation server reserve, listavailability, listavailabilityWithDay and display. It has no methods.

RerservationServer: The ReservationServer class is the main class for the reservation server. It has a single field, RESERVATION_FILE, which is a File object representing the file containing the database of reservations made by the server.

The main() method is the entry point of the program. It starts an infinite loop that listens for incoming connections on port 8081 using a ServerSocket object. For each incoming connection, it creates a new SocketHandler object and starts a new thread to handle the request. This allows the server to handle multiple requests concurrently.

SocketHandler: This is a class that handles incoming requests for reserving rooms and displays information about reservations. The SocketHandler class implements the Runnable interface and overrides the run() method to handle requests from a client socket.

To process the request, the run method calls one of several helper methods:

run() method:

It reads the request from the client using a BufferedReader object.

It calls the getRequestType() method to parse the request and determine the type of request.

It then performs a switch statement based on the type of request and calls the appropriate method to handle the request.

getRequestType() method:

It takes a string containing the request as an argument and returns an object of type Request.

The Request object has two fields: name, a string representing the type of request, and parameters, a HashMap containing the parameters of the request.

The method parses the request string to extract the type of request and its parameters.

ValidateReserveRequest() method:

It takes a HashMap object containing the parameters of a reserve request as an argument.

It checks that all the required parameters for a reserve request are present in the HashMap. If any required parameter is missing, it throws an exception.

CheckActivityExist() method:

It takes a string representing the name of an activity as an argument and returns a boolean indicating whether the activity exists or not.

MakeReservation() method:

It takes a HashMap object containing the parameters of a reserve request as an argument and returns a boolean indicating whether the reservation was successful or not.

ListAvailableHoursWithDay() method:

It takes a Socket object representing the client and two strings representing a room name and a day as arguments.

It creates a new socket to communicate with the server running on localhost, port 8080.

It sends a GET request to the localhost server to check the availability of the specified room on the specified day.

It reads the response from the localhost server and determines the status code (200, 400, or 404).

If the status code is 200, it calls the Send200Message() method to send a success message to the client.

If the status code is 400, it calls the Send400Message() method to send a bad request message to the client.

If the status code is 404, it calls the Send404Message() method to send a not found message to the client.

ListAvailableHours() method:

It takes a Socket object representing the client and a string representing a room name as arguments.

It works in a similar way to the ListAvailableHoursWithDay() method, except it sends a GET request to the localhost server to check the availability of the specified room on all days.

reservationExistInDB() method:

It takes a string representing the ID of a reservation as an argument and returns an object of type ReservationInfo if the reservation exists, or null if it does not.

Send200Message() method:

The Send200Message() method is a helper method that sends a success message to the client with a specified message as the body of the response. The message is passed as a string argument to the method, and the Socket object representing the client is also passed as an argument. The method sends the success message to the client by writing the appropriate HTTP response to the output stream of the Socket object.

Send404Message() method:

It takes a Socket object representing the client and a string containing an error message as arguments.

It sends a not found error message to the client with the specified error message as the body of the response.

Send403Message() method:

It takes a Socket object representing the client as an argument.

It sends a forbidden error message to the client.

Send400Message() method:

It takes a Socket object representing the client and a string containing an error message as arguments.

It sends a bad request error message to the client with the specified error message as the body of the response.

Send200MessageWithReservationInfo() method:

It takes a Socket object representing the client and an object of type ReservationInfo as arguments.

It sends a success message to the client with the information contained in the ReservationInfo object as the body of the response.

GetStatusCode() method:

It takes a string containing the response from the localhost server as an argument and returns the status code (200, 400, or 404) as a string.

ReservationInfo: The ReservationInfo class is a simple class that represents information about a reservation made by the server. It has five fields:

-id, a string representing the ID of the reservation.

-name, a string representing the name of the room.

-day, a string representing the day of the week when the reservation was made.

-hour, a string representing the hour when the reservation starts.

-duration, a string representing the duration of the reservation.

The ReservationInfo class also has a toString() method that returns a string representation of the reservation information in a human-readable format. The string contains the reservation ID, the name of the room, and the day and time when the reservation starts and ends. The toString() method uses the daysOfWeek array to convert the day number to a day of the week.

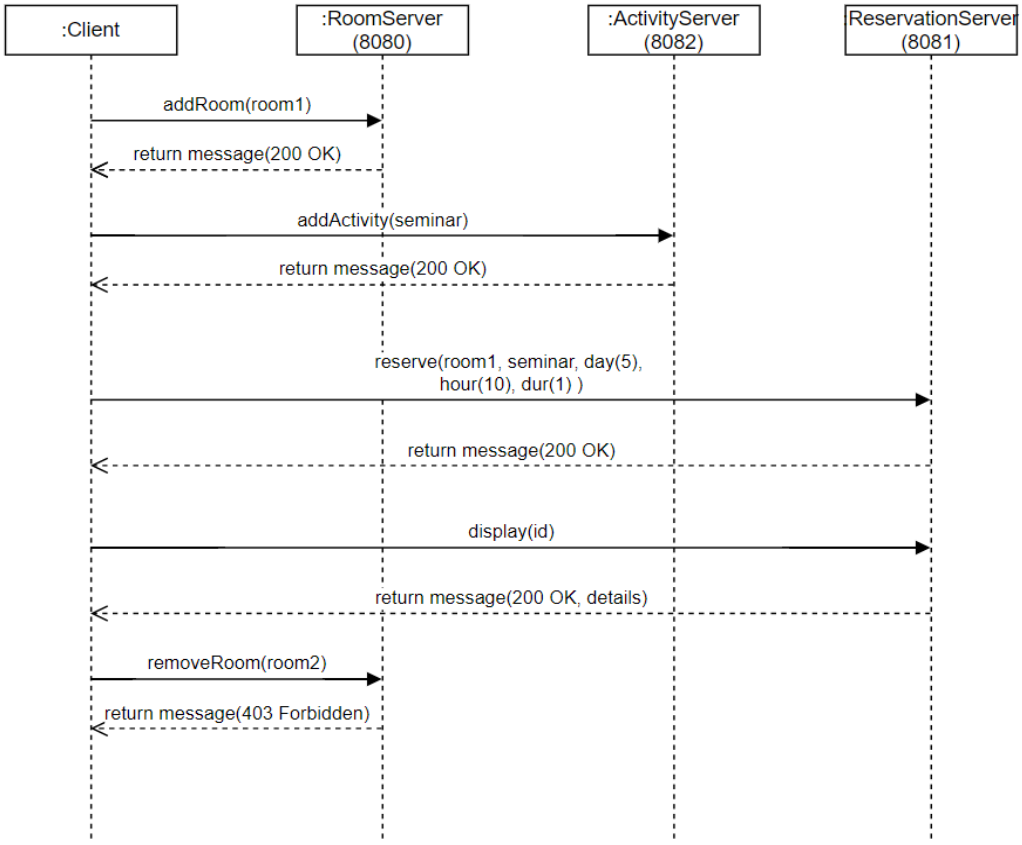
Request: The Request class is a simple class that represents a request made to the server. It has two fields:
-name, a string representing the type of request.
-parameters, a HashMap containing the parameters of the request.

The Request class has a single constructor that takes two arguments: a string representing the name of the request and a HashMap containing the parameters of the request. The constructor assigns these values to the name and parameters fields, respectively.

The Request class is used in the SocketHandler class to represent the requests made by clients to the server. The name field is used to determine the type of request, and the parameters field is used to access the specific parameters of the request.

It should be noted that our project is implemented in a multi-threaded way to solve the concurrency issues among multiple users and since it is a multi-threaded implementation there might be issues with resource sharing. Therefore to prevent possible race conditions we used the synchronized keyword to acquire a lock on the required objects before accessing the shared resource (room, activity, reservation databases). This ensures that only one thread can access the database at a time, preventing race conditions and ensuring that the database is accessed safely by multiple threads.

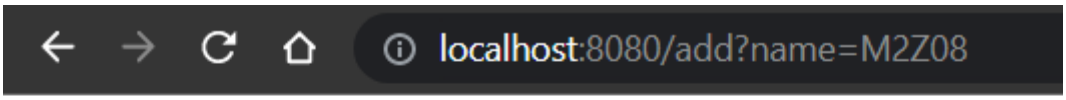
3) System Sequence Diagram (SSD for design document)



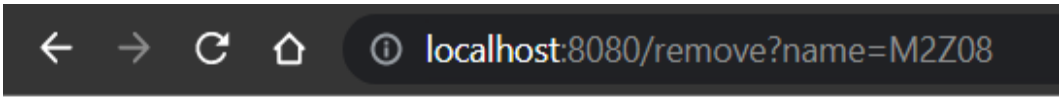
4) Sample Outputs

Here is some screenshots of the outputs:

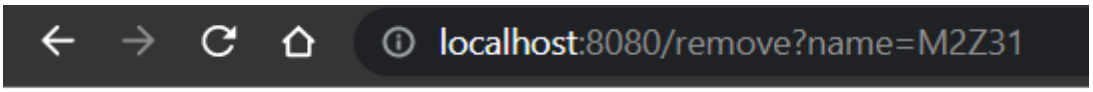
RoomServer



M2Z08 is added successfully



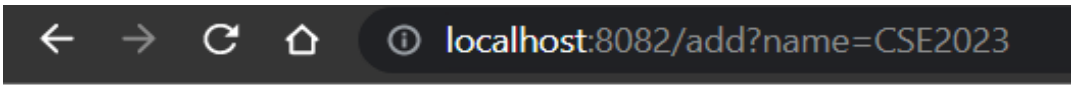
M2Z08 is removed successfully



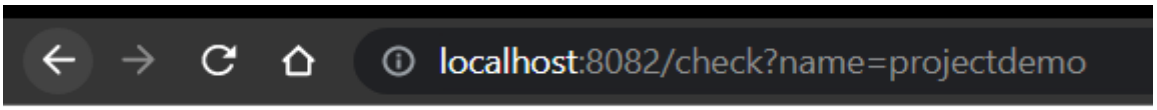
Error, Room with name M2Z31 does not exist

ActivityServer

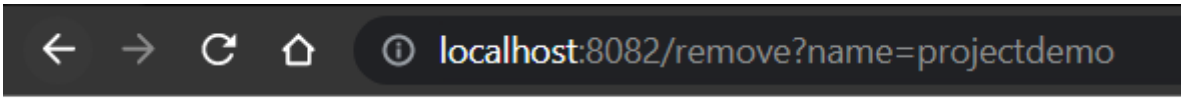
```
C:\Users\byzjdr>curl http://localhost:8082/add?name=CSE2023
<body>CSE2023 is added successfully</body>
```



Activity with name CSE2023 is already added

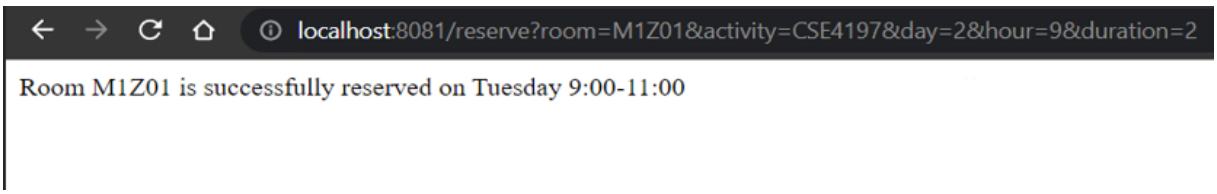


projectdemo exists



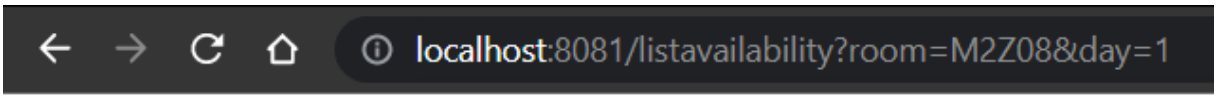
projectdemo is removed successfully

Reservation Server

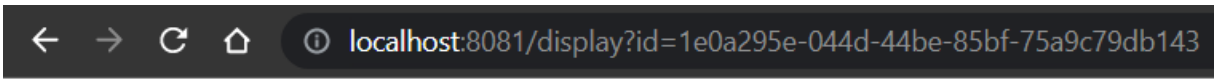


Available Hours On Friday, Room M1Z01

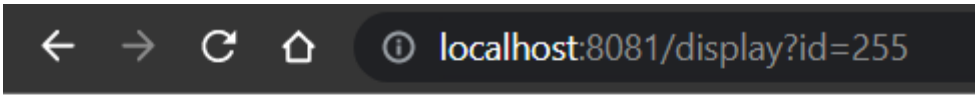
- 9
- 14
- 15
- 16
- 17



Error: Inputs are not valid.



Reservation ID: 1e0a295e-044d-44be-85bf-75a9c79db143
Room: M1Z01
When: Friday 10:00-12:00



Error: Reservation with ID 255 is not found