# CmpE 436 Term Project Report

Irmak Kavasoglu

**Abstract**

The Boun Room Allocator application is an Android application, aimed to help students and teachers find available rooms for the classes or problem sessions. It creates a weekly schedule for several rooms in several buildings, and can provide all types of users with the availability information. The application also takes into consideration the capacity of the rooms while showing the available rooms for a given slot.

## 1 Introduction

As my term project, I have decided to make an Android application for scheduling lectures in different rooms. With this application;

- Facility managers are in control of the rooms and buildings
- Teachers are able to search for available rooms for their lectures or meetings
- Students are able to see which classroom is occupied when

The details will be explained in the following sections.

## 2 Overview

For the project's purpose, I have chosen Android platform for my client application, and used Android Studio as my development tool. Therefore, the client application is written in Java mostly (Android uses xml for UI screens).

The client application communicates with a server application. For testing purposes, I have run this server application on my own computer, but it can easily be transferred to an Amazon machine since it does not require any dependencies. The server application is also written in Java, but the environment I used for it is Eclipse.

For communication between them, I use Java sockets. For messages and database, I use JSON objects. Details for each part of the project will be explained in the following sections.

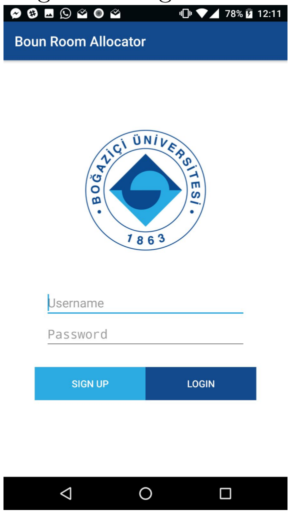
# 3 Detailed work

## 3.1 Client Side

As mentioned before, my application is an Android application. In this chapter I will explain the screens and the logic behind them.

### 3.1.1 Login and Sign Up Screens

Figure 1: Login Screen



The first screen that user sees after opening the application is the Login Screen, shown in Figure 1. This screen uses a linear layout with weights to keep percentages of the views the same in different devices with different screen sizes.

It has a Boun logo on top, an editable username field and an editable password field. Below the credentials, there are two buttons. The login button is on the right because it is expected to be used more frequently and the sign up button is on the left.

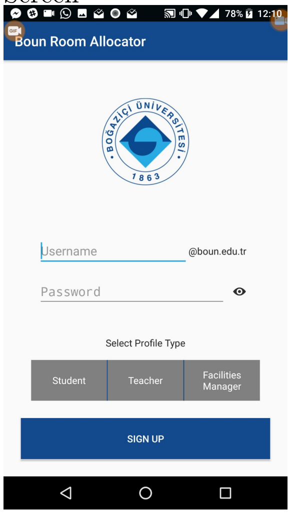
This screen is handled by the *LoginActivity.java* class. It implements two onClickListeners for the two buttons. Sign Up button's listener redirects user to the Sign Up Screen, which is shown in Figure 2. Login button's listener starts the login process.

Let's start with the Sign Up button's action. It starts the *SignUpActivity.java* when it is clicked. The Sign Up page has the logo as the Login Page. It has the username field, and next to it has an *@boun.edu.tr*. This gives user the information that the university email is required for sign up. I do not check if the email exists or not, but this is useful for both being able to give users unique usernames and understanding if they are actually the user type they have selected.

Below the username field, we see the password field. It starts as masked characters, but if user wants to see the typed password, it can be unmasked by clicking the eye icon next to it.

Below the credential fields, we see a selection with three options. This is the part where the user selects his/her user type. The application has three user types, these are: facilities manager, teacher and student. The gray box containing the option turns blue upon selection, and sets the user type. If user clicks another option, the first box turns back to gray and second one becomes blue, updating the user type.

Figure 2: Sign Up Screen



There is a sign up button in the bottom of the screen. When user click the sign up button, the first check its listener does is to make sure that the three required fields are filled. If not, it stops the signing up process and shows user a warning, indicating that they need to fill the fields first.

If user did fill all the fields, it starts up the signing up process. When the process starts, sign up button becomes a spinning process bar and lets user know that something is happening. Until the process bar is hidden, user interaction is blocked.

Since connections can be long, it is not a good idea to run them in the main thread. Therefore, sign up button starts an *ASyncTask* named *SignUpTask*.

*SignUpTask* first opens a Socket connection, using the host IP and port, which are declared in *Constants.java* file. When the connection is established, it sends out a JSONObject to the server. This object has the connection type field as *signUpConnection*. It has the username, password and usertype as the other fields.

After sending the message, it waits for a response. If the response arrives and the success field on the returned object is true, user is notified, *SignUpActivity* is killed and the screen goes back to login screen. Otherwise, user is informed that the process failed, the process bar becomes the sign up button again and user can try again.

Back to the login screen, we now inspect the login button. Similar to the sign up button, after checking if the fields are filled, it starts a new thread called *LoginTask*. In this task, it sets the connection type to *loginConnection* and adds the username and password fields into the JSONObject. Then waits for a response from server.

The server is expected to return the user type along with the success field. Depending on this field, *LoginActivity* starts one of the following activities: *StudentActivity* for type "student", *ManagerActivity* for type "manager" and *TeacherActivity* for type "teacher". The following sections explain these flows in detail.

### 3.1.2 Student Screen and Calendar

One of the three screens that login button redirects to is the student screen, shown in Figure 3, controlled by *StudentActivity*. This screen has the welcome message on top left, the refresh button on top right, and the buildings and rooms in the rest of the screen.

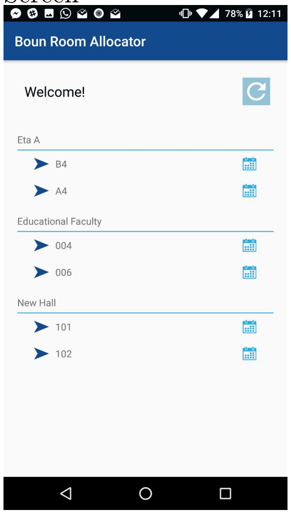


Figure 3: Student Screen

When this screen is shown, it performs a click on the refresh button. The refresh button is tied to a onClickListener, which starts a thread called *GetRoomsTask*. This task sends a message to the server with the connection type *getRoomsConnection*. With the response, it expects a JSON containing the buildings and rooms information. While waiting for the response, the refresh button becomes a spinning progress bar and block user information.

When the information arrives, the layout is processed. There is a container linear layout inside a vertically scrollable scrollview, starting from under the welcome text and occupies the rest of the screen. So if there are too many rooms or buildings, user can scroll down easily.

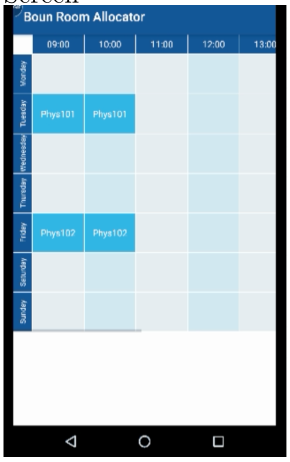
For buildings, I have a layout named *layout_ building_ container.xml*. This layout contains a textview for the name of the building and a light blue line under it.

For rooms, I have a layout named *layout_ room_ container.xml*. It contains a dark blue bullet point, margined from left. Next to it, it has a textview for the name of the room. And to the end of the view, it has an imageview. For the student screen, this imageview is selected as a light blue calendar icon.

Processing the information returned from the server, I create new instances of the building and room layouts by using *LayoutInflator*. These dynamically created views are then added to the container layout.

The calendar icons in every room, has a onClickListener. When user clicks on them, the *RoomCalendarActivity* is started with room and building information.

Figure 4: Calendar Screen



When calendar screen is started, it is started with the information of the name of the room and building. With this information, *RoomCalendarActivity* starts a new thread called *GetWeekTask*.

Using the information it was given in the start, the calendar sends a message to server. It has the connection type *getWeekConnection*, the name of the building and the name of the room. The expected result is a JSON object, containing the information of the occupied days and hours of the specified room, and the name of the lecture during that time.

To create a calendar view, I again use the *LayoutInflater*. As the screenshot shows, every row of the week is layout-wise the same. This layout row for one single day is named *layout_ day_ container.xml*.

For each day of the week, one day layout is created and added to a container view. Using the information from the response, the corresponding box in the schedule is painted blue and the name of the lecture is written on it. Since the timetable is always seven rows but it has columns for every hour from 9 to 18, I decided to place the container in a horizontally scrollable view.

When user goes back to the student screen, the refresh button is automatically clicked again, so that the user will have the latest changes.

### 3.1.3 Manager Screen and Building/Room Control

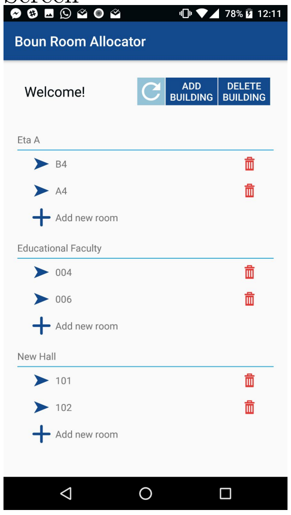
The second type of users are the facilities managers. If the logged in user is a manager, then the *ManagerActivity* is started, which is shown in Figure 5.

When started, this activity also uses *GetWeekTask*, just like the *StudentActivity*. When the result is received, it creates the layout in a similar manner; within a container wrapped by a vertically scrollable view.

The manager screen has several differences from student screen. The room layout is inflated in here too, but the icon on the right of the layout is replaced with a red trash icon in this screen. It gives away the purpose pretty clearly: a button to delete the corresponding room from the database.

Another difference is the two buttons to the right of the refresh button. One is for deleting buildings and the other is for adding new buildings. We want the facilities managers to be able to actively add new rooms, buildings as well as deleting the existing ones.

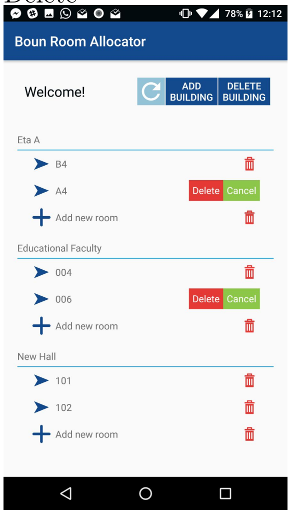
Figure 5: Manager Screen



**Room Deletion** Let's start with the room deletion. On the screen, to the right of each room, we see a red trash icon. This is an imageview with an onClickListener. It is very accessible, which is actually both an advantage and disadvantage.

It is an advantage, because we always want functionalities to be as few steps away as possible from the user. Putting the trash cans next to the rooms makes it easy for the user to delete any room s/he wants, easily.

Figure 6: Manager Screen with Room Delete



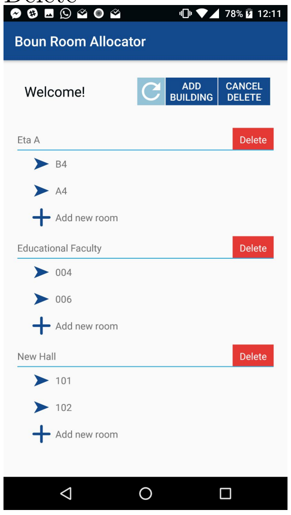
But there is also a disadvantage. The deletion is an action which is usually undoable, at least in this application's case it is undoable. Therefore, putting the icon this close may lead to mistakenly deleted classes.

To prevent this, I have added a second step to deleting a room, which is seen in Figure 6. When user clicks on the thrash icon, the icon is replaced with two buttons; one of them is the delete and the other one is cancel. The cancel button is to the right, and this is to make the delete button in a different place that it was before. This approach makes sure that user doesn't mistakenly delete a room, s/he needs to click the trash icon, find the delete button and delete it by using the delete icon.

If user clicks the cancel button, the trash can comes back and nothing happens. If user clicks on the delete button, a new thread named *DeleteRoomTask* is started. This task sends a message to the server, containing the connection type as *deleteRoomConnection*, the building and the room information.

When the server sends a response, the user is informed with a toast about if the deletion was successful or not. Also, after the operation is done, the task clicks the refresh icon to get the latest list of rooms.

Figure 7: Manager Screen with Building Delete



After having a thrash can next to every room, implementing the same pattern for buildings wouldn't look nice, it would be pretty crowded. Therefore, I have decided to use a slightly different approach for building deletion.

**Building Deletion** When a user wants to delete a building, they first use the delete building button on top. Instead of navigating to a new activity, I stay in the same screen, hide all of the existing trash icons, and put the delete buttons next to every building, as shown in Figure 7.

When user enters the building deletion mode, the delete building button becomes cancel button. User can cancel the operation any time, and the screen goes back to its original state.

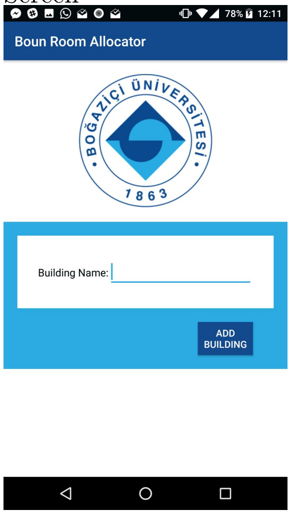
If user clicks on one of the delete buttons, a new task named *DeleteBuildingTask* is started and an object with connection type *deleteBuildingConnection* and building name with the selected building is sent to the server. Upon response, user is informed and page is refreshed.

**Adding Buildings** Now that deletion of both rooms and buildings are explained, let's consider how the application allows addition of these. Since adding rooms is dependent on buildings, we will start with adding buildings.

In Figures 5, 6 and 7, we see that manager screen has an add building button next to refresh button. This button has an onClickListener and when clicked, starts a new activity called *AddBuildingActivity*.

The *AddBuildingActivity* has the layout shown in Figure 8. It has the logo on top, it has an editable field for the building name, and it has an add building button.

Figure 8: Add Building Screen



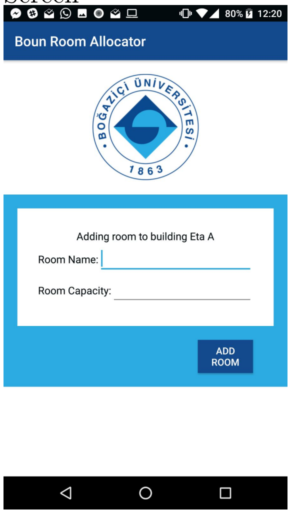
When user clicks on add building button, first, the building name is checked. If the field is empty, user is given a warning saying that the building name cannot be empty, and the process is stopped.

If the building name is not empty, the activity starts a new thread named *AddBuildingTask*. This task connects to server and sends a message. In the message, the connection type is declared as *AddBuilding-Connection*, and the building name is sent as well.

If the response from server is successful, the activity is killed, the manager screen is brought back and the refresh button is clicked so that the user can get the latest changes. If not, then user is notified and screen stays the same.

**Adding Rooms** In Figures 5, 6 and 7, we can see the add new room line under every building, placed under the existing rooms, indicated with a plus icon and a text. These add new room lines are clickable areas in the manager screen and when user clicks on them, they start a new activity called *AddRoomActivity*. While starting the new activity, they also send the building name that the room is going to be added.

Figure 9: Add Room Screen



When the add room screen is started as shown in Figure 9, the layout is created with a logo on top, two editable fields for room name and room capacity, and an add room button. The text on top of the editable fields is edited according to the received building name. The room capacity field only accepts numbers and not letters.

When user clicks add room, the editable fields are checked if they are empty or not. If they are not filled, user is warned and addition process stops. If they are filled, a new thread called *AddRoomTask* is started.

The *AddRoomTask* creates a connection between client and server, and sends a message. This message specifies the connection type as *addRoomConnection* and also contains the fields room name and building name.

If the response is successful, the activity is killed, user is notified with a Toast and the manager screen is brought back with a click of refresh button so that user can get latest changes.

### 3.1.4 Teacher Screen and Lectures

Boun Room Allocator's third and last user type is the teacher type. When user logs in with a teacher account, the screen shown can be seen in Figure 10.

The layout is a lot similar to the layouts of student and manager screens. User has the list of buildings and rooms, created in the same way as they were created in other screens (with *LayoutInflator* and *GetRoomsTask*).

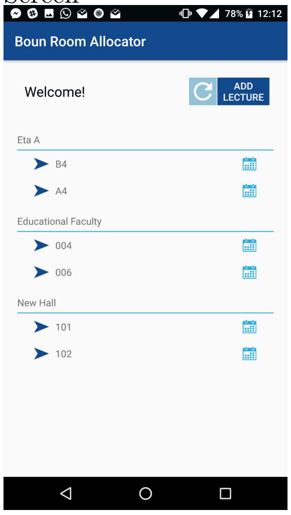


Figure 10: Teacher Screen

The image icon on the right is left as the calendar icon. The functionality of it is the same as the student screen, it starts the corresponding calendar of the selected room.

Next to the refresh icon, we see one button, which is the add lecture button. When a teacher wants to add a lecture to the weekly schedule, this button is going to start the process.

When the teacher clicks the add lecture icon, the *AddLectureActivity* is started. *AddLectureActivity* sets the layout as shown in Figure 11.

**Adding Lecture** The add lecture screen has one field for the lecture name, and under that it has a field for minimum capacity. The capacity field only accepts integers. Capacity is important because one class being available is not enough, it has to have enough space for the students who are taking that class. Remember the room capacity field in manager mode, since we know the class capacities from there, we can filter the classes using this required minimum capacity field.

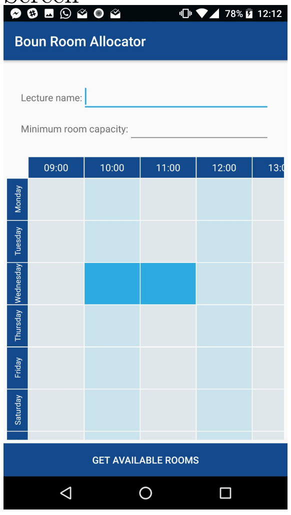


Figure 11: Add Lecture Screen

Below the two fields, we see a calendar. This calendar is created in a similar manner to the calendar in student page. It uses day layouts to make week view.

One different from the existing calendar view is that, here, every block has an onClickListener for itself. This listener is triggered when user clicks that field and toggles the cell color. This is a useful user interface for selecting the weekly schedule slots for the lecture.

The calendar has a horizontal scroll but since this activity contains more views than the calendar screen, there is also a vertical scroll. The vertical scroll contains all views except the button in bottom.

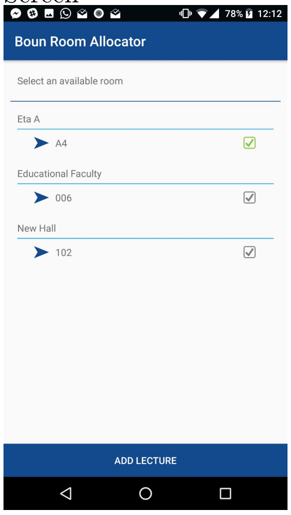
The button in the bottom is the get available rooms button. As the name states it, this button is going to start the process to get the information about selected slots, required capacity and lecture name, then communicate with the server and return the available rooms for the user to select one of them.

So, user fills in the required fields, then clicks the button. The button starts a new thread called *GetAvailableRoomsTask*. This task connects to server through a socket, and sends a message. The message has the connection type as *getAvailableRoomsConnection*. Message also contains the information about the minimum required capacity and the selected slots. When the results arrive, a new activity called *SelectRoomActivity* is started.

**Selecting Rooms** After user provides the constraints, the server checks the rooms and filters out the rooms which are available at the specified times and meet the capacity constraints.

With the resulting rooms, the select room screen arrives, seen in Figure 12. The layout starts with a text to tell user what this page is about, which is selecting a room.

Figure 12: Select Room Screen



Select an available room

Eta A

A4

Educational Faculty

006

New Hall

102

The screen's layout is designed similar to the student screen. The available rooms arrive with the buildings they are in, and the layout is constructed with room names under each building. Remember the action icon next to the rooms; we used them as thrash bins in manager screen and calendars in student screen. Here, they are used as selection icons.

Initially, all icons are gray. When user clicks on one of them, the selection is updated and the icon turns green. User can select only one of the options.

Of course, the room list in this screen is also in a vertical scroll view, because there may be a lot of available rooms and user should be able to see all of them without any problems.

Under the page, there is the add lecture button. This button starts a new thread called *AddLectureTask*. This thread starts a socket connection with server and sets the connection type as *AddLectureConnection*. As well as the selected room and building, previous data of lecture name and times are also included in the message.

When the response arrives from the server, the user is notified with the result and the activity is killed. The screen goes back to the teacher screen and presses the refresh button. If the process was successful, user will see the new lecture on the calendar. If not, they will be notified and will be able to try again.

## 3.2 Server Side

The server side of this project was written in Java. I have used Eclipse IDE as my working environment. In this section, I will explain the server side operations.

### 3.2.1 Room Allocator Server

The *RoomAllocatorServer* class is the server's initial class, only it has the main function. The main function only has one line of code, where it calls the start server function.

The start server function is made of two parts. The initialize server part and the endless loop for accepting connections. The server initialization opens a *ServerSocket* at port *60015.* Then in a while loop, server accepts connections from sockets, which are sent from the Android application.

For every accepted connection, the Room Allocator Server class starts a new thread by creating a new instance of the class *ConnectionHandleThread.* As it's name suggests, this class extends the Thread class. With this approach, server does not block the new requests while handling the older ones.

### 3.2.2 Connection Handle Thread

Connection Handle Thread takes in a parameter Socket in the Constructor. This thread is alive as long as the connection is alive.

When it is first run, it initializes the in and out streams from the socket, because in this application every socket connection is made of one request and one response. Then it reads in a JSON object. The communication in my application uses JSON objects because they are easy to convert from and to Strings, and this is useful for socket connections.

After receiving the JSON, Connection Handle Thread looks for the *connectionType* field in the object. This field will contain the information about from where this object is sent and what is its purpose.

Depending on the connection type, this class calls a function from the *DatabaseAccess.* The function name is simply the connection name, without the word *Connection.* Each connection type requires a different name for the extra data in the JSON that was received. The following table shows all possible connection types and their related extra's names.

| Connection's name | Extra's name |
|---|---:|
| *loginConnection* | credential |
| *signUpConnection* | credential |
| *addBuildingConnection* | building |
| *addLectureConnection* | details |
| *getRoomsConnection* | *no extra* |
| *getWeekConnection* | *no extra* |
| *addRoomConnection* | room |
| *deleteRoomConnection* | room |
| *deleteBuildingConnection* | buildingName |
| *getAvailableRoomsConnection* | details |

After getting the related extra from the received JSON, this extra is sent to the corresponding function in the Database Access class. The return value of the functions of Database Access class are always JSON objects, and all of them has a field called *success.* This result is then serialized and sent back to the Android client.

The details of database access functions are explained in the following sections.

## 3.3 Database Structures

### 3.3.1 Users Database

The users database is kept in a file called *users.dat.* This file contains one JSONObject. When there is a database access to users, this object is read.

Users database object contains key-value sets. The keys of the object are the usernames. This approach also helps to keep usernames unique. The values of these key usernames are again JSON objects, containing details about that user.

The JSON object which contains details about the user has 2 fields: one is the *password* and the other is the *userType.* Password is needed for the login verification and user type determines which page the user will see after they login to the Android application.

### 3.3.2 Rooms Database

The rooms database is kept in a file called *rooms.dat.* All of the building, room and capacity information is kept here. This file contains one JSONObject. When there is a database access to rooms, this object is read.

The rooms JSON object has key-value pairs, similar to users database. There, the key was the username, here, the key is the *buildingName.* This way, we make sure that there are no two buildings in the same name.

The building name corresponds to a JSON object, which has its own key-value pair set. This second level object has the *roomName* as its keys; therefore we know that this building contains these rooms.

And the values of the room name keys are again, JSON objects. These objects contain information about that room. Currently, the only key-value pair here is the *capacity* field and its integer value. But this is open for improvement.

### 3.3.3 Week Database

The week database is also a JSON object, kept in a file named *week.dat.* Unlike the others, this object contains a fixed number of fields inside it, which are the days of the week.

Every day of the week has one JSON object to hold information about that day's schedule. As the first object divides the week into 7 days, this second level object divides the day into hours, starting from 09 until 18. If there is nothing scheduled for a given hour, then that key does not exist in the day object.

The hours also have their own JSON objects. These objects are now specific to that day and hour. They contain the building names as their keys, and in their value sets, they contain the <*roomName*, *lectureName*> pairs.

This way, we know that on which day, at which hour, in which building, which classroom is occupied with which lecture.

## 3.4 Database Handling

### 3.4.1 Database Helper

*DatabaseHelper* class contains two mandatory functions: reading from database and writing to database. The *getDatabase* function takes in the database file's path as string and returns the JSON inside it.

The *updateDatabase* method works similarly. It takes in the path of the database and also a JSON object, which is expected to be the newer version of the database. Then it writes that JSON to a temporary file in the same path. This is to make sure that we do not lose any data because of an error. When the file is successfully created, it renames the temporary file to the actual database name; overwriting the existing database file.

### 3.4.2 Database Access

The *DatabaseAccess* class has one function for each connection type. All of the functions use a Lock for the databases they access. This section will walk through each connection's handler function.

**Login Connection** This type of connection's handler is the *logUserIn* function. This function takes in a JSON object. It expects to find the *username* and *password* fields inside. Then it gets the users database, checks if the user exists and if so, checks if the passwords match. If the answer is yes, it returns a JSON with success value true and along with it the *userType* of that user.

**Sign Up Connection** This connection's handler is the *signUserUp* function. It takes in a JSON object, expects to find *username*, *password* and *userType* inside. Then it checks if the user exists already. If not, it adds the user with its password and user type to the database and returns a JSON with success.

**Get Rooms Connection** This connection is handled by *getRooms* function. The get rooms function does not expect any input, since its only purpose is to return the current database. It reads the rooms database and returns it in a JSON, along with a variable success added to the result object.

**Get Week Connection** This connection is also very straightforward. It uses the *getWeek* function with no parameters. The week database is read, written into a JSON object, added the success field to that JSON object and the result is sent back.

**Delete Building Connection** This connection is handled by *deleteBuilding* function. It expects a string *buildingName* as input parameter. If rooms database contains that building, it removes that building from the database.

**Add Building Connection** Add building connection is handled by *addBuilding* function. This function expects a JSON object with a *buildingName* variable inside. It then checks the rooms database for that building. If it does not already exist, it adds the building to the database and returns successful JSON result.

**Delete Room Connection** This connection is handled by the *deleteRoom* function. It expect the *buildingName* and *roomName* to be contained in the parameter JSON. Then it does two checks; first it checks if the building exists, second it checks if the room exists. If both of these statements are true, it deletes the room from the database and sends back a success result.

**Add Room Connection** Add room connection is handled by *addRoom* function. The function expects a JSON object as input. This object should have the *buildingName*, *roomName* and *capacity* in it. Then comes the checks; first the building is checked because a room cannot be added to a non-existing building. Secondly, the room is checked, rooms have to have unique names within buildings. If all is fine, database is updated with the new room and success result is sent back.

**Get Available Rooms Connection** This connection is redirected to the *getAvailableRooms* function. This function is longest function among the database access functions. As input, it expects a JSON object, containing several fields.

Initially, this function gets the *capacity* value to be able to filter the rooms. Even if the room is available, capacity may block the process to continue, therefore the first thing to do is to filter the rooms by capacity. The function creates a JSON object called *result* and puts in the buildings and rooms which satisfy the minimum capacity constraint.

Now that all the rooms in the result object is feasible capacity-wise, next thing should be filter the by their availability on the specified times.

The received JSON object is expected to contain a *week* object inside it. This object is structured similarly to the week database. It has the days, inside them the hours, telling the receiver that it requires a room which is available on those day's those hours.

The function iterates over the days and inside days, iterates over the hours. For every <*day*, *hour*> pair, it checks the week database to see if there is any lectures on that time. If there is, it gets the room name of the occupied room, and if it existed in the result JSON object before, it deletes it, since it is not available at one if the requested hours.

In the end, the result object is left with only rooms which have enough capacity and are available for every hour the user specified. This object is sent back as the response.

**Add Lecture Connection** The add lecture connection is redirected to the *addLecture* function. It expects a JSON with the information of the *roomName*, *buildingName*, *lectureName* and a *week* object. The first three of these are straightforward strings, the fourth one is a JSON object. It contains days as keys and JSON objects containing the hours occupied in that day as the values.

The lecture is added to the week database by getting the specified days and hours, adding the lecture name and updating the database. The response is a success JSON object.

## 4 Conclusion

This project helped me learn how to work with multiple clients at the same time. I didn't have the luxury of handling one client at a time, and I have learned that using socket connections in new threads solves this problem nicely. Also, protecting the database from simultaneous access was important, because several threads can run at the same time with this approach and they may try to write to the same database at the same time. Therefore, I have added locks to the database access logics.

Overall, this was a great project to learn a lot from. It has parts that can be improved, but there is always a better and more featured version of every project, therefore I stayed with the initial functionality and I tried to do my best for the base version of the project.