

Bookify

Bookify


REGISTERLOGIN

09/20/2023 to 09/20/2023


1 person

Search


Top rated




SYNTAGMA;GREAT;WIFI CheckIN24Hour
★★★★★(3)
2 🛏 Entire Home
Total Price: \$231.00




Plaka art house
★★★★★(3)
2 🛏 Entire Home
Total Price: \$270.00




Walk to Plaka, terrace,garden, WiFi
★★★★★(3)
1 🛏 Entire Home
Total Price: \$240.00




Acropolis cosy 1bedrm apt wi-fi, AC
★★★★★(2)
1 🛏 Entire Home
Total Price: \$240.00




Apt next to Plaka, balcony, WiFi
★★★★★(2)
1 🛏 Private Room
Total Price: \$300.00




Cosy neighborhood, big apartment!
★★★★★(2)
1 🛏 Entire Home
Total Price: \$150.00



Uber Modern Comfy Apartment-Athens
★★★★★(2)
1 🛏 Entire Home
Total Price: \$210.00



Charming Suite in Historic Athens
★★★★★(2)
1 🛏 Entire Home
Total Price: \$135.00



beautiful & spacious central appt
★★★★★(2)
1 🛏 Private Room
Total Price: \$75.00

Attributions

Templates and icons from Material-UI.
Icons from www.flaticon.com.
Icons from Font Awesome.
Icons from icons8.

Creators

Georgios Alexandros Kostas
Konstantinos Arkoulis

Map data © OpenStreetMap contributors.
Map data © Geoapify.
Geocoding provided by Nominatim.

© 2023 Bookify. All Rights Reserved.

Created By:

Konstantinos Arkoulis (sdi2100008)
Georgios-Alexandros Kostas (sdi2100080)

For the purposes of Online Application Technologies Class

University of Athens, 2022-23

Contents

Introduction	4
Installation Guide	4
Run in the Cloud.....	5
Run locally from pre-built files	5
Build and Run Locally	6
Run Setup Scripts	7
Build the Website	8
Application Features	9
Architecture Breakdown	15
Security	15
Authentication	16
Authorization	16
HTTPS/TLS	18
Database	18
Backend/Server	19
General Architecture	19
Registration -Login	20
Users	21
Rooms	22
Booking and Availability	23
Search.....	24
Reviews	25
Recommendation System	26
Administrator	28
Images	29
Messages.....	30
Utilities and Configuration	31
Frontend/Website	32
General Structure.....	32
Security and Authentication.....	33

Other Contexts	34
Setup Scripts	34
Conclusion.....	35
Attributions	36
API Reference	36

Introduction

In this assignment we created an online booking application similar to Airbnb or Booking.com. Our app, called Bookify, provides most of the features one would expect to see in an application of this type including but not limited to user registration and profile, profile pictures, an administrator page, room creation page, a page to view all the room details (description, map, photos), reviews, an availability and booking system and a search page that allows various filters and supports pagination. We implemented all the requirements of the assignment's specification along with some extra features we considered useful.

The application is developed using the client-server model. The server was created using Spring Boot and offers a RESTful API that can be used by various client applications such as websites or mobile apps. The website, created using React and JavaScript, functions as the client, providing a nice and intuitive interface for the average user to interact with the app. In this document we will be providing installation and execution instructions as well as explaining the design of our codebase, along with the main design dilemmas that led to our current implementation. Following is a brief rundown of the main sections of this report:

- [Installation Guide](#): all the information you need to know for the various ways to install and run Bookify.
- [Application Features](#): in this section we will be presenting the various capabilities our app offers and how to use them.
- [Architecture Breakdown](#): next up we are breaking down the design of our codebase explaining the high-level structure of both the [server](#) and the [website](#) as well as the [database](#) and the [security](#) methods we used.
- [Setup Scripts](#): in this section we will briefly be examining the scripts we created to populate the database using the provided datasets, what they do and how they work.
- [Conclusion](#): here we will be summing up the process of developing Bookify and the challenges we faced.
- [Attributions](#): we could not end this report without providing proper attribution to the creators of the various open-source libraries, APIs and icons we used.
- [API Reference](#): this chapter includes the documentation of the server's API and the permissions required to access each endpoint.

Installation Guide

The following installation instructions assume a Linux Debian-based operating system. However, installing and running the application in a Windows environment should be a similar process. The main differences concern the installation of various programs which will require you to download and run the installer from the vendor's official website. We will be trying to cover all the information required to install and run the application no matter what operating system you are on. We tested those installation instructions on Ubuntu 22.04. If you are on an older version of Ubuntu, almost all the instructions should still work the same, except it may not be possible to install the required Java Development Kit (JDK) version directly from the package manager. In this case, you may need to install the JDK-18 directly from the vendor's site.

There are various ways to run the application:

1. From the provided link running in our own virtual machine in the cloud.
2. Downloading the pre-built .jar file and running the app from there.
3. Downloading the source code and building it locally.

In the following subsections we will be providing detailed instructions for each of those options. If you are running the server locally, please refer to [this](#) subsection to check how you can use the provided setup scripts to preload some useful data to the database. Finally, section there is also a [subsection](#) on how to build the website yourself, if necessary to do so.

Run in the Cloud

The easiest way to quickly and effortlessly test our application is by using this link: <https://bookify.duckdns.org>. This will be the same as running any other online application from the browser and - latency aside - you will be able to use it as if it was running on your local machine. Please note that the application is running on our own rented virtual machine in the cloud which has very limited resources. Therefore, we can not guarantee uptime or response time when using this method. We also ask that you do not overload the server with a lot of big image files, as storage space is already limited (uploading a few images is fine of course). However, this method should be enough to demonstrate all the functionality required by the assignment.

Run locally from pre-built files

Another option is to use the provided pre-compiled .jar file to run the server locally. However, this option will require significantly more effort on your part as it is necessary to download various components and set up the database. Please note that you will need to have administrator rights to the machine you are using. **Please try to avoid copying and pasting the Linux commands directly on a terminal as the pdf format might affect some of the special characters.**

1. If you are using Linux, make sure the package manager is up to date with this command:
sudo apt update
2. Install the Java Development Kit (JDK) if not already installed. For this you can either use the installer provided in the official website or on Linux you can run the following command:
sudo apt install openjdk-18-jdk

Note: if the package manager is not able to find the requested JDK-18 version, try running this command instead: *sudo apt-get install openjdk-18-jdk*

If that fails too, you will have to manually install JDK-18 from the vendor's website. We can not guarantee that the program can be correctly run and built with an older JDK version, although you might get away with running it using JDK-17.

3. Install MySQL Server either via the official website or with this command if you are on Linux:
sudo apt install mysql-server
4. If on Linux, start up the MySQL Service with the command: *sudo service mysql start*
5. To set up the database you need to open a MySQL console if you are on Windows or execute the following command on Linux: *sudo mysql*. You will then need to execute the following commands (same in both operating systems):
 - a. *create database db_bookify;*

- b. *create user 'admin'@'localhost' identified by '[password]';* where *[password]* will be the password you want to set for the database user.
 - c. *grant all privileges on *.* to 'admin'@'localhost';*
6. You can now close the MySQL console and use the terminal of your operating system to navigate to the build folder of the deliverable we provided. An example, that still requires you to provide the appropriate path, is this command: `cd C:/Users/[USER]/Documents/bookify/build`.
7. After you have **successfully** executed **all the steps** described above you can run the executable using this command:

```
java -jar bookify-0.0.1-SNAPSHOT.jar --upload.directory.root=
C:/Users/[USER]/Documents/bookify --spring.datasource.password=[password]
```

The provided options are crucial to the correct execution of the application. `--upload.directory.root` defines a path to the parent folder the application can use to store the required data such as images, recommendation results etc. If the given path does not exist, the application will create it **if and only if** it has the required permissions to do so. As a result, please make sure you only provide a path the application will actually have access to write to.

The `--spring.datasource.password` option defines the password of the database user as created in step 5b. The app assumes the password '1234' for the database user, so if your password is different than that you will need to set it using this option. Other options that may be useful are

- `--spring.datasource.username` which defines the username of the database user (default: admin).
- `--server.port` which defines the port the server will be listening to (default: 8443).

After successfully executing those instructions, you should have a Spring Boot application up and running in your terminal. It will take between 5-20 seconds to startup. Immediately after, it will start running the recommendation algorithm in the background if any rooms and users are available.

We have set up the server to provide all the pages of the website to the browser when requested. Therefore, it is not required to create any additional static server to serve the website content. The only action that should be required to use the app once the server is running locally is to open your browser and type the following into the address bar: <https://localhost:8443>. If you are running the server on a virtual machine you will need to substitute 'localhost' with the IP address of your VM in the above link.

IMPORTANT: due to our SSL certificate being self-signed most of the browsers will reject those requests. It is therefore necessary to add an exception to the browser for our website. If the browser shows you a message rejecting the request, click on the advanced options button and then click on proceed/add exception. If you do not manage to get the exception done this way, you will need to manually set it via your browser's security settings.

Build and Run Locally

It is also possible to compile and run the code yourself on your local machine. The steps are similar to those of the [previous](#) section, but you will now need to execute the following steps instead of step 6 to build the project locally:

1. Install Maven (the build tool). In Linux this can be done with this command: *sudo apt install maven*.
2. In the terminal of your operating system, navigate to the bookifySystem subfolder in the provided deliverable.
3. Use the following command to build and run the project:

```
mvn spring-boot:run
-Dspring-boot.run.arguments="--upload.directory.root=C:/Users/[USER]/Documents/Bookify
--spring.datasource.password=[password]"
```

You will need to properly set the provided options for the application to run correctly as explained in step 7 of the [previous](#) section. Please note that the first time this command is executed, Maven may need some time to download all the required dependencies and then compile and run the project. When it is done you should have a Spring Application running in your terminal. Refer to the [previous](#) section for instructions on how to use the app from your browser.

4. Other useful Maven commands include:
 - a. *mvn clean*: cleans up all the build files.
 - b. *mvn package*: packages the application into a .jar file that can be easily distributed, deployed or moved around.

Run Setup Scripts

When running the application locally on your machine for the first time, the database will contain no useful data such as rooms and users (apart from the pre-installed admin user). We have created a python script to fill in the database with reviews, users and rooms from the dataset that came with the assignment.

CAUTION: This script will delete all existing users, rooms, reviews and availability-related information from the database. Make sure to use this script BEFORE you start using the app to avoid loss of data! This script is meant to be used when first setting up the application. If used later on, it is possible it may corrupt the database.

To use this script, you will need to follow these instructions:

1. Install Python 3 via the official website or using this Linux command: *sudo apt install python3-pip*. Make sure that Python is also added to PATH if you are using Windows.
2. Install the required package to allow python to connect to the database with this command:
pip install mysql-connector-python
3. Open the populateDatabase.py script (in the *scripts* folder of the deliverable) in any text editor and scroll down to the main function. At the start of this function, you will need to edit the database credentials to match those you set up when creating the database. The default host is "localhost", the default username "admin", default password is "1234" and the database username is "db_bookify". If any of these are different to your settings, you will need to change them in the script and save any modifications before moving on to the next step.
4. Run the server application locally at least once following the instructions in the previous sections so that the spring application will create all the necessary tables in the database. Wait for the startup process to finish before you run the script. The tables are created the first time the application runs, so if you have already started the server, you can skip this step.

5. In the terminal of your operating system navigate to the *scripts* subfolder containing the python script along with the required datasets. Execute the script with the following command:
python populateDatabase.py

Depending on the python version you installed, you may need to start the command with *python3* instead of *python*.

Depending on how fast your machine is, this script should need a couple of minutes to load all the necessary data. When it is done, you will find that the website now includes about 2500 rooms and 32000 reviews associated with them.

Build the Website

In all the install/run options mentioned above, there is no need to build the front-end yourself. The website is already built directly into the backend's *resources* folder so that it can be served automatically by the backend itself, without the need to set up a dedicated server. If you, however, wish to build the frontend from the source code you can follow these instructions:

- Install nodejs directly from the website or using the following command in Linux:
sudo apt install nodejs
- Install Node Package Manager (npm). Can be installed with this Linux command:
sudo apt install npm
- Navigate to the *bookifyfrontend* folder of the deliverable in the terminal of your operating system.
- Install the necessary npm dependencies with this command: *npm install --force*. This command may take some time to complete.
- You can now setup a local development server using *npm run* or build the project with *npm run build*. If you choose to build the project, please be aware that the build system is configured to build the project directly into the *resources* folder of the backend. You can change this by editing the *.env* file in the *bookifyfrontend* folder.

Application Features

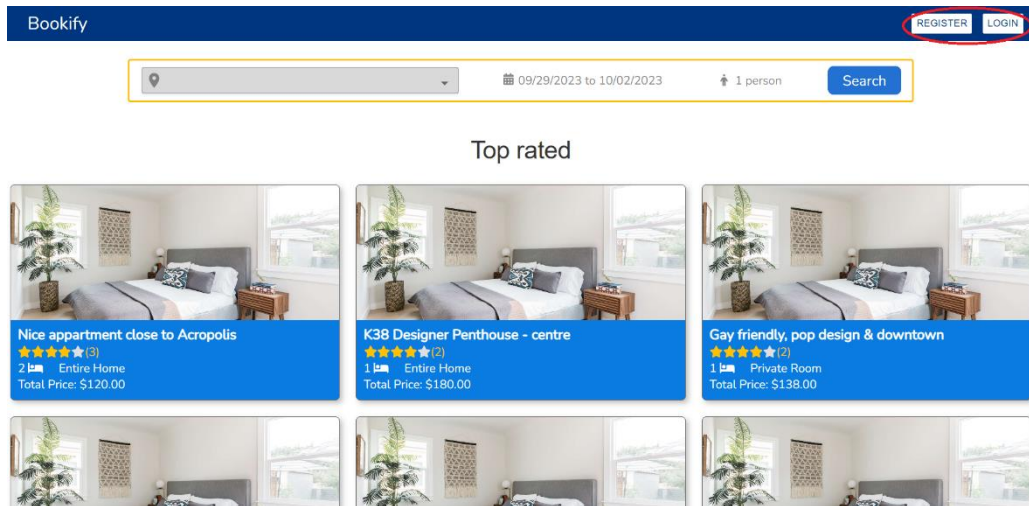


Figure 1: Welcoming page, user Registration/Login

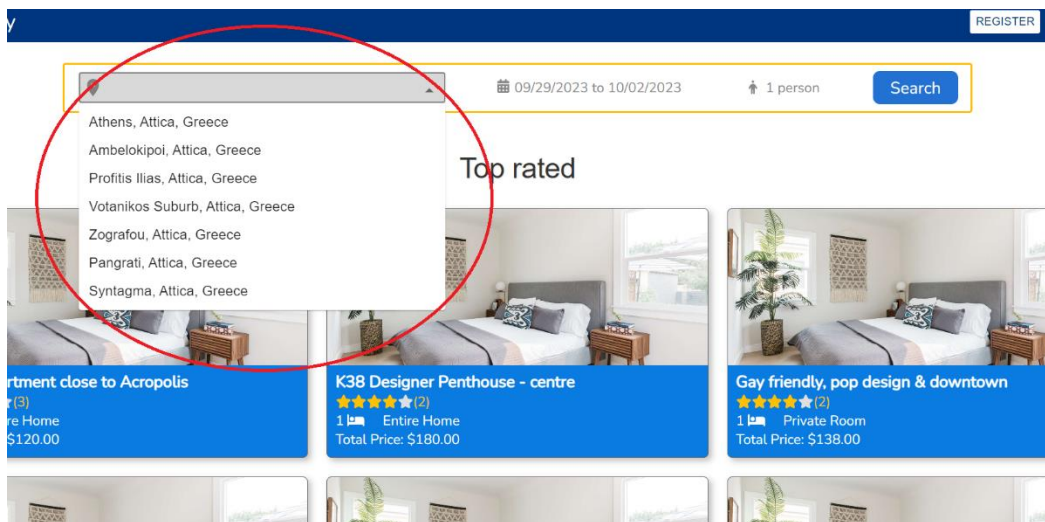


Figure 2: Searching locations with autocomplete

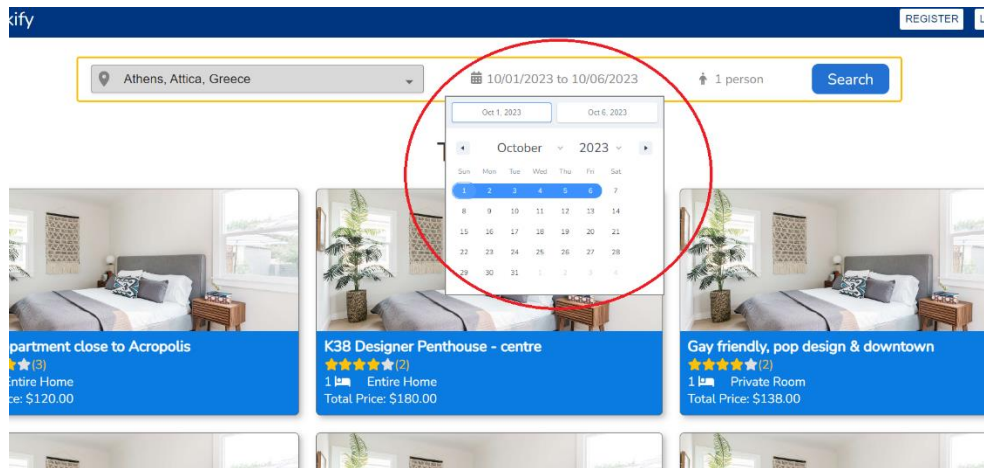


Figure 3: Choosing dates

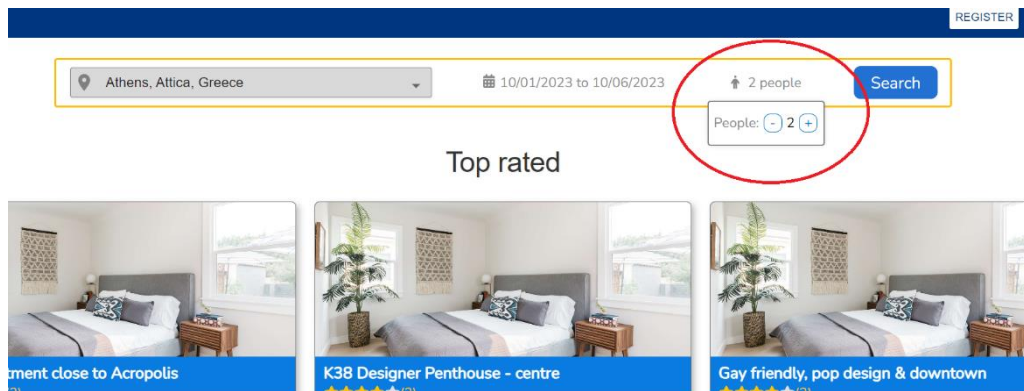


Figure 4: Clicking search and.... Voilà

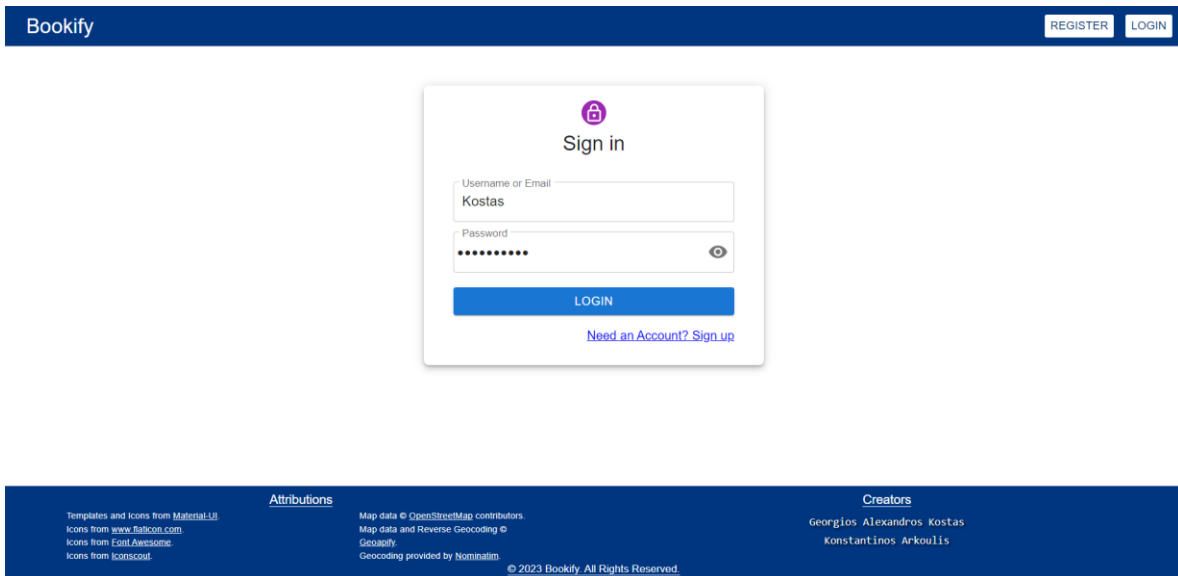


Figure 7: User login

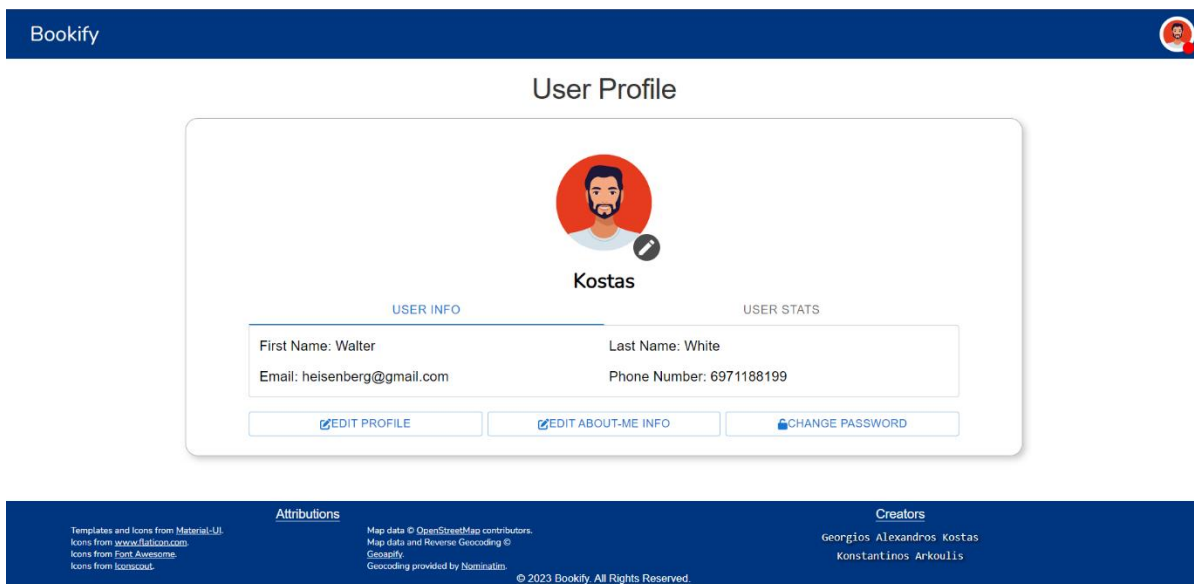


Figure 8: User's Profile

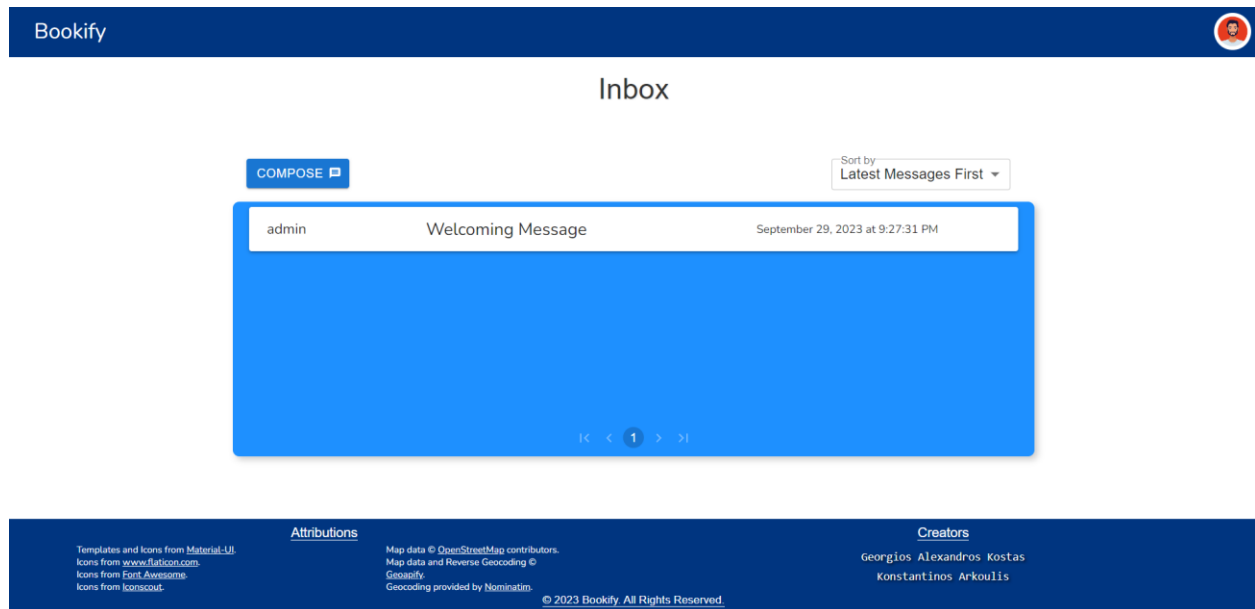


Figure 9: User's inbox

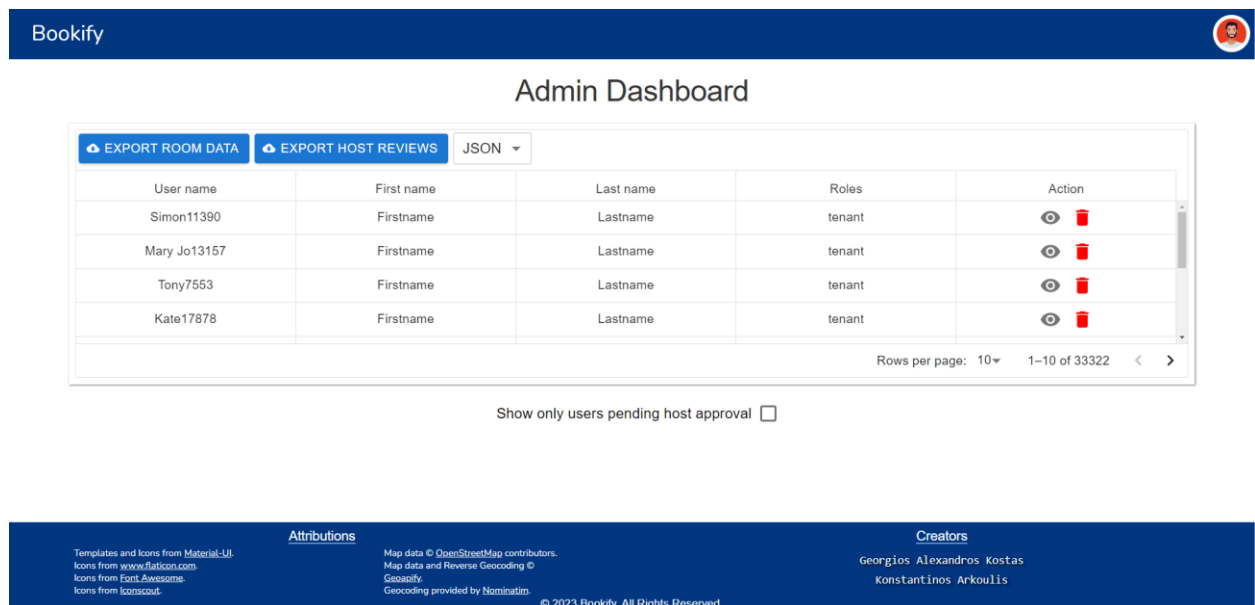


Figure 10: Admin's Dashboard,

Admin's credentials are: username: "admin", password: "1234"

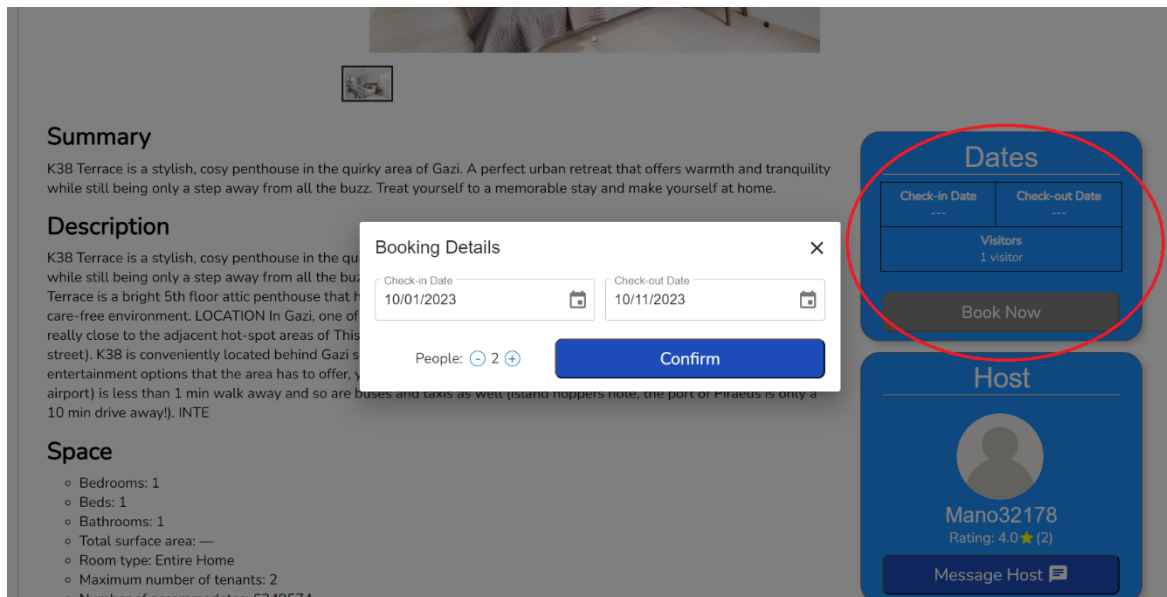


Figure 11: Room booking

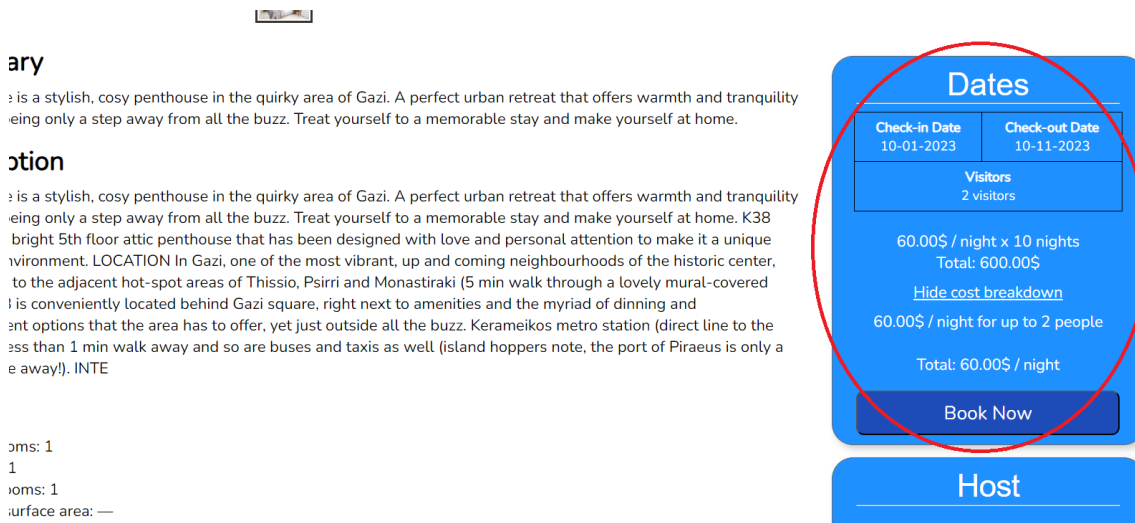


Figure 12: Total cost calculation

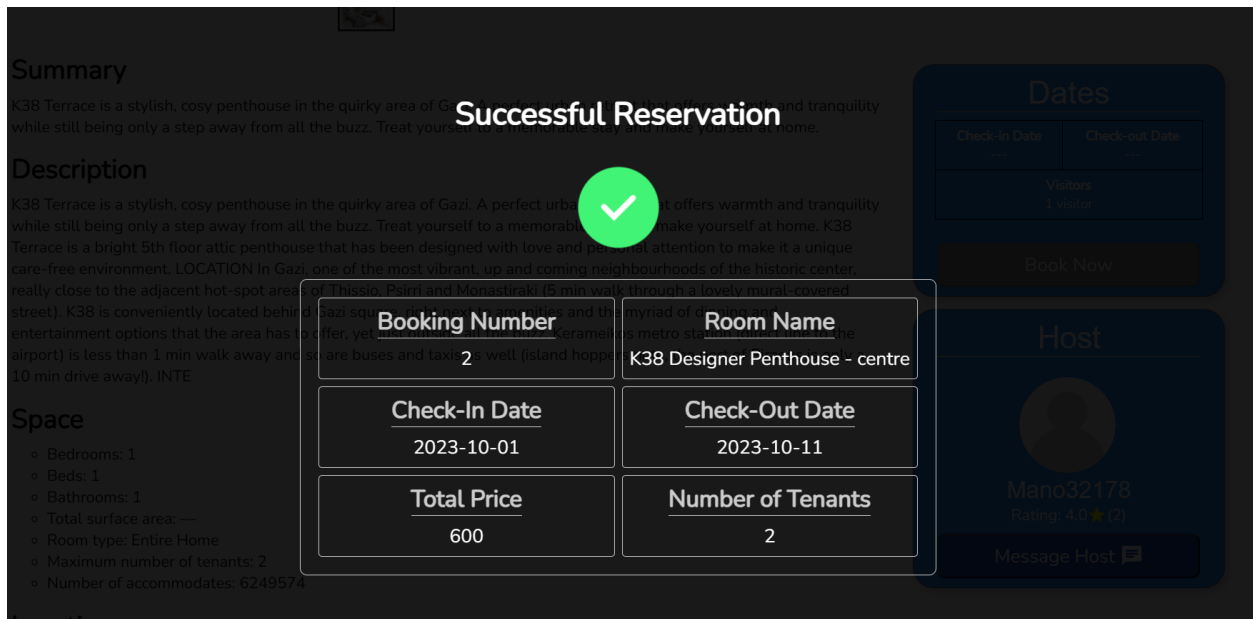


Figure 13: Clicking "Book Now" and... we are ready to go!

Architecture Breakdown

Security

As with any online application, security is one of the most fundamental requirements. For the purposes of this assignment, we will mainly be looking at two main types of security considerations:

- **Authentication:** the process and mechanisms required to verify who a user making a request is.
- **Authorization:** the process and mechanisms required to control what resources and operations a given user has access to.

Before looking into those two sections in more detail, we need to highlight that security and access control are implemented in both the server and the website. The server will not allow unauthorized users to access protected endpoints and will also perform all the required security and sanity checks (that we could think of!) before carrying out any operation. At the same time, the website does not allow unauthorized users to view privileged pages, such as the admin dashboard, or perform unauthorized requests via the website's

interface. For example, the Review and Book buttons are disabled if the user is not authenticated or does not have the tenant role or if the room is already booked for the given dates. Even if the user somehow manages to bypass those protections, the server will still reject the request thus guaranteeing the integrity of the application.

Authentication

As required by the assignment, the authentication in this application is performed using a mechanism called [JSON Web Token \(JWT\)](#). Using this mechanism, whenever a user signs up or logs into our application the server can create and pass back to the user a String, called token, that encapsulates all the authentication-related information that concern that given user. Such information can include username, roles, token expiration date etc. The user can store this token and pass it along with any subsequent requests. The server can then verify the validity of this token and figure out who that user is and what resources they can access.

For security reasons this token is short lived with an expiration duration of about 3 minutes. This way, if the JWT is compromised it will expire very soon, thus limiting the damage. In this context the JWT is called an access token. When the access token expires, the server will reject any request containing that token as unauthenticated. When that happens, the client needs to ask the server to refresh their access token by hitting a special endpoint created for this purpose.

To issue a new access token, the server needs to verify that the user requesting the refresh is who they say they are. This is done by sending along with the refresh request another kind of token, called a refresh token. The refresh token is created on application signup and is related only to one given user. The user gets that refresh token along with their access token when they log in or register. The server can capture the refresh token from the request and check it against the user's token from the database to issue a new access token if they match.

To implement all this behavior in our server, we relied heavily on Spring Boot's Spring Security Framework. The *authentication* package and the *SecurityConfiguration* class in our source code contain code related to the implementation of this functionality. Of particular note in this section are the following classes:

- *TokenService*: encapsulates the logic related to creating new JWT Tokens. The claims of our JWTs include issue datetime, expiration date, subject (concerned user), expiration date and roles that are used to decide permissions (such as admin, tenant etc.).
- *SecurityConfiguration*: contains Spring beans that handle authentication related components such as AuthenticationManagers and JWT or password encoders and converters.

At this point, we need to note that the bulk of the authentication related work is done by the Spring Security framework, and we only need some code to configure those components to match the design we described.

Authorization

With the authentication part of our server being taken care of we now need to worry about the authorization part and make sure each user can only access resources and API endpoints they have permission to use. This is done in 3 layers, each of them specifying endpoint permissions at different levels of granularity:

1. **SecurityFilterChain**: the filter chain is a part of the Spring Security framework that allows us to define a set of security filters for any incoming request. This is a very powerful tool, but we used

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
    http.csrf(csrf -> csrf.disable());
    authorizeHttpRequests(auth -> {
        auth.requestMatchers(new AntPathRequestMatcher("/{**}", HttpMethod.OPTIONS.name())).permitAll();
        auth.requestMatchers("/api/upload/**").permitAll();
        auth.requestMatchers("/api/registration/**").permitAll();
        auth.requestMatchers("/api/room/**").permitAll();
        auth.requestMatchers("/api/amenities/**").permitAll();
        auth.requestMatchers("/api/roomPhotos/**").permitAll();
        auth.requestMatchers("/api/search/**").permitAll();
        auth.requestMatchers("/api/reviews/**").permitAll();
        auth.requestMatchers("/api/book/**").permitAll();
        auth.requestMatchers("/api/roomType/**").permitAll();
        auth.requestMatchers("/api/admin/**").hasRole(Constants.ADMIN_ROLE);
        auth.requestMatchers("/api/messages").authenticated();
        auth.requestMatchers("/api/user/**").permitAll();
        auth.requestMatchers("/api/recommendation/**").permitAll();
        auth.anyRequest().permitAll();
    });
    oauth2ResourceServer().jwt().jwtAuthenticationConverter(jwtAuthenticationConverter());
    http.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
    return http.build();
}
```

Figure 15: The Security Filter Chain of the Bookify server

only a small subset to allow us to lock out a whole category of API endpoints to the users that do not have permission to see them. Following is a screenshot of our Security Filter Chain defined in the *SecurityConfiguration* class. It clearly shows how it can be used to permit only users with the appropriate role to access endpoint classes such as `/api/admin`, which is only accessible to users with the admin role. It is also important to note that, unless defined differently, the default endpoint behavior is to allow all incoming requests. This is done to allow users to access static resources served by the backend, such as the different website pages.

2. Because most of the endpoint classes contain endpoints that differ from each other as to which roles have permission to access them, we need a way to define permissions at the endpoint level. This is done using Method-Level security and `@PreAuthorize` annotations. The following screenshot shows those annotations in action. It allows the `deleteRoom` endpoint to be accessed only by users with the admin or host role.

```
@DeleteMapping("/{roomID}")
@PreAuthorize("hasRole('admin') or hasRole('host')")
public ResponseEntity<?> deleteRoom(@PathVariable Integer roomID){
    try {
        roomService.deleteRoom(roomID);
        return ResponseEntity.ok("Room " + roomID + " deleted successfully");
    }
}
```

Figure 16: `@PreAuthorize` annotations in action

3. Sometimes determining permissions is not as easy as checking whether a user has a specific role. An example where that might be the case is replying to a message in a given conversation of the in-app messaging system. Without proper checks, if a bad actor was to somehow get access to the entity ID of a conversation, they could hit the corresponding endpoint with that ID and get access to all the messages of this conversation and even post replies to it! There are countless similar edge cases in our backend that need careful consideration to guarantee adequate security. The way to handle those edge cases is to manually perform any required logic and sanity checks and return a 403 Forbidden HTTP response if the action to be performed is not allowed. Here is an

example of this being done programmatically, to verify a user has privileges to perform read, update or delete operations in a conversation:

```
private void verifyConversationPrivileges(Conversation conversation, User user) throws IllegalAccessException {  
    if(!conversation.userBelongsToConversation(user))  
        throw new IllegalAccessException("User " + user.getUsername() + " does not belong " +  
            "to conversation with id " + conversation.getConversationID());  
}
```

Figure 17: Verifying that a user has the required privileges to access, update or delete a conversation

As we can see, there is no one way to guarantee correct authorization to all the endpoints in our app. Most of the endpoints make use of a variation of all those 3 layers described above. It was important to carefully consider and handle all the edge cases we could think of when creating a new endpoint. Some of them are not even strictly authentication related, such as not allowing a user to review or book their own room. We will be trying to provide a detailed rundown of all those authorization checks in the [API Reference](#) section of this document.

HTTPS/TLS

As per the specification of the assignment, all requests coming to and from the backend are encrypted using the SSL/TLS protocol. This means that if a request was somehow stolen, sensitive data such as passwords or private messages would never be exposed as they are securely encrypted. Configuring Spring Boot to use TLS was a matter of adding a few attributes in the application.properties file after creating a self-signed certificate (stored in `/bookifySystem/src/main/resources/keystore` folder of the deliverable). The problem with this certificate is that most browsers will initially reject it, requiring the user to add an exception to the browser. For the purposes of this assignment, however, we judged that to be an acceptable compromise.

Database

The database is an integral part of any application of this size. One of the first questions we ran into was whether to use a relational or non-relational database. Upon careful consideration, we decided that the data and entities we would need to create contain a lot of relationships between each other. Therefore, in our judgment, a relational database was the best way to represent our application's data. We decided to use MySQL for our database, as it is a very mature and well documented database management system that we were already familiar with from previous courses.

The database **is not separately designed** in another application such as MySQL Workbench. Instead, it is created automatically based on our classes using the Hibernate Object Relational Mapping (ORM) framework, which ensures the application classes remain consistent with the tables and entries in the database. We did not hesitate to create a lot of different entity types whenever we decided it was necessary. This provided us with a more consistent and extensible way to represent our data. For example, apart from the usual entities one might expect, such as rooms, users, reviews etc., we also have tables to represent room amenities or room types. This allows us to easily add new types of amenities or rooms and makes searching by them easier and less error prone compared to, for example, using a string. It also helps with reducing repetition of data in our database.

We will not go into any more detail on the design of this database in this section. We will be trying to cover the major database decisions for each subsystem into its own corresponding subsection under

[Backend/Server](#). Following is a schema of our database, imported in MySQL workbench for **demonstration purposes only**:

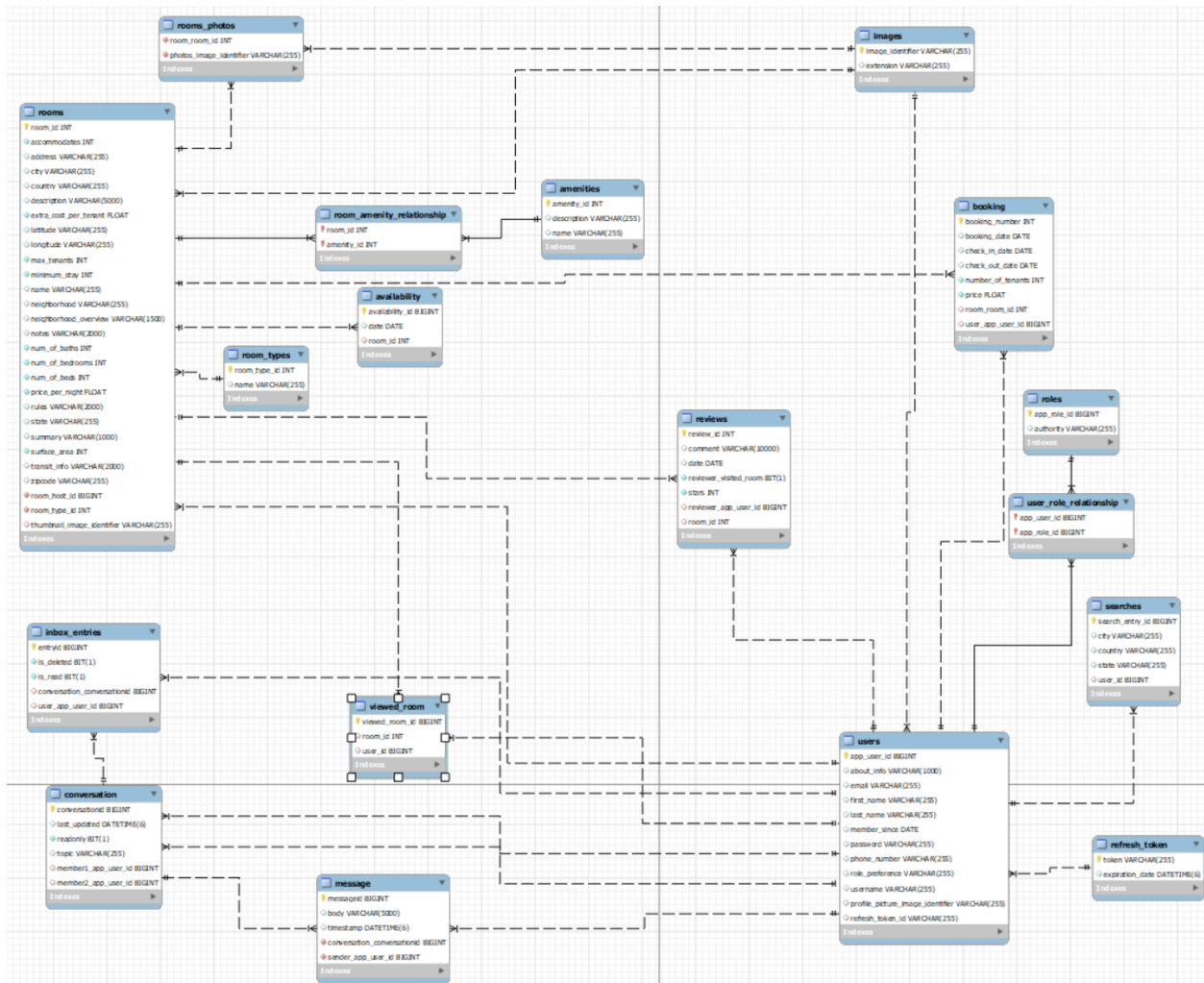


Figure 18: The database design of Bookify

Backend/Server

We decided to use Spring Boot as the framework for the development of the server, as it is well-documented, was recommended by the professor and we were already familiar with Java. In the following subsections we will be briefly presenting the main subsystems of the backend part of our application, along with the main design considerations that led us to those specific implementations. We will also be exploring the limitations of various components of the backend.

General Architecture

The backend code can be found in the `/bookifySystem` subfolder of the deliverable and is organized into packages. Each main subsystem or feature is contained in its own package, so that all the classes and code related to a given feature are close together and logically organized. There are 4 main types of classes that can be found in our codebase:

- **Data Classes and DTOs:** these classes define various entities for our database that are automatically converted to database tables and relations using the Hibernate Object Relational Mapping (ORM) Framework. They may also define classes called Data Transfer Objects (DTOs) that are used to conveniently pass a lot of data (such as the registration information of the user) around or capture this data from incoming HTTP requests.
- **Repositories:** our repositories are extensions of the classes provided by Hibernate and are used to provide Create, Read, Update and Delete (CRUD) operations on the database tables to the rest of the program, thus removing the need to write SQL calls for most of the required operations. However, they can easily be extended to run custom queries in the database in an SQL-like language (HQL) for more complex operations. Each database entity needs to have its own repository associated with it.
- **Services:** these are the classes that handle the business logic of the application. They are responsible for carrying out all the steps required to perform a given operation. Such steps include, but are not limited to, doing necessary checks on the data, asking the repositories to perform some subset of the CRUD operations and handling different error cases.
- **Controllers:** these classes provide access to the application's API. They are responsible for defining the various API endpoints, catching incoming requests, passing the request for a given operation down to the appropriate service and returning the result in the form of an HTTP message. They also catch various exceptions thrown by the services and return the appropriate HTTP response code for each. Essentially the controllers define the interface of our app to the outside world.

We make use of Spring Boot's automatic dependency injection system to make the setup of our application as effortless as possible. The process of handling any given request can briefly be described as follows: the controller corresponding to the endpoint we hit catches the incoming HTTP requests along with any query parameters, path variables or body arguments. It passes all the necessary information down to the corresponding service responsible for handling this operation. The service will make all the required checks to ensure the operation can be performed and that the user has the required access to perform said operation. It then performs all the required logic, sometimes making use of operations provided by other services, communicates with the corresponding repositories for any database-related operations, checks for errors and passes the response back to the controller. The controller then encapsulates and sends the response back in an appropriate HTTP message.

Registration-Login

Two of the most commonly used actions in Bookify are registering or logging in to the website. This is implemented by the code in the *registration* package and especially the *RegistrationService*. The following actions are provided by this service (make sure you give the [Security](#) section a read first):

- **Registration:** this function needs to communicate with the UserService (described in more detail in [this](#) section) to create a new User in the database with the specified attributes. It also needs to generate a refresh and access token that will be returned in the registration response. This response is the same as the login response to allow the user to be immediately signed into the website upon registration.
- **Login:** the user can log in using either their username or email as well as their password. The *RegistrationService* needs to ensure the provided credentials are valid and then issue a new access token and pull the refresh token from the database to generate the log in response.

- Refresh: the refresh function is an integral part of the security of our application. It accepts the refresh token of a user, checks it against the refresh token stored in the database and if they match, the service issues a new access token that is then sent back to the user.
- Logout: when a user logs out of an application their refresh token needs to be invalidated or deleted from the database.

The *RegistrationController* class catches all authentication-related requests and passes them down to the *RegistrationService* that is responsible for handling them.

Users

The *User* is one of the most fundamental entities in our application. It needs to hold all the relevant information such as username, email, first and last name, phone number, hashed password as well as relationships to the refresh token, the profile picture (many-to-one as the default profile picture is used by more than one user) and a user's roles. The roles that our application supports are administrator, tenant, host and inactive host (a host that is still waiting for approval). The roles are defined in a separate database table and are connected to the *User* table with a many-to-many relationship (easy to see why). If not already stored, the roles are preloaded into the database on application startup.

The *UserService*, part of the *user* package, needs to provide quite a few user-related operations to be used by the client or even other parts of the codebase. These operations include but are not limited to:

- Creating a new user: this function needs to ensure the requested username and email are not already in use or reserved for other purposes, verify that the password is long enough and if all sanity checks pass, the service must encode the password, assign to the user the default profile picture, their requested roles (inactive host if they requested to be a host) and then create the corresponding new entry in the database.
- Retrieving the details and stats of a user: requires a query to the database as well as verifying that the requested user actually exists.
- Updating the user profile: this functionality is more complex than it may seem at first. The new email and username need to be checked for uniqueness, the roles need to be correctly updated taking care of edge cases (what happens to a host's rooms when they decide to remove that role), and a new access token needs to be generated since the authentication credentials might have changed.
- Changing password: needs to ensure the old password from the request matches the user's password stored in the database and also ensure the new password is valid before encoding the new password and storing it in the database. A new access token also needs to be issued.
- Deleting a user: because the user is related to other entities, such as messages and reviews, that should not be deleted if the user associated with them leaves, it is not possible to simply remove the corresponding user entry from the database. Instead, we are performing a so called "soft delete". This means that the user is not actually deleted but set to inactive. Logging in and authentication is disabled for that user and the various queries on the *User* table will skip users that are marked as deleted. To make sure a user will not be able to continue authenticating with their existing JWT when they are deleted, we created a custom Spring Security Filter called *UserNotDeletedFilter* in the *authentication* package to reject authenticated requests coming from

any deleted users. This filter is registered into the `SecurityFilterChain` using a custom Bean in the `SecurityConfiguration` class.

We will conclude the review of the `user` package with the `ProfilePictureService` that provides functionality for uploading, retrieving and deleting profile pictures. The bulk of this work is handled by the [Image Subsystem](#), but certain edge cases need to be handled, such as automatically assigning the default profile picture to a user that chose to delete theirs and making sure the previous profile picture is deleted when a new one is uploaded.

Rooms

The room is the most fundamental entity of our application, as to be expected of an Airbnb clone. This entity is defined in the `Room` class. It includes too many attributes to list here, such as the Room ID (primary key), name, description, location fields, space-related information (such as number of rooms and beds) etc. Instead, we are going to focus more on the relationships with other entities, mapped using the Hibernate ORM framework:

- Many-to-many relationship with the `Amenities` table. Defines the amenities a room provides.
- Many-to-one relationship with the `Image` entity for the thumbnail (the default thumbnail can be used by more than one room).
- One-to-many relationship with the `Image` entity for the room photos (each room photo is unique to a given room).
- Many-to-one relationship with the `User` table to define the room's host.
- One-to-many relationship with the `Bookings` table to hold all the reservations for a given room.
- One-to-many relationship with the `Review` entity.

At this point we need to highlight that the room's amenities and type are represented by separate entities, related to each room. This facilitates an easier implementation of the search system and allows for easily extending the app with more amenities or room types. Apart from the room repository, controllers and various DTOs used to conveniently move data around, the `rooms` package also contains these important service classes:

- `RoomService`: handles the creation, editing and deletion of a room, allows the retrieval of room data and provides various useful functions related to rooms such as retrieving pages of rooms from a given host (used to populate the host dashboard on the website).
- `RoomPhotoService`: handles functionality related to the photos and thumbnails of the rooms such as adding, retrieving or deleting a photo and getting a list of all the rooms' photo GUIDs.

Very special care needs to be taken so a bad actor will not be allowed to edit or delete a room that he has no access to (due to being unauthenticated or not being the host of that room). To this end, we use the `RoomAuthenticationUtility` to verify that the current user making the request has the required permissions to perform privileged operations.

Finally, we need to note that when creating a room, the photos are not included in the original request. Instead, the server accepts a room registration request with the required fields to create a room and then the client (the website) needs to add the photos to the newly created room separately. This is also the case when viewing a room. The initial request returns only the various fields and descriptions of the room

as well as a list of the thumbnail and all the photo GUIDs that need to be requested separately by the client.

Booking and Availability

One of the toughest design challenges we faced when creating the backend was the representation of the availability of a room. This functionality is critical to an online booking application and needs to be carefully implemented as other crucial features of the app, such as search and bookings, depend on it. To represent the availability, we created a table of *Availability* entries. Each entry consists of a date and a many-to-one relationship to the room it belongs to. When adding availability to a given room, we simply need to create one entry in this table for each day in the availability range. When a room is booked, we just need to remove the *Availability* entries of that room that correspond to the requested stay. Checking whether a room is available or not in a given date range is also a matter of simply verifying that all the corresponding entries in the *Availability* table for that room exist.

The logic described here is implemented in the *AvailabilityService* which uses the *AvailabilityRepository* to handle more complex database interactions with the *Availability* table. In the following pictures we are showing the code that implements two availability-related functions: checking whether a room is available in the given date range and marking a room as unavailable for that range of dates.

```
2 usages  AlexKostas +1
public boolean isRoomAvailable(int roomID, LocalDate startDate, LocalDate endDate){
    if(roomRepository.findById(roomID).isEmpty())
        throw new EntityNotFoundException("Room with id " + roomID + " not found");

    long daysBetween = Utils.getDaysBetween(startDate, endDate);
    return availabilityRepository.countAvailableDaysBetween(roomID, startDate, endDate) == daysBetween;
}

1 usage  AlexKostas
@Transactional
public void markRoomAsUnavailable(int roomID, LocalDate startDate, LocalDate endDate){
    availabilityRepository.deleteAvailabilitiesForRoomBetweenDates(roomID, startDate, endDate);
}
```

Figure 19: A code snippet of the Availability Service

```
1 usage  AlexKostas
@Query("select count(a) from Availability a " +
        "where a.room.roomID = :roomID " +
        "and a.date >= :startDate and a.date < :endDate")
long countAvailableDaysBetween(int roomID, LocalDate startDate, LocalDate endDate);

1 usage  AlexKostas
@Modifying
@Query("delete from Availability a " +
        "where a.room.roomID = :roomID " +
        "and a.date >= :startDate and a.date < :endDate")
void deleteAvailabilitiesForRoomBetweenDates(int roomID, LocalDate startDate, LocalDate endDate);
```

Figure 20: A code snippet of the Availability Repository

The Availability System we just described makes the implementation of the Booking System a lot easier. A *Booking* is an entity that represents a reservation and holds many-to-one relationships with the

corresponding user and room as well as the check-in/check-out dates, the booking date, the number of tenants that will stay in the room and the final price. The functionality of booking a room is handled by the *BookingService*. The logic required to book a room is trivial: the service needs to perform several sanity and security checks and verify with the *AvailabilityService* that the room is indeed available. If no check fails, the service will ask the *AvailabilityService* to mark the room as unavailable for the requested date range and then create and save the Booking entry with the corresponding information. This information can be passed back to the user to confirm the reservation.

Search

There are two types of search our application needs to support:

- Searching rooms: the normal search functionality a user would expect a booking application to have. The user needs to be able to search for rooms in a given location and also apply filters such as maximum price, amenities and room type.
- Location autocomplete: in order for the location-based search to work appropriately, the location attribute given to the backend needs to have a given format so that it can be compared against the entries in the database. For this reason, we decided to implement an autocomplete feature for the location search bar on the website to guarantee consistent location queries.

Let's dive more into how the autocomplete search feature works. To generate the list of autocomplete suggestions, the server accepts the input of the user and looks for entries in the database that match the already typed input. This is done using the following query in the *RoomRepository*.

```
@Query(nativeQuery = true, value =
    "SELECT DISTINCT r.city, r.state, r.country FROM rooms r " +
    "WHERE LOWER(r.city) LIKE CONCAT('%', LOWER(:input), '%') " +
    "    OR LOWER(r.state) LIKE CONCAT('%', LOWER(:input), '%') " +
    "    OR LOWER(r.country) LIKE CONCAT('%', LOWER(:input), '%') " +
    "LIMIT " + Configuration.MAX_LOCATION_AUTOCOMPLETE_SUGGESTIONS)
List<String[]> findAutocompleteLocationSuggestions(@Param("input") String input);
```

Figure 21: The autocomplete database query

Since the autocomplete suggestions come from the database itself, it is guaranteed that whatever input the user selects will have matching results to return. This also has the side effect of not allowing the user to search for locations there are no rooms for.

Let's now examine more closely the inner workings of the normal room search. The results need to be paginated, so we are making use of Spring's built-in pagination system when querying the database. The *SearchController* needs to accept and pass down to the *SearchService* all the pagination parameters (current page, page size), the requested filters and whether the results need to be ordered in ascending or descending price order. The price is calculated dynamically based on the length of the requested stay and the number of tenants. The *SearchService* then makes use of this long query in the *RoomRepository* to return a page of matching rooms:


```
@Query("SELECT r FROM Room r " +
      "LEFT JOIN r.amenities a " +
      "WHERE ((:filterAmenityCount = 0 OR a IN :filterAmenities) OR a IS NULL) " +
      "AND (r.roomType IN :roomTypeFilter OR :roomTypeFilterCount = 0) " +
      "AND :nights >= r.minimumStay " +
      "AND r.city = :city AND r.state = :state AND r.country = :country " +
      "AND (r.pricePerNight + r.extraCostPerTenant * GREATEST(0, :tenants - r.maxTenants)) * :nights <= :maxPrice " +
      "AND r IN ( " +
      "  SELECT a.room FROM Availability a " +
      "  WHERE a.date >= :startDate AND a.date < :endDate " +
      "  GROUP BY a.room " +
      "  HAVING COUNT(a) = :nights " +
      ") " +
      "GROUP BY r " +
      "HAVING COUNT(a) = :filterAmenityCount OR :filterAmenityCount = 0 " +
      "ORDER BY " +
      "(r.pricePerNight + r.extraCostPerTenant * GREATEST(0, :tenants - r.maxTenants)) * :nights ASC")
```

Figure 22: The query that makes Bookify's search functionality work

This query does the following for each room in the database:

- Checks whether the room provides **all** the requested amenities.
- Verifies that the room matches the requested room type.
- Ensures that the given date range is longer than the minimum stay of the room.
- Verifies that the price (calculated dynamically) is lower than the max price requested by the user.
- Checks that the room is available in the requested date range.
- Sorts the results in ascending or descending price order. Please note that due to JPL not allowing named parameters to be used in the ORDER BY section of the query, we had to create and maintain two, otherwise identical copies of the search query, one for each direction of price ordering.

For each result by the aforementioned query, the server collects the preview info (thumbnail, number of beds, rating etc.) and sends it back to the user.

Reviews

Bookify allows its users to post reviews to various rooms. This is a feature very useful to the tenants that are looking for a room. It also allows for the generation of more accurate [recommendations](#). Reviews can only be posted by logged-in users with the tenant role. However, any user - authenticated or not - is able to see them. For demonstration purposes we decided to allow the tenants to review rooms they have not booked before. A green verification checkmark is used on the website to indicate that the review is done by a user that has previously stayed in the room at the time their review was last edited.

The *Review* class is fairly simple. It contains an ID (primary key), a rating between 1-5 and a comment made by the user. It also contains a relationship to the user that created it and to the room it refers to. We are also storing whether the review is made by a user that has previously booked that room. The rating of the room is for now calculated simply by averaging out the rating of all the reviews for that room. We considered giving greater weight to more recent or verified reviews, but we ran out of time to implement such features.

The *ReviewService* handles the logic for storing, retrieving and editing reviews, verifying the user has the privileges required to perform actions on existing reviews and handling special error cases such as trying

to edit a review that does not exist or has been deleted. The *ReviewRepository* provides useful queries on the *Reviews* table such as calculating the average rating of a host and counting the reviews of a given host (used in the profile stats page of the website):

```
@Query("SELECT AVG(r.stars) FROM Review r " +
      "INNER JOIN r.room rm " +
      "INNER JOIN rm.roomHost h " +
      "WHERE h =:host")
Double calculateAverageStarsByHost(User host);

1 usage  arkou
@Query("SELECT COUNT(r) FROM Review r " +
      "INNER JOIN r.room rm " +
      "INNER JOIN rm.roomHost h " +
      "WHERE h =:host")
int countByHost(User host);
```

Figure 23: Examples of some useful custom queries in the *ReviewRepository*

Recommendation System

The recommendation system, contained in the *recommendation* package, is responsible for providing a predefined number of rooms that may be relevant or of interest to the user. As required by the specification, the recommender is based on the [Matrix Factorization](#) algorithm, implemented using [Gradient Descent](#). Before diving deeper into the implementation details of the recommendation system it is worth examining closely the following problems:

- Providing recommendations to unauthenticated users: to provide recommendations to users that have not signed up or logged in to the app we would need to find a way to track their behavior while using our website. Although this is technically possible using cookies, the privacy concerns and the implementation complexity made us decide against pursuing such an option. Instead, we decided to provide the rooms with the highest rating (tie-break using number of reviews) as recommendations to the unauthenticated user.
- Can the recommendation algorithm run quickly enough to provide suggested rooms with an acceptable amount of delay to the user? The simple answer to this question is no. No matter how much we optimize the code or tinker with the learning parameters, there is no way the gradient descent part of the algorithm will run in an acceptable amount of time for a sufficiently large database (about 2-3 seconds for the dataset provided by the professor, ignoring any other delays).

To solve the second of those problems we decided to run the factorization part of the algorithm, which makes up the bulk of the work, periodically as a background task. For demonstration purposes we have set up our Spring Backend to call the matrix factorization routine every 30 minutes and do 500 iterations each time, which should take about 3 minutes to complete.

When the algorithm is done, it saves the user and item matrices, along with 2 dictionaries that map user and room IDs to their corresponding row/column in the matrix, on disk for later use. Whenever the server needs to suggest rooms for a given user, it loads up the latest algorithm results from disk and runs the necessary calculations to return the most relevant rooms. This operation can be performed fast enough

for the user to not notice any significant delay, with the small penalty of the recommendations being slightly out of date. However, there are a few edge cases we need to consider:

- The user has signed up to the application after the latest recommendation run.
- The recommendation algorithm, that is also executed on startup, is not yet finished.
- Some I/O operation fails when trying to load the algorithm results from disk.

Our solution to those problems was to simply return the top-rated rooms when something goes wrong. This way the user will never be left without any room to check out on the home page. The query that retrieves the best-rated rooms is contained in the *RoomRepository* and is shown below:

```
@Query("SELECT r FROM Room r " +
        "LEFT JOIN r.reviews re " +
        "GROUP BY r.roomID " +
        "ORDER BY AVG(re.stars) DESC, COUNT(*) DESC LIMIT 20")
List<Room> findBestRooms();
```

Figure 24: The query for retrieving the rooms with the best rating

For shortness' sake we will not cover the details of the matrix factorization algorithm here. [This](#) is a very helpful article that describes the method and the math behind the algorithm in more detail. The main classes that handle the recommendation part of the backend are the following:

- *RecommendationService*: provides the main logic for generating the suggested rooms for the various cases described above and periodically runs the recommendation algorithm in the background.
- *MatrixFactorizer*: this is the class that performs the actual computations needed to factorize the initial rating matrix. Having this in a separate class allows us to abstract away from the computation-heavy factorizing code and to focus on the business logic of the recommendation generation. This class also includes the number of latent features (K) and the various hyperparameters. We ended up with those values after a lot of experimentation and measurements.
- *MatrixUtility*: contains utility functions related to various matrix operations such as dot product, randomization of an initial matrix and printing a matrix on the screen for testing purposes.

The last important aspect of the recommender that we need to examine is the generation of the initial rating matrix that is later factorized into the user and room utility matrices. Relying on only a user's reviews or reservations is not a viable strategy as the initial matrix would be too sparse to get results of any value. Therefore, we make use of a few different datapoints, each weighted appropriately according to their importance:

- **Bookings**: if a user has booked a room, it is pretty much sure that the user would be interested in such a listing. It is also easy to retrieve all the bookings with a simple query to the database. Each reservation is related to a room and a user so that gives us all the information we need.
- **Reviews**: a user's review of a room is a very accurate representation of how much they liked it. Those datapoints are also easy to get as the Reviews table is related to the user and the room.

- Viewed Room: if the user has clicked on and viewed the details of a room, it is likely they are interested in it and possibly like it. However, this is not as clear cut as the previous two cases, so it will receive a lower weight. To get the necessary data for this datapoint we need to keep track of all the rooms a user has viewed. This is done in a separate table contained in the *rooms_viewed* package. We created the following query in the corresponding repository to count how many times a user has viewed each room:

```
@Query("SELECT NEW com.bookify.rooms_viewed.ViewedRoomDTO(v.user.userID, v.room.roomID, COUNT(*)) " +
      "FROM ViewedRoom v " +
      "GROUP BY v.room.roomID, v.user.userID ")
List<ViewedRoomDTO> getViewedRoomPairs();
```

Figure 25: The query that counts how many times a user has viewed a given room

- Searches in the same location: A user might be interested in a room if they have previously made a search for the same location. However, this is far from certain, therefore, this data point will receive the lowest weight. Similarly, to the previous case, we need to keep track of all the searches made by the authenticated users. This is done in *SearchEntry* table and the number of relevant searches for a given user and room are retrieved using the following query:

```
@Query("SELECT NEW com.bookify.search.SearchEntryDTO(s.user.userID, r.roomID, COUNT(*)) " +
      "FROM SearchEntry s, Room r " +
      "WHERE s.city = r.city AND s.state = r.state AND s.country = r.country " +
      "GROUP BY s.user.userID, r.roomID ")
List<SearchEntryDTO> getSearches();
```

Figure 26: The query that counts the number of relevant searches made by a user for a given room

Having defined the datapoints we are going to need, the creation of the rating matrix is now a matter of retrieving all this information from the database, weighing them appropriately and populating the matrix accordingly. Any user-room entry that has no datapoints related to it will be initialized to zero.

Administrator

One of the requirements of this assignment was to implement an administration page to allow the admin to examine the details of a user, approve or reject their host application or remove them from the app. The administrator is also able to export all the room information and the reviews related to each host in both XML and JSON format. To use this page, one would need to log in using the administrator credentials (username: admin, password: 1234). The website should automatically redirect the user to the administrator dashboard, which is also accessible via the dropdown menu of the website.

The *AdminService* class in the *admin* package handles all this functionality. Approving or removing a host is a matter of deleting the inactive host role and assigning them the host role (if approved). The [messaging](#) system is also instructed to send a message to the user to inform them that their application has been processed. This service also provides the endpoints required to populate the admin's dashboard with paginated users, possibly filtered by having their host application pending.

Creating the XML and JSON files is done using the Jackson Library that recursively traverses the database hierarchy and serializes the objects it encounters. Proper annotations need to be added to the back

references of relationships to other entities to avoid endless recursion (*@JSONBackReference*). We also used *@JSONIgnore* annotations so we don't include certain attributes to the generated documents. To avoid duplicating the logic for serializing to XML instead of JSON, we used the Underscore library to convert the generated JSON to XML. Once we generate the raw JSON or XML String, we need to dump it into a file which we will then return to the user. It is the responsibility of the client to properly format or display the content with the appropriate indentation.

The assignment specification asks that the exported data include the attributes of all rooms along with the reviews and bookings made for each room as well as the reviews per host. As it is not possible to organize the data in a way that satisfies both structures mentioned above, we offer two different kinds of data that can be downloaded:

- Room Info: This is a list of all rooms, with each room containing its attributes and the reviews and bookings associated with it. These two entities will respectively include their relevant attributes as well as the user that made the review or booking.
- Host Reviews: This is a list of all hosts, for each of which we show a list of their rooms and all the Reviews and review info associated with each room.

We should note that serializing the database populated with all the entries from the dataset is not a fast process, requiring at least one minute to complete. However, given that this feature is meant to be used seldomly and only by the administrator, we decided not to optimize this process any further. Please keep in mind that with the full dataset, such as the one you can preload the database with by following [these](#) instructions, the generated files will be quite large (50+ MBs) and can easily crash a text editor that is not suitable for this kind of big file sizes.

Images

The requirements of our application mean that the server will need to store images that the users upload, such as profile pictures and room photos. Those images will inevitably need to be served back to the users when viewing a certain room or examining the profile of a host.

The most important design decision for this subsystem was whether to store the images directly in the database or on the disk. Storing large files such as an image in the database, especially MySQL, is far from optimal in terms of performance. Therefore, we decided to store the images on disk instead. There is a price to pay for that, in terms of horizontal scalability, but for the purposes of this assignment we judged that to be an acceptable compromise.

However, we created an Image entity that is stored into the database so we can create relationships between images and other entities such as rooms and users. This entity includes a reference to the image file stored on disk rather than the whole content of the file. This reference is in the form of a unique string called a Global Unique Identifier (GUID). This GUID also serves as the name of the image file stored on disk. As a result, when uploading a new image, the server needs to create the corresponding entity into the database as well as store the file in a directory that can be accessed later.

To encapsulate all this image-related logic we created the *ImageStorage* class inside the *images* package. This class provides functionality such as creating, retrieving and deleting an image to the rest of the codebase. This way, the rest of the code can easily work with Image entities without having to worry about

the underlying handling of the files on disk and keeping the database and the file structure consistent with each other. The most important operations provided by this class are:

- *saveImage*: takes an image file as a parameter and saves it to the appropriate directory for later access, as well as creating and returning the corresponding Image entity in the database.
- *loadImage*: returns the Image entity from the database for a given image identifier (GUID).
- *loadImageFileByGUID*: like the function above, but it returns the image file instead.
- *loadImageFile*: takes an Image entity as a parameter and returns the corresponding image file.
- *deleteImage*: takes an Image entity as its parameter and deletes the entry from the database as well as the corresponding file.
- *deleteImages*: performs the same operation as above, but takes as a parameter a list of images to be deleted.

Finally, we are noting that our app is preloaded with a default profile picture and a default room thumbnail. These images come with the application itself and can not be deleted, as they would not be accessible anymore to the rest of the users or rooms that may make use of them.

Messages

As the specification requested the feature of allowing potential tenants to message hosts for more information about their rooms, we decided to add our own in-app messaging system that works like an email application. For the purposes of this project, we decided to allow all the users to message each other provided they know the recipient's username. We added a button in the room view page of the website to message the host, which will redirect the user to the messaging page, similar to clicking an email link.

When sending a message to the user using the compose button on the website, a new *Conversation* will be started. This conversation can contain many *Messages* as the users are replying to each other. It is important to highlight that a Message can not exist outside the context of a Conversation. Furthermore, a conversation can be marked as read-only, such as the welcome messages when registering to Bookify. A user is obviously not allowed post a reply to a read only conversation.

The *Conversation* is represented by a database entity containing the topic, the last-updated timestamp, the read-only status and relationships to the two users of this conversation. The *Message* entity contains the body and the timestamp of the message, as well as a many-to-one relationship to the user (a user can send multiple messages) and the conversation (a conversation includes many messages).

The main design problem we run into when developing this subsystem is what happens when a user deletes a given conversation. We can not simply delete the corresponding entity from the database, as this would mean the other user would no longer have access to it. This problem extends to other features such as marking a conversation as read. One way to solve this would be to create two copies of each conversation, assigning each of them to one of the participants. However, this would mean unnecessarily repeating the same data.

Our solution to this problem was to create a separate table of *InboxEntries*. When a new conversation is started, the server creates two inbox entries for that conversation, one associated with each user. Those entries hold the read and delete status variables. A user can only see conversations for which their corresponding inbox entry has the *isDeleted* variable set to false. When they wish to delete a conversation,

the server only needs to update the status variable of the corresponding inbox entry, thus solving the problems described above.

The bulk of the logic that makes the messaging system work is contained in the *MessageService* class. This class is responsible for creating and retrieving the corresponding database entries according to the method described above. It also needs to perform various sanity and security checks, such as ensuring that the conversation a user is trying to reply to exists and that the user is a party of this conversation and thus allowed to actually reply. Please refer to the code in this class for more details on the implementation of this system.

This is by no means a fully-fledged messaging system. There are many more features we could have added, such as editing or deleting individual messages. Since the messaging part is not the main concern of the application though, we decided to offer only the basic features.

Utilities and Configuration

In our codebase one can find some classes that do not belong to any of the Data, Repository, Service or Controller types we mentioned [earlier](#). Those classes contain either utility functions and constants that are useful to other classes in the rest of the codebase, or configure a given part of Spring Boot to fit the needs of our application. The Utilities and Configuration classes are contained in their own packages, separated from, but accessible to the rest of the code.

Here is a very quick rundown of the purposes of each class in the *configuration* package:

- *AsyncConfig*: enables and configures the execution of asynchronous tasks in the Spring framework, such as the recommendation algorithm described [here](#). Various parameters are set such as the thread pool size and queue capacity.
- *Configuration*: contains a set of constants that configure several parts of the application in a centralized place. Examples include the subfolders for images and data, the username and password of the administrator and the duration of the access and refresh tokens described in [this section](#).
- *CorsConfig*: allows Cross Origin Resource Sharing (CORS) for development purposes and only from the local host in port 3000. This is not necessary for the app to function when it is built and deployed but is left there regardless, in case someone wants to run the frontend from a npm development server.
- *InitializeDatabase*: the code in this class is executed immediately after application startup and loads the necessary info such as roles, amenities, room types and the admin user to the database, if they are not already added.
- *MethodSecurityConfig*: enables method-level security annotations such as those described [here](#).
- *SecurityConfiguration*: defines security-related beans and the security filter chain. More details can be found in the [Security](#) section.
- *WebConfig*: redirects any non-API request to root path ('/') in order to allow the backend to correctly serve react router pages.

Following is a brief description of each of the main utility classes:

- *Constants*: defines a set of constants, such as the strings describing the various role requests from the user, useful for the inner workings of the app.

- *GUIDGenerator*: provides functionality to generate Global Unique Identifiers (GUIDs) that are used as filenames for images or refresh tokens for the users.
- *ImageFormatDetector*: detects whether a given image file is a PNG or JPEG image.
- *IOUtility*: encapsulates Input/Output related operations such as creating the path where the application can write its data and saving and retrieving the recommendation files from disk.
- *UtilityComponent*: contains functions that are useful throughout the program such as getting the current authenticated user or creating a Search Response DTO from a given Room object. We need to note at this point that this class is a Spring Component as it needs to make use of Spring Boot's built-in dependency injection framework to access various repositories.

This section concludes our overview of the backend.

Frontend/Website

The website is a crucial part of Bookify and also the only (convenient) way for any user to interact with the application. Therefore, the site needs to look as nice as possible and provide an intuitive, interactive interface to allow the user to easily access all the features we provide. The front-end was created using the React framework as it is extremely popular, well documented and flexible.

React allows us to separate our app into individual reusable components and compose our application using those components. Examples of custom components we created include the navbar, the registration and login forms, the search bar and a lot more. The various pages of the website are also components themselves! Fortunately, there are a lot of libraries out there, such as Material UI, providing us with ready-to-use components such as text/input fields, icons, tooltips, image carousels, date range pickers etc.

We will not be examining the design of the front-end in as much detail as the backend, as each component is basically a mix of JSX (React's HTML variant), CSS and JavaScript that handles the interactivity of the component and performs any necessary communication with the server. We will be looking at the general structure of the front-end code, how authentication, server requests and protected routes/pages are implemented and at some useful hooks (reusable react functions) we created. The rest of the codebase features no major design decisions and its development was a matter of creating the JSX elements that make up each component, painstakingly styling it using CSS and adding the required JavaScript to provide the appropriate interactivity and communication with the server.

General Structure

The code for the website is located in the */bookifyFrontend* folder of the deliverable we provided. The source code can be found in the */bookifyFrontend/src* subfolder. It is organized in the following folders:

- *api*: contains the script that configures Axios (the library we used to send requests) to communicate with the backend. The URL to the server is also defined here.
- *components*: includes all the components that make up our website. Each component is contained in its own folder along with the CSS file that defines its styling.
- *context*: the definitions of the various React contexts we used are included here. A context is a React feature that allows us to pass parameters, called props in React, to all the children components without actually having to pass them manually at each level of the nested component tree.

- *hooks*: contains various useful React hooks that provide easy access to functionality like fetching various data from the backend, such as images, user info or performing other kinds of operations such as logging out, refreshing a token etc.
- *images*: contains a few custom icons that we use throughout the app.
- *pages*: this folder contains the components that define the various pages of the application. Each page is a component on its own and can also contain other components inside it. Some pages also need their own styling sheets, which are contained in the */styles* subfolder.

Two very important scripts that hold the whole app together are the following:

- *index.js*: this file asks the *ReactDOM* to render our app. It needs to include certain custom components such as the *Auth* and *SearchProvider* and *Filter* contexts we created. It also includes some built-in components such as the *BrowserRouter* from *react-router-dom*, that is used to allow us to navigate to different pages of the app.
- *App.jsx*: this crucial script defines all the different pages of the website, the path to each page, the custom component that defines that page as well as what roles have access to it.

Security and Authentication

In the frontend part of Bookify implementing Security was a matter of locking users out of pages they are not allowed to access using *react-router-dom*'s protected routes and disabling/hiding buttons or options that perform operations the user is not authorized to do. Some examples of this are booking a room without being logged in, trying to book your own room or trying to post a review without being a tenant. It is also important to mention again that these checks are also performed in the backend to ensure a bad actor can not perform an unauthorized operation by bypassing the website restrictions.

To implement the techniques mentioned above, however, we need a way to know whether a user is authenticated as well as their application roles. This is the responsibility of the Authentication system which also enables us to send authenticated requests to protected API endpoints to perform critical operations available only to certain user or user roles.

The website's authentication system stores all authentication-related information (refresh and access tokens, username and roles) in a global React context called *AuthContext*. The *useAuth* hook provides functions to retrieve and edit the authentication information from any component that may need to do so. With this system, the process of verifying a user is logged in (so they can perform a certain privileged operation) is reduced to checking whether the username of the current logged in user, retrieved from the *useAuth* hook, is not empty or null. Similarly, the process of logging a user into the website is a matter of retrieving the authentication information by a request from the backend, and then using the *useAuth* hook to store them into the global authentication context. Finally, the logout functionality, implemented in its own *useLogout* hook, needs to perform the necessary request to log out from the backend and then reset the authentication information in the *AuthContext* and redirect the user to the home page.

Performing a request to an unprotected endpoint is a matter of simply calling in the appropriate Axios function and filling in the required parameters such as endpoint URL and body parameters. Performing an authenticated request, however, using the JWT token described in [this](#) section, is a much more complicated process. The access token needs to be retrieved from the authentication context and passed along with the request. If the server rejects the request, it is possible the access token expired and we

need to refresh it and resend the request with the new access token instead. If the refresh request fails too, we need to completely log the user out of the application since their refresh token expired. This whole process is automated using the *useAxiosPrivate* hook. This hook will return an object called *axiosPrivate* that can be used to send authenticated requests to the server wherever necessary, just like a normal *axios* object. To make this happen, *useAxiosPrivate* makes use of the *useRefreshToken* hook that makes the refresh request to the server, updates the authentication context and returns the new access token issued by the server.

The few hooks described above take care of all the work necessary to handle all authentication-related functionality allowing us to implement the rest of the website without having to constantly worry about authentication details. Our implementation of this subsystem was inspired by [this](#) tutorial.

Other Contexts

As we saw, React's contexts are an incredibly useful tool that can make the implementation of certain features a much easier process. Here are two other examples of React contexts we created:

- *SearchContext*: when a user makes a search for a room in a given range of dates and for a specified number of tenants and then clicks to see more details of one of the search results, it is probably a good idea to not ask them to enter that information again when booking the room, as they can easily be inferred from the search the user made. To that end, when making a search we store the parameters the user entered into a global search context to allow the booking options panel to be automatically preloaded with that information, if available. We need to ensure we reset this information when making a new search or visiting the home page.
- *FilterOptionsContext*: in this context, we are storing the various filters and pagination options of the user when they visit a room. This enables the user to view a room and then return to the search page without having to re-enter all the desired filters or manually navigating to the page they were previously on. We are also saving this info to session storage, so they can persist even if the user refreshes the page.

This concludes our short overview of the front-end's architecture. As promised, this section was short, as explaining how each of the dozens of individual components work would fill a whole book without providing any real value to understanding the front-end code. Instead, we focused only on the few crucial systems, such as authentication and sending requests, that form the core of all the functionality of the website.

Setup Scripts

Our application would be incredibly dull if it had no data in it. It would also be practically impossible to load it up manually with enough data to make sure the server performs adequately with large amounts of data. For this reason, we decided to populate our database with the information (reviews, rooms, users) from the provided datasets. It was not possible to directly import them to our database, so we had to make a custom Python script to perform this task. The script writes directly to the database instead of sending the appropriate requests through the server both for convenience and performance reasons.

We had to manually go through the provided spreadsheets and figure out which columns we would use, as well as match each column of our csv to the corresponding column of the database. We then needed

to write custom SQL queries to insert the data from the csv into the database. We did not make use of the calendar.csv file at all but we used many of the columns of the other two files (listings and reviews). Since each review and room needs to be associated with a user, we used whatever data was available in the spreadsheet to create the corresponding user for each review or room. Therefore, we first go through the two spreadsheets to create the users and assign to them the appropriate roles and then go through them again to generate the reviews and listings. We also note that the setup scripts will mark any listing it adds as available for the next 90 days after the current date for demonstration purposes.

One major problem we had was that the location data in the spreadsheet were incredibly inconsistent. For example, the same city would be spelled differently or in another language which is a big issue for our location-based [search system](#). We solved that problem by using the [Geoapify Reverse Geocoding API](#) to figure out the city, state and country of each listing from the provided coordinates. This ensured consistent, although slightly inaccurate, location data for each listing. The Geoapify API is rate-limited, so we had to create a separate script to perform the reverse geocoding requests and save the results into the *locations.json* file on disk. The setup script can then access this file whenever it needs to, since the spreadsheet data stays the same, without needing to request any information from Geoapify. For the reviews, there was no rating available in the dataset, so we had to choose one at random.

Before loading any data into the database, the script needs to delete all existing users (bar the administrator), reviews, availability and listings. It also needs to temporarily disable the foreign key checks and auto-commit. After that, the script can start going through the csv files and add the appropriate entities to the database. As described earlier, it is important to edit the *populateDatabase.py* script to set the correct database credentials for your setup. We will also highlight, once more, that this script is meant to be used when first setting up the application. If used when the database is filled with data generated after using the application for a while, there is a strong possibility the app functionality will be irreparably corrupted, thus requiring a complete reset of the database.

Conclusion

Developing Bookify has been a long process. We started by setting up a basic Spring Boot project and configuring Spring Security to handle authentication and role-based security using JWTs. Once that was done, we could move onto creating the user entity and the registration and login related code. We tested our app using manual Postman HTTP requests and once we verified we got the basics right, we started expanding the backend with more features such as rooms, images, reviews and search. At the same time, we started experimenting with React and tried to make it interact with the server, starting again with registration and other authentication related features. Once we got the hang of React, the rest of the development was a matter of implementing, integrating and testing all the requested features one by one. We both worked on pretty much all parts of the code and were in constant contact with each other to synchronize our work. We used git and GitHub for version control as well as branches, pull requests and code reviews to maintain proper code quality and stay up to date with each other's work.

The biggest challenge in developing Bookify was the mere scale of it. The assignment required a ton of features, each of them needing to be adequately tested, secured, checked against edge cases and connected to an intuitive and presentable interface on the website. We felt that the volume of different features that needed to be implemented prevented us from looking into certain aspects of our app, such as efficiency and scalability, more closely. The second biggest challenge was the creation of the website

and especially the styling of it. After creating the first few pages of the site, this process was painfully tedious and took up the bulk of the development time.

Attributions

Our many thanks to:

- [Openstreetmap](#), [Geoapify](#) and [Nominatim](#) contributors for their map data and reverse geocoding services.
- [Font Awesome](#), [Iconscout](#) and [flaticon](#) for the free icons they provide.
- [Material UI](#) for their templates, icons and various React components that we used heavily on our website.

API Reference