Bookify

[Image of Home Page]

Created By: Konstantinos Arkoulis, Georgios-Alexandros Kostas

For the purposes of Internet Application Technologies Class

University of Athens, 2022-23

Contents

[Installation Guide 3](#_Toc145379073)

[Run in the Cloud 3](#_Toc145379074)

[Run locally from pre-built files 3](#_Toc145379075)

[Build and Run Locally 5](#_Toc145379076)

[Run Setup Scripts 5](#_Toc145379077)

[Application Features 6](#_Toc145379078)

[Architecture Breakdown 6](#_Toc145379079)

[Security 6](#_Toc145379080)

[Authentication 6](#_Toc145379081)

[Authorization 7](#_Toc145379082)

[Database 9](#_Toc145379083)

[Backend/Server 9](#_Toc145379084)

[General Architecture 9](#_Toc145379085)

[Authentication 11](#_Toc145379086)

[Users 11](#_Toc145379087)

[Rooms 11](#_Toc145379088)

[Booking and Availability 11](#_Toc145379089)

[Search 11](#_Toc145379090)

[Reviews 11](#_Toc145379091)

[Recommendation System 11](#_Toc145379092)

[Administrator 12](#_Toc145379093)

[Images 12](#_Toc145379094)

[Messages 12](#_Toc145379095)

[Utilities and Configuration 12](#_Toc145379096)

[Frontend/Website 12](#_Toc145379097)

[API Reference 12](#_Toc145379098)

[Attributions 12](#_Toc145379099)

# 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.

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 sections we will be providing detailed instructions for each of those options. If you are running the server locally, please refer to [this](#_Run_Setup_Scripts) subsection to check how you can use the provided setup scripts to preload some useful data to the database.

## 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. 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 will probably 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*

1. 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*

1. Install MySQL Server either via the official website or with this command if you are on Linux:

*sudo apt install mysql-server*

1. 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):
   1. *create database db\_bookify;*
   2. *create user ‘admin’@’localhost’ identified by ‘[password]’;* where *[password]* will be the password you want to set for the database user.
   3. *grant all privileges on \*.\* to ‘admin’@’localhost’;*
2. 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.
3. After you have **correctly** 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 4b. 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=8443 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 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 thing that should be required to use the app once the server is running is to open your browser and type the following into the address bar: <https://localhost:8443>.

**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 5 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. An example of this command, that still requires you to provide the correct path, is the following: *cd C:/Users/[USER]/Documents/*bookify/bookifySystem.
3. Use the following command to build and run the project: *mvn spring-boot:run.* 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:
   1. *mvn clean:* cleans up all the build files.
   2. *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 preinstalled 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!**

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*

1. Open the populateDatabase.py script 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.
2. 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.
3. 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.

# Application Features

# 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.

### Authentication

As required by the assignment, the authentication in this application is performed using a mechanism called [JSON Web Token (JWT).](https://en.wikipedia.org/wiki/JSON_Web_Token) 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 (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 even 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 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 but are not limited to issue datetime, expiration date, subject (concerned user), expiration date and roles that are used to decide permissions.
* 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 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 A screen shot of a computer

   Description automatically generatedthe *SecurityConfiguration* class. It clearly shows how it can be used to permit only users with the given role to access endpoint classes such as /api/admin, that 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.

Figure : The Security Filter Chain of the Bookify server

1. 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 be users with the admin or host role.

A screen shot of a computer program

Description automatically generated

Figure : @PreAuthorize annotations in action

1. 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 database 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 provide 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:

A screen shot of a computer code

Description automatically generated

Figure : 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](#_API_Documentation) section of this document.

## 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 we were already familiar with it from previous courses, and it is a very mature and well documented database management system.

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

A computer screen shot of a computer program

Description automatically generated

Figure : The database of the Bookify application

## 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 logically organized and also close together. 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, handling errors and much more.
* 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 sends the response back in an appropriate HTTP message.

### Authentication

### Users

### Rooms

### Booking and Availability

### Search

### Reviews

### 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](https://en.wikipedia.org/wiki/Matrix_factorization_(recommender_systems)) algorithm, implemented using [Gradient Descent](https://en.wikipedia.org/wiki/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 ignoring any other delays for the dataset provided by the professor).

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.

For shortness’ sake we will not cover the details of the matrix factorization algorithm here. [This](http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/) 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 any relevant results. 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:

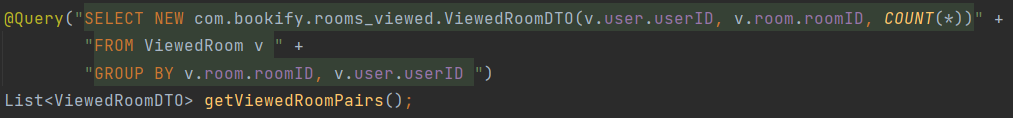


Figure : The query that counts how many times a user has viewed a given room

* 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:

A computer screen shot of a computer code

Description automatically generated

Figure : 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

### 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 examine 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

### Utilities and Configuration

In our codebase one can find some classed that do not belong to any of the Data, Repository, Service or Controller types we mentioned [earlier](#_General_Architecture). 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 brief 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](#_Recommendation_System). 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](#_Authentication).
* *CorsConfig*: allows Cross Origin Resource Sharing (CORS) for development purposes and only from the local host in the port 3000. This is not necessary for the app to function when it is built and deployed but is left regardless in case someone wants to run the frontend from a npm development server.
* *InitializeDatabase*: the code in this classes 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](#_Authorization).
* *SecurityConfiguration*: defines security-related beans and the security filter chain. More details can be found in the [Security](#_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 defining 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

# Attributions

# API Reference