The application is a simulation of the freelancer market application. We have two types of users:

**Entities:**

- **User**: All users must have a shared email and password. All users must have an email and password. User types must be one of the following 2 types (we call this extra information profile), and authorization is based on role in operations:
    - **Freelancer**: has fields name, bio, and skills (which is a set)
    - **employer**: has fields company name and bio
- **Project**: It must only be created by the employer and has name and description fields and status (active and inactive) and a list of skills.
- **Application**: For each project, a request can be sent by the freelancer, which includes a message. The request status can also have 3 states: PENDING, ACCEPTED, and REJECTED.
- **Skill**: It contains a word like Java. At the beginning of the program there should be defined skills like the following:
    - "Java", "Spring Boot", "Database Design", "Frontend Development"

**Operations**:

- **User registration** (Should be available without authentication): Only the email and password, a name for the freelancer, and a company name for the employer should be received.
- **User login** (Should be available without authentication): This operation must occur with username and password and return a jwt after login.
- **User authentication**: JWT must be validated.
- **Get userprofile by id** (Only authenticated users): User information should be returned (you should design a separate dto for each type)
- **update profile** (only profile owner): Separately for each type of user, also for updating skills of freelancer if a skill doesn't exist in database you should add it to database
- **Get skills** (Only authenticated users): each type of users should able to get all skills
- **Project creating** (just for employer): He/She should be able to create a new

project. If he/she uses a skill that is not in the database, you must add it to the database.

- **Project Listing** (Should be available without authentication): Only the ID, name, and skills should be returned here.
- **Project Detail** (Only authenticated users): Users should be able to view the full details of a specific project.
- **Apply to Project** (just for freelancers): A notification must also be sent to the project owner.
- **View Applications** (Only the employer who owns the project): See all project-related applications
- **View Applications** (Only the freelancer who owns the applications): see all applications freelancer made
- **Accept an application for a project** (Only the employer who owns the project): Change the project status to INACTIVE and the application status to ACCEPTED. All the other applications must be REJECTED. Everyone who applies for this project should be informed about this via notification.
- **Reject an application for a project** (Only the employer who owns the project): reject a special application
- **Inactive a project without accept anyone** (Only the employer who owns the project): maybe employer regret about this project

**Notification**:

The notification system should communicate with the rest of the rabbitMQ. The notification should be printed as an email on the terminal.

From: example@freelancer.market.com (email we set on .env file)
To: jack@gmail.com
Content:
hello world

Tips:

- I also want you to use JPA and PostgreSQL for the database (if there is a place for more speed it would be better to use nosql besides telling me to use mongodb)

- You must implement JUnit and Mockito for unit and integration tests to ensure the correctness of core functionalities.
- You should use open API in all user-side endpoints and write a complete schema so that the user understands the API.
- Also use the java doc format to comment the entire program.
- I want you to use external libraries wherever needed, especially lombok and new java features like records, var, and stream. Also, if code is used between multiple services, it is better to create a shared library that can be loaded onto maven with jitpack and github.
- You will need to optimize the performance of the system in order to lower its response time under high load (e.g. 50 request per second)
- You may need to integrate Redis for caching frequently accessed data (e.g., popular project listings, user profiles) to improve performance and reduce database load. (use it)
- You may implement different caching mechanisms like cache-aside, read-through, etc.
- In microservices, use a regular structure and separation of concerns as much as possible, so that the controller, service, and repository are separated from each other.