Be Connected

National and Kapodistrian

University of Athens

Internet Technologies and Applications

Assignment

Creators:	AM:
Dimitrios Nikolaos Boutzounis	1115202200112
Evangelos Argyropoulos	1115202200010

2023-2024

Contents

Introduction

For this assignment, we developed a professional networking app named BeConnected, which is similar to LinkedIn. This project was completed as part of the Internet Technologies and Applications course at the National and Kapodistrian University of Athens. The application supports two roles: Administrator and Professional. Administrators, assigned during installation, use their interface to manage users and export data. Professionals utilize their interface to manage their training and experience details, handle connections, and engage with their network. They can send and receive connection requests, post articles with images and videos, comment on posts, express interest in content, receive notifications, and conduct private discussions. Additionally, they can view other professionals' profiles and manage their connection settings.

The application is built on a client-server model. The server, developed with Spring Boot and MySQL, provides a RESTful API that supports various client applications. The client-side of the application is designed using React and JavaScript, offering a user-friendly interface for easy interaction. This document will cover the installation and execution instructions, explain the design of our codebase, and discuss the key design decisions that shaped our current implementation.

Installation Guide

Backend Setup

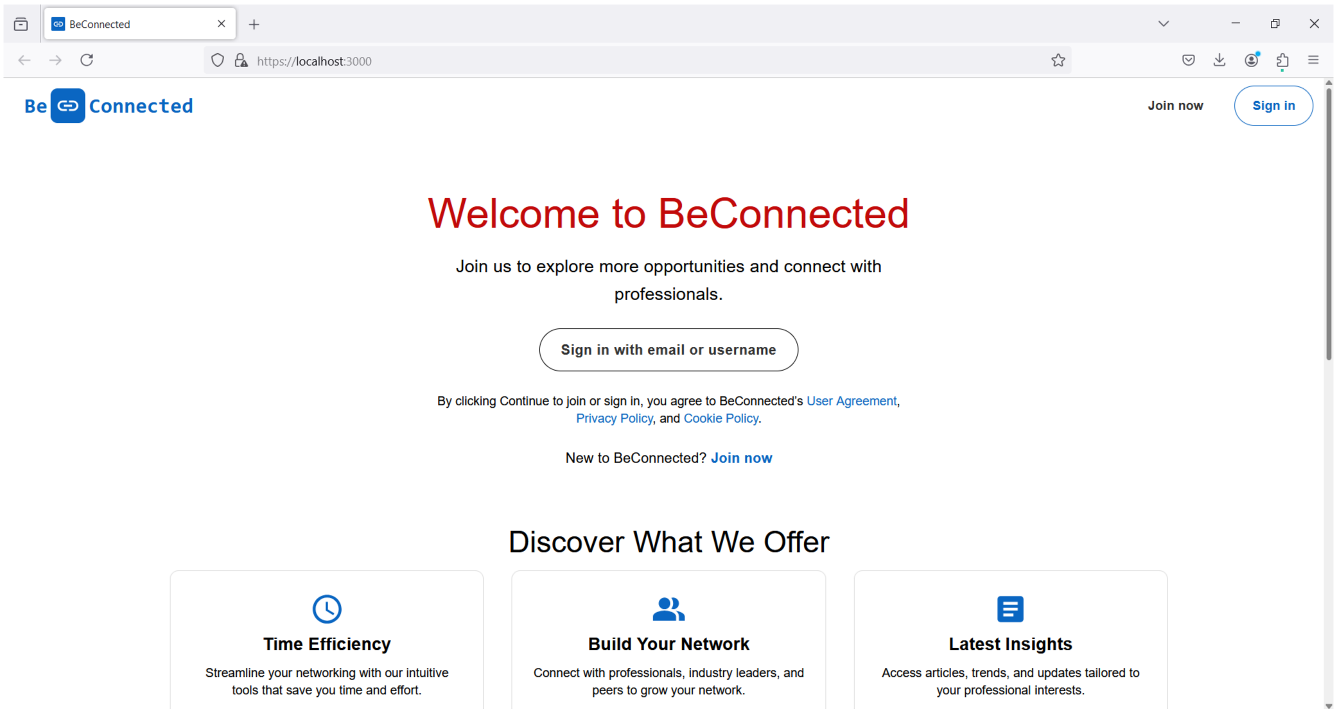
- If you haven't already installed the Java Development Kit (JDK), make sure to get version 21 or higher. You can download and install it from the official website. On Linux, you can install it by running the following command: `sudo apt install openjdk-21-jdk`
- To install MySQL Server, you can either download it from the official website or use the following command on Linux: `sudo apt install mysql-server`
- On Linux, start the MySQL service using the following command: `sudo service mysql start`
- To set up the database, open a MySQL console if you're on Windows or run `sudo mysql` on Linux. Then, execute the following commands (the same for both operating systems):
  - `create user 'admin'@'localhost' identified by 'password';` (where `password` is 1234)
  - `grant all privileges on *.* to 'admin'@'localhost';`
  - `create database beconnected_db;`
- You can now run the application in any Java-supported IDE, such as IntelliJ IDEA

Frontend Setup

- If you haven't installed Node.js and npm yet, you can download and install it from the official website. Alternatively, on Linux, you can use this command: `sudo apt install nodejs npm`
- Open your terminal and navigate to the beconnected-frontend folder in the deliverable. Then, install the required npm dependencies by running the command: `npm install --force`
- You can now start the frontend by using the following command: `npm start`

Note: Because our SSL certificate is self-signed, most browsers will reject the connection. To access our website, you'll need to add an exception in your browser. If you see a rejection message, click on the advanced options button and choose "proceed/add exception". If that doesn't work, you'll need to manually set the exception through your browser's security settings.

Application Features



Welcome to BeConnected

Join us to explore more opportunities and connect with professionals.

By clicking Continue to join or sign in, you agree to BeConnected's User Agreement, Privacy Policy, and Cookie Policy.

New to BeConnected? Join now

Discover What We Offer

Time Efficiency

Streamline your networking with our intuitive tools that save you time and effort.

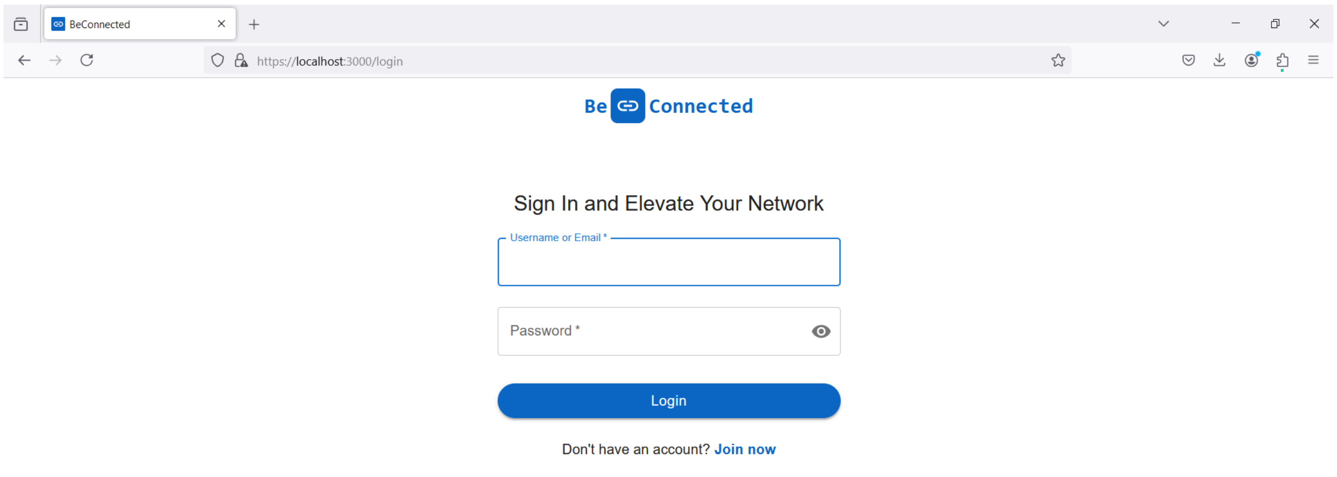
Build Your Network

Connect with professionals, industry leaders, and peers to grow your network.

Latest Insights

Access articles, trends, and updates tailored to your professional interests.

Welcome Page


BeConnected

https://localhost:3000/login

Be Connected

Sign In and Elevate Your Network

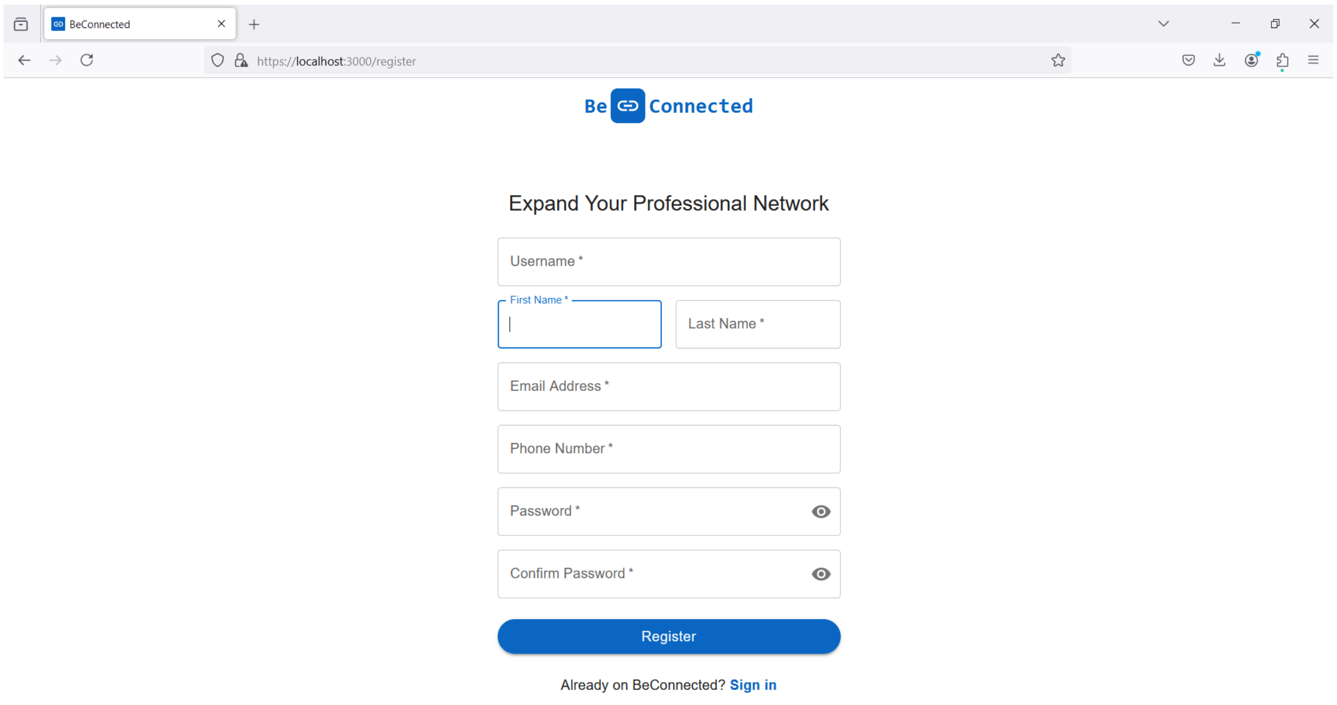
Username or Email *

Password *

Login

Don't have an account? Join now

Login Page


BeConnected

https://localhost:3000/register

Be Connected

Expand Your Professional Network

Username *

First Name *

Last Name *

Email Address *

Phone Number *

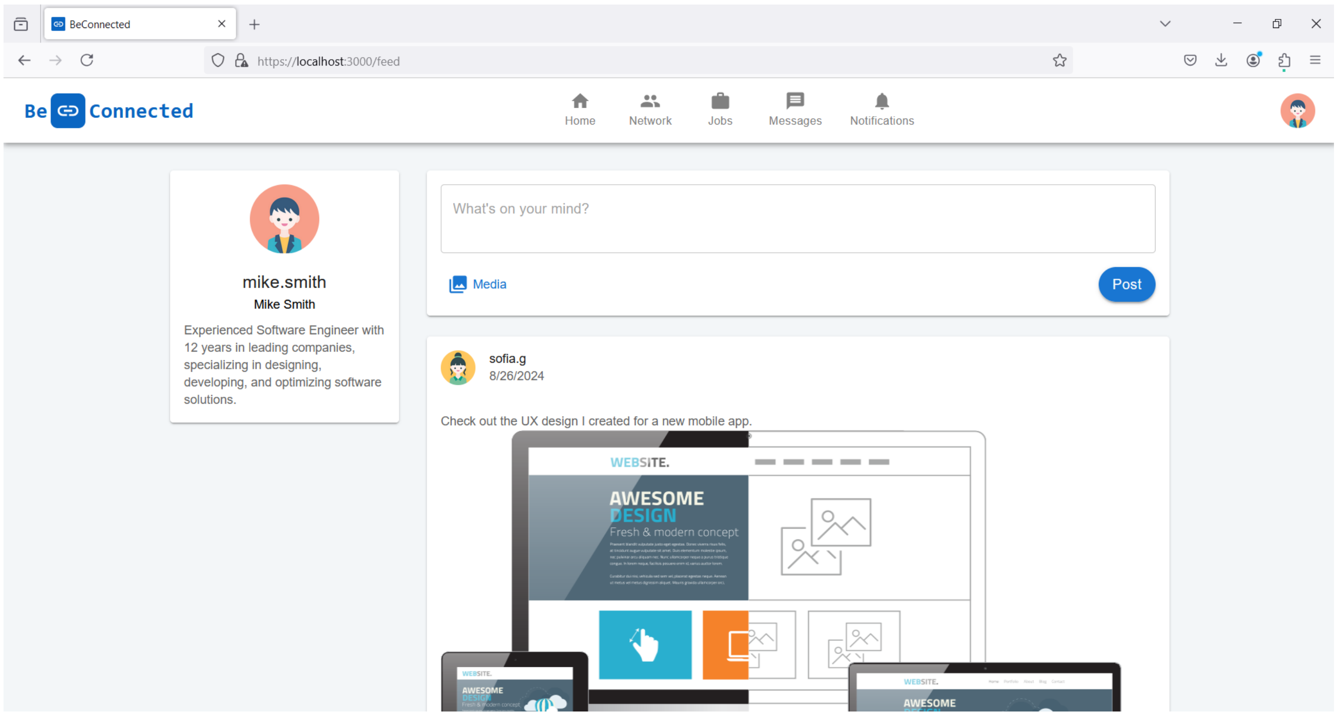
Password *

Confirm Password *

Register

Already on BeConnected? Sign in

Registration Page



mike.smith

Mike Smith

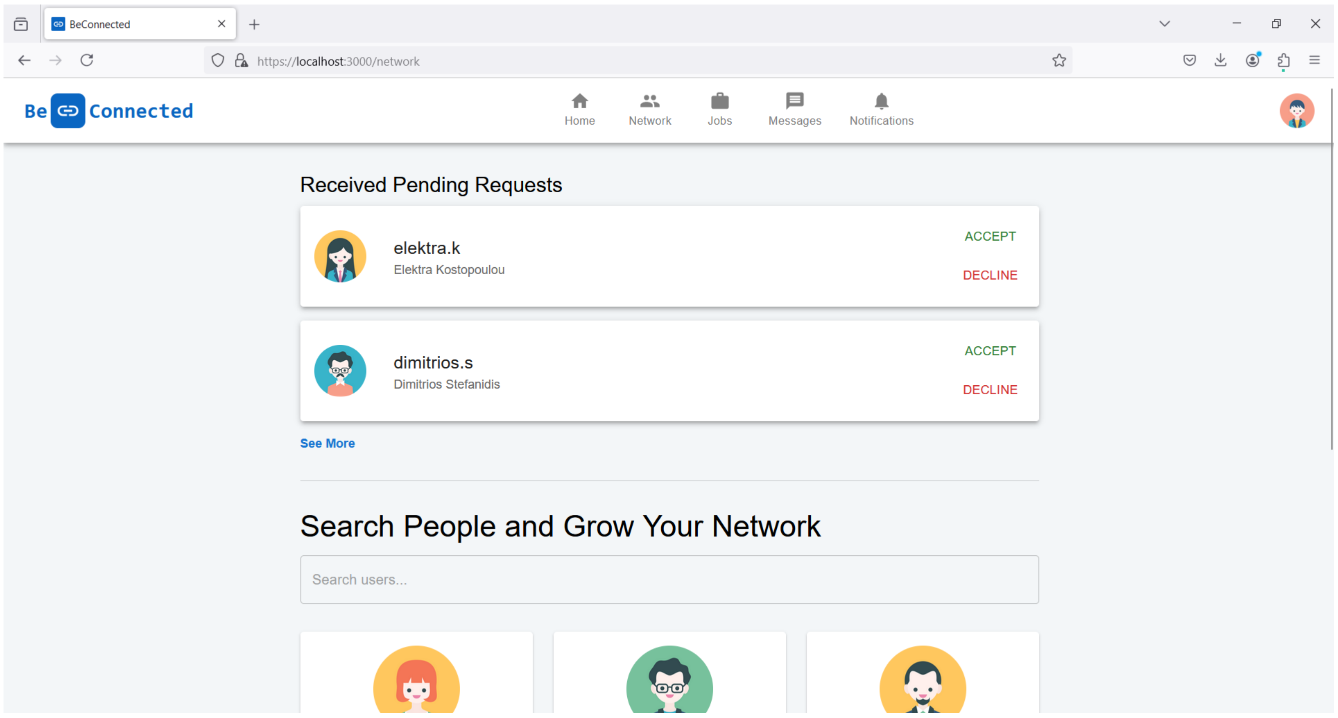
Experienced Software Engineer with 12 years in leading companies, specializing in designing, developing, and optimizing software solutions.

sofia.g

8/26/2024

Check out the UX design I created for a new mobile app.

Feed Page



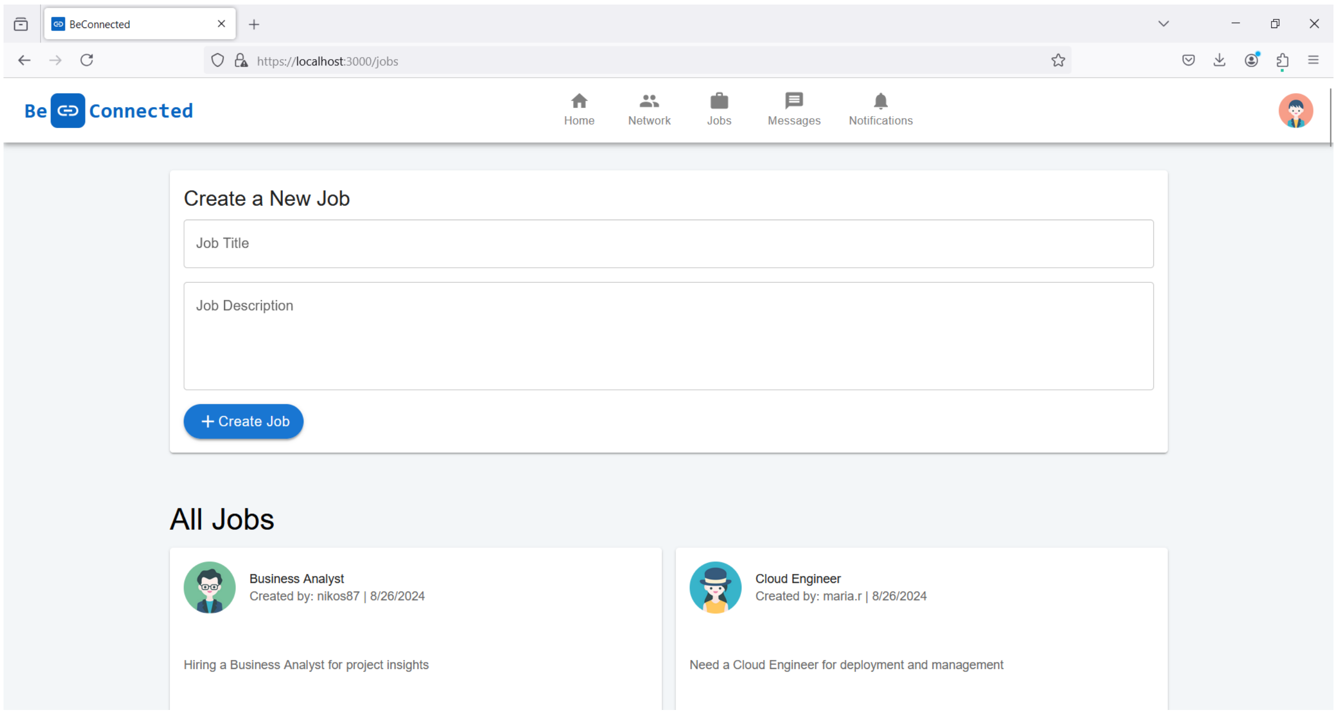
Received Pending Requests

elektra.k
Elektra Kostopoulou

dimitrios.s
Dimitrios Stefanidis

Search People and Grow Your Network

Network Page



Create a New Job

Job Title

Job Description

+ Create Job

All Jobs

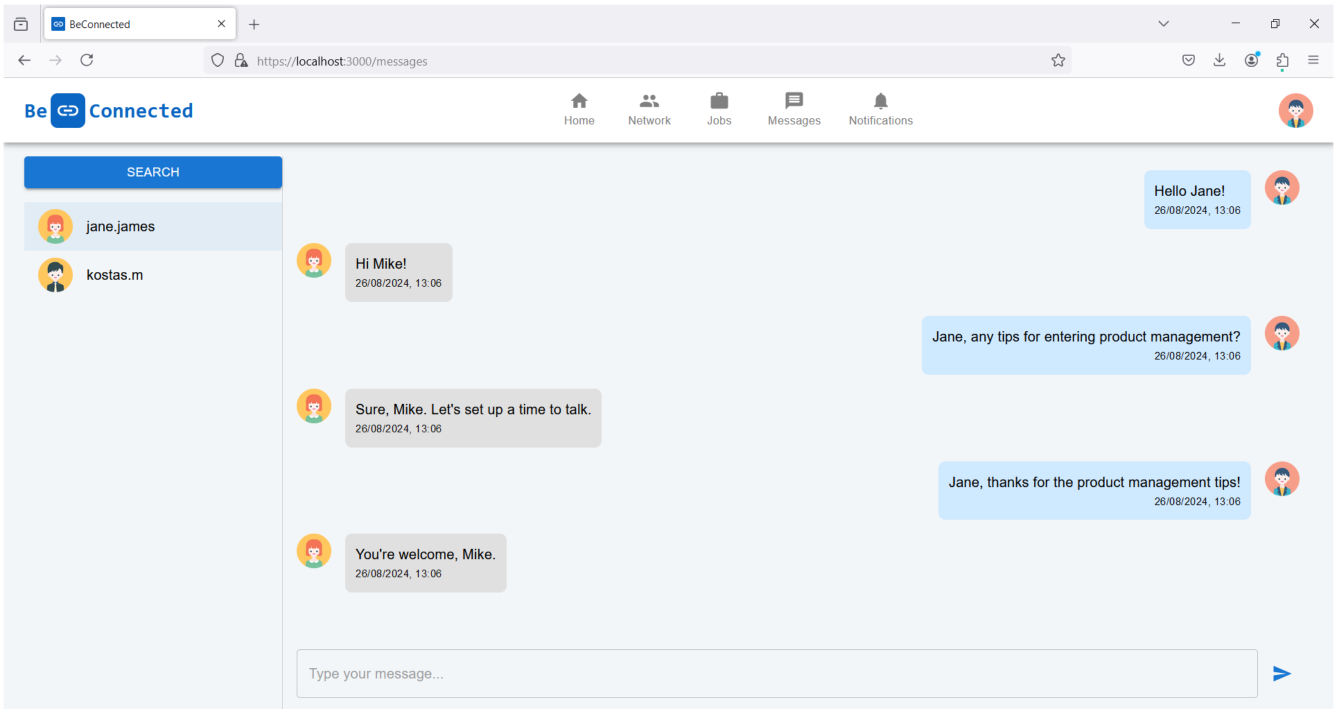
Business Analyst
Created by: nikos87 | 8/26/2024

Hiring a Business Analyst for project insights

Cloud Engineer
Created by: maria.r | 8/26/2024

Need a Cloud Engineer for deployment and management

Jobs Page



Hello Jane!
26/08/2024, 13:06

Hi Mike!
26/08/2024, 13:06

Jane, any tips for entering product management?
26/08/2024, 13:06

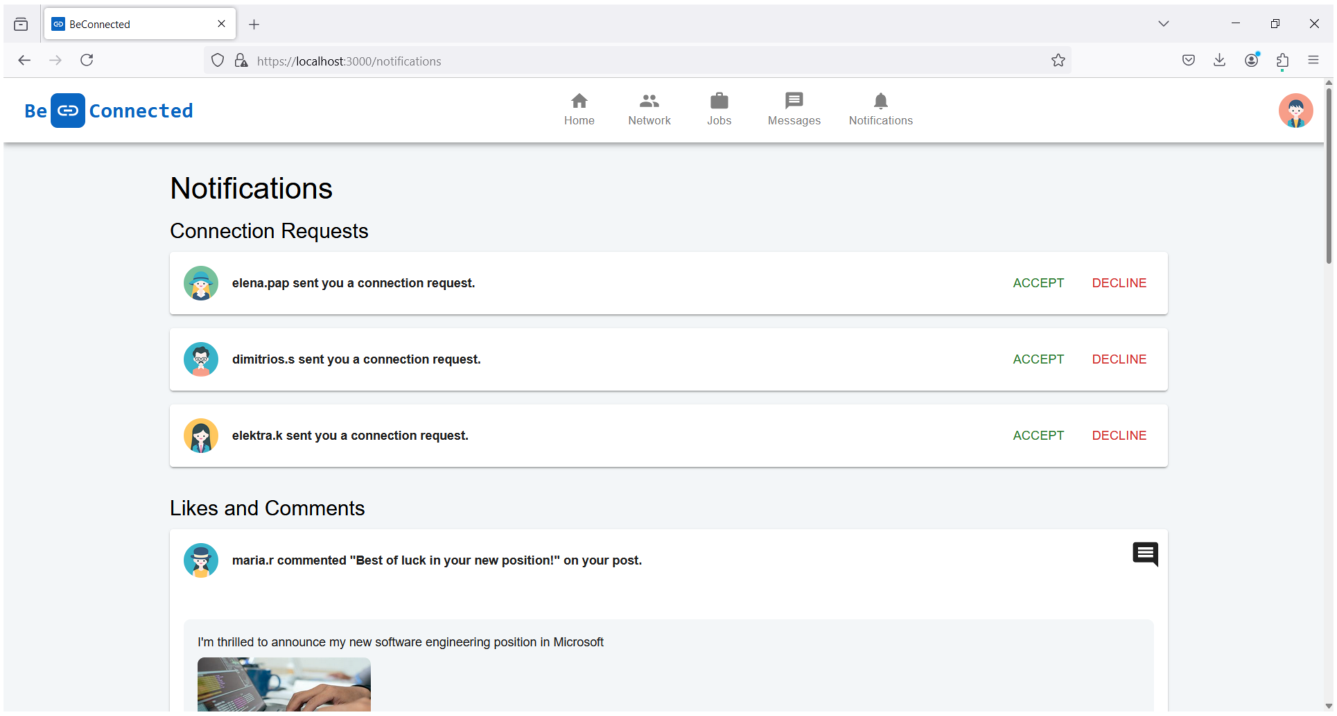
Sure, Mike. Let's set up a time to talk.
26/08/2024, 13:06

Jane, thanks for the product management tips!
26/08/2024, 13:06

You're welcome, Mike.
26/08/2024, 13:06

Type your message...

Chat Page


BeConnected

https://localhost:3000/notifications

Notifications

Connection Requests

elena.pap sent you a connection request. ACCEPT DECLINE

dimitrios.s sent you a connection request. ACCEPT DECLINE

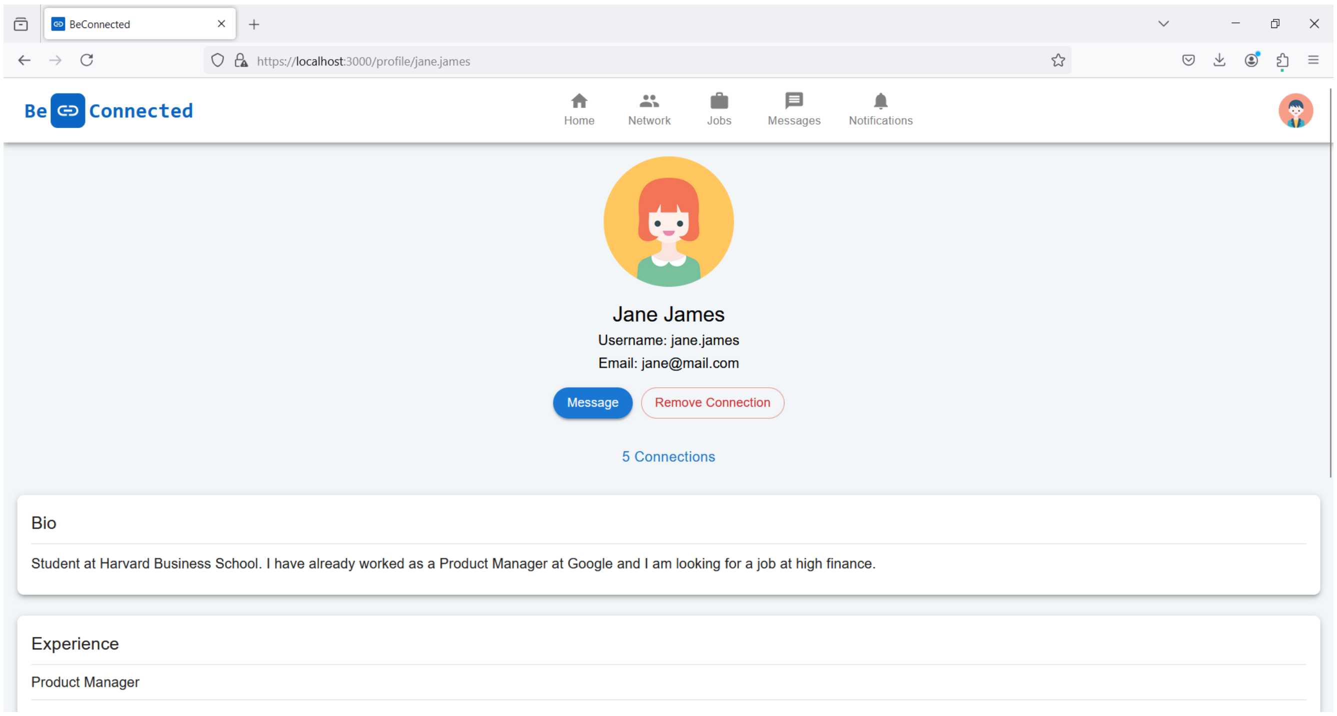
elektra.k sent you a connection request. ACCEPT DECLINE

Likes and Comments

maria.r commented "Best of luck in your new position!" on your post.

I'm thrilled to announce my new software engineering position in Microsoft

Notifications Page


BeConnected

https://localhost:3000/profile/jane.james

Jane James

Username: jane.james

Email: jane@mail.com

5 Connections

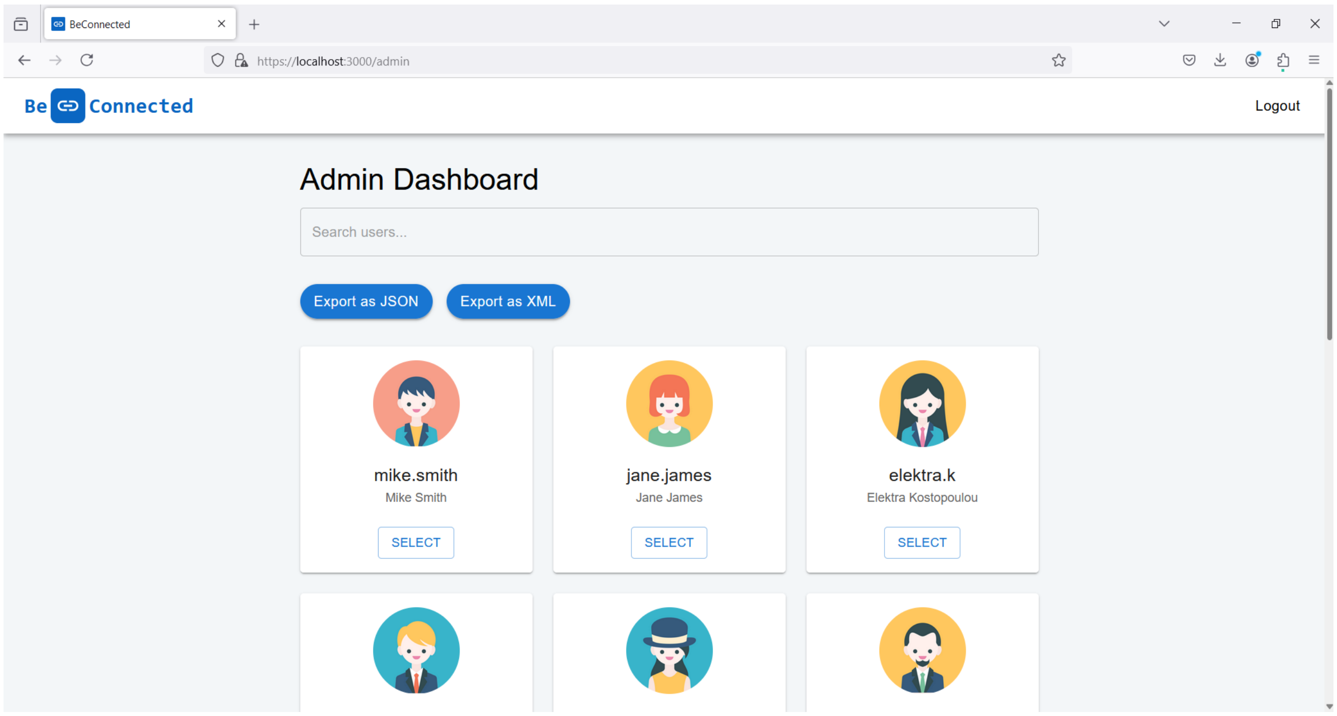
Bio

Student at Harvard Business School. I have already worked as a Product Manager at Google and I am looking for a job at high finance.

Experience

Product Manager

Profile Page



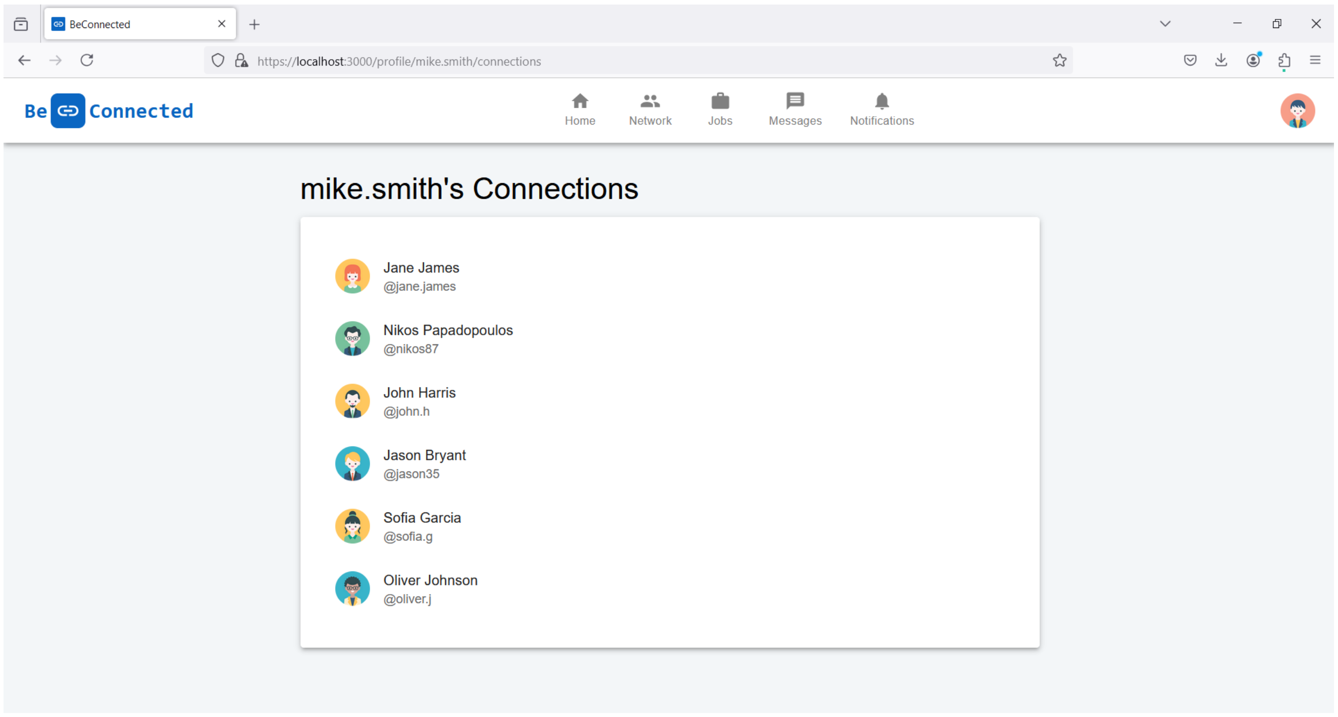
Admin Dashboard

Search users...

Export as JSON Export as XML

mike.smith	jane.james	elektra.k
Mike Smith	Jane James	Elektra Kostopoulou
SELECT	SELECT	SELECT

Admin Page



mike.smith's Connections

Jane James
@jane.james

Nikos Papadopoulos
@nikos87

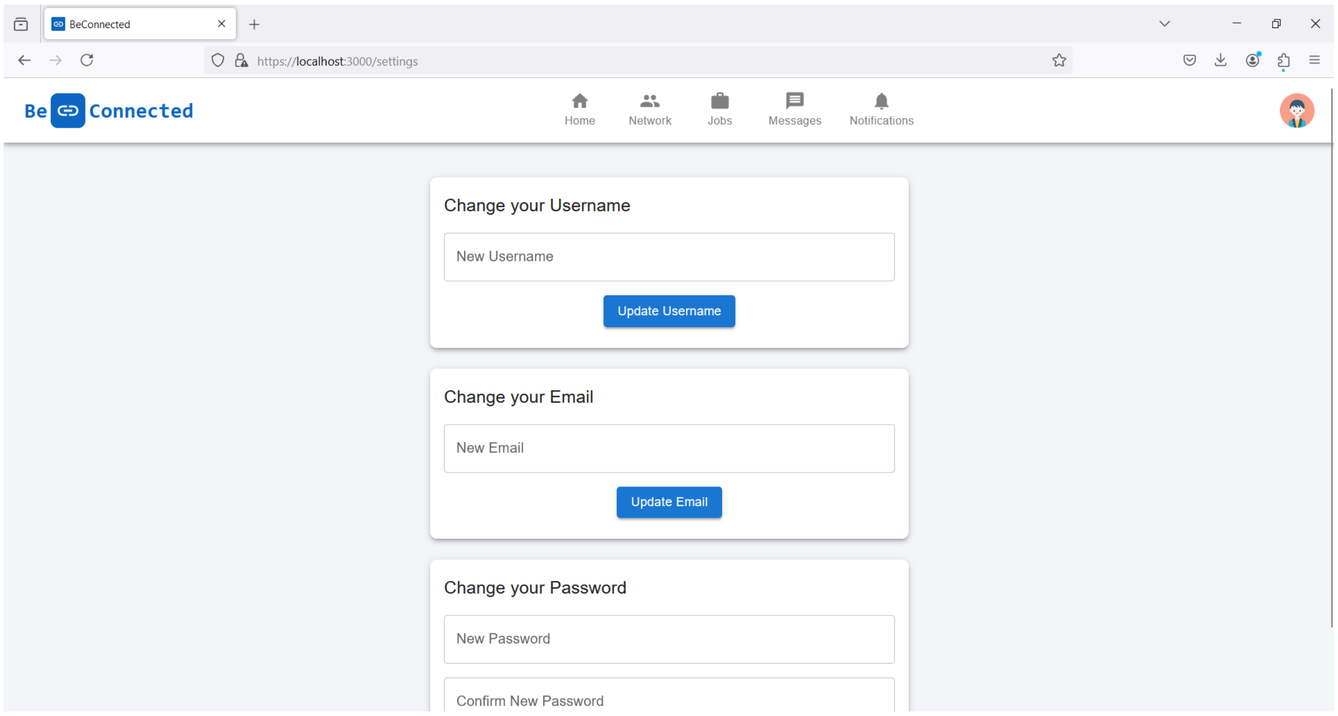
John Harris
@john.h

Jason Bryant
@jason35

Sofia Garcia
@sofia.g

Oliver Johnson
@oliver.j

Connections Page



Settings Page

Structural Overview

Security

Security is one of the most essential requirements for any online application. The two key aspects of security we are considering in present assignment are:

- **Authentication:** verifies a user's identity.
- **Authorization:** determines what actions or resources the user is allowed to access.

Authentication

The authentication system in this application uses JWT-based tokens to ensure secure access control. Upon registration or login, users receive an accessToken for short-term access and a refreshToken to obtain a new accessToken without reauthentication, minimizing security risks if a token is compromised. During registration, the application verifies the uniqueness of the username and email, securely hashes the password, and stores both tokens in the database. Old tokens are revoked upon login to keep only the latest tokens active. To maintain user sessions securely, the `/api/refresh_token` endpoint allows users to refresh their accessToken, with the system validating the refreshToken before issuing a new one. Additionally, the JwtAuthenticationFilter intercepts requests to validate tokens, ensuring that only authenticated users can access protected endpoints, thereby providing a robust security layer against unauthorized access.

Authorization

The authorization in our application is managed through Spring Security, specifically within the SecurityConfiguration class. It defines access rules for different API endpoints. Public endpoints like `/api/login/**`, `/api/register/**`, and `/api/refresh_token/**` are accessible to everyone without authentication. However, endpoints under `/api/admin/**` are restricted to users with the "ADMIN" authority. All other requests require the user to be authenticated. The application also uses a JWT-based authentication filter to secure requests and manages sessions in a stateless manner, ensuring that each request is independently authenticated.

SSL/TLS

In line with the project requirements, we've implemented SSL/TLS to encrypt all communications between the backend and clients. This means that even if someone intercepts a request, sensitive data like passwords or personal information will remain secure. We set up TLS in our Spring Boot application by adding the necessary configurations in the `application.yml` file and generating a self-signed certificate, which you can find in the `beconnected-backend/src/main/resources` directory.

Database

The database design for the "BeConnected" project, implemented using MySQL, is structured as a relational database to manage the complex and interconnected data generated by the social networking platform. We chose the relational approach because it allows for well-organized data storage, ensures data integrity through normalization, and supports efficient querying, especially in scenarios involving complex relationships between entities like users, posts, messages, and connections.

At the core of the schema is the User table, which holds the primary user data, including usernames, passwords, emails, and other personal details. Each user can have a profile picture linked through a one-to-one relationship with the Picture table, where image data, filenames, and content types are stored. The User table is central to many relationships across the schema, reflecting the user-centric nature of the application.

The Token table manages user authentication through a many-to-one relationship with the User table, where each token entry stores an access token and a refresh token tied to a specific user. This setup allows secure session management, ensuring that only the latest tokens are valid and reducing the risk of unauthorized access.

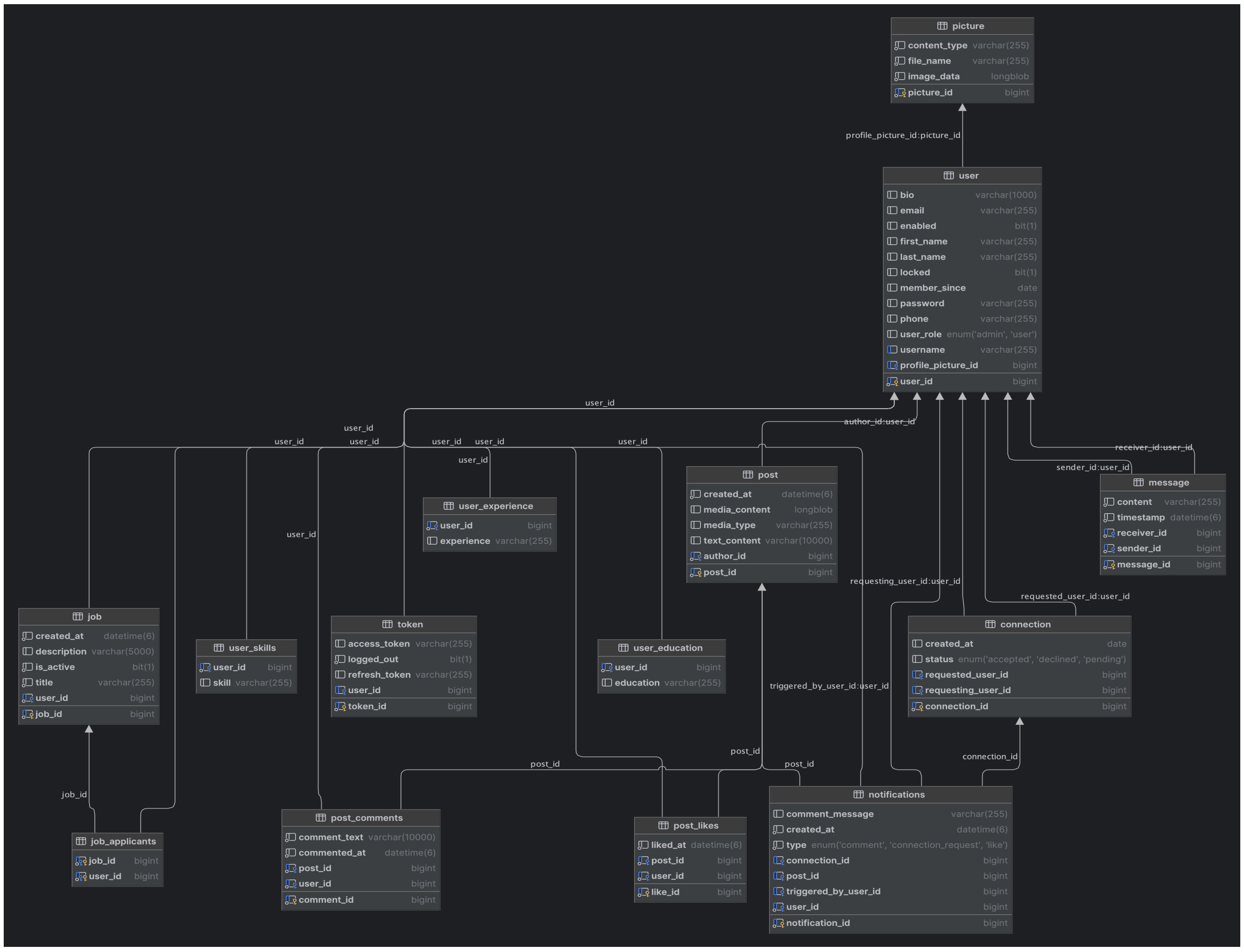
Users interact with each other through the Connection table, which tracks friend requests and their statuses. This table has two foreign keys, both linking back to the User table, representing the requesting and requested users, respectively. The status of these connections is managed via an enumerated field, allowing for easy tracking of whether a connection request is pending or accepted. Content creation is managed through the Post table, where users can author posts that include text and media. The Post table has a many-to-one relationship with the User table, linking each post to its author. Users can interact with posts by liking them (tracked in the Post_Likes table) or commenting on them (stored in the Post_Comments table). Both Post_Likes and Post_Comments tables have many-to-one relationships with the User table and the Post table, establishing an effective system for content interaction.

The Job table supports a job posting feature where users can create job listings. The relationship between Job and User is many-to-one, with each job post linked to the user who created it. The Job_Applicants table, which connects users with job postings they have applied for, implements a many-to-many relationship between the User and Job tables. This is achieved through a join table, capturing the dynamic nature of job applications.

Messaging between users is facilitated through the Message table, which stores the content of messages, timestamps, and sender-receiver relationships. This table has two many-to-one relationships with the User table, linking each message to its sender and receiver.

Lastly, the Notification table enhances user engagement by notifying users of various events like connection requests, likes, and comments. This table can be linked to multiple entities such as User, Post, and Connection, allowing it to deliver context-specific notifications based on the type of activity that triggered it.

Below we provide you the database diagram.



Database for BeConnected Server

Backend

We selected Spring Boot for our backend development due to its strong documentation and our professor's recommendation. In the next sections, we'll provide a brief overview of the main subsystems in our application's backend.

User

The User class defines the structure of the user entity within the application. It represents a user with attributes such as userId, username, email, password, and other personal details. This class also maintains collections of user experiences, education, and skills, providing a comprehensive profile for each user. The model supports relationships with other entities, such as connections and profile pictures, ensuring that user data is well-organized and easily accessible.

The system also includes a Picture entity that models user profile pictures. This class captures essential details such as pictureId, the actual image data, file name, and content type. The Picture Repository manages data access for this entity, while the Picture Service provides methods to delete pictures, maintaining clean and efficient image management within the application.

The User Repository interface is responsible for data access operations related to the User entity. It extends from a base repository to provide standard CRUD operations. In addition to these basic operations, it includes custom methods for querying users by username or email, searching users based on a query string while excluding the current user and administrators, and retrieving users who are not administrators. This repository ensures efficient and flexible data retrieval, catering to various application needs.

The User Service class handles business logic related to user operations. It interacts with the User Repository to perform tasks such as finding users by different criteria and throwing an exception if the user is not found. It includes methods for updating user details, such as updateUsername and updateEmail, which handle validation to ensure the uniqueness of the new values. Additionally, the User Service manages password changes through updatePassword, ensuring passwords are securely encoded before saving. For updating profile pictures, the updateProfilePicture method handles the integration with file handling. Overall, the service acts as the central point for executing user-related operations, ensuring that the application's business rules are applied correctly.

The User Controller class is designed to handle user-related operations through various RESTful API functions. It allows clients to interact with user data by invoking methods that correspond to specific operations.

The getUserById method retrieves user information by their unique identifier, while getUserByUsername fetches a user based on their username. For searching users, the searchUsers method leverages the UserService to find users that match a given query string, excluding the current user and administrators.

The controller also supports profile management. The getCurrentUser method returns details of the currently authenticated user by extracting their ID from the JWT token. The updateCurrentUser method enables users to modify their profile details with the changes being saved through the User Service.

To manage user-specific attributes like username, email, and password, the updateUsername, updateEmail, and updatePassword methods allow updates to these fields. Each of these methods verifies the user's identity using the JWT token and applies the updates accordingly.

For profile pictures, the getProfilePicture method retrieves an existing profile picture for a user, returning it with appropriate media type information. The updateProfilePicture method allows users to upload a new profile picture, handling file uploads and integration with the User Service to update the user's profile picture. The deleteProfilePicture method removes the current profile picture, updates the user's profile, and deletes the picture data through the Picture Service.

Connection

The Connection class is the entity that represents user connectivity within the application. Mapped to the connection table, the struct contains the most important details of a connection request. The connectionId is the only unique identifier, the requestedUser and the requestingUser are the users involved in the request. The status attribute is the comparer between the state of the connection (e.g. pending, accepted) and createdAt that is a date field storing the time the connection was started.

The Connection Repository interface has methods to find connections based on the users involved, such as getting connections where a user is either the requester or the requestee. In addition, it can also query connections according to their status, which is good for filtering out unnecessary connections and only showing the ones that are active or pending. The repository has other methods like to check if a connection between two users already exists or if it has a particular status.

The Connection Service class encapsulates the business logic associated with user connections. It has a method for creating a connection, either by creating a new request or by updating an existing one if it is still pending. The service also handles the acceptance or decline of connection requests, updating the status accordingly and managing notifications. For users seeking to manage their connections, the service provides methods to retrieve accepted connections and to remove existing ones. Furthermore, it can retrieve the pending connection requests, making it easier for the user to review their connecting requests and see the connection status.

The Connection Controller class facilitates user interaction with connection features through its RESTful API functions. It allows the user to receive a connection request and, in response, allows the user to either approve or decline the connection request. In addition to this, users can check a list of their accepted connections or view all the pending requests they have received. Besides that, the controller enables users to delete a connection if they want so.

Feed

The Post class models a social media post, representing user-generated content within the application. This class maps to the post table and includes attributes such as postId for the unique identifier, textContent for textual content, and mediaContent for any associated media, stored as a byte array. The mediaType attribute indicates the format of the media content. Each post is associated with an author (of type User), and the createdAt attribute records when the post was created. The class provides constructors for initializing post objects with various attributes.

The Like entity models user interactions with posts through likes. It captures attributes like likeId, the associated post, the user who liked the post, and the timestamp of the like. This entity facilitates tracking of user engagement with posts and supports relationships with both the Post and User entities.

The Comment entity represents user comments on posts, with attributes including commentId, the associated post, the user who made the comment, the comment text, and the timestamp of the comment. This class manages the relationship between posts and users, allowing for detailed tracking of user feedback on posts.

The Post Repository interface provides methods to find posts by author, ordered by creation date, and to retrieve posts by a list of author IDs. The repository supports querying posts by their ID and filtering posts authored by users in a given list, sorted by creation date. The Like Repository includes methods for finding likes based on post ID or user ID, and supports querying likes for a list of user IDs. The repository also provides a method to check if a specific user has liked a particular post. The Comment Repository interface handles data access for comments. It includes methods to retrieve comments by post ID or user ID and supports querying comments for a list of user IDs.

The Post Service class provides business logic for managing posts, likes, and comments. It includes methods for creating a post, finding posts by ID, and retrieving posts authored by a specific user. The service offers functionality to add comments and likes to posts, as well as to remove them. It also supports deleting posts and calculating relevance scores for recommending posts to users. The recommendation system uses matrix factorization to suggest posts based on user interactions and content similarity.

The Feed Controller class provides RESTful API endpoints for interacting with posts, comments, and likes. It allows users to create posts, including the option to upload media content through the createPost method. Users can retrieve their personalized feed by invoking the getFeedForCurrentUser function, which uses the JWT token to identify the current user and fetch relevant posts.

The controller also supports adding comments with the addComment method and liking posts via the likePost function. To handle media content associated with posts, the getMediaPost method retrieves the media data if it exists. Users can view comments and likes on posts through the getCommentsByPost and getLikesByPost methods, respectively.

For managing interactions, the controller includes functionality to remove likes with removeLike and delete comments through removeComment. The deletePost method allows users to delete their posts, provided they have the necessary authorization. Each of these actions is secured by JWT-based authentication, which is verified through the Authorization header in the request.

Notification

The Notification class models notifications in the application, linking each notification to a specific user and detailing its type, which could be a connection request, like, or comment. It captures the time when the notification was rendered and contains the comment, post or connection information.

The notification repository interface is responsible for the management of notifications through the use of methods which retrieve notifications based on the user, connection or post.

The Notification Service class implements the creation of notifications corresponding to connection request, likes, comments and allows to fetch notifications concerning the user activities. It also ensures users receive timely updates about interactions relevant to them.

The Notification Controller class provides RESTful endpoints to fetch notifications related to user connections, likes, and comments, ensuring that the retrieved data is pertinent to the authenticated user.

Job

The Job class represents a job posting within the system. It features various attributes, including an ID, title, description, the user who created the job, the creation date, and a set of applicants. It allows for adding and removing applicants through its methods, and it maintains a boolean flag to indicate whether the job is active.

The Job Repository provides methods to fetch jobs based on their active status or the username of the user who created them. These methods enable efficient querying of job listings from the database.

The Job Service class provides business logic for handling job-related operations. It includes methods to retrieve all jobs, fetch jobs by their ID, and get jobs based on a user's username. It also supports job creation and deletion. Additionally, it allows users to apply for and remove their applications from jobs. The service calculates job relevance scores using cosine similarity and matrix factorization techniques to recommend jobs to users based on their skills and bio.

The Job Controller class exposes RESTful API endpoints for job management. It provides functionality to create a job, retrieve jobs for the current user, delete a job if it was created by the user, and handle job applications. Each endpoint ensures that the appropriate user is authenticated via JWT tokens and performs the necessary actions while adhering to authorization rules.

Chat

The Message class represents a communication record between two users in the system. It includes attributes such as a unique message ID, sender and receiver user entities, the content of the message, and a timestamp indicating when the message was sent. The class is designed to facilitate the creation of new messages and maintain a record of interactions between users.

The Message Repository provides methods to query messages and identify users who have interacted with a specific user. It includes custom queries to find distinct chatted user IDs and to retrieve messages based on sender and receiver relationships. This repository supports efficient retrieval and management of message data.

The Message Service class encapsulates the business logic related to messaging functionalities. It provides methods to obtain a list of users with whom a specific user has previously communicated, send messages between users, and fetch the complete conversation history between two users. It handles user lookups and ensures that messages are persisted correctly. The service ensures transactional integrity when sending messages and processes message retrieval by combining and sorting message records.

The Message Controller class exposes RESTful API endpoints for managing messages. It offers functionalities to get a list of users with whom the current user has communicated, send messages to other users, and retrieve message conversations between the current user and another specified user. Each endpoint ensures that operations are performed securely by validating the JWT token to identify the authenticated user and applying appropriate authorization checks.

Admin

The AdminService class is designed to provide administrative functionalities for managing user data within the system. It includes methods for retrieving all users while excluding administrators and exporting user data in various formats. The exportUsersDataByIds method allows for exporting detailed information about users based on their IDs and supports multiple formats such as JSON and XML. It employs reflection to gather user data, including their profile picture, posts, connections, jobs, liked posts, and commented posts. The service handles errors gracefully and logs issues related to data export and field access. Additionally, the convertUserToDetailedMap method organizes user-related data into a map, which is then used for exporting purposes.

The AdminController class exposes REST API endpoints for administrative operations related to user management. It provides endpoints to retrieve all users and export user data by their IDs. The getAllUsers method returns a list of all non-administrator users, while the exportUsersDataByIds method allows exporting user data in specified formats, handling exceptions by returning an appropriate error message. The controller ensures that administrative tasks are performed through secure and efficient API calls, leveraging the functionalities provided by AdminService.

Recommendation System

Our recommendation system combines Matrix Factorization and Cosine Similarity to deliver personalized content, like posts and job listings, to users. Matrix factorization is at the central of our approach, where we predict user preferences by breaking down a user-item interaction matrix into two smaller matrices: one for users and another for items. These matrices capture the hidden factors that influence what users like.

The matrix factorization process is iterative, adjusting these matrices over several rounds to reduce the gap between predicted and actual preferences. We use key parameters like the number of features, learning rate, and regularization to fine-tune the process and prevent overfitting. The algorithm stops once the error falls below a certain level, ensuring we don't waste time on unnecessary calculations.

We also use cosine similarity to give each user-item pair an initial relevance score based on text content. For posts, this means comparing user bios, skills, and past interactions with the post's content. For jobs, it matches the user's skills and bio against job descriptions and titles. We even adjust the relevance score based on factors like how recent the post is or whether the user has already interacted with it.

Frontend

The frontend plays a vital role in BeConnected, being the primary and most convenient means for users to engage with the application. It's essential that the site is visually appealing and offers a user-friendly, interactive interface to ensure easy access to all the available features. To achieve this, the front-end was developed using the React framework, chosen for its widespread popularity, extensive documentation, and versatility. We won't dive into the frontend design as deeply as the backend since it mainly involves a mix of CSS and JavaScript for interactivity and server communication.

Api

In API file we set up a client using the Axios library to handle user authentication and interactions. We implemented an Axios instance with a base URL and configured an interceptor to handle token refresh when a 401 error occurs. If the access token expires, we retrieve the refresh token, request a new access token, and retry the original request. We also handled user actions like login, registration, logout, profile updates, messaging, post interactions, and job management, ensuring that each request includes the necessary authentication tokens and handles errors gracefully.

Components

Our components organize the user interface and manage interactions within the web application. We handle data retrieval, state management, and user input to display dynamic content such as user profiles, connections, and feeds. By doing this, we ensure that the application remains both functional and user-friendly.

Conclusion

Working on BeConnected has been a valuable journey, containing a variety of challenges as well as lessons learned. We began by setting up the fundamental architecture using Spring Boot for the backend and React for the frontend. Implementing secure authentication mechanisms with JWTs and managing user sessions was a critical first step. As we progressed, we focused on the core functionalities, including user connections, messaging, and profile management, and tested these features using manual Postman HTTP requests. At the same time, we worked on building the frontend with React and ensuring it interacted efficiently with the server. We also used Git and GitHub for version control, utilizing branches, pull requests, and code reviews to maintain code quality and stay updated with each other's work.

The biggest challenge we encountered during the project was setting up security. At first, defining and implementing JWT tokens for authentication was tough since we weren't familiar with them. Additionally, we struggled with implementing security filters for authorization. Another major challenge was displaying profile pictures and media on the frontend pages. We resolved this issue by setting up secure endpoints to deliver the media bytes and generating object URLs for them. Despite these challenges, the project went smoothly overall.