

# National and Kapodistrian University of Athens

Department of Informatics and Telecommunications



**E-commerce technologies - Assignment final report**

**Summer semester 2023-2024**

***Worknet***

|           |                          |
|-----------|--------------------------|
| Student 1 | Syrios Konstantinos-Zois |
|           | 7115112300035            |
| Student 2 | Tsiompikas Dimitris      |
|           | 7115112300036            |

## Table of Contents

|   |           |
|---|-----------|
| <b>Introduction.....</b>                  | <b>3</b>  |
| <b>Initial UML diagram.....</b>           | <b>4</b>  |
| <b>Backend Architecture.....</b>          | <b>5</b>  |
| User.....                                 | 5         |
| Comment.....                              | 5         |
| Custom File.....                          | 5         |
| Education.....                            | 5         |
| Work Experience.....                      | 5         |
| Skill.....                                | 6         |
| Job.....                                  | 6         |
| Like.....                                 | 6         |
| Message.....                              | 6         |
| Notification.....                         | 6         |
| Notification Type.....                    | 6         |
| Post.....                                 | 6         |
| View.....                                 | 6         |
| Employment Type.....                      | 7         |
| Workplace Type.....                       | 7         |
| <b>File Saving System.....</b>            | <b>7</b>  |
| <b>Security.....</b>                      | <b>8</b>  |
| <b>Recommendation System.....</b>         | <b>14</b> |
| <b>Android Frontend Architecture.....</b> | <b>20</b> |
| Entering Worknet.....                     | 20        |
| Main Activity.....                        | 21        |
| Home Fragment.....                        | 23        |
| Network Fragment.....                     | 23        |
| New Post Fragment.....                    | 24        |
| Notifications Fragment.....               | 24        |
| Job Postings Fragment.....                | 25        |
| Messages Fragment.....                    | 26        |
| Settings.....                             | 27        |
| Profile.....                              | 28        |
| Education Fragment.....                   | 29        |
| Skills Fragment.....                      | 29        |
| Work Experience Fragment.....             | 29        |
| Dynamic UI entries.....                   | 29        |
| Example Service call.....                 | 30        |
| <b>Installation.....</b>                  | <b>31</b> |
| <b>Conclusion.....</b>                    | <b>32</b> |
| <b>Bibliography.....</b>                  | <b>33</b> |

# Introduction

In this report, we will describe the architecture and decisions we took in order to implement the mobile application for this course's assignment.

The application's name is **Worknet** and it functions as a social networking app for professionals who want to apply to jobs, see posts relevant to their job, or talk to others and increase their networking.

The application was developed with the following tools:

- Java Spring Boot for the backend
- Android Studio (Java) for the frontend

We began by creating an initial UML diagram, as presented in the first chapter of the report, to get an initial idea of the application's architecture and to start building the backend system. With this initial architecture in mind, we started implementing the backend which evolved during development, by removing or adding classes and relationships. In the second chapter there will be a detailed overview of the backend.

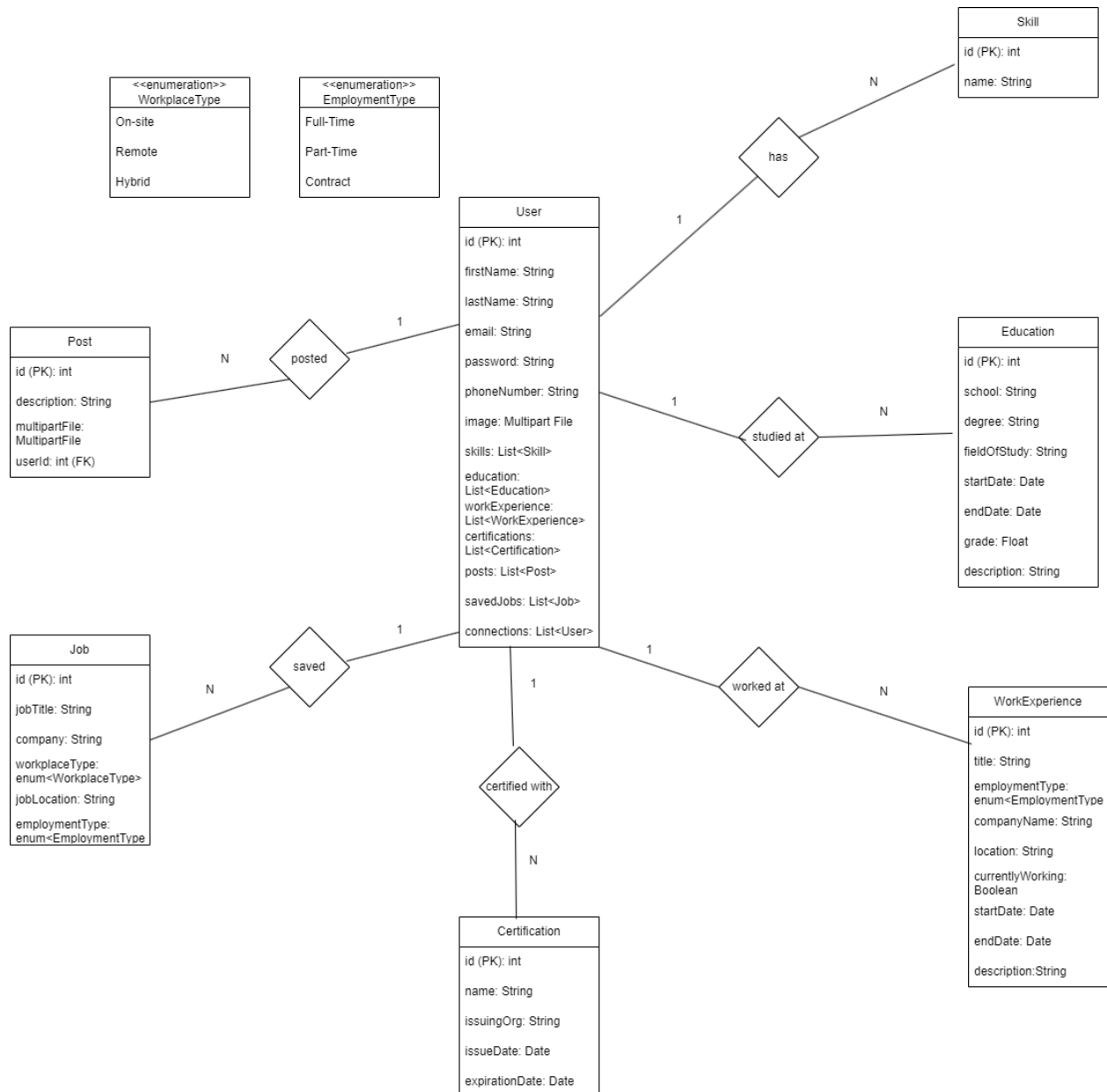
In the third chapter, we present our file saving system, and in the fourth chapter we describe the implementation of the required security features, in order to have TLS and SSL for both the backend and the frontend. In the fifth chapter, we describe the recommendation system that was added on the backend side, which is used through service calls to get the desired results.

In the sixth chapter, we will see the frontend design and implementation which uses android activities and fragments, in order to provide an easy-to-use and robust user interface.

In the last and seventh chapter, we provide details about our application's installation.

We would like to note that in order to have more time for testing and bug-fixing, we began to implement both the frontend and backend concurrently. Initially, the frontend part would have only the UI implemented, and the various services were added later on, as the backend implementation was progressing.

## Initial UML diagram



Above you can see the first version of the UML diagram, which was used to conceive the initial form of our backend's architecture. The *User* class is the main class, which describes the professionals that will use the application. Its fields serve both the Login/Logout/Register parts of the application (with the email and password fields), and of course the Profile page of each user, where other users and themselves can see or add information about their education, work experience, and skills.

The certification class was removed as it was deemed out of the scope of this assignment.

## Backend Architecture

The backend architecture, as mentioned before, was created with Java Spring Boot, and for the Database we used MySQL and Hibernate. We created the classes (except Certifications) as seen in the above UML diagram and added some new classes as needed. Each class is an entity (apart from the Enums), and each entity is represented in the frontend with a **Data Transfer Object** (DTO).

DTOs allow us to show only certain fields to the frontend through service calls, minimizing the need to always have all fields present. The overall scheme is the classic backend architecture used with Spring Boot, meaning that each entity has a JPA repository, a Service and a Controller for transferring data to the frontend part.

Below, we'll be describing each class that was implemented and what it offers to the system:

### User

The User class represents the professionals who want to connect, apply to jobs and create posts in the application. It offers fields (email, password, jwtToken) for logging in, logging out and registration while also offering all required information for the Profile page and Network page (examples being education, work experience, skills profilePicture etc).

### Comment

The comment class is used to show the comments in user created posts.

### Custom File

The custom file class represents the files used for profile pictures and the files uploaded with user posts.

### Education

The Education class represents the users' education entries (university degrees, high schools, etc.)

### Work Experience

The WorkExperience class is used to represent work experience entries in user profiles.

## **Skill**

The skill class is used to show the user's skill entries on the profile page for each user and in job postings.

## **Job**

The Job class is used to hold the job postings of users.

## **Like**

The Like class represents likes added by users to user created posts.

## **Message**

The message class represents messages sent by users to other users.

## **Notification**

The notification class is used to show notifications received through certain user actions which will be described in the associated Enum below.

## **Notification Type**

This enum has the following types:

- CONNECTION,
- APPLY\_TO\_JOB\_POST,
- LIKE\_POST,
- MESSAGE, and
- COMMENT.

All are used for notifications, regarding their purpose.

## **Post**

The post class is used to represent posts created by users that appear on the front page.

## **View**

The view class is used to count the number of times a job posting has been viewed. This is used in the recommendation system later on.

## Employment Type

This enum has the following types:

- FULL\_TIME,
- PART\_TIME,
- CONTRACT.

It is used in job postings and work experience entries.

## Workplace Type

This enum has the following types:

- ON\_SITE,
- REMOTE,
- HYBRID.

It is used in job postings.

## File Saving System

Since we discussed the concept of the File Transfer Protocol in class, we thought of using a folder functioning as an “SFTP”. This folder is called “FileStorage”, and all profile images and post files are saved there.

However, due to Android’s limitations security-wise, more specifically android studio is not allowed to look for PC folders even with external storage permissions, the idea was scrapped.

The initial plan was to save the file paths in the database as it was shown in the course labs, however due to the above difficulties we saved a LONGBLOB object, after we compressed it to hold less space in the database, and we decoded it in the frontend part to use the files.

The folder still exists however, so that the endpoints can save the files somewhere and show the naming conventions used for files. Images get saved with the following naming convention: *<user id>\_profile\_picture*, and files uploaded with user created posts are saved as: *post\_<post id>\_<type of file>\_<file counter>*, with the type being either image, audio, or video, and the file counter representing how many of these similar files we have in the same post.

# Security

The security layer of the project was implemented in both the backend and frontend parts of the application, in order to accommodate the usage of HTTPS and TLS handshakes.

For the backend part we created the following configuration file:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.authorizeHttpRequests(auth -> {
            auth.requestMatchers(...patterns:"/**").permitAll();
            auth.anyRequest().authenticated();
        }).csrf(AbstractHttpConfigurer::disable)
        .build();
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    JwtGenerator generator(AuthenticationConfiguration authenticationConfiguration) throws Exception {
        return new JwtGenerator(authenticationManager(authenticationConfiguration));
    }
}
```

The file contains the security filter chain, which is newly added in Spring Boot 3 and is the new non-deprecated practice to use for security.

We disabled CSRF to enable HTTPS calls with postman, and since we do not have any roles for this application, anyone who gets authenticated will be able to use all endpoints.

The config file also contains an authentication manager for login/logout, a BCrypt password encoder for encoding the passwords in the database and a JWT generator used for generating the JWTs used later on for keeping the user logged in the app.



```

@PostMapping("/login")
public ResponseEntity<> loginUser(@RequestBody LoginUserDTO loginUserDTO) {
    User user = userService.getUserByEmail(loginUserDTO.getEmail());
    if (user == null){
        return new ResponseEntity<>(body:"User not found. Try other credentials.", HttpStatus.NOT_FOUND);
    }

    String token = jwtGenerator.generateToken(loginUserDTO);

    user.setJwtToken(token);
    userService.updateUser(user.getId(), user);

    return new ResponseEntity<>("User logged in successfully. Bearer " + token, HttpStatus.OK);
}

```

**Login** searches for the user in the database via email and, if the user exists, a JWT is generated for the user. It is then added to the respective field and persisted to the database. The mobile application later on searches for the JWT and, if it is still in the database, it will remember the latest logged in user and open that user's main page even if the application closes and gets reopened.

```

Tsiobieman +2
@Override
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    // jwt token is added for authorization purposes.
    // email is added so that the app can "remember" the user's email if they close the app
    // without logging out so their data can be persisted and re-shown on the app.
    SharedPreferences sharedPreferences = getSharedPreferences( name: "MyAppPrefs", MODE_PRIVATE);
    String jwtToken = sharedPreferences.getString( key: "jwt_token", defValue: null);
    String email = sharedPreferences.getString( key: "email", defValue: null);

    if (jwtToken != null){
        Intent intent = new Intent( packageContext: this, MainActivity.class);
        intent.putExtra(getString(R.string.e_mail), email);
        finishAffinity();
        startActivity(intent);
    }else{
        setContentView(R.layout.activity_start);

        buttonRegister = findViewById(R.id.buttonRegister);
        buttonLogin = findViewById(R.id.buttonLogin);

        SetupButtonListeners();
    }
}

```

```

@PostMapping("/register")
public ResponseEntity<> registerUser(@RequestBody RegisterUserDTO registerUserDTO) {
    User user = userService.getUserByEmail(registerUserDTO.getEmail());
    if (user != null) {
        return new ResponseEntity<>("user with email: " + user.getEmail() + " already exists.", HttpStatus.BAD_REQUEST);
    }

    User newUser = new User();
    newUser.setPassword(passwordEncoder.encode(registerUserDTO.getPassword()));
    newUser.setEmail(registerUserDTO.getEmail());
    newUser.setFirstName(registerUserDTO.getFirstName());
    newUser.setLastName(registerUserDTO.getLastName());
    newUser.setPhoneNumber(registerUserDTO.getPhoneNumber());

    userService.addUser(newUser);
    return ResponseEntity.status(HttpStatus.CREATED).body("User registered successfully!");
}

@GetMapping("/logout")
public ResponseEntity<> logoutUser(@RequestParam String email) {
    User user = userService.getUserByEmail(email);
    if (user == null) {
        return new ResponseEntity<>("User does not exist.", HttpStatus.BAD_REQUEST);
    }

    user.setJwtToken(jwtToken:null);
    userService.updateUser(user.getId(), user);

    return new ResponseEntity<>("User logged out successfully.", HttpStatus.OK);
}

```

In the screenshot above, the register and logout functionalities are shown. Register gets a RegisterUserDTO and fills it with the data obtained from the frontend, allowing for a new user to be created in the database.

Logout finds the user given as a parameter by email, then the JWT is being set to null so the frontend check can bring back the start activity and the user gets updated to be persisted in the database.

```

## Spring security config (TLS)
server.port = 8443

server.ssl.enabled=true
server.ssl.key-store=worknet.jks
server.ssl.key-store-password=worknet
server.ssl.key-store-type=PKCS12
server.ssl.key-alias=worknet_ssl

```

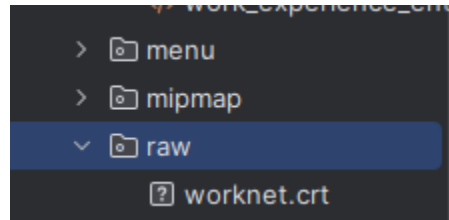
```

🔒 worknet.crt
📁 worknet.jks

```

There have also been changes in the *application.properties* file. The server port was changed to 8443 for HTTPS, since 8080 is used by HTTP. We created a keystore called *worknet.jks* and a certificate called *worknet.crt* (by following instructions from the web and the course labs), in order to be able to call endpoints with HTTPS and ensure the TLS handshake.

For the frontend part, in order to be able to use the services with the android virtual device (AVD) we also used the certificate on the android part:



We also created this XML file called *network\_security\_config* which is also needed for the same purpose and detects the certificate:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <base-config>
    <trust-anchors>
      <certificates src="@raw/worknet"/>
      <certificates src="system"/>
    </trust-anchors>
  </base-config>
</network-security-config>
```

```

public class RetrofitService {

    // Created custom Trust manager in order to bypass hostname not verified error.
    // common practice in Android dev - used only for dev purposes. SSL/TLS handshake still stands.
    // Retrofit is used to connect the android app with the backend endpoints in the Spring Boot app.
    // Tsiobieman +1
    public static Retrofit getRetrofitInstance(Context context) {
        try {
            // Load CAs from an InputStream
            CertificateFactory cf = CertificateFactory.getInstance("X.509");

            // Load the certificate from res/raw
            InputStream caInput = context.getResources().openRawResource(R.raw.worknet);
            Certificate ca;
            try {
                ca = cf.generateCertificate(caInput);
            } finally {
                caInput.close();
            }

            // Create a KeyStore containing the trusted CAs
            String keyStoreType = KeyStore.getDefaultType();
            KeyStore keyStore = KeyStore.getInstance(keyStoreType);
            keyStore.load(stream: null, password: null);
            keyStore.setCertificateEntry(alias: "ca", ca);

            // Create a TrustManager that trusts the CAs in the KeyStore
            String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
            TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
            tmf.init(keyStore);

            SSLContext sslContext = SSLContext.getInstance(protocol: "TLS");
            sslContext.init(km: null, tmf.getTrustManagers(), random: null);

            // Hostname verifier to avoid the error.
            OkHttpClient client = new OkHttpClient.Builder()
                .sslSocketFactory(sslContext.getSocketFactory(), (X509TrustManager) tmf.getTrustManagers()[0])
                // Tsiobieman
                .hostnameVerifier(new HostnameVerifier() {
                    // Tsiobieman
                    @Override
                    public boolean verify(String hostname, javax.net.ssl.SSLSession session) {
                        return hostname.equals("10.0.2.2");
                    }
                })
                .build();

            // Build Retrofit instance
            return new Retrofit.Builder()
                .baseUrl("https://10.0.2.2:8443/")
                .client(client)
                .addConverterFactory(ScalarsConverterFactory.create()) // used for running Strings from calls. (
                .addConverterFactory(GsonConverterFactory.create())
                .build();

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

The above code is used for the configuration of Retrofit. Retrofit is used to call the services from the backend. The IP used is 10.2.2.2:8443 to hit the HTTPS endpoints, and a custom trust manager had to be created in order to avoid the *hostname not verified* error. *Custom Converter Factories* had to be created for the conversion from JSON to GSON for android. *Scalar Converter Factory* is used for calls that have `Call<String>` as the return type.

```

<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.READ_MEDIA_IMAGES" />
<uses-permission android:name="android.permission.READ_MEDIA_VISUAL_USER_SELECTED" />
<uses-permission android:name="android.permission.READ_MEDIA_AUDIO" />
<uses-permission android:name="android.permission.READ_MEDIA_VIDEO" />

<application
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/worknet_label"
    android:networkSecurityConfig="@xml/network_security_config"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportRtl="true"
    android:theme="@style/Theme.Worknet"
    tools:targetApi="31">

```

We have also added special configurations in AndroidManifest.xml as well, the first two *uses-permission* entries are there for HTTPS calls, and the rest are for accessing folders in the android device in order to get files for posts and profile pictures.

# Recommendation System

The recommendation system was built using the algorithm of Matrix Factorization with Gradient Descent for optimization. We used the course's slides as inspiration and we used [1] and [2] as extra resources.

The recommendation system was built on the backend and it is called via service on the frontend using an endpoint. For the assignment we had to implement two recommendation algorithms. One with Matrix Factorization using user views on job postings, from users that are connected to the user who we want to recommend postings to, and one algorithm that takes the skills of a user and recommends job postings based on them.

We implemented the RecommendationSystem class as follows:

```
private double[][] P; 8 usages
private double[][] Q; 8 usages
private int[][] userJobMatrix; 4 usages

public void createInteractionMatrix(List<User> users, List<Job> jobs) {
    int numUsers = users.size();
    int numJobs = jobs.size();

    this.userJobMatrix = new int[numUsers][numJobs];

    // Create mappings from IDs to indices
    Map<Long, Integer> userIdToIndex = new HashMap<>();
    Map<Long, Integer> jobIdToIndex = new HashMap<>();

    // Populate the userIdToIndex map
    for (int i = 0; i < numUsers; i++) {
        userIdToIndex.put(users.get(i).getId(), i);
    }

    // Populate the jobIdToIndex map
    for (int i = 0; i < numJobs; i++) {
        jobIdToIndex.put(jobs.get(i).getId(), i);
    }

    // Populate user job interaction matrix
    for (User user : users) {
        for (Job job : jobs) {
            Long userId = user.getId();
            Long jobId = job.getId();
            int viewCount = user.countViewsForJob(job.getId());

            // Get the mapped indices
            int userIndex = userIdToIndex.get(userId);
            int jobIndex = jobIdToIndex.get(jobId);

            this.userJobMatrix[userIndex][jobIndex] = viewCount;
        }
    }
}
```

The above method initializes the user job interaction matrix. The table has *numberOfUsers* \* *numberOfJobs* dimensions, and each cell contains the views each user has for a job. The users in this case are connections of the user who we want to recommend job postings to.

```

public void matrixFactorization(int numUsers, int numJobs, int numFactors) { 1 usage 2 Tsobleman +1
    this.P = new double[numUsers][numFactors];
    this.Q = new double[numJobs][numFactors];

    initializeMatrix(this.P);
    initializeMatrix(this.Q);

    double learningRate = 0.01;
    double lambda = 0.1;
    int numIterations = 100;

    for (int iter = 0; iter < numIterations; iter++) {
        for (int u = 0; u < numUsers - 1; u++) {
            for (int j = 0; j < numJobs; j++) {
                if (this.userJobMatrix[u][j] > 0) {
                    double predictedRating = dotProduct(this.P[u], this.Q[j]);
                    double error = this.userJobMatrix[u][j] - predictedRating;
                    for (int k = 0; k < numFactors; k++) { // implemented gradient descent to improve results
                        this.P[u][k] += learningRate * (error * this.Q[j][k] - lambda * this.P[u][k]);
                        this.Q[j][k] += learningRate * (error * this.P[u][k] - lambda * this.Q[j][k]);
                    }
                }
            }
        }
    }
}

```

The above method handles the matrix factorization.

Here, we initialize the P and Q tables with *numberOfUsers \* numberOfFactors*, and *numberOfJobs \* numberOfFactors* dimensions respectively.

We initialized the hyperparameters with the shown values because we believe that with the current system these are sufficient. However, even if the database was scaled to handle large amounts of data, these hyperparameters would still give good results.

The initial part of the algorithm happens when the *predictedRating* is found. Everything else after that is the Gradient Descent method that we use to optimize the returned results.

We could have also used Social Regularization here probably by using the connections, however it was deemed unnecessary due to the small complexity of the data.

Below, the first method returns the top N recommendations for our user, and the second method returns job postings based on the user's skills:

```
private static List<Integer> getTopNRecommendations(double[] scores, int N) { 1 usage  ▲ Tsiobleman +1
    PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> Double.compare(scores[b], scores[a]));
    for (int i = 0; i < scores.length; i++) {
        pq.offer(i);
    }

    List<Integer> recommendations = new ArrayList<>();
    for (int i = 0; i < N; i++) {
        if (!pq.isEmpty()) {
            recommendations.add(pq.poll());
        }
    }
    return recommendations;
}

public HashSet<Job> recommendJobsBySkill(User user, List<Job> jobs) { 1 usage  ▲ dits +1
    List<Skill> userSkills = user.getSkills();
    HashSet<Job> recommendedJobs = new HashSet<Job>(); // hash set to remove duplicates.

    Set<String> userSkillNames = new HashSet<>();
    for (Skill userSkill : userSkills) {
        userSkillNames.add(userSkill.getName().trim().toLowerCase()); // case insensitive skill search.
    }

    for (Job job : jobs){
        List<Skill> individualJobSkills = job.getSkills();
        for (Skill skill : individualJobSkills){
            if (userSkillNames.contains(skill.getName().trim().toLowerCase())) {
                recommendedJobs.add(job);
            }
        }
    }

    return recommendedJobs;
}
```



The last two methods are the dot product between two vectors, and the *initializeMatrix* method which is used by P and Q tables to initialize them with random double values and use them later on for Matrix Factorization.

```
private static double dotProduct(double[] vec1, double[] vec2) {
    double result = 0.0;
    for (int i = 0; i < vec1.length; i++) {
        result += vec1[i] * vec2[i];
    }
    return result;
}

private void initializeMatrix(double[][] matrix) {
    Random random = new Random();
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++) {
            matrix[i][j] = random.nextDouble();
        }
    }
}
```

Below, we show the controller method for job recommendations. Starting off, we get the user's connections and skills, we get all jobs from the database and we remove all jobs created by the user who will get the recommendation.

```
public ResponseEntity<?> recommendJobs(@RequestParam Long userId) {
    User user = userService.getUserById(userId);
    if (user == null) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body("User with ID " + userId + " not found.");
    }

    List<User> connections = user.getConnections();
    List<Skill> userSkills = user.getSkills();
    List<Job> jobs = jobService.getAllJobs(); // used for recommendation by skills

    // remove jobs created by the user who will get the recommended jobs.
    Iterator<Job> iterator = jobs.iterator();
    while (iterator.hasNext()) {
        Job job = iterator.next();
        if (job.getJobPoster().getId() == userId) {
            iterator.remove();
        }
    }

    List<JobDTO> recommendedJobDTOS = new ArrayList<>();
}
```

Below, we show the recommendation by connections (Matrix Factorization):

```
// recommendation by matrix factorization on user's connections
if (!connections.isEmpty()) {
    List<View> connectionViews = new ArrayList<>();
    for (User connection : connections) {
        connectionViews.addAll(connection.getViews());
    }

    HashSet<Job> noDupesJobs = new HashSet<>(); // remove duplicates
    for (View connectionView : connectionViews) {
        noDupesJobs.add(jobService.getJobById(connectionView.getJob().getId()));
    }

    List<Job> connectionJobs = new ArrayList<>(noDupesJobs);

    RecommendationSystem recommendationSystem = new RecommendationSystem();

    recommendationSystem.createInteractionMatrix(connections, connectionJobs);

    connections.add(user); // add this to get results later on.

    recommendationSystem.matrixFactorization(connections.size(), connectionJobs.size(), numFactors: 10);

    List<Job> recommendedJobs = recommendationSystem.getRecommendedJobs(user, connectionJobs);

    // turn them to DTOs
    for (Job recommendedJob : recommendedJobs) {
        JobDTO recommendedJobDTO = modelMapper.map(recommendedJob, JobDTO.class);
        recommendedJobDTOs.add(recommendedJobDTO);
    }

    // remove jobs created by the user who will get the recommended jobs.
    Iterator<JobDTO> connectionIterator = recommendedJobDTOs.iterator();
    while (connectionIterator.hasNext()) {
        JobDTO job = connectionIterator.next();
        if (job.getJobPoster().getId() == userId) {
            connectionIterator.remove();
        }
    }
}
```

We get the user's connections and the jobs they've viewed. We, then, remove duplicate jobs and the jobs created by the user who will get the recommendations. The recommendation system gets initialized, the interaction matrix is created with the connections and jobs, then the matrix factorization happens, and lastly we add the jobs to the list so they can be returned.

Lastly we show the recommendation by skills. We simply get the user's skills and add jobs that have said skills to the list. If the list is empty (user has no connections or skills) we show a respective text on the front end otherwise we show the jobs.

```
// recommendation by user skills
if (!userSkills.isEmpty()) {

    RecommendationSystem recommendationSystem = new RecommendationSystem();

    HashSet<Job> recommendedJobs = recommendationSystem.recommendJobsBySkill(user, jobs);

    for (Job recommendedJob : recommendedJobs) {
        JobDTO recommendedJobDTO = modelMapper.map(recommendedJob, JobDTO.class);
        recommendedJobDTOs.add(recommendedJobDTO);
    }
}

List<JobDTO> uniqueRecommendedJobDTOs = recommendedJobDTOs.stream().distinct().toList();

return ResponseEntity.ok(uniqueRecommendedJobDTOs);
}
```

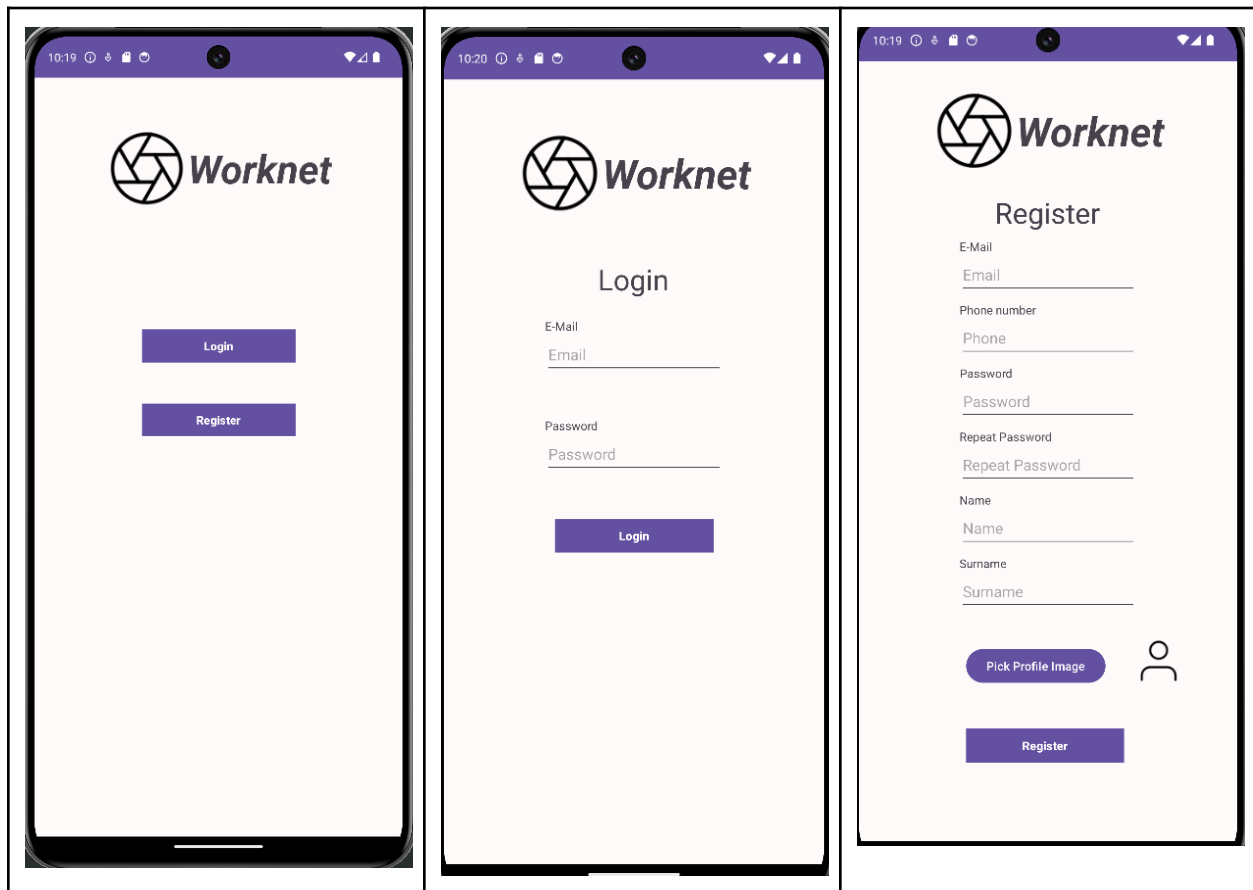
## Android Frontend Architecture

According to the assignment's requirements, **Worknet** is a mobile application for Android. It was developed using *Android Studio Jellyfish Edition (2023.3.1 Patch 1)* with minimum SDK: *API 34 ("UpsideDownCake"; Android 14.0)*.

The application's UI was structured using both android activities and fragments according to our needs. For example, a "main" activity was used as a central hub, which swapped between the available fragments for the various functionalities of the application. We used fragments for most "screens"/functionalities of our application, as it was deemed that the overhead of creating and finishing new activities for those functionalities was unnecessary.

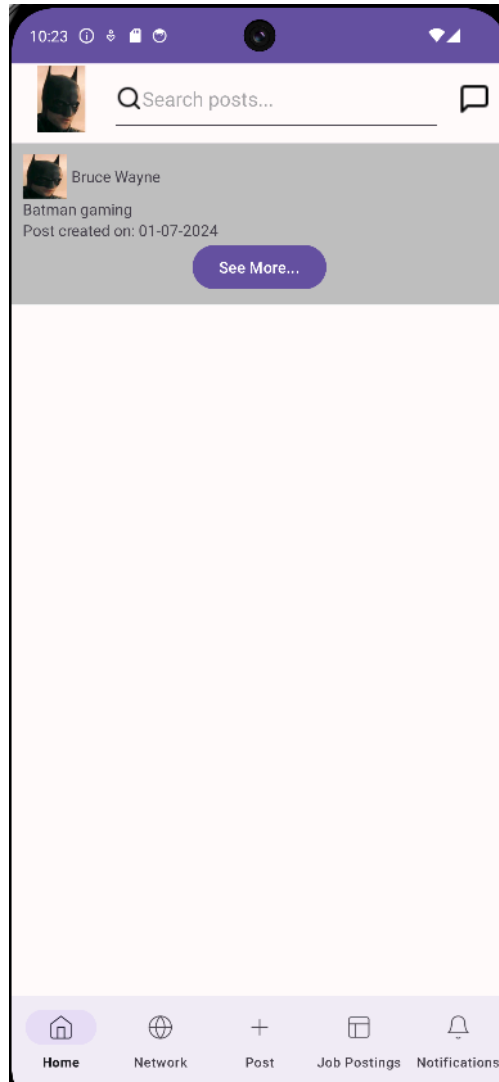
## Entering Worknet

When opening the application the user is presented with the choice of logging in the app or registering for an account. This sequence is implemented using activities which are all finished when the user successfully enters the application using a valid account.



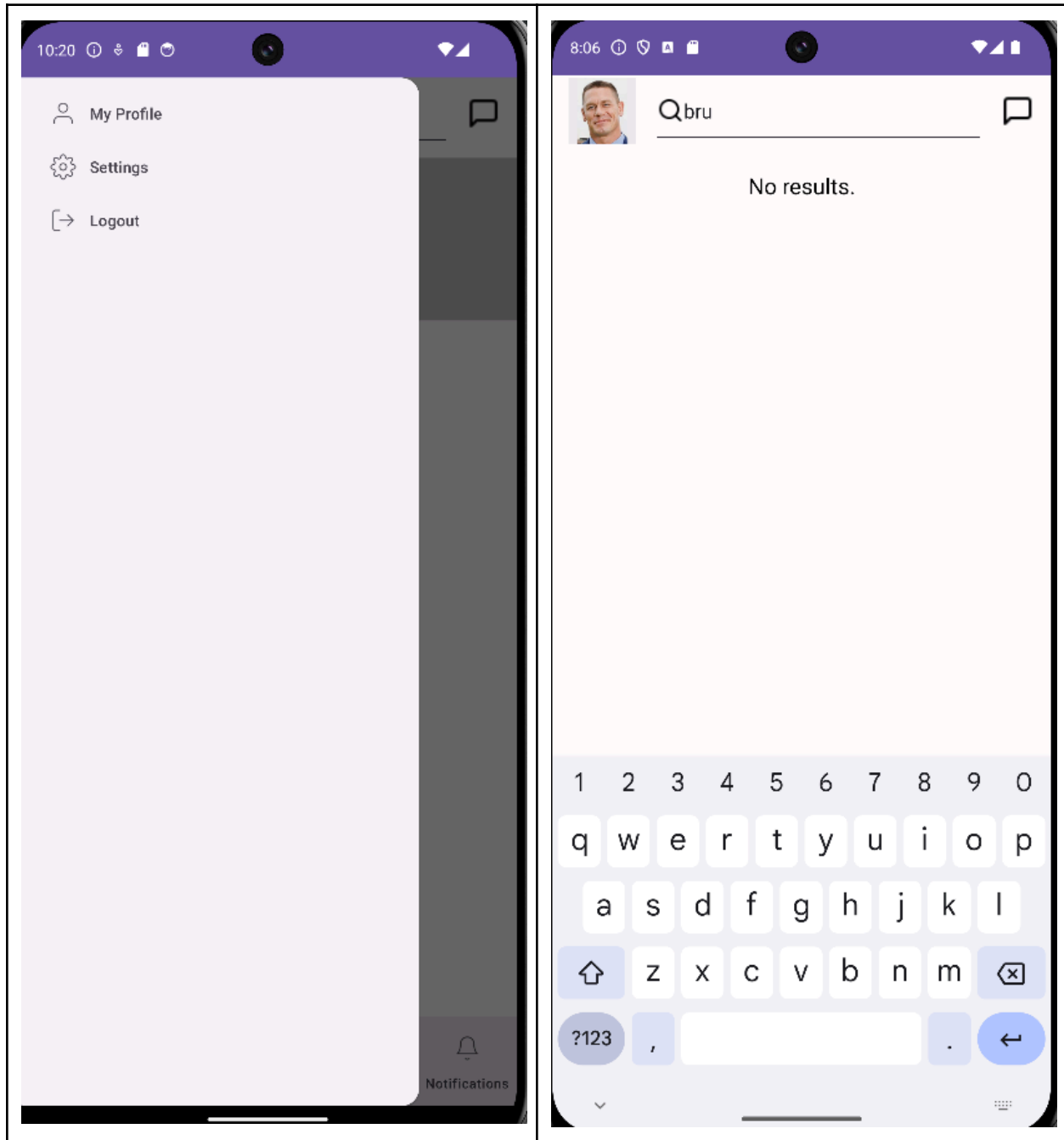
## Main Activity

As previously mentioned, the central hub of our application is the “main” activity. It’s the activity that starts after a user logs in, and initializes with the **home** fragments which shows relevant posts. Part of its UI is divided into a top bar and bottom bar for navigating the application.



The bottom navigation bar contains five buttons that can swap between the **home**, **network**, **new post**, **job postings**, and **notifications** fragments.

For the top bar, the user can click on their profile picture, which will open the side menu which contains buttons to navigate to the user's profile and settings, or logout. The top bar also contains a search bar that can be used to search for posts or users depending on the context (i.e. which fragment is displayed), and a button to navigate to the user's chats.

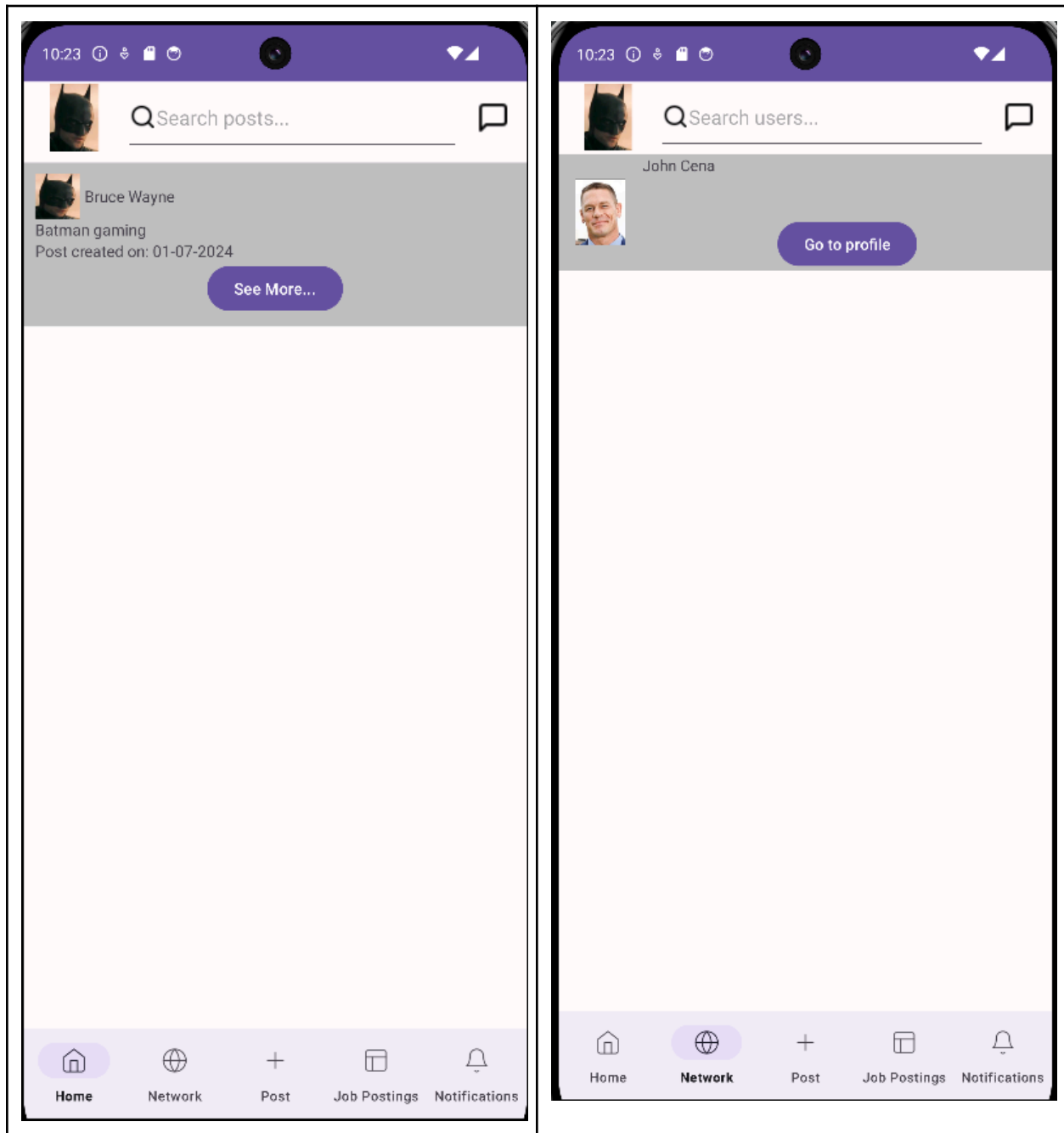


## Home Fragment

This fragment displays posts that are relevant to the user.

## Network Fragment

The network fragment displays a list with the other users that the currently logged in user is connected with. The user has the choice to navigate to their profiles if they want.

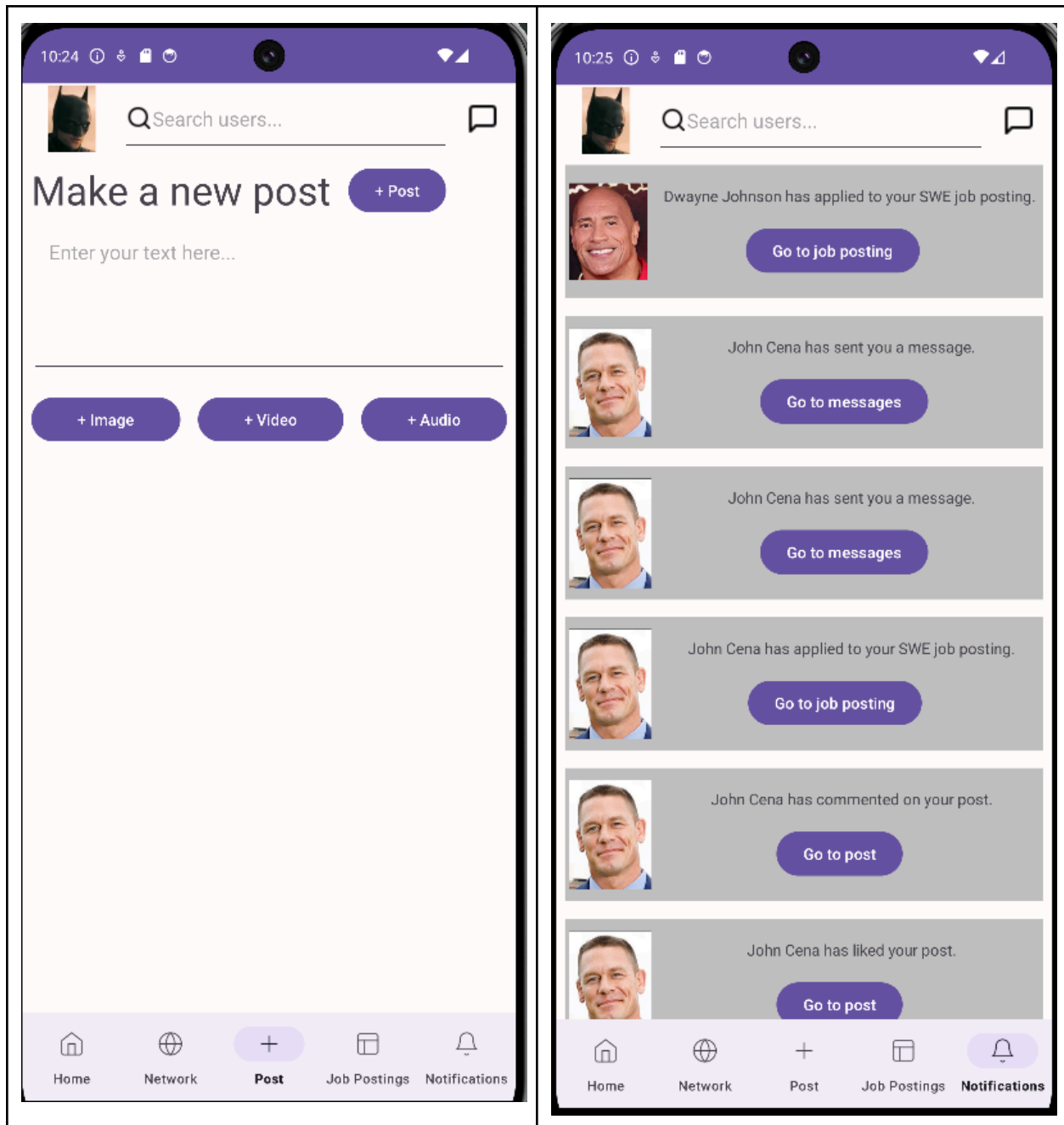


## New Post Fragment

In this fragment the user can create a new post and the media they want to upload from their device.

## Notifications Fragment

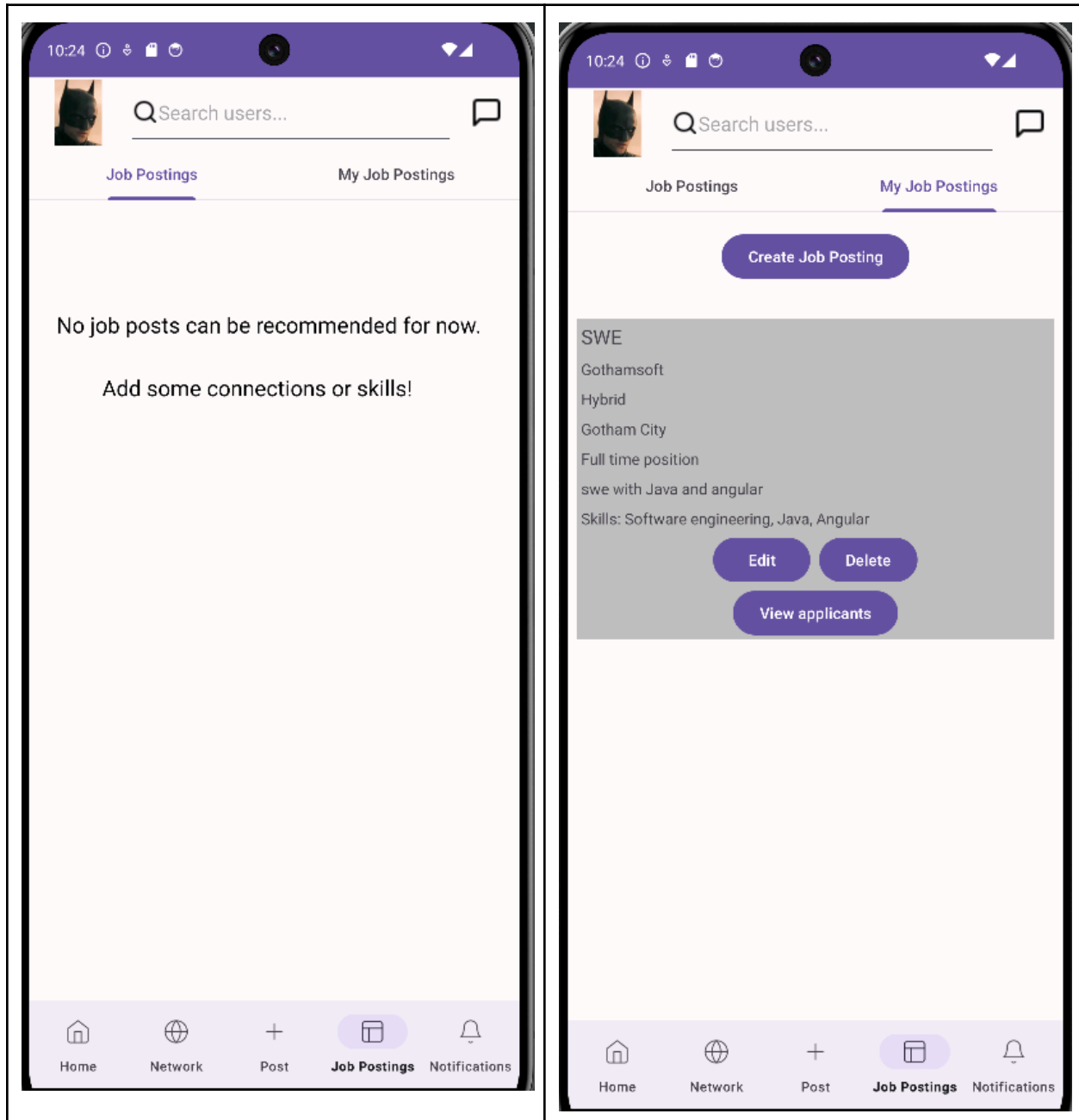
In the notifications fragment, the user can see a list of their notifications. Each notification has an action associated with it (navigate to messages) according to its type.





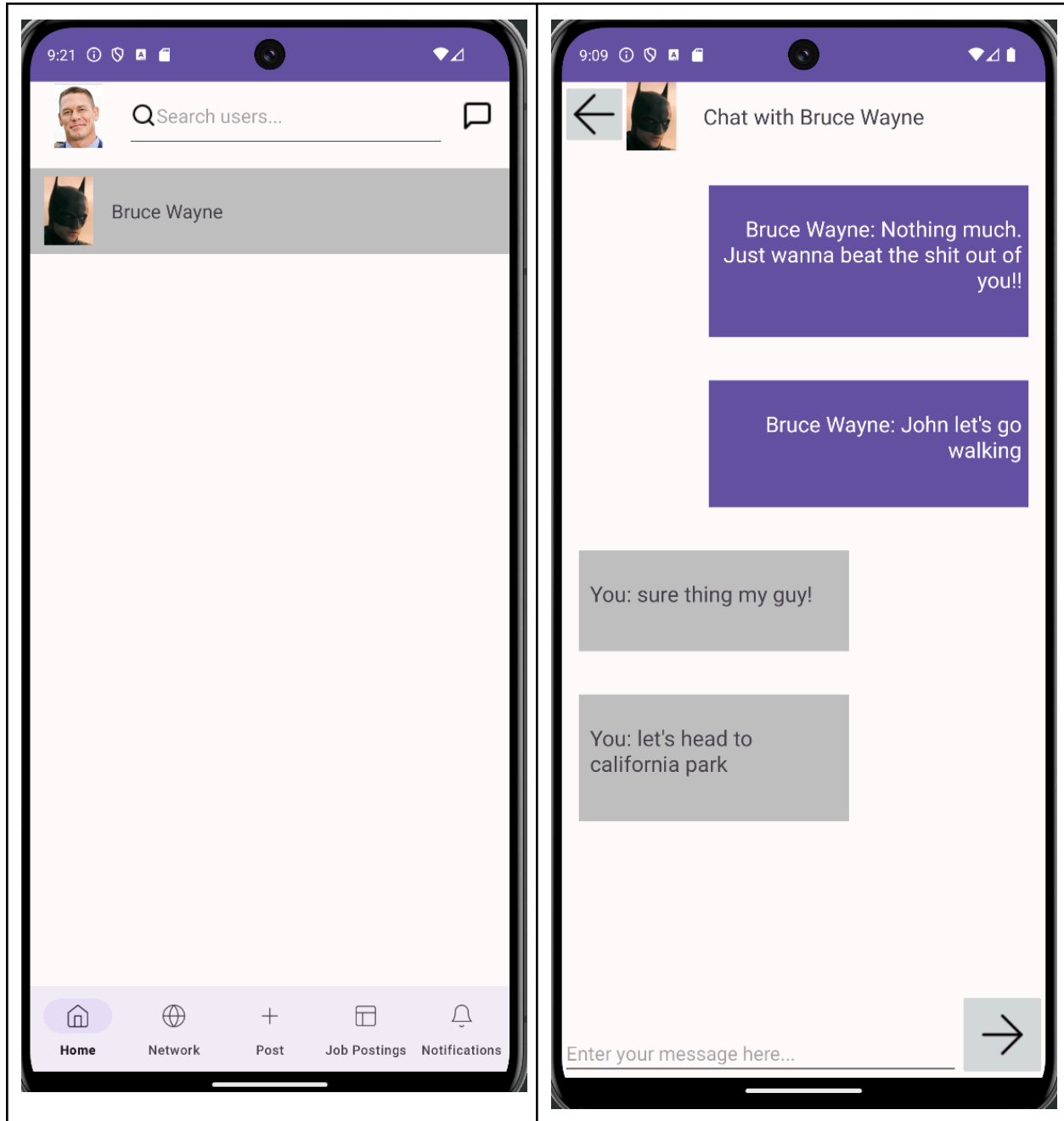
## Job Postings Fragment

The job postings fragment is a little more complex. It uses a view pager adapter in order to swap between two “modes”. The user can either view relevant job postings created by others and click on them for more details or to apply, or view (or create) their own job postings.



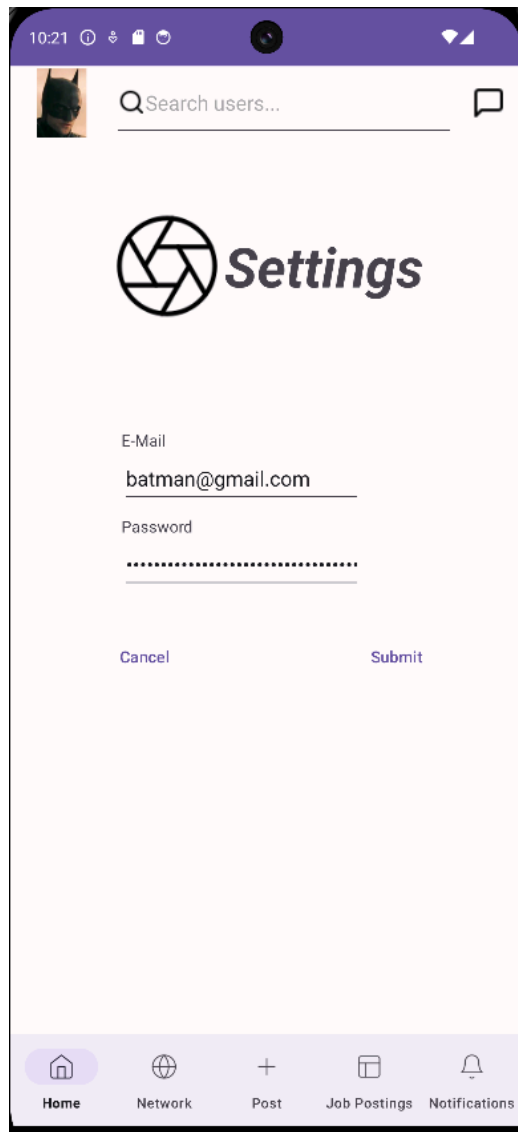
## Messages Fragment

The messages fragment has a list with the user's existing chats. He can select each entry to view their chat and send new messages.



## Settings

The settings is a fragment where the user can change their login information.

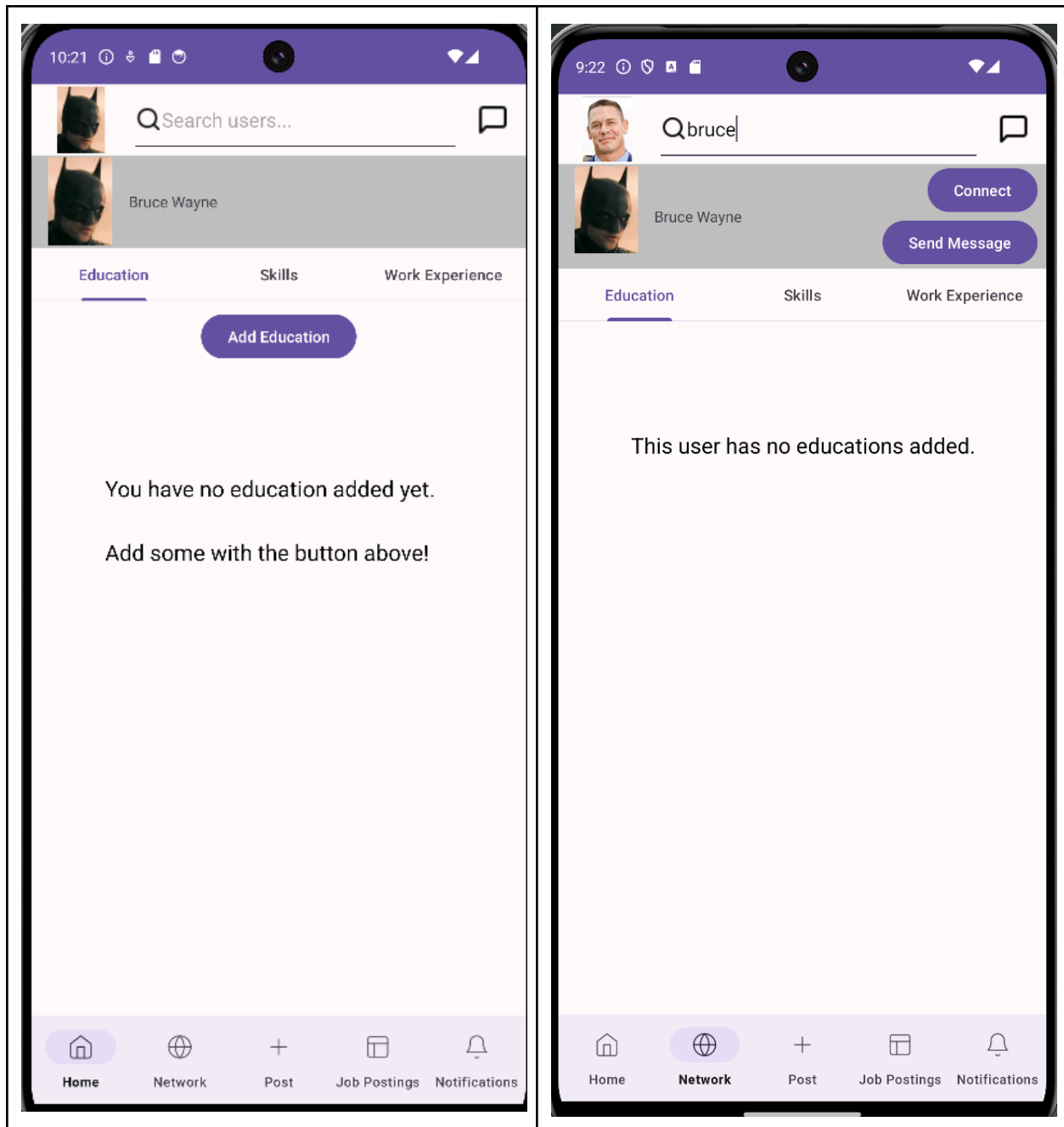


The screenshot shows a mobile application interface with a purple header bar. The status bar at the top displays the time 10:21 and various system icons. Below the header, there is a search bar with a magnifying glass icon and the placeholder text "Search users...". To the left of the search bar is a small profile picture of a person with dark hair. Below the search bar, the word "Settings" is displayed in a large, bold, black font, preceded by a circular icon containing a camera shutter design. Underneath the title, there are two input fields. The first is labeled "E-Mail" and contains the text "batman@gmail.com". The second is labeled "Password" and contains a series of dots, indicating a masked password. At the bottom of the form, there are two buttons: "Cancel" on the left and "Submit" on the right. The bottom of the screen features a navigation bar with five icons: a house icon for "Home", a globe icon for "Network", a plus icon for "Post", a document icon for "Job Postings", and a bell icon for "Notifications".

## Profile

The profile is one of the more complex fragments in our application. It has two “modes”: a) for when the user views their own profile - with additional functionality such as adding information, and b) for when the user views someone else’s profile - with additional functionality such as connecting with the user.

It uses a view pager adapter to swap between fragments that display (and can add depending on the profile’s mode) information relevant to the user.



## Education Fragment

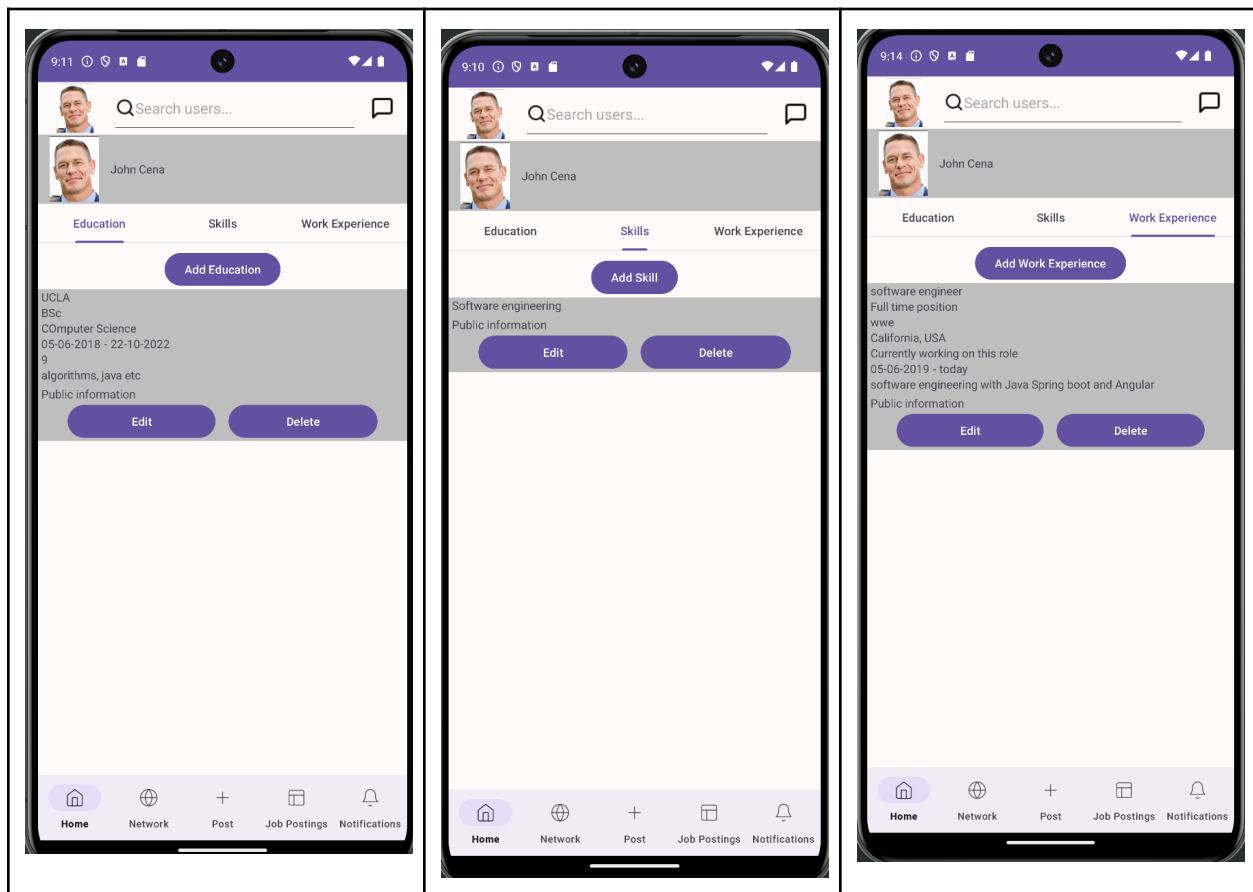
The education fragment contains entries with information about the user's education.

## Skills Fragment

The skills fragment contains entries with information about the user's skills.

## Work Experience Fragment

The work experience fragment contains entries with information about the user's work experience.



## Dynamic UI entries

A number of activities and fragments need to show a dynamic list of items that we get from the database (e.g. the list of the user's connections). To achieve this, the required entries' templates are in separate XML files, and are inflated in the appropriate containers to populate the application's screens.

## Example Service call

In the photo below we show an example service call using *retrofit's onResponse* and *onFailure* architecture.

If the response is successful, then the education is added successfully to the database.

If the response is unsuccessful, that means that we have added either the EducationDTO or the user email in a wrong format (null, wrong data type, etc.), and we get error 400 from the backend.

If we get to the *onFailure* case, we get an error 500 and that means the server somehow failed.

```
educationService.addEducation(educationDTO, userDTO.getEmail()).enqueue(new Callback<String>() {  
    ± Tsiobleman +1  
    @Override  
    public void onResponse(Call<String> call, Response<String> response) {  
        if (response.isSuccessful()) {  
            Toast.makeText(context: AddEditEducation.this, text: "added education successfully.", Toast.LENGTH_LONG).show();  
        } else {  
            Toast.makeText(context: AddEditEducation.this, text: "education addition failed. You cannot add the same education twice.",  
            }  
        }  
    }  
    ± Tsiobleman  
    @Override  
    public void onFailure(Call<String> call, Throwable t) {  
        Log.e(tag: "fail: ", t.getLocalizedMessage());  
        // Handle the error  
        Toast.makeText(context: AddEditEducation.this, text: "education addition failed. Server failure.", Toast.LENGTH_LONG).show();  
    }  
});  
if (t instanceof IOException) {
```

# Installation

In order to install and run our app you need to have the following:

*MySQL Workbench* (would be good but optional)

*MySQL Server* (default installation on server port 3306 is optimal)

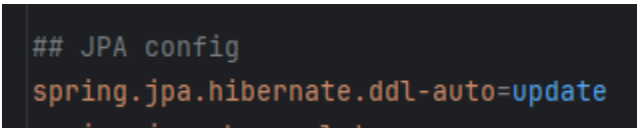
*Java 17 SDK* on IntelliJ

*Android Studio Jellyfish Edition (2023.3.1 Patch 1)* with minimum SDK: *API 34* (*"UpsideDownCake"*; *Android 14.0*).

Then you have to do the following:

1. Go to MySQL workbench or mysql console and create a database. Then add a schema called "worknet".
2. Open the project (*worknet*) in IntelliJ.
3. In the *application.properties* add the username and password you use to connect to your database, through the workbench or the console.

```
## JPA config
spring.jpa.hibernate.ddl-auto=update
```

4.  Change this to create and run the application.
5. Once the application ends, close it and change it back to update, then re-run the app.
6. Now the server works!
7. Open the worknet-android folder with Android Studio
8. Create an AVD and add some photos, videos and mp3 files by using the given instructions here: *add\_files\_to\_phone\_instructions.txt*.
9. Run the app and you're good to go.

## Conclusion

For the **Worknet** application, as was mentioned, we began by designing and taking some initial decisions regarding the backend structure.

After that, we split the necessary tasks, mainly using the backend-frontend divide, and each of us took one part of the application. Development was done concurrently, and when sufficient progress was made to each part, we began connecting them.

In the end, we tested the application using emulators from Android Studio, and we detected and fixed bugs, using pair programming.

The most significant problem we met was that of the structure of the **File Saving System**, as was described in the chapter with the same name.

We cooperated using teleconferencing tools to schedule regular meetings, and co-authored code asynchronously using the GIT version control system, hosted on GitHub.



## Bibliography

- [1] Recommender Systems: Matrix Factorization from scratch, September 2020, Aakanksha NS, ([link](#))
- [2] Recommender Systems: Matrix Factorization, July 2020, Denise Chen, ([link](#))