NATIONAL INSTITUTE OF POSTS AND TELECOMMUNICATIONS

Web Project Report

"SnapJob"

Created by:
Oualid AMTOUT

Supervised by:
Pr. Tarik Fissaa

Academic Year: 2025 - 2026

Contents

General Introduction

Project Context

The employment market is evolving rapidly, yet traditional recruitment methods often lack the fluidity and efficiency demanded by modern job seekers and employers. In an increasingly digital world, the disconnect between talented candidates and suitable opportunities represents a significant challenge for both parties. Organizations struggle to efficiently identify qualified candidates, while job seekers face difficulties in presenting their complete professional profiles and discovering relevant opportunities.

This project, entitled **"SnapJob"**, addresses these challenges by creating a dynamic web platform that facilitates seamless connections between recruiters and candidates. Drawing inspiration from professional networking platforms such as LinkedIn, SnapJob integrates social networking features with advanced recruitment functionalities to create a comprehensive employment ecosystem.

Objectives

The primary objectives of this project encompass both functional and technical dimensions. The platform aims to enable users to construct comprehensive professional profiles that incorporate their certifications, educational background, and professional experience. The system provides flexibility through dual functionality, allowing users to either publish job opportunities as recruiters or apply to existing positions as candidates. The recruitment process concludes when offer creators review generated application cards and select the most suitable candidates.

From a technical perspective, the project implements a full-stack web application utilizing modern development practices and architectures. The system employs the Model-View-Controller (MVC) design pattern, implements secure authentication mechanisms using JSON Web Tokens (JWT), and leverages a NoSQL database for flexible data management.

Problem Statement

Several persistent challenges characterize the current recruitment landscape. Traditional application processes require candidates to repeatedly enter the same information across multiple platforms, creating inefficiency and frustration. Recruiters often lack comprehensive visibility into candidate qualifications beyond basic resume information, making it difficult to assess fit accurately.

SnapJob addresses these issues through a card-based application system that allows candidates to selectively present relevant qualifications for each position. By combining social networking features with specialized recruitment functionality, SnapJob creates a cohesive environment that serves the needs of all stakeholders.

Chapter 1

Data Base(CDM & MLD)

1.1 Requirement Analysis

The development of SnapJob began with a comprehensive requirement analysis and deep understanding of the use case

- **User Management:** Registration with the important informations , login, profile editing, and secure authentication.
- **Profile Construction:** Adding experiences, education, skills (capabilities), and certifications.
- **Content Publication:** Publishing job offers and creating social posts.
- **Application Management:** Creating customized application cards for applying to the offers .
- **Recruitment Process:** Reviewing applicants accepting/rejecting candidates and save that in the data base .

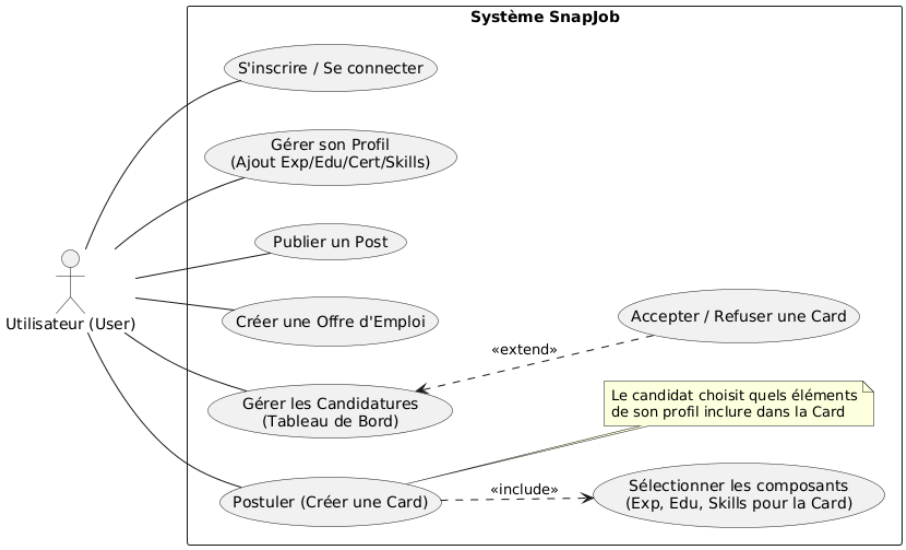


Figure 1.1: Use Case Diagram

1.2 Conceptual Data Model (CDM)

The SnapJob application is user-centric, offering the possibility to build a complete profile by integrating certifications, educational background, and professional experiences. This system allows for great flexibility: a user can both publish job offers as a recruiter or apply to existing announcements by generating a candidate "Card". The recruitment process is finalized when the offer creator reviews these cards to select and accept the most relevant candidates.

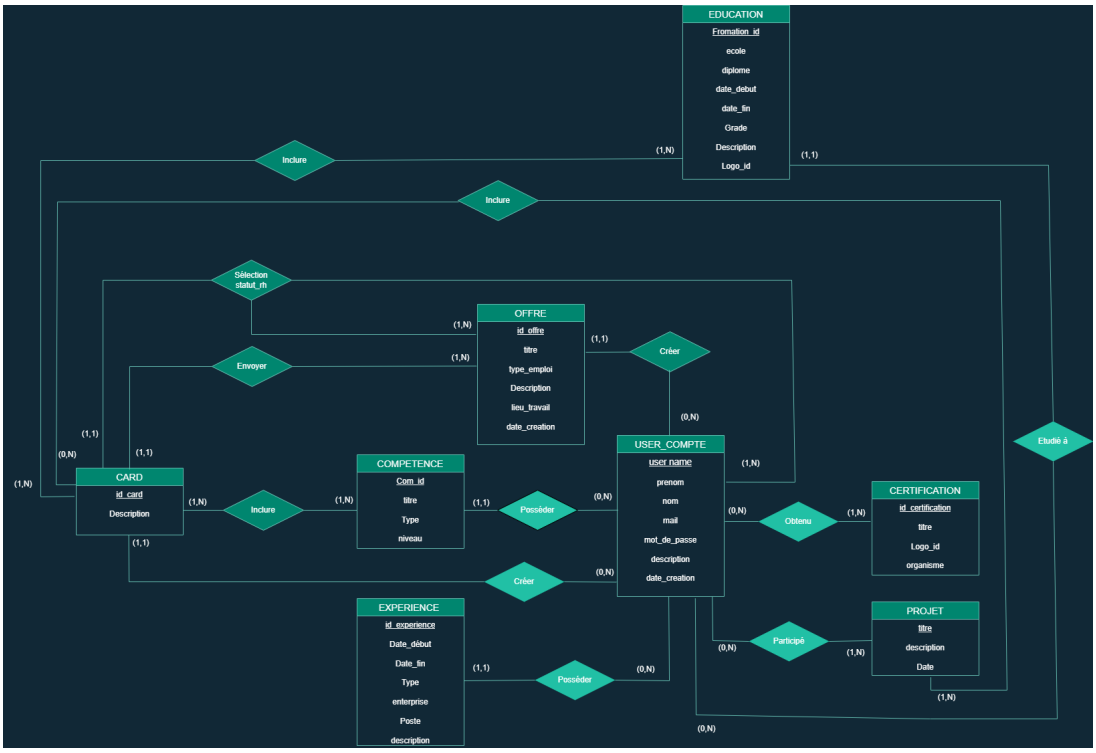


Figure 1.2: SnapJob Conceptual Data Model (CDM)

Remark: the entity and structure related to the **"Post"** functionality have been added just in the Logical Data Model (MLD).

1.3 Logical Data Model (MLD)

Here is the logical data model of the SnapJob database (**adapted to the application's needs**). Primary keys are underlined and foreign keys are preceded by a hash symbol.

- **USER** (__id__, first_name, last_name, email, password, user_name, description, isAdmin, profilePic, coverPicture, followers, following, createdAt, updatedAt)

- **EXPERIENCE** (__id__, Date_debut, Date_fin, Type, enterprise, Poste, description, createdAt, updatedAt, #userid)

- **EDUCATION** (__id__, ecole, diplome, date_debut, date_fin, Grade, Description, createdAt, updatedAt, #userid)

- **CAPABILITIES** (_id, title, type, niveau, createdAt, updatedAt, #userid)
- **CERTIFICATION** (_id, titre, organisme, createdAt, updatedAt, #userid)
- **PROJECT** (_id, titre, description, date, createdAt, updatedAt, #userid)
- **OFFRE** (_id, titre, type_emploi, Description, lieu_travail, username, profilePic, createdAt, updatedAt, #userid)
- **CARD** (_id, username, Description, createdAt, updatedAt, #id_offre, #userid)
- **STATUE** (_id, user_name, status, createdAt, updatedAt, #id_card, #id_offre)
- **INCLURE_COMP** (_id, #id_card, #Comp_id)
- **INCLURE_EDUCATION** (_id, #id_card, #Formation_id)
- **INCLURE_CERT** (_id, #id_card, #Cert_id)
- **INCLURE_EXP** (_id, #id_card, #Exp_id)
- **INCLURE_PROJ** (_id, #id_card, #title)
- + **POST** (_id, content, username, profilePic, likes, createdAt, updatedAt, #userid) to add the option to create a post (article)

Explanation of Key Concepts of the LDM

- **Central User Entity:** The **USER** entity represents the heart of the system. All main information revolves around the user, making it the central pillar of the architecture.
- **Profile Components:** The entities **EXPERIENCE**, **EDUCATION**, **CAPABILITIES**, **CERTIFICATION**, and **PROJECT** are modules linked to the user, allowing them to enrich their professional profile.
- **OFFRE Entity:** It represents job opportunities published by recruiters. Each offer is linked to its creator and serves as a reference point for applications, thus centralizing all applications for a given position.
- **Junction Tables (INCLURE_*):** These tables allow the user to create a customized application (**CARD**). They link the card ID to the specific component ID (e.g., a specific experience), offering the possibility to choose which profile elements to include for each application.

- **CARD Entity:** It materializes the application by grouping the user, the targeted offer, and the profile elements selected via the junction tables.

- **STATUE Entity:** It manages the lifecycle of the application (accepted, refused, pending) by linking the recruiter's decision to the specific card.

Chapter 2

Architecture

2.1 Architecture

Inspired by the dynamics of professional social networks such as **LinkedIn**.

2.1.1 Application Architecture

The entry point of the application is the `server.js` file.

1. **Entry Point (`server.js`):** This file initializes the Express server, configures middlewares, establishes the connection with MongoDB, and registers route handlers.
2. **Routing (Routes):** Requests are delegated according to their URL prefix:
   - `/auth`: Handled by `authRoutes.js`.
   - `/user`: Handled by `userRoutes.js`.
3. **Controller (Controllers):** Routes call specific functions containing business logic. The project mainly relies on two controllers: `userController.js` for profile and application management, and `authController.js` for managing authentication (account creation, login, and logout).
4. **Model (Models):** This defines the data structure via Mongoose (files in the `/models/`). It concretizes the Logical Data Model (MLD) described earlier by transforming entities (User, Offre, Card, etc.) into schemas usable by the application.
5. **View (Views):** Finally, data is sent to EJS templates in the `views/` folder to generate the final HTML rendering.
6. **Middleware:** The `verifyJWT.js` file functions as a security checkpoint that intercepts incoming requests to check if they come from authenticated users (by validating the JWT Token) before authorizing access to protected resources.

2.2 File Structure

The project organization follows this hierarchy:

```
  SnapJob/
|--   server.js (Entry Point)
|--   package.json
|
|--   routes/ (URL Management)
|   |--   root.js
|   |--   authRoutes.js
|   |--   userRoutes.js
|
|--   controllers/ (Business Logic)
|   |--   authController.js
|   |--   userContriller.js
|
|--   models/ (LDM Schemas)
|   |--   user.js
|   |--   offre.js
|   |--   card.js
|   |--   statue.js
|   |-- ... (other models)
|
|--   middleware/
|   |--   verifyJWT.js (Security)
|
|--   views/ (EJS Templates)
|   |--   index.ejs
|   |--   profil.ejs
|   |--   Appliedorjob.ejs
|   |-- ...
|
|--   public/ (Static Files)
    |--   Styles/
    |--   Scripts/
    |--   imgs/
```

2.3 Sitemap (API Development)

Here is the hierarchy of links and features accessible in the application:

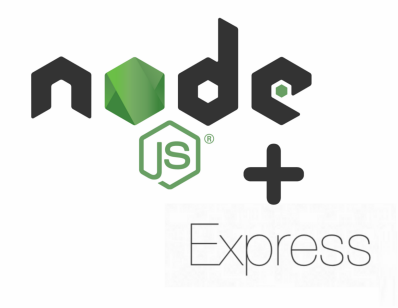
```
 http://localhost:3000/
|
|__ /auth (Public)
|    |__ /login (GET/POST give the Token)
|    |__ /register (GET/POST for creating new user in database)
|    |__ /logout (remove Token)
|
|__ /user (Protected by JWT)
     |__ /home/:id (Homepage / Feed)
     |    |__ /offre (POST - Create an offer)
     |    |__ /post (POST - Create a post)
     |
     |__ /profil
     |    |__ / (read my profile using the id from the token)
     |    |__ /:id (Another user's profile)
     |    |__ /update/:id (Update info)
     |    |__ /addExp, /addEdu... (Add education, exp...)
     |    |__ /creatCard (Create an application)
     |
     |__ /Applied/:id (see created and applied offers)
          |__ ?offerId: (Accept/Refuse cards)
```

2.4 Technology Stack

The architectural choices for **SnapJob** were driven by the need for performance, scalability, and rapid development. While inspired by the popular **MERN** stack, this project utilizes **EJS** instead of React for the frontend logic, effectively implementing the **MEN** stack.

1. Backend & Application Server

Node.js & Express.js (v5.2.1)
The core of the application is powered by **Node.js**, selected for its non-blocking I/O model. We utilize the **Express.js** framework to structure the REST API, manage routing, and integrate essential middlewares such as `cookie-parser` for secure token handling.



2. Database & ODM

MongoDB & Mongoose (v9.0.2)
Data persistence is handled by **MongoDB** (NoSQL), which provides the flexibility needed to store complex user profiles. We use **Mongoose** as an Object Data Modeling (ODM) library to enforce strict schemas (User, Offer, Card) and manage relationships between documents via the `INCLURE_*` join tables.

3. User Interface (Frontend)

EJS, CSS3 & Vanilla JavaScript
Instead of a client-side framework like React, we use **EJS** (Embedded JavaScript) for Server-Side Rendering. This allows the server to inject data directly into the HTML templates before sending them to the client. The interface is styled with modern CSS3 (utilizing CSS variables) and made interactive with lightweight JavaScript.

4. Security & Authentication

JWT & Bcrypt
Security is implemented using a stateless authentication mechanism via **JSON Web Tokens (JWT)**. These tokens are securely stored in HTTP-Only cookies to prevent XSS attacks. Additionally, user passwords are salted and hashed using **Bcrypt** prior to storage.

5. Development Tools

The development workflow relied on standard industry tools:

- **VS Code:** Primary code editor.
- **Postman:** Used for testing API endpoints and simulation.
- **MongoDB Compass:** GUI for visualizing database data.
- **Nodemon:** For automatic server restarts during development.
- **Dotenv:** For secure environment variable management.

Chapter 3

Implementation and Business Logic

This chapter details the functional core of the SnapJob application. Business logic is mainly distributed between two controllers: `authController.js` for security and `userController.js` for data management.

3.1 Account Management and Security

The authentication controller (`authController.js`) manages the user access lifecycle via secure JWT Tokens.

- **`register`**: Creates a new user account. This function validates data, checks username uniqueness, hashes the password with *bcrypt* for security, generates JWT tokens (access and refresh), and redirects the user to login.

```
const register = async (req, res) => {
    try {
        const { first_name, last_name,user_name, email, password,description } = req.body;
        if (!email || !password || !first_name || !last_name || !user_name){ return res.render('Signup', { error: 'you should fille all the gaps' })};
        const finduser = await user.findOne({ user_name: user_name }).exec()
        if (finduser) {
            return res.render('Signup', { error: 'User already exists' });
        }
        const hashedpassword = await bcrypt.hash(password, 10);
        const User = await user.create({
            first_name,
            last_name,
            email,
            password: hashedpassword,
            user_name,
            description
        });

        const accessToken = jwt.sign({ UserInfo: { id: User._id } }, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '20m' })

        const refreshToken = jwt.sign({ UserInfo: { id: User._id } }, process.env.REFRESH_TOKEN_SECRET, { expiresIn: '7d' })

        res.cookie('jwt', refreshToken, { httpOnly: true, secure: true, sameSite: "None", maxAge: 60 * 60 * 1000 * 24 * 7 })
        User.save()

        res.redirect('/auth/login');
    } catch (error) {
        console.log(error)
    }
```

Figure 3.1: Register

- **`login`**: Authenticates a user. It verifies credentials, compares the hashed password, generates JWT tokens, and stores them in *HTTP-only* cookies before redirecting to the homepage.

Figure 3.2: login

- **`logout`**: Logs the user out by deleting the *refresh token* cookie to invalidate the session.
- **`refresh`**: Ensures connection persistence. It regenerates a new access token by validating the stored *refresh token*, allowing the session to remain active after the main token expires (20 minutes).
- **`updateUser`**: Allows updating profile information (name, email, description, photos). Strict verification ensures that the user only modifies their own profile.

3.2 Skills and Profile Management

The SnapJob profile is modular. The following functions (`userController.js`) allow dynamically adding or removing components:

- **`createxperience`** / **`removeexperience`**: Manages the addition and removal of professional experiences (dates, company, position, description).
- **`createCertification`** / **`removecertification`**: Manages certifications and issuing organizations linked to the profile.
- **`creatEducation`** / **`removeeducation`**: Manages academic background (school, diploma, dates, grade).
- **`creatcapabilitie`** / **`removecreatcapabilitie`**: Allows adding or removing skills (Hard Skills/Soft Skills) and their level.

```
const creatcapabilitie = async (req, res) => {

    try {
        const { title, type, niveau } = req.body
        if (!title || !type || !niveau) { return res.send("Please fill all the gaps") }
        const newcapabilitie = await capabilitie.create({ title, type, niveau, userid: req.user.toString() })
        console.log(newcapabilitie);
        res.redirect('/user/profil/');

    } catch (error) {
        res.send("Server Error");
    }
}

const removecreatcapabilitie = async (req, res) => {

    try {
        const { title } = req.body
        if (!title) { return res.send("Please fill all the gaps") }
        const newcapabilitie = await capabilitie.findOneAndDelete({ title })
        res.redirect('/user/profil/');

    } catch (error) {
        res.send("Server Error");
    }
}
```

Figure 3.3: Create component

```
const removecreatcapabilitie = async (req, res) => {

    try {
        const { title } = req.body
        if (!title) { return res.send("Please fill all the gaps") }
        const newcapabilitie = await capabilitie.findOneAndDelete({ title })
        res.redirect('/user/profil/');
                (local var) error: unknown
    } catch (error) {
        res.send("Server Error");
    }
}
```

Figure 3.4: Delete component

3.3 Publication and Application System

This section covers the key ”Recruiter-Candidate” interaction features.

- **`postoffre`**: Allows recruiters to create a new job offer. The function validates mandatory fields and associates the offer with the creating user.

```
const postoffre = async (req, res) => {
    try {
        const userid = req.params.id
        const { title, type_emploi, description, place } = req.body
        if (!title || !type_emploi || !place || !description) { return res.send("Please fill all the gaps"); };

        const findoffre = await offre.findOne({ title: title }).exec()

        if (findoffre) {
            return res.send('this offre allredy exist change the title');
        }
        const foundUser = await user.findById(userid);
        const Offre = await offre.create({ title, type_emploi, Description: description, lieu_travail: place, userid, username: foundUser.user_name, profilePic
        res.redirect('/user/home/' + userid);
    } catch (error) {
        console.log(error);
        res.send("Server Error");
    }
}
```

Figure 3.5: creation

- **`postpost`**: Handles the creation of social publications (posts) on the news feed to encourage community engagement.

- **`creatCard` (Key Function):** This function materializes the act of applying. It generates a "Card" which groups:
  - the candidate's information
  - A specific selection of profile elements (via `INCLURE_*` junction tables).

- **`updateCandidateStatus`**: Allows the recruiter to process an application. It updates the status (Accepted/Rejected) in the `STATUE` table and redirects to the next candidate.

```
const updateCandidateStatus = async (req, res) => {
    const { cardId, offerId, status, index } = req.body;
    if (!cardId || !offerId || !status) {
        return res.send("Please fill the gaps");
    }
    const findOffre = await offre.findById(offerId);
    if (!findOffre) {
        return res.send("not found");
    }
    if (findOffre.userid.toString() !== req.user.toString()) {
        return res.send("Unauthorized");
    }
    const findCard = await card.findById(cardId);
    if (!findCard)
        return res.send("not found");
    }
    const existingStatus = await statue.findOne({
        id_card: cardId,
        id_offre: offerId
    });
    if (existingStatus) {
        return res.send("This candidate already reviewed");
    }
    const myUser = await user.findById(req.user).select('-password');
    const newstatue = await statue.create({
        id_card: cardId,
        user_name: myUser.user_name,
        status: status,
        id_offre: offerId
    });

    const nextIndex = parseInt(index) + 1;
    res.redirect('/user/Applied/' + req.user.toString() + '?offerId=' + offerId + '&index=0');
```

Figure 3.6: updateCandidateStatus

3.4 Display and Navigation (Display)

These functions prepare data (Data Fetching) before rendering EJS views.

- **`gethomepage`**: Generates the news feed (Feed) by mixing posts and offers, sorted by creation date, and displays user suggestions after authentication.

```
try {
    const finduser = await user.findById(req.params.id).select('-password');
    const posts = await post.find().sort({ createdAt: -1 });
    const offres = await offre.find().sort({ createdAt: -1 });
    const myUser = await user.findById(req.params.id).select('-password');
    const suggestions = await user.find({}).limit(3);

    let mylist = [];
    let i = 0;
    let j = 0;
    while (i < posts.length || j < offres.length) {
        let randomint = Math.floor(Math.random() * 100);
        let addPost = false;
        if (i < posts.length && j < offres.length) {
            addPost = (randomint % 2 === 0);
        } else if (i < posts.length) {
            addPost = true;
        } else {
            addPost = false;
        }

        if (addPost) {
            mylist.push(['post', posts[i]]);
            i++;
        } else {
            mylist.push(['offre', offres[j]]);
            j++;
        }
    }
```

Figure 3.7: creation of a random list (Feed)

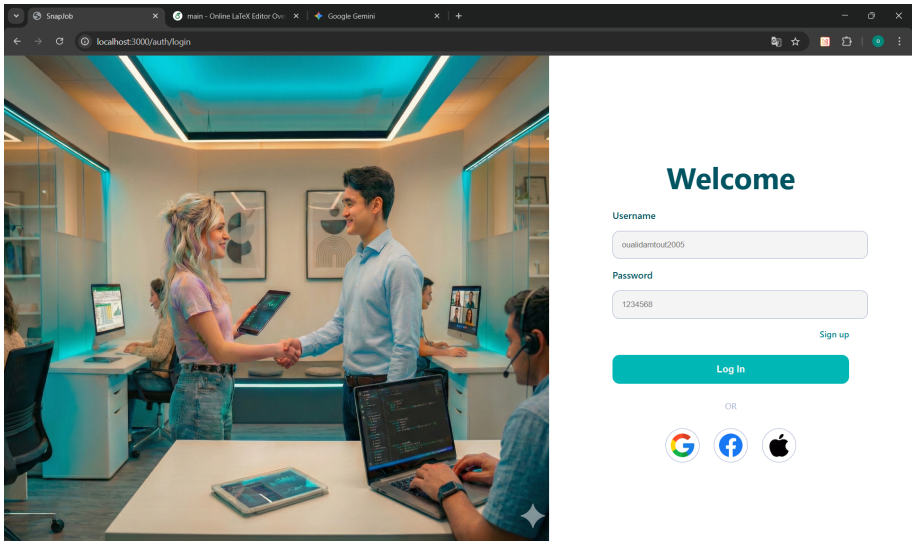
- **`getmyprofil` / `getprofil`**: The first displays the private profile of the connected user using the ID in cookies, while the second allows viewing another user's public profile using params.id.
- **`getApplied` (Dashboard):** Manages the hybrid dashboard display:
  - For the candidate: Lists the statuses of their applications.
  - For the recruiter: Displays pending candidates for a specific offer, with their full profiles for evaluation.

Figure 3.8: Application management interface generated by `getApplied`

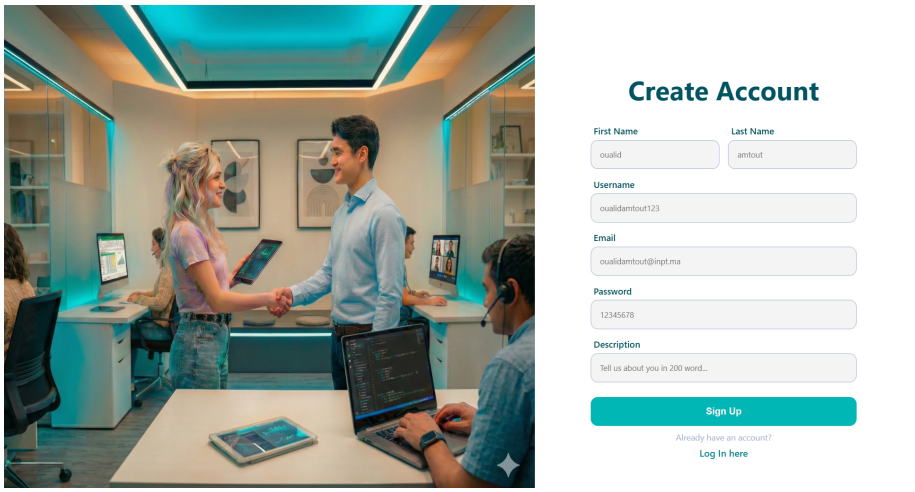
Chapter 4

Demonstration and User Experience

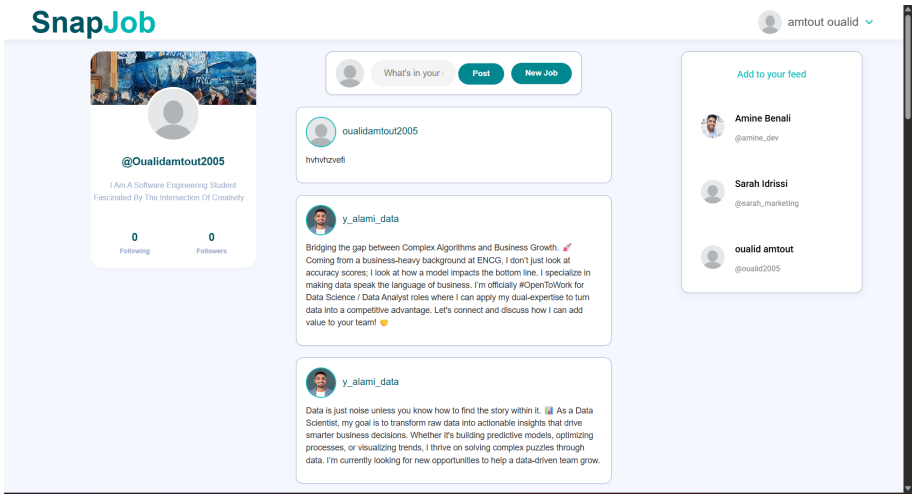
This section details the user journey through the SnapJob platform, from authentication to recruitment management.



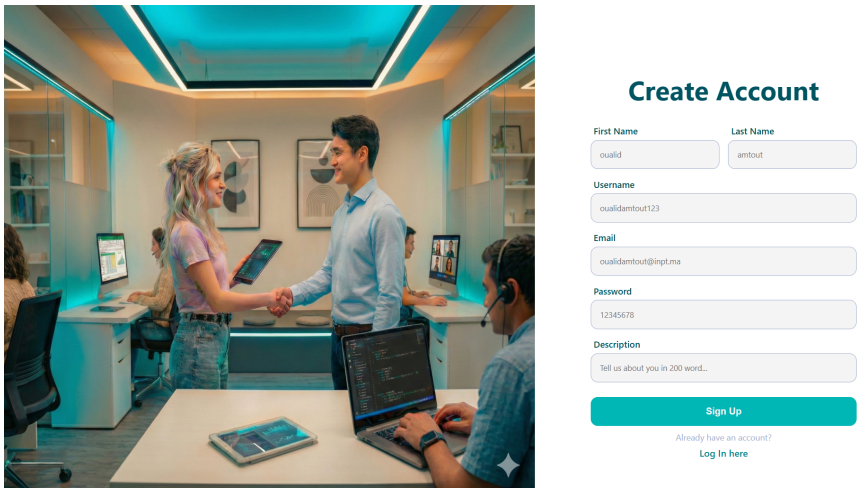
Login page: The application entry point is a secure login page. Upon successful login, the server issues a session token stored in the browser's cookies to maintain the user's authenticated state New users can register via the sign-up button.



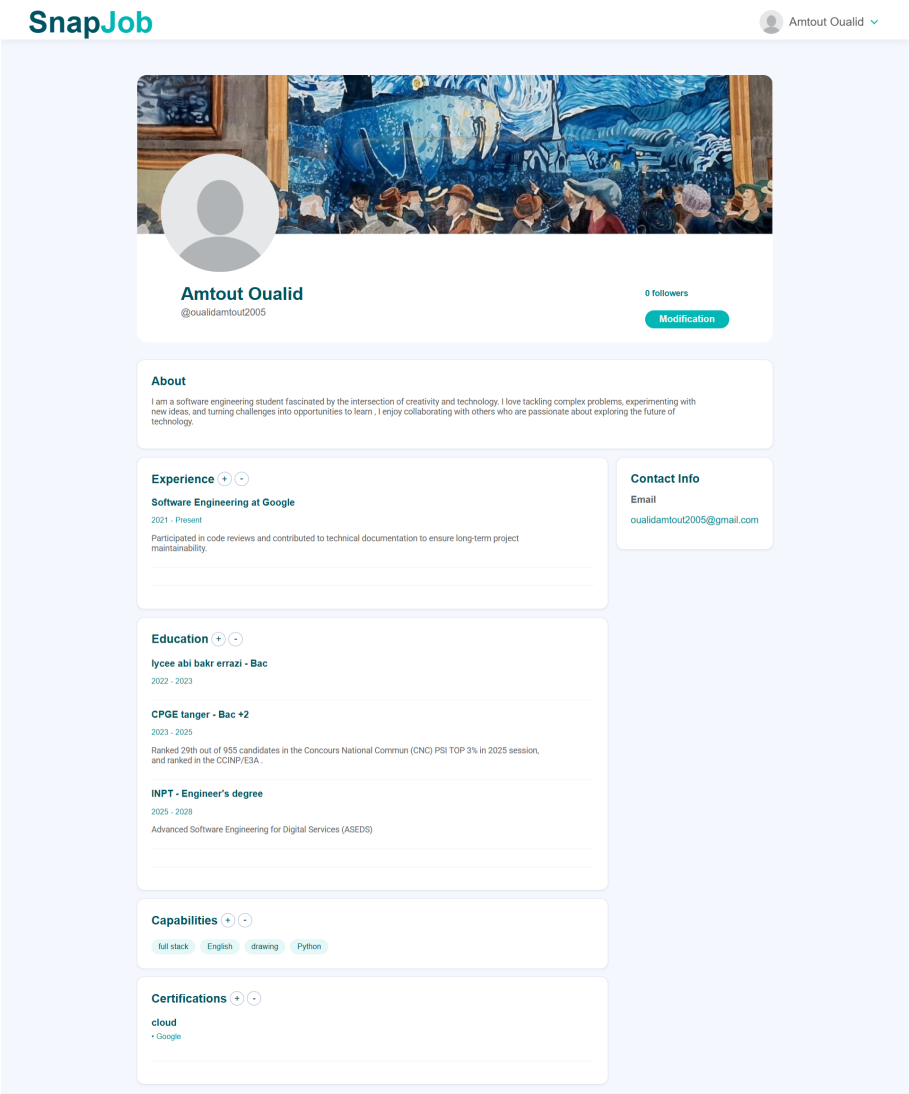
Register New users can register via the sign-up form, which creates a new entry in the MongoDB database with Bcrypt-encrypted passwords for enhanced security.



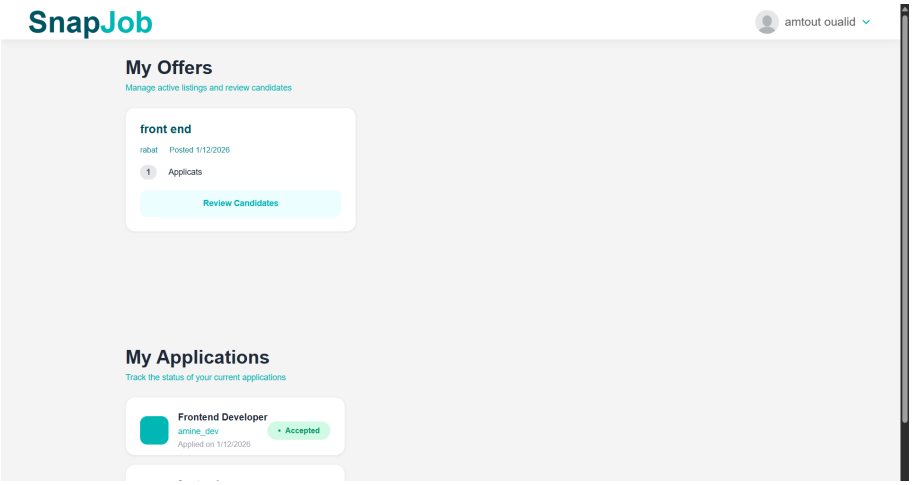
Home page: Upon successfully logging in, the user is immediately greeted by the Home Page, the central hub of SnapJob. The interface is designed for intuitive navigation, featuring a three-column layout. On the left, users see a quick summary of their personal profile. The center displays the dynamic feed, where the algorithm mixes posts and job offers to keep content engaging. Finally, the right sidebar provides network suggestions, allowing users to easily discover and connect with others. This design ensures all key features are accessible at a glance. and the user can also create post and offers by himself



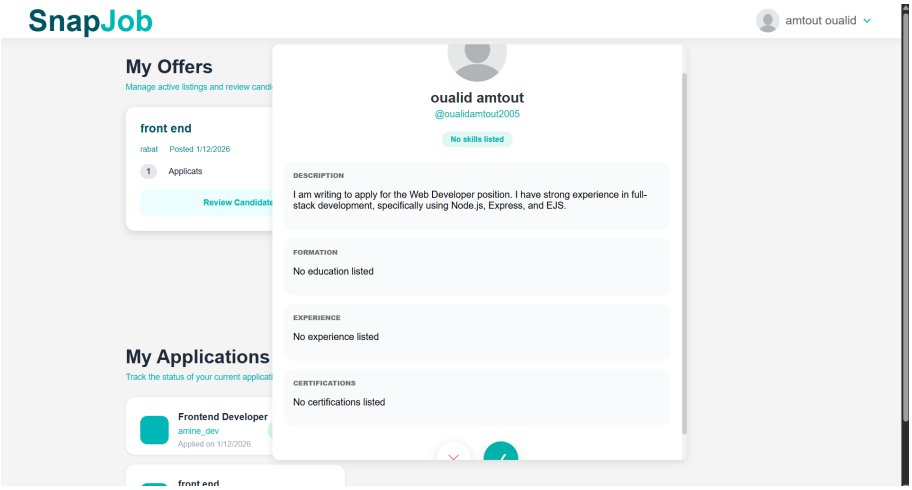
Register: New users can register via the sign-up form, which creates a new entry in the MongoDB database with Bcrypt-encrypted passwords for enhanced security.



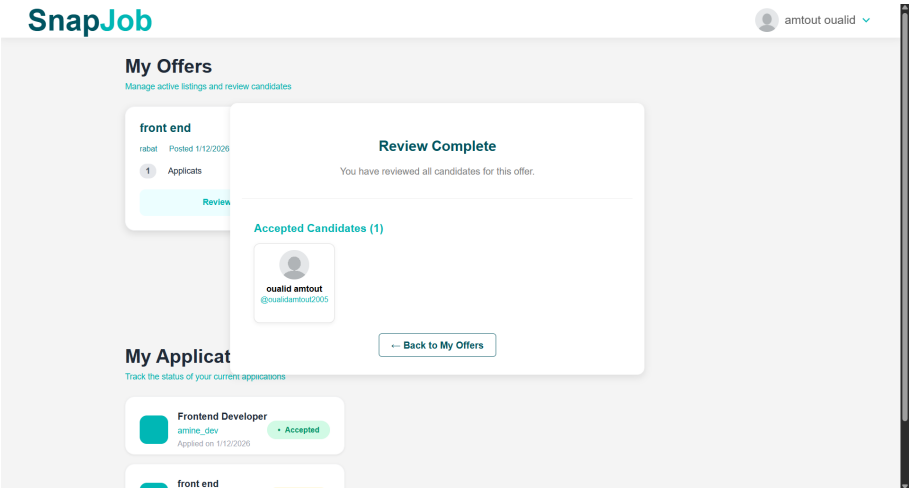
Porfil page: This Profile Page serves as a dynamic CV where users have full control over their data. By clicking the 'Modification' button, they can instantly update core details like their Name, Username, Profile Picture, and Email. Furthermore, for professional sections like Experience, Certifications, and Skills, I implemented interactive Plus (+) and Minus (-) buttons. These allow the user to add new entries or delete existing ones in real-time, directly updating the database..



Applid page: In My Applications , candidates track the real-time status of their submissions, such as seeing an 'Accepted' tag. Meanwhile, the 'My Offers' section allows recruiters to manage their listings and monitor the live count of applicants for each job they have posted



Applid page: Clicking 'Review Candidates' triggers this pop-up displaying the applicant's Cards .This focused view allows the recruiter to evaluate the candidate and immediately decide to Accept or Refuse.



Applid page: After the review process, the system generates a final list of 'Accepted Candidates'. This summary provides immediate utility for the recruiter, displaying the candidates email. It also links directly to their full profiles for any final verification needed before hiring.

General Conclusion

This project allowed the implementation of a complete social media architecture with Node.js and MongoDB, offering a robust solution for recruitment management. The use of JWT ensures security, while the modular architecture allows for easy future evolution.

Appendix A

Appendices

Useful Links

- **Source Code (GitHub):** https://github.com/oialid25/backend-SnapJob
- **Demonstration Video:** https://drive.google.com/drive/folders/1SN9eDUECy0DWgOJHHC5lPdiSDKzeMUC_?usp=sharing