CS 5934: Final Product Report

Product Name: Quantify

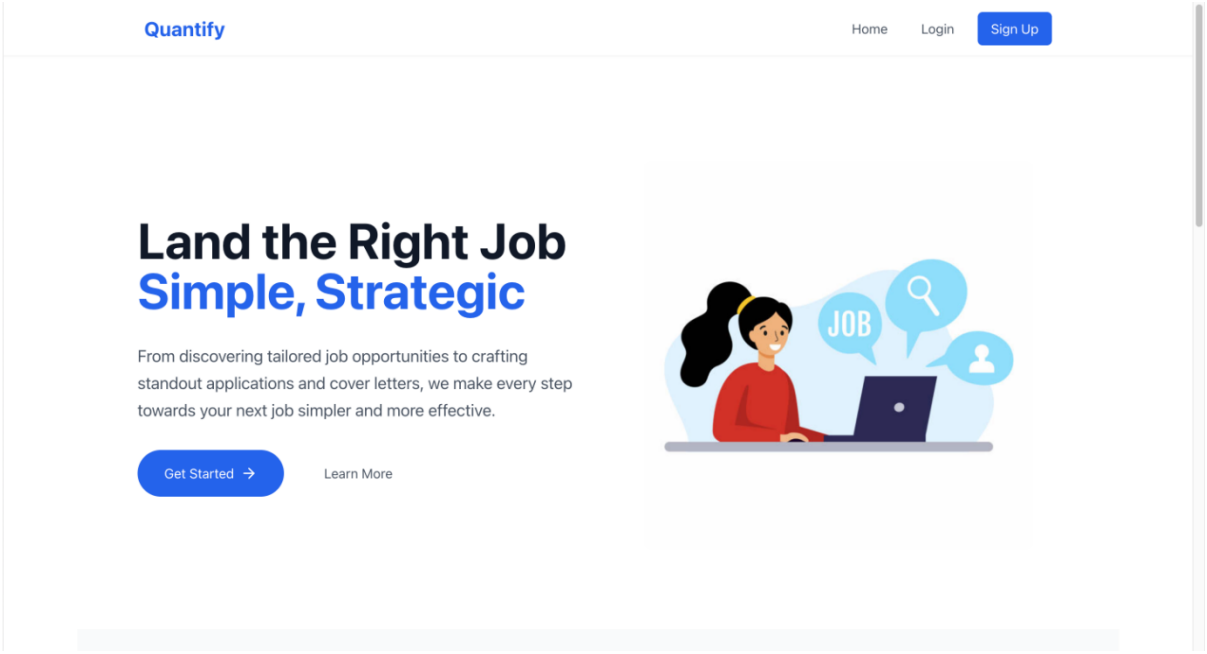
Project members:

Aditya Reddy Sabbella
Khavin Krishnan Kalpana
Krishna Vamsi Dhulipalla
Uma Sruthy Gajula
Vineela Yerrabelli

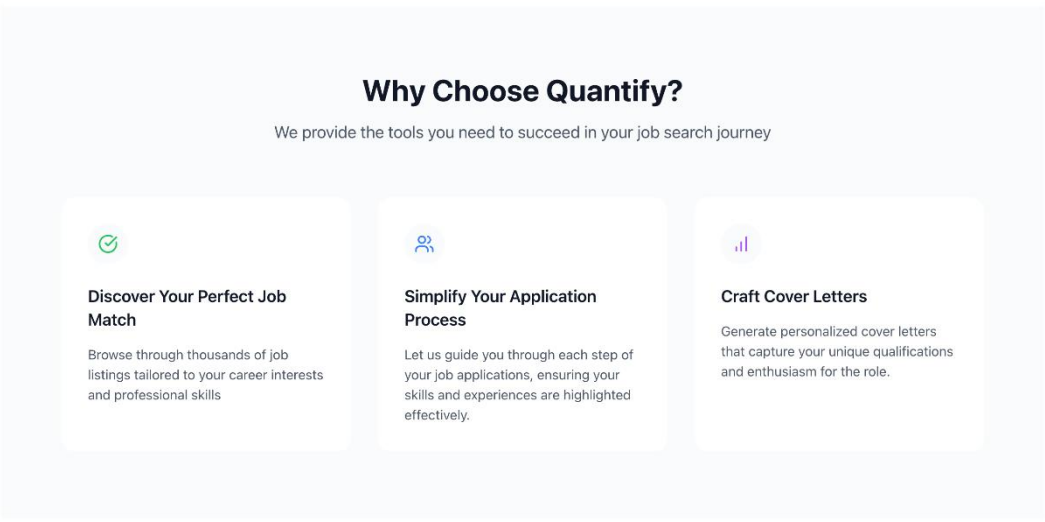
1. Product Description

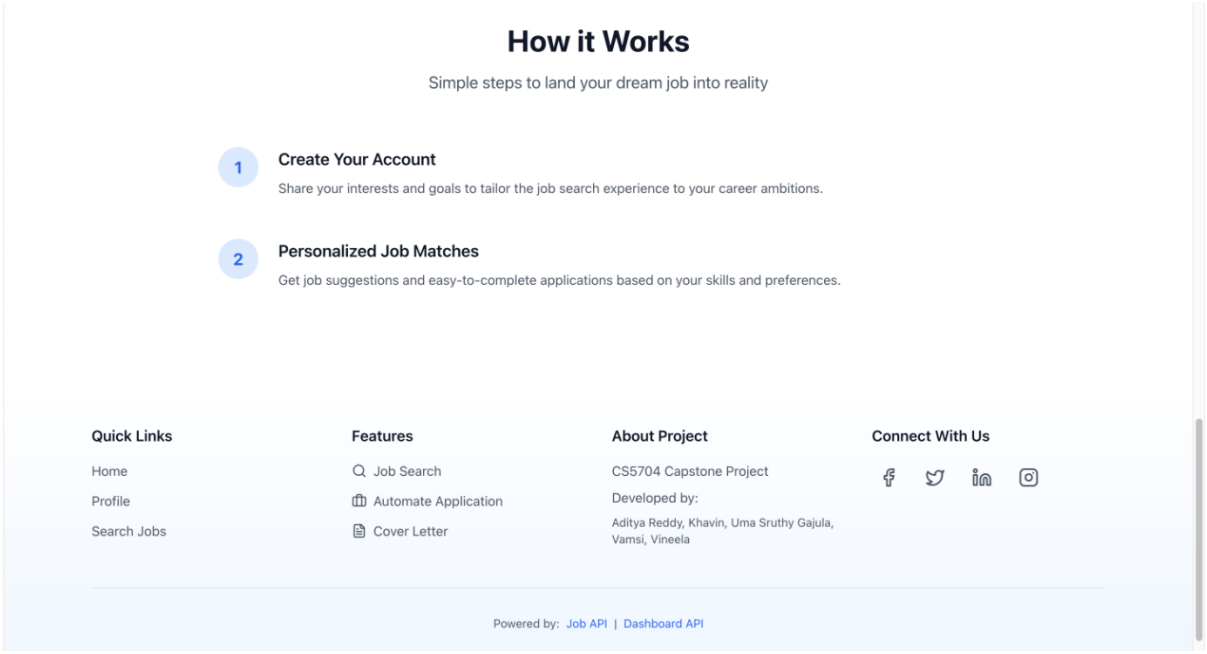
Quantify aims to automate and streamline the job search and application process. By aggregating job listings from different platforms (such as LinkedIn and Indeed), filtering them based on the candidate's profile, and generating personalized cover letters, Quantify reduces the time spent on job applications. Users will be presented with a curated list of relevant jobs and can click on the jobs they want to apply for. The browser extension will then fill in the application forms and submit them with one click. The project aims to save users time, reduce job application fatigue, and improve the efficiency of the application process.

2. Product Functionalities

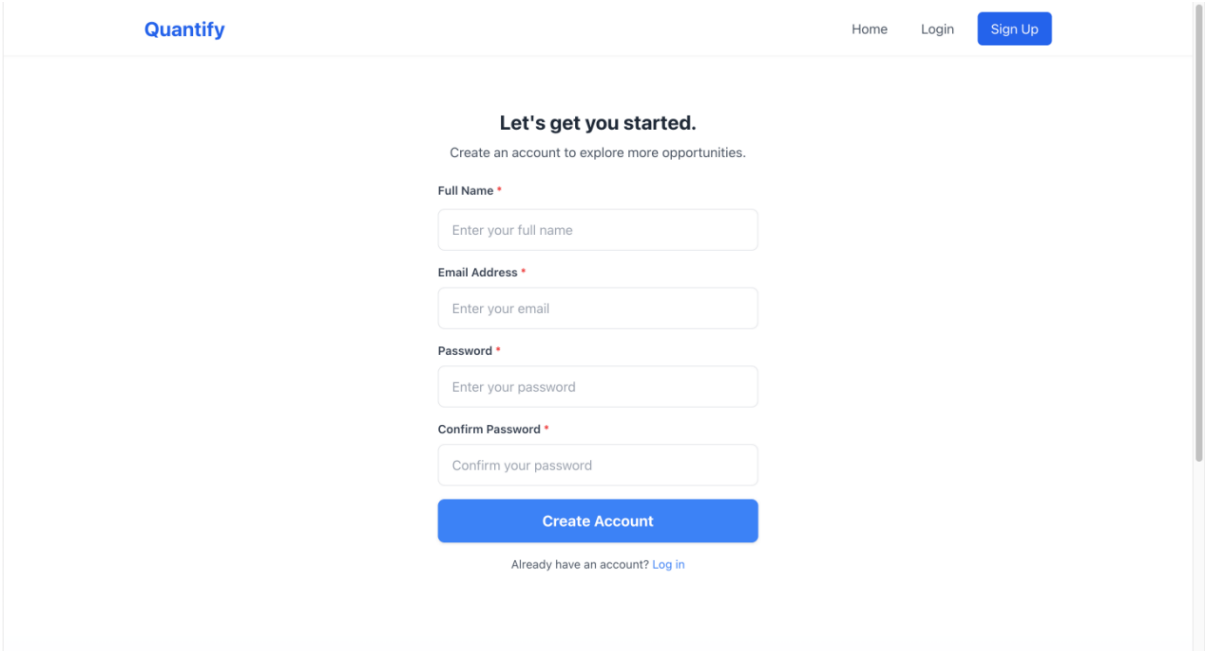


The hero content and image of our home page describes the need for a tool like Quantify. The home page also lists out different features offered by "Quantify".

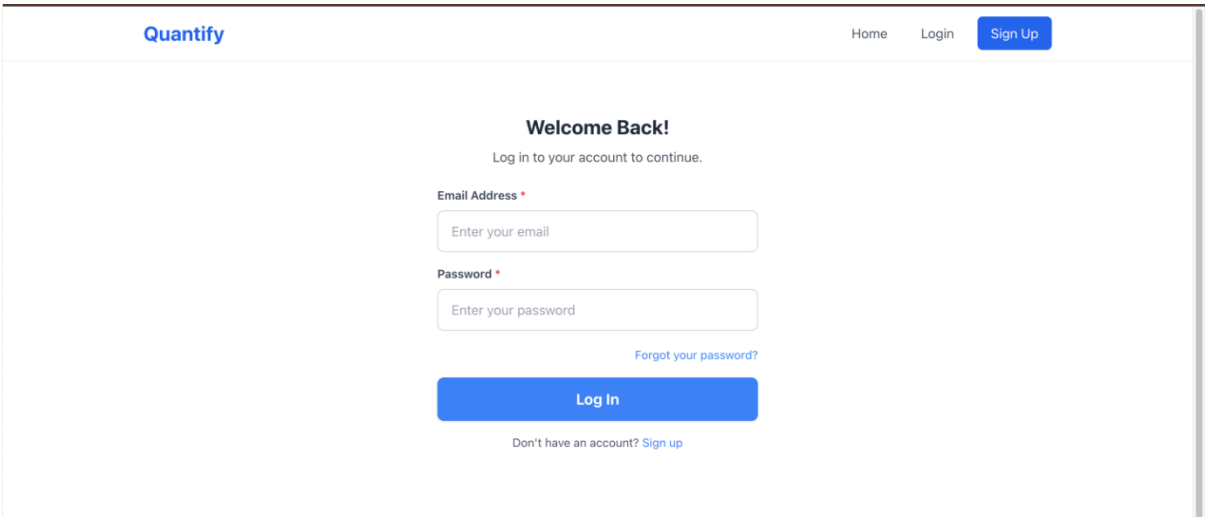




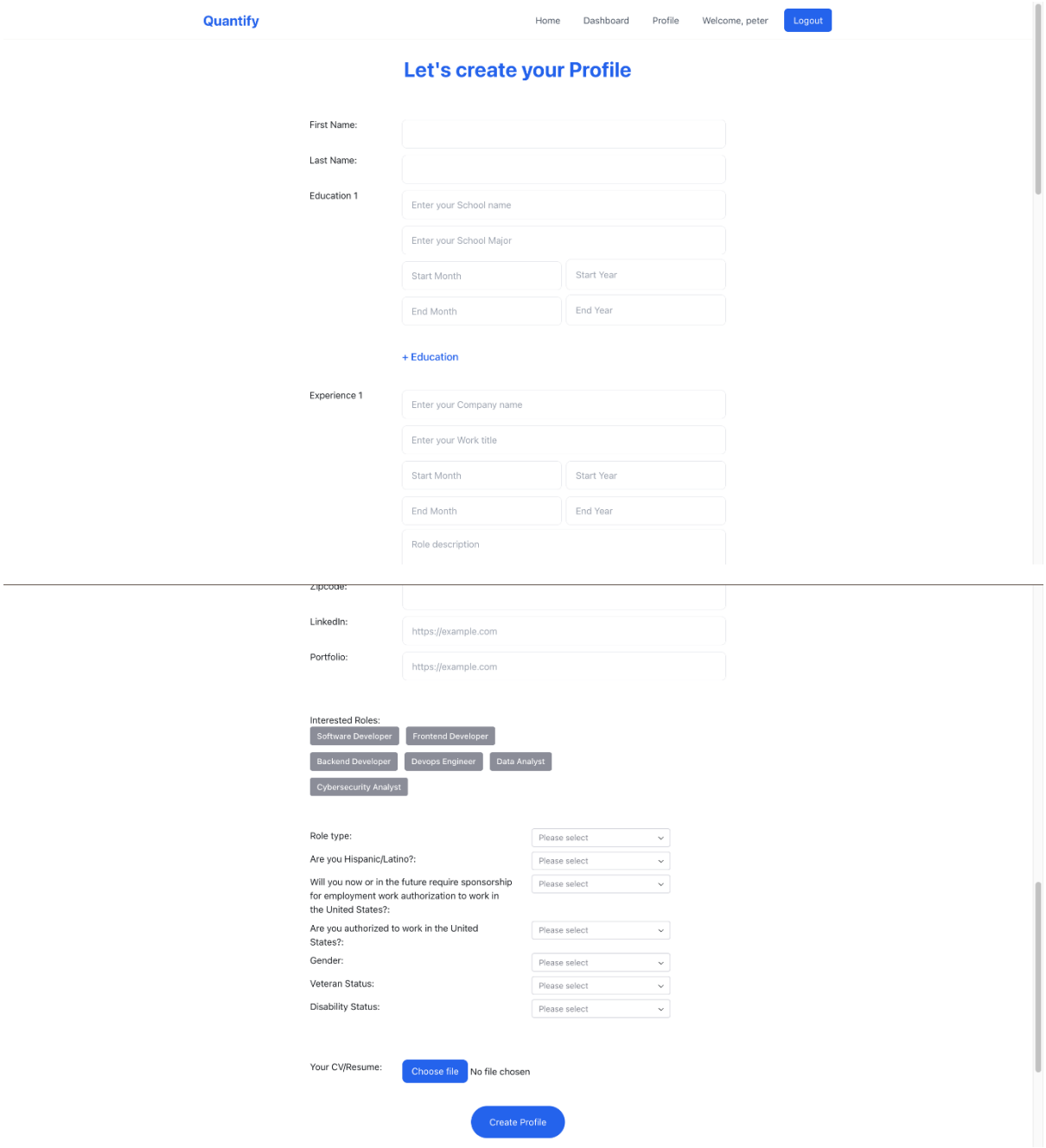
At the end of the home page, we have listed out the simple steps that the user must take to start auto filling job applications. We have used libraries like Radix UI and Tailwind CSS for creating UI elements and Icons. The frontend is built using React.



During the Sign-up process, the user will be prompted to enter their Full name, Email and password. We are using Firebase for the authentication purpose. Firebase will store the user's email address and password in a secure manner by appending a salt string to the password and using hashing algorithms like MD5 to create a hashed string. Firebase will only store the email address, the hashed password and the salt. In this way, we avoid storing the passwords in plain text and thereby improving security.

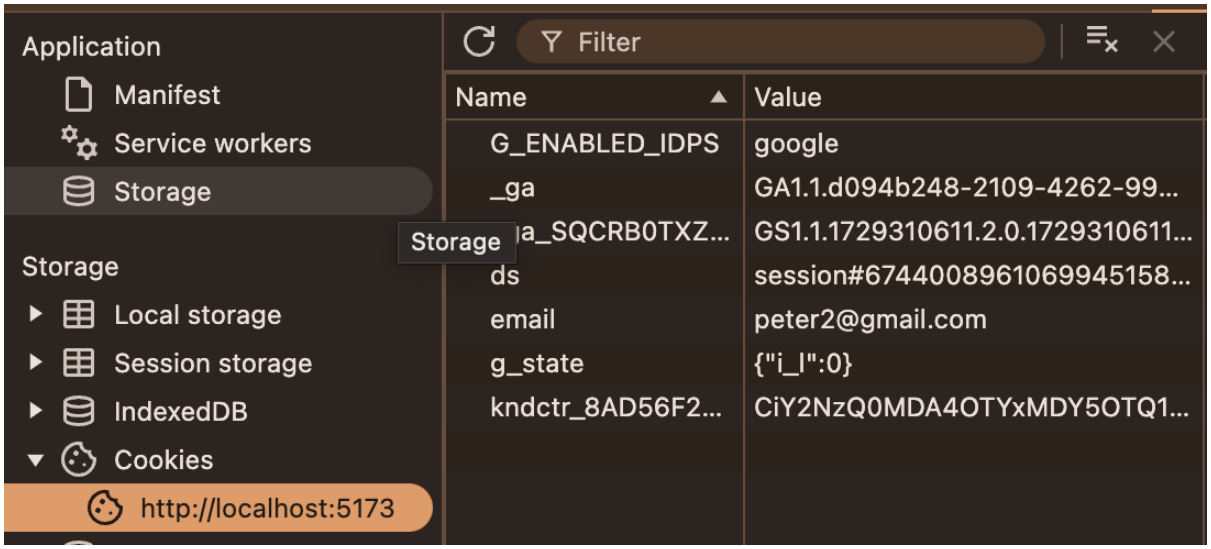


If the user already has an account, then can sign in with their email and password. The email address and the password combination are sent to Firebase for authentication. Firebase appends the salt string to the password and generates a hash. If the generated hash matches the stored hash, the user is successfully authenticated. If the hashes do not match, Firebase returns the appropriate error message.

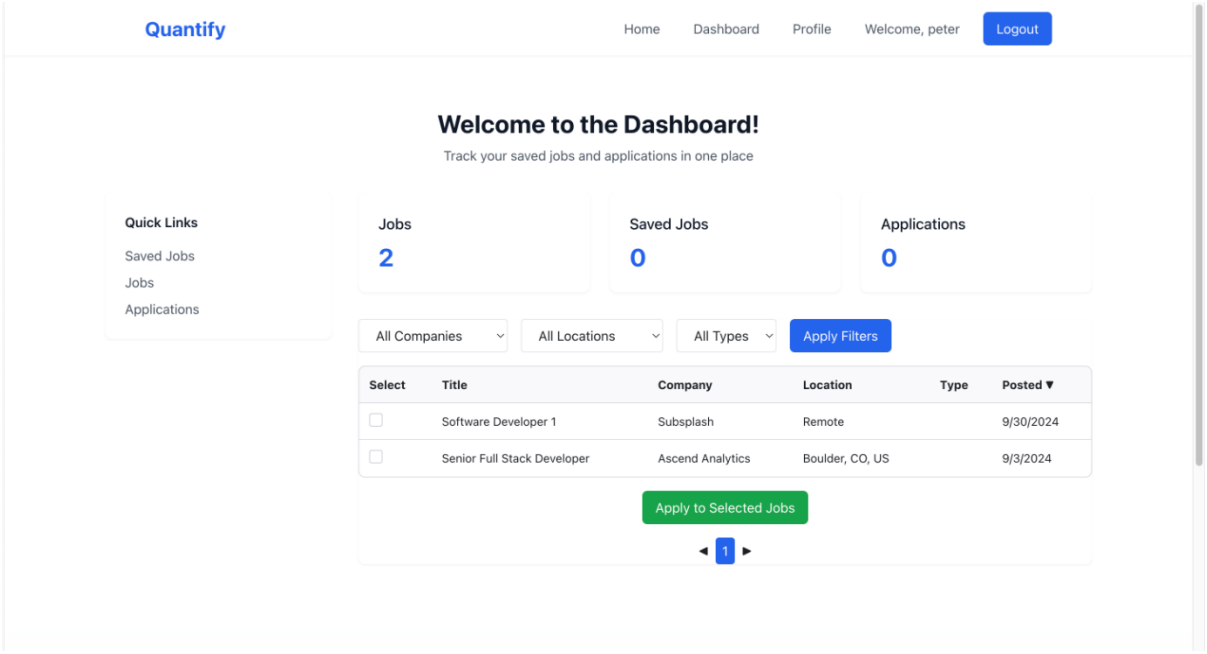


Once the user is logged in, they can create a profile. For every user, we collect their personal information, educational information, professional experience, skills, ethnicity, resume, etc., This information is stored in our database. We have used a MongoDB's Cloud enabled solution – Atlas, as our database because of its flexibility related to schema and also it lets us scale the cluster if in case the user base increases in the near future. The above two screenshots show the profile page.

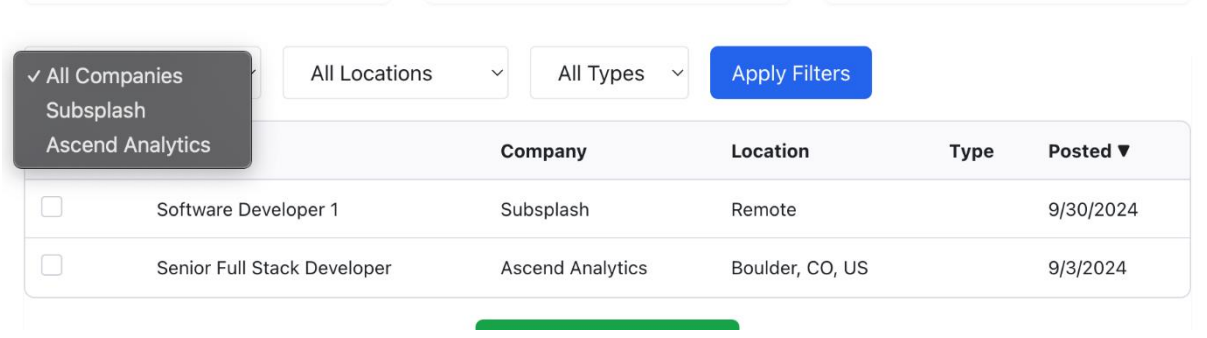
The user resume is stored in the server's file system and the path of the file is stored as part of the user's profile in MongoDB. The backend which is built using Python's Flask framework serves the resume as a static file.



Whenever a user logs in the dashboard, we create a cookie with the key “email” and the value is the user’s email address. This information is necessary because the browser extension will make use of the email address to fetch the user’s profile information and resume file from the backend.

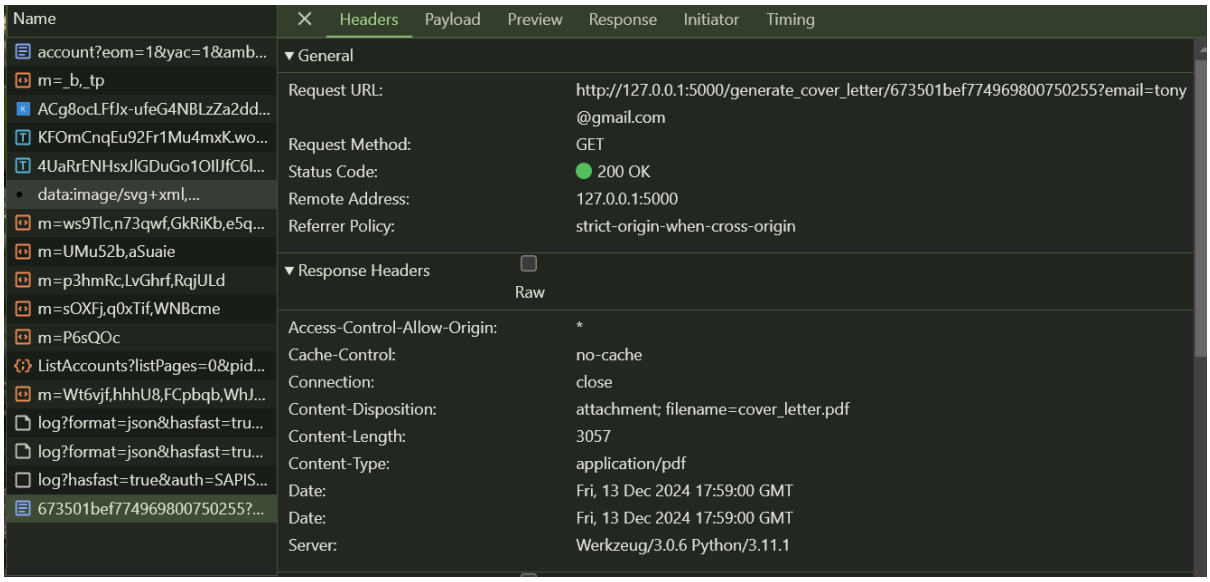


The dashboard page shows the jobs relevant to the user. The job scraper collects the job listings information from the “JSearch” API. As of now, the users can only see the job listings posted on the greenhouse.io platform since the browser extension only supports the greenhouse platform currently.

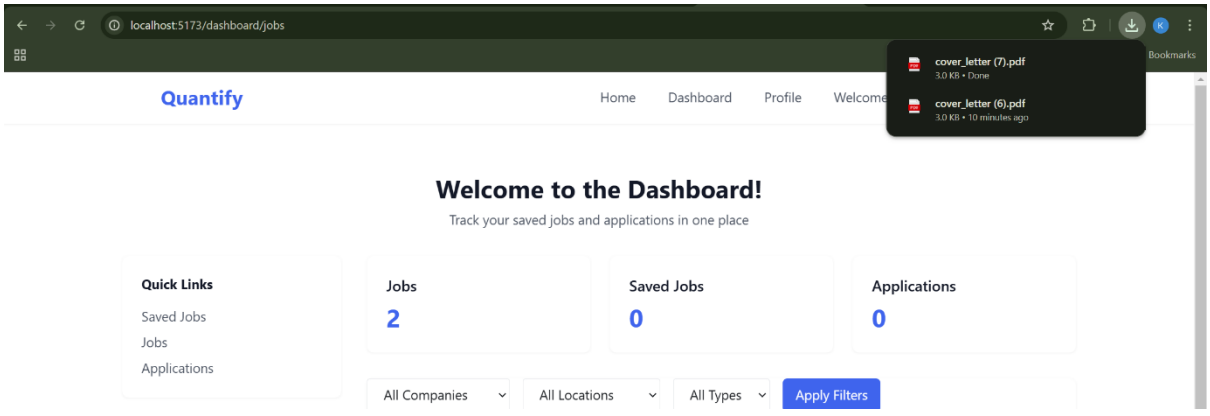


In the dashboard page, the user can filter the job listings by company, location and role type. The user can also sort the job listings based on the posted date by clicking the “Posted” column header. We have also created an API to generate tailored cover letter based on the job description. For a job listing, the system retrieves both the job description and the user's resume. Using a Retrieval-Augmented Generation (RAG) pipeline, the bot first generates embeddings for the job description and resume using HuggingFaceEmbeddings. These embeddings are then passed to the LLaMA 3-70b model to generate a personalized cover letter. To ensure the most relevant context is used, the bot uses cosine similarity to query an FAISS vector store to retrieve the job-resume context, ensuring that the generated content matches the user's profile

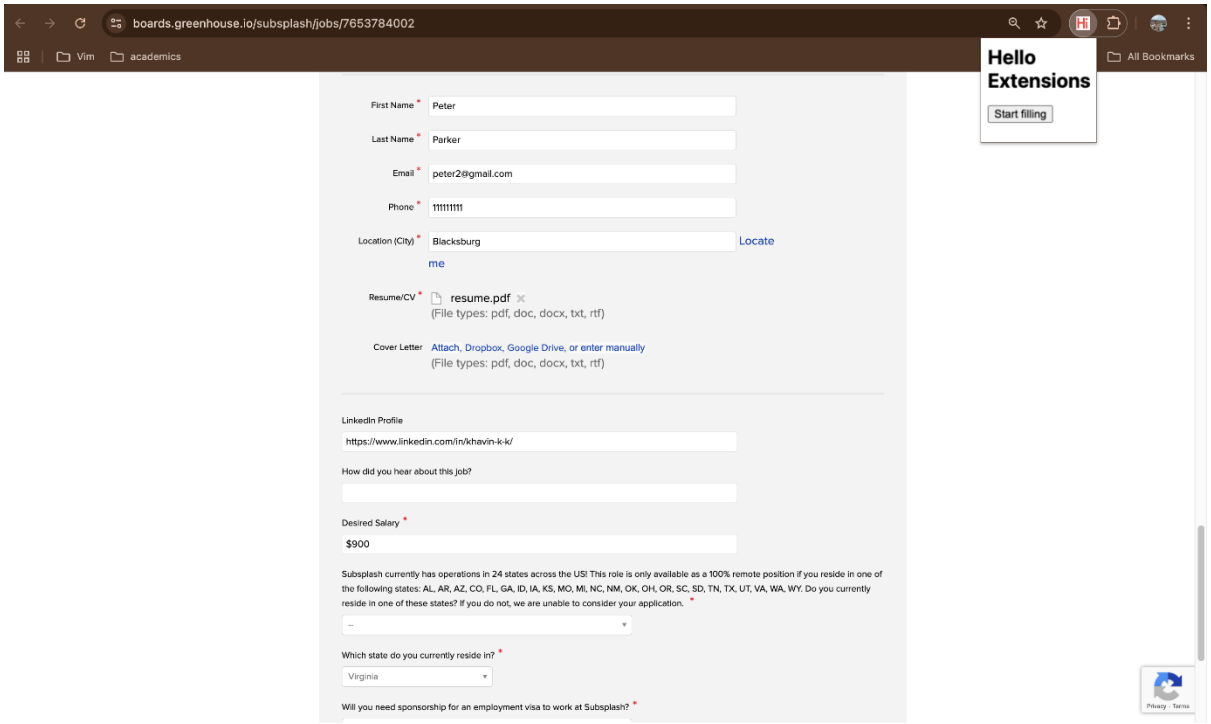
Once the cover letter is generated, it is formatted into a well-structured PDF using ReportLab's tool, Currently, the API works as expected and we can create a tailored cover letter based on the job description, however this feature is not yet integrated into the UI because of time constraints. In the future, we plan to incorporate this functionality directly into the dashboard.



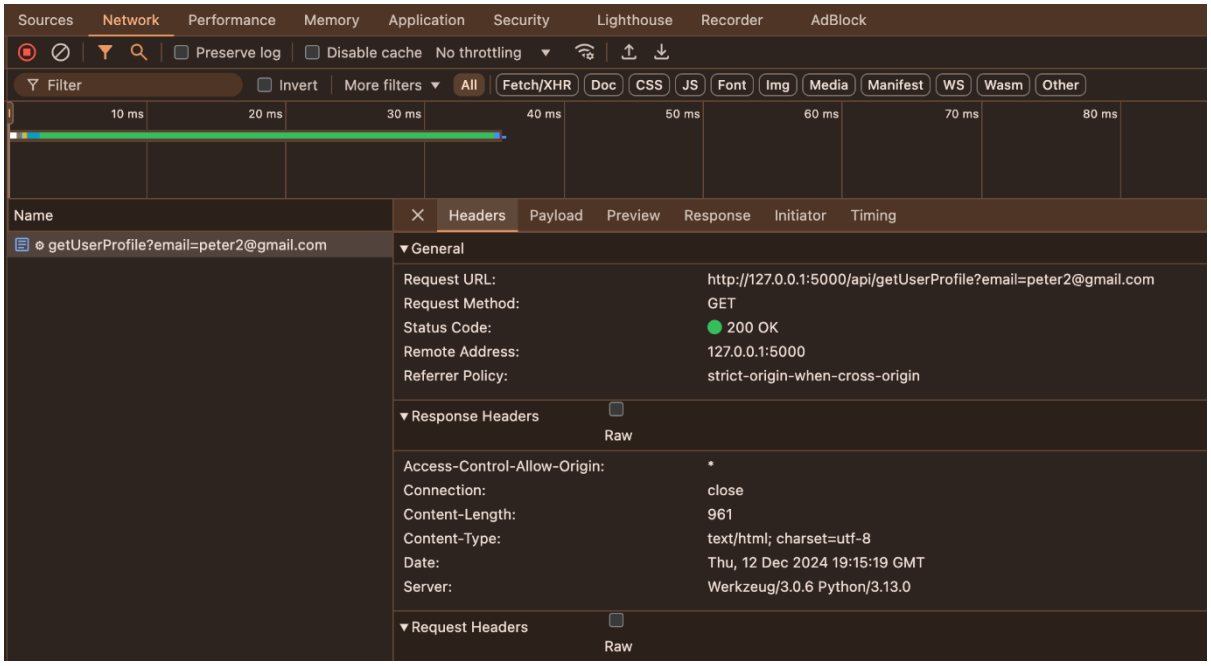
In the image above, we pass the job id as a path param to the API, which then returns the cover letter as a response.



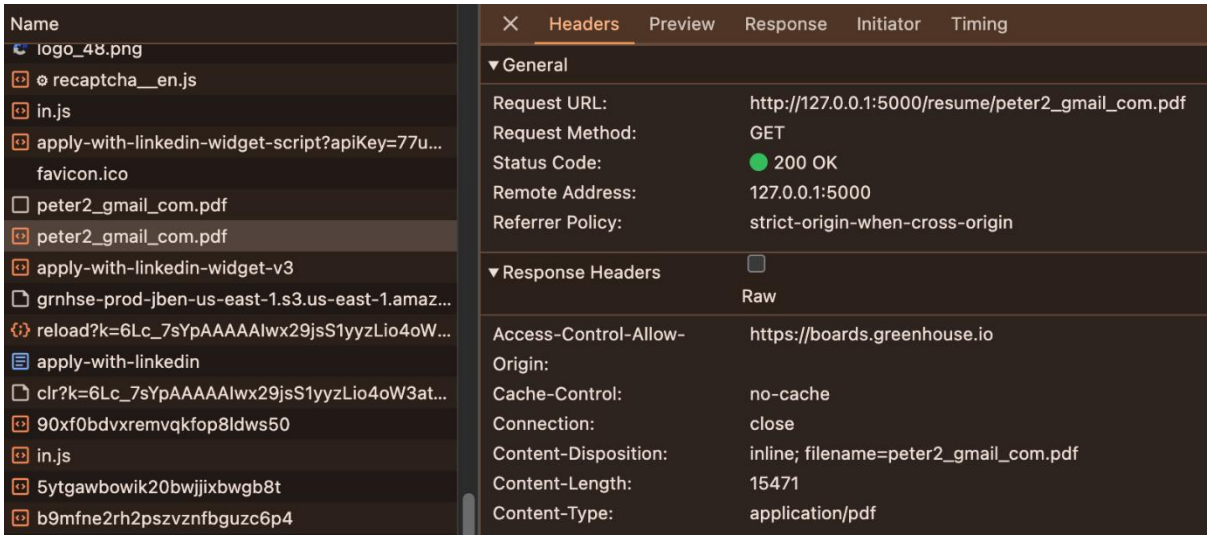
The user can select all the jobs that they would like to apply and click on “Apply to Selected Jobs” button. This in turn will open the job listings in different browser windows.



Once the job listings are opened in a browser window, the browser extension will autofill the application form with the user's profile data and uploads the user's resume.



The browser extension will fetch the logged in user information from the "email" cookie. From the above screenshot, you can see the browser extension fetching the user profile information from the backend while passing the logged in user's email address as a query param.



From the above screenshot, you can see the browser extension fetching the resume file from the backend.

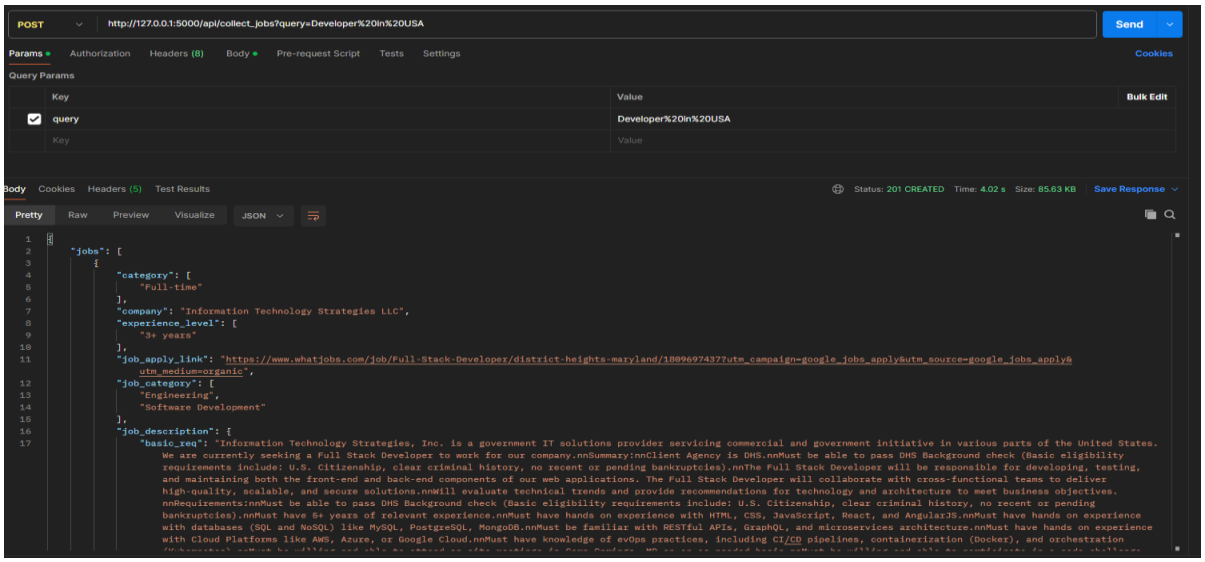
3. Design

3.1 Key Components

Quantify has three main components: the dashboard, the job scraper and the chromium browser extension.

Job Scraper

The main purpose of the Job Scraper is to periodically fetch the job listings from different job portals like Indeed, LinkedIn, etc., and push the listings to MongoDB. We are using the “JSearch” API to fetch the relevant job listings. The scraper collects key details such as job titles, companies, locations, salary ranges, and application links. It also allows us to search the stored data using queries, making it easy to retrieve relevant job listings. This scraper automates the job collection process and plays a crucial role in streamlining our job management system. It gives us the response something like below which then will be put in the DB.



Tech stack used: Python, Flask, MongoDB

From the start, we wanted to go with a NoSQL database because of their flexible schema structure. MongoDB is a popular NoSQL database with good support for Python. We are using the MongoDB Atlas service provider for hosting our MongoDB cluster.

Dashboard

The dashboard allows a user to Sign Up and create a profile. Once the user logs in to the dashboard, they can see the list of jobs that are relevant to them. After going through the list, the user can select the jobs that they wish to apply and click apply. This in turn will launch different browser windows, where the Chromium extension will take care of completing the application form. Whenever a user logs in to the dashboard, a cookie with the key "email" is set. The chromium browser extension will make use of this cookie to fetch the logged in user's profile data.

Tech stack used: React, Tailwind CSS, Radix UI, Python, Flask, Firebase

We have made use of Firebase for user authentication and storing credentials. We have also used libraries like Radix UI and Tailwind CSS to speed up the UI development process. We chose React and Flask because of their ease of use and the availability of large number of external libraries.

Chromium browser extension

The main purpose of the browser extension is to autofill the job application forms. The extension uses the above mentioned "email" cookie to identify the logged in user. The extension passes the logged in user to the Flask backend to fetch the user profile info and the resume.

Tech stack used: HTML, CSS, JS

We chose to build a chromium extension because the chromium-based browsers like Chrome and Edge accounts for nearly 72% of the browser traffic (https://gs.statcounter.com/browser-market-share). The current scope of this project targets job listings which have been hosted on the greenhouse.io platform. We plan to expand it further to other job boards in the future.

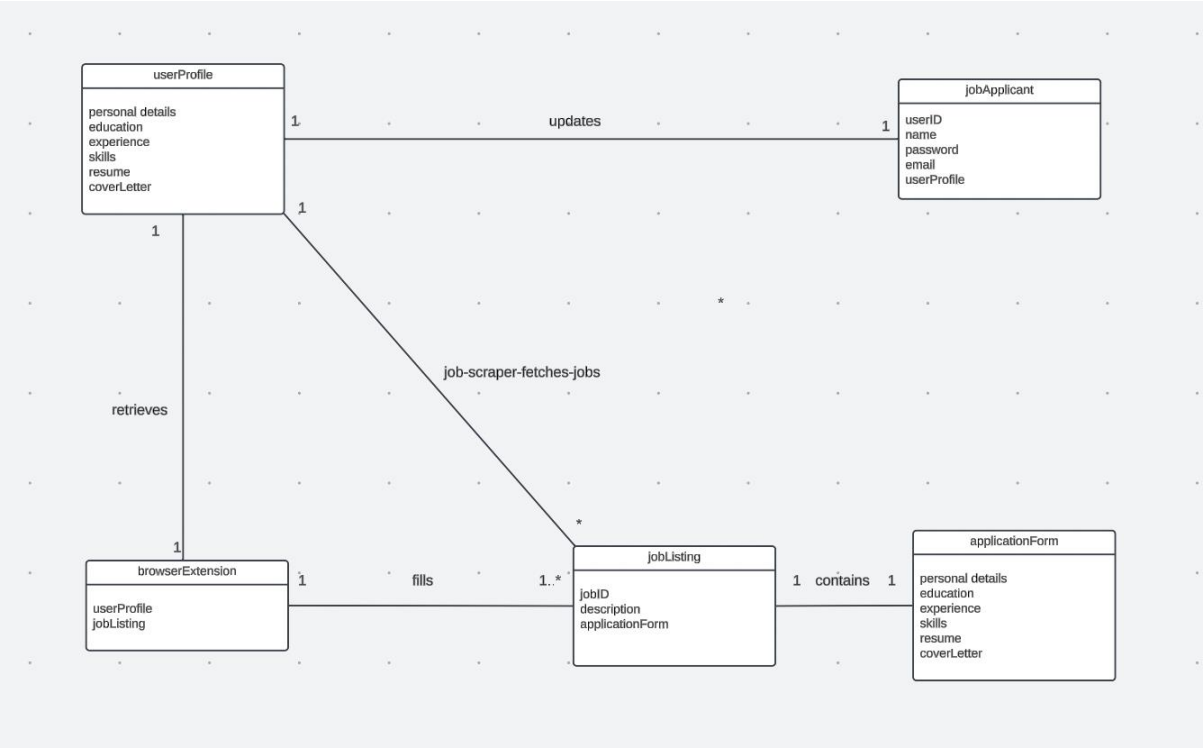
3.2 Cloud hosting

We've hosted our project on Azure, setting up resources to manage various components of the project. For this project, we've used Azure App service as it is an easy way to deploy our project and make the application go live. We have containerized the application using Docker and pushed our images to Azure Container Registry. In our App Service Plan, we have used Web App to deploy our services on a Single Container and storing all of our secrets as environment variables. Additionally, Azure App Service does all the heavy lifting by setting up a Load Balancer, auto scaling the application based on traffic and setting up DNS to access the application from the Internet.

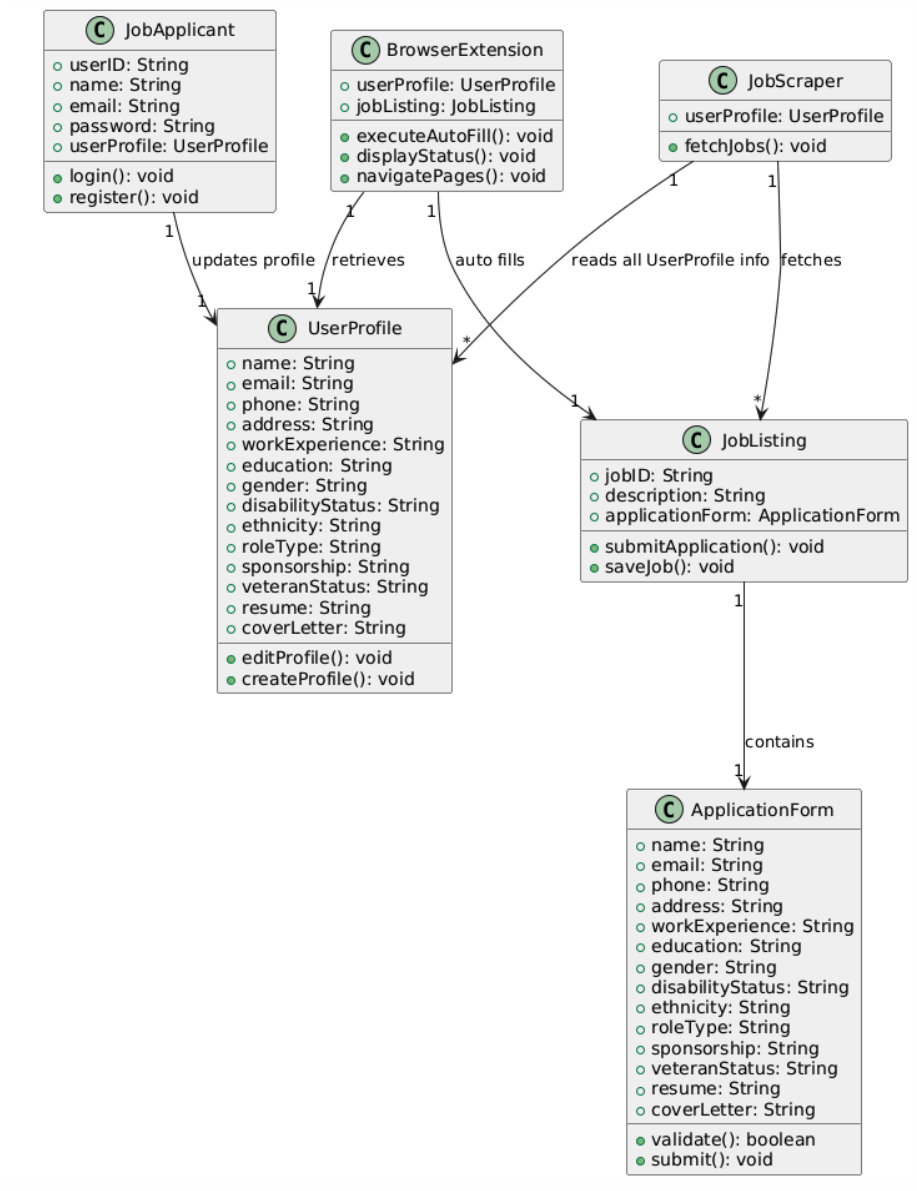
Most of our resources are in the West US region, though the container registry is in East US 2, because of resource availability and cost optimization in the West US region. This setup allows our Dashboard and Chrome extension to work hand in hand without any hiccups for the end user.

Name	Type	Location
ASP-Quantify-9ce5	App Service plan	West US
quantify	Container registry	East US 2
quantify	App Service	West US
quantify-api	App Service	West US
quantify-app_key	SSH key	West US

3.3 Domain model

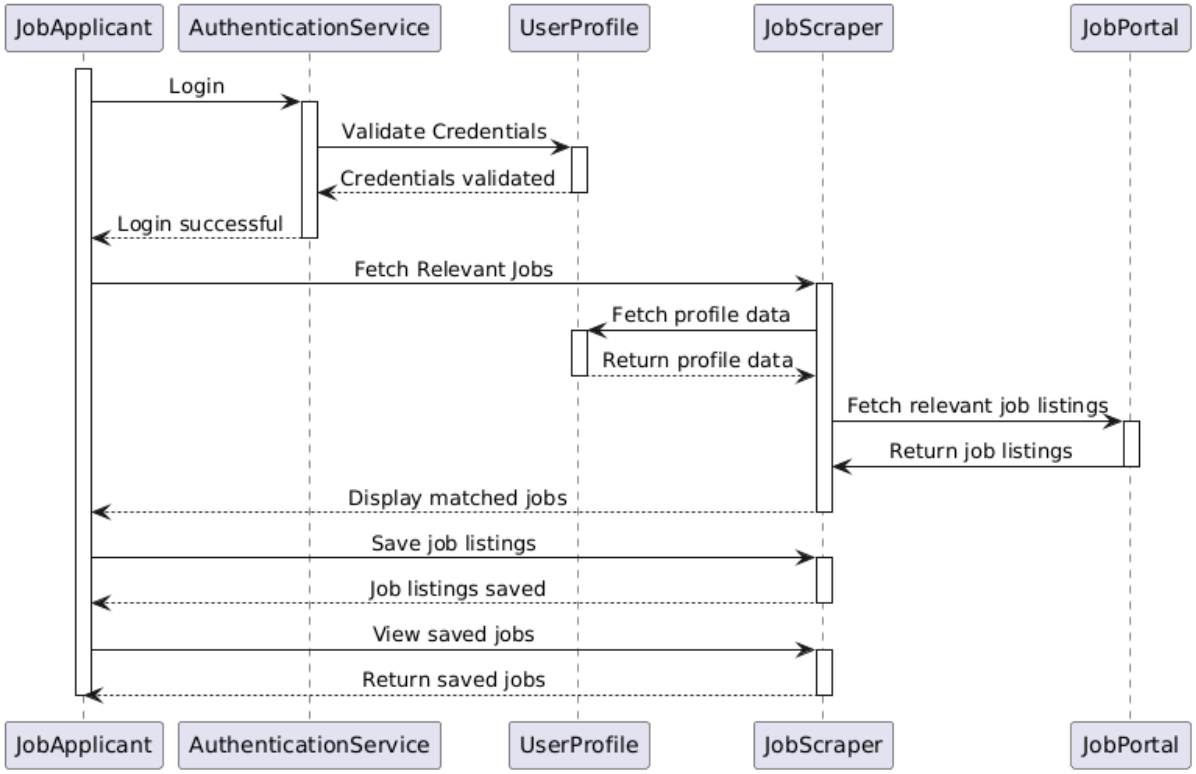


3.4 Design class diagram



The above diagram is our Design Class diagram.

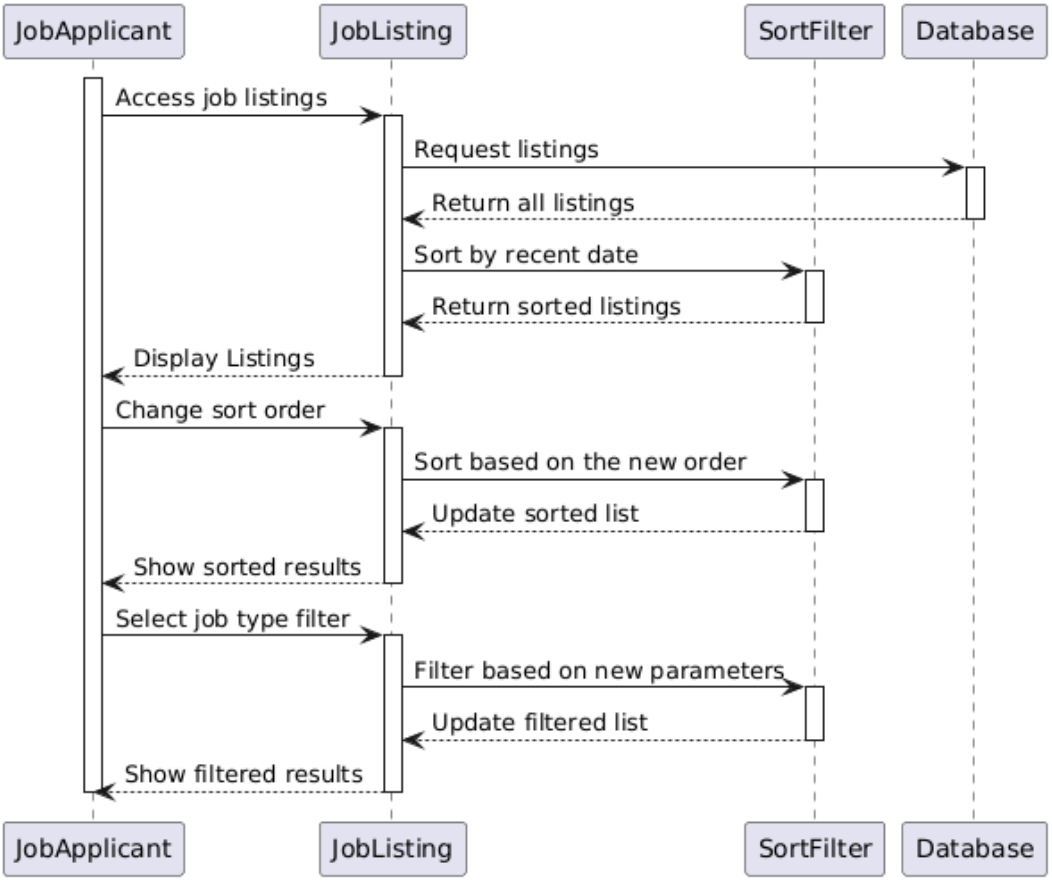
3.5 Sequence diagrams



The sequence diagram shows the time using rectangular blocks to denote that the objects were active during that time-period. User stories represented with the above sequence diagram

1. As a job applicant, I want to be able to Register/Login to the application.
2. As a job applicant, I should see jobs which matches my profile in the dashboard.
3. As a job applicant, I want to save jobs that interest me so that I can apply to them later.

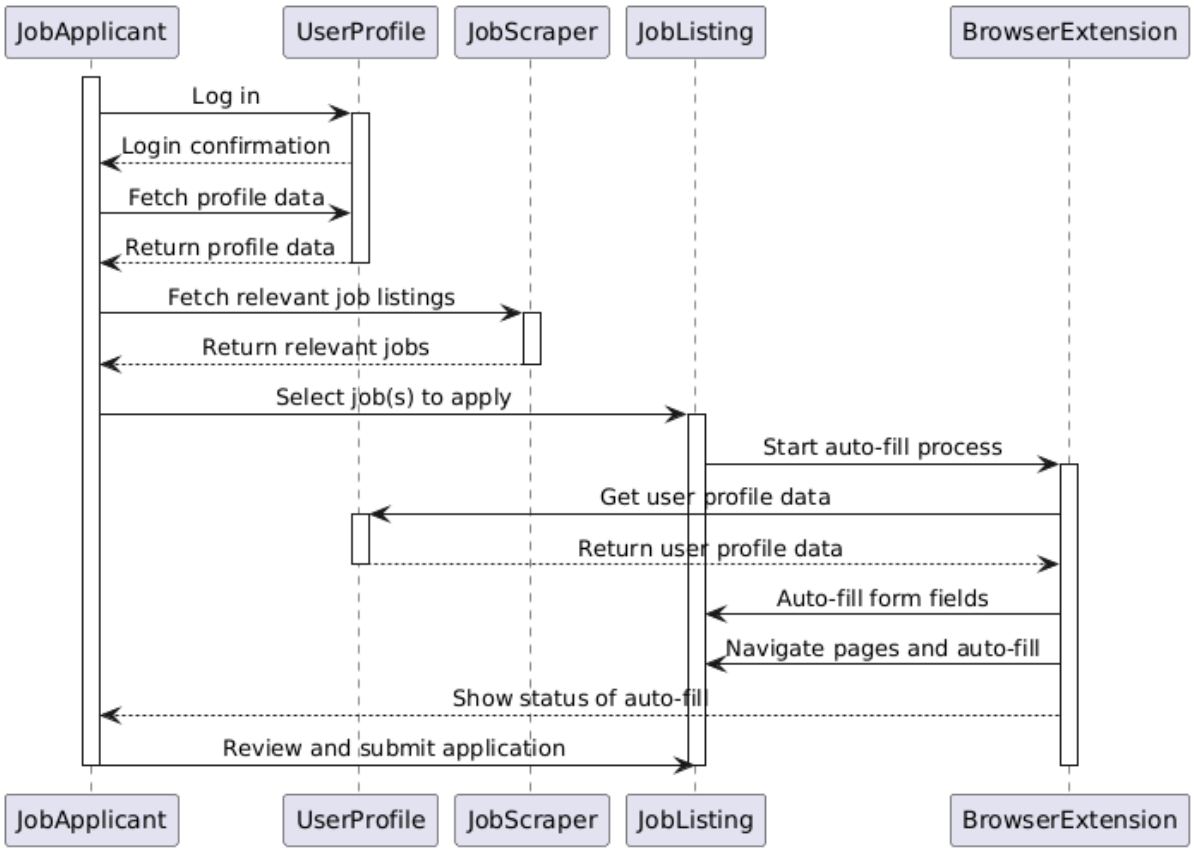
Here the domain object “JobPortal” refers to external job portals like LinkedIn, Indeed, etc.,



User stories represented with the above sequence diagram:

1. As a job applicant, I want to sort job listings by the date they were posted so that I can see the most recent opportunities.
2. As a job applicant, I want to filter jobs by type (e.g., full-time, part-time, contract) so that I can find jobs that fit my preference.

The main function of “SortFilter” domain object is to sort and filter the job listings based on the sort or filter requirements. The main function of the “Database” domain object is to store the job listings.



User stories represented with the above sequence diagram:

1. As a bot, I want to handle job application errors gracefully.
2. As a bot, I want to automatically fill out job application forms
3. As a bot, I want to fetch user information to auto-fill job applications.

3.6 Tools used

Frontend: React, React-router (for routing in the dashboard), RadixUI, Tailwind CSS

Backend: Flask, Python

Database: MongoDB (as part of MongoDB Atlas)

Machine Learning: LangChain, Llama 3-70B

Authentication: Firebase

IDEs used: VS code, Vim

Collaboration: Jira

Meetings: Zoom

3.7 Uniqueness

The product which is closest to our tool “Quantify” is an application called “Simplify”. However, “Simplify” does not provide any option to autofill multiple job applications at once. Quantify allows a user to multi select the jobs that they are interested in and autofill all of them with one single click. This

saves the user the manual hassle of opening a job listing in a new window and clicking an extension to autofill the application. With Quantify, the user simply must go through different windows, verify if the auto filled information is correct and click the Submit button. We have intentionally made the last step manual, as it gives users the opportunity to review the information before submitting their applications.

4. Retrospection

4.1 What went well and What went wrong?

During the initial phases of the project, our team struggled with communication, which led to duplication of efforts and confusion about task ownership. However, by the end of Sprint 1, we learned from this experience and started communicating much better. We decided to meet twice a week to discuss progress and work on blockers. Instead of just assigning the stories to a team member, we also assigned sub-tasks, this resulted in much better ownership of the tasks and removed ambiguities.

Regarding what went well, we were able to identify the missing skill sets properly and the team spent a lot of time bridging their knowledge gaps. The team members also aided others whenever necessary, and this resulted in a highly collaborative environment and as a side effect this also kept the team's morale high.

4.2 Learnings:

One of the major learnings from this class was project management. We learned how to organize a project, from creating user stories to managing sprints and conducting sprint retrospectives. On the technical side, we gained experience in developing a Chromium extension, building a professional-looking dashboard using React, and utilizing UI libraries. In addition to that, we also learned how to deploy an application to a cloud environment like Azure.

4.3 Actions that should have been taken to avoid the problems encountered while working on the project:

We miscalculated the generalizability of the Chromium extension to multiple job portals like Workday, ICIMS, Greenhouse, etc., The Document Object Model (DOM) structures of the job portals are completely different which required customized code for each one of them. In retrospect, we could have spent more time studying the DOMs of different job portals. This might have helped make the extension much more generalizable.

4.4 How could the product get better?

As of now, the browser extension supports only the job portal "boards.greenhouse.io". In future, we plan to extend this functionality to other popular job portals like Workday and ICIMS. This expansion will help reduce the amount of manual effort for users and enable them to apply for jobs more quickly and accurately. Additionally, we aim to develop extensions for Firefox and Safari, which will cover over 90% of the browser market share. Also, instead of depending on third-party API to scrape jobs, we plan to host an in-house solution to reduce cost and further enhancing the user experience.

5. GitLab Link

https://code.vt.edu/adityars/Quantify