PROJECT 3

Bhav Beri, Divij, Harshit Aggarwal, Jhalak Banzal, Pranav Agrawal

link to repository

TASK 1

Functional and Non-functional Requirements

Functional requirements:

- Job Listing addition/removal by admin
- Candidate login/registration
- View job listing
- Apply for listings by a candidate
- The recruiter can see details of candidates
- Candidate can see the current status of his application
- LinkedIn skill report generation of candidate
- Personality analysis using the Twitter profile of the candidate
- Report generation of analysis using an LLM service

Non Functional Requirements:

- Response time < 0.1ms
- Latency < 0.1ms
- Security- Role-based authentication and user data cannot be publicly accessed.
- Extensibility- The application should allow for the extension of validation checks and report generation methods without affecting other services/methods.
- Portability- The application should work across multiple OSes: Windows, Linux, and MacOS.

Key Requirements:

- **Report Generation system**

  This is a complex process that leverages several resources from external services that include computationally intensive operations. This is due to the existence of multiple LLMs that parse, analyse, and present the data.

  Due to the sheer scale of this system, it is indispensable that the system resources are allocated judiciously. The different computational requirements of different parts of the system necessitates them being scaled independently of each other, making it the perfect architecture for us.

- **Candidate Application portal**

  The candidate can apply for a listing simply by providing his social media handles. This is very little information travelling over the network, and would usually involve many people applying often.

  Thus, this makes the usage of REST ideal in this scenario, as it would ensure efficiency in our purposes and also maintain better security.

- **Confidentiality of information**

  As this system would store sensitive data about its users, security becomes one of our primary concerns.

  Moreover, malicious use of our backend APIs would give unfair advantages to our candidates, leading to an unfair and potentially unqualified workforce in the companies of the industry. Therefore, a need for proper separation of the frontend and backend arises, such that none of our API endpoints are exposed to any external party.

  Additionally, having a separated system ensures that a malicious party gaining access to a single node would not compromise the entire system.

Subsystem Overview

1. **User Profile**: Users can create and edit their profile, which consists of various details. We have 3 different kinds of users namely:

   a. Admin: Admin can Make/ Delete new job listings.

   b. Recruiter: The recruiter can see all the applications, generate reports for the application and approve them.

   c. Candidate: The candidate can apply in a particular listing and check all the listings.

2. **Sentiment Analysis**: Allows to predict the sentiment (negative, neutral, positive) of candidates' social media posts using BERT.

3. **Red Flags Detection**: The Recruiter can identify potential issues like sexism and hate speech using the candidates' social media activity.
4. **MBTI Classification**: The Recruiter can see the candidates' MBTI personality types based on the scraped Twitter posts by the candidate.
5. **Skill Analysis**: This scans candidates' LinkedIn profiles to identify skills and suggest suitable job roles based on a mapping provided by LinkedIn.
6. **Report Generation with LLM**: The recruiter can generate a human-readable report summarising candidates' personalities, behaviours, and cultural fit using LLMs and suitable jobs.

TASK 2

Stakeholder Identification

stakeholder	concerns	viewpoints	views addressing these concerns
Technical Recruiters and hiring managers	Need the system to provide brief and accurate assessment of the technical abilities of the candidates to be able to go through many candidates quickly	They want the information obtained from the linkedin profile to be relevant and accurate	The skills and experience of the candidates in the generated report
HR professionals	Wants the candidates to be polite and professional and not be a nuisance to the workspace	They want the information about the personality of the candidate taken from twitter to be relevant	Sentiment of the twitter profile of the user and problematic tweets made by the candidate
Candidates	Needs a platform to apply for new jobs and to have the system profile an accurate assessment of abilities and personality	They want a useful application interface	Interface for applying to job postings

Major Design Decisions

ADRs:

1. Choose REST over GraphQL
2. Microservices Architecture
3. MongoDB
4. Python - Flask, FastAPI

Attached at the end.

TASK 3

Architectural Tactics

Availability Tactics:

- **Ping**: *Availability*: Ping sends a network packet to a destination and measures the time taken for a response, ensuring timely availability of the network resource.
- **Exception**: *Reliability*: Exception handling ensures reliability by gracefully managing unexpected errors or situations, preventing system failures, and maintaining stability.

Performance Tactics:

- **Introduce Concurrency**: *Availability and Scalability*: Concurrency enables availability by allowing multiple tasks to run simultaneously, reducing downtime, and enabling scalability by efficiently utilising resources to handle increasing workloads without performance degradation.
- **Maintain Multiple Copies**: *Availability and Scalability*: Maintaining multiple copies ensures Availability by providing redundancy, allowing quick access to data or services in case of failures, and Scalability by distributing load across replicas, preventing bottlenecks and accommodating growing demand.
- **Increase Available Resources**: *Availability and Scalability*: Increasing available resources enhances Availability by ensuring sufficient capacity to handle requests, reducing downtime due to resource constraints, and Scalability by enabling the system to accommodate growing workloads without performance degradation.

Security Tactics:

- **Authenticate: Security**: *Authentication* verifies the identity of users or systems, enhancing Security by ensuring that only authorized individuals or entities access resources or data, preventing unauthorized access and potential breaches.

- **Authorise: Security**: *Authorization* specifies what actions users or systems are permitted to perform, enhancing Security by ensuring that only authorised entities have access to specific resources or functionalities, preventing unauthorised activities and maintaining data integrity.

- **Maintain Data Confidentiality**: *Security*: Maintaining data confidentiality ensures Security by encrypting sensitive information, restricting access to authorised users, and preventing unauthorised disclosure or leakage of sensitive data, safeguarding privacy and compliance with regulations.

- **Maintain Integrity**: *Securit*: Maintaining integrity ensures Security by validating data to ensure it remains unchanged and uncorrupted, preventing unauthorised modifications or tampering, and maintaining trustworthiness and accuracy of information.

- **Limit Exposure**: *Security*: Limiting exposure reduces the attack surface by minimising the accessibility of sensitive resources or information, enhancing Security by reducing the potential for exploitation and unauthorised access, and safeguarding against security threats and vulnerabilities.

- **Limit Access**: *Security*: Limiting access restricts user permissions to only necessary resources or functionalities, enhancing Security by minimising the risk of unauthorised access, preventing data breaches, and ensuring confidentiality and integrity of sensitive information.

Modifiability Tactics:

- **Semantic Coherence**: *Maintainability*: Semantic coherence ensures Maintainability by maintaining consistency and clarity in code or documentation, facilitating easier understanding and modification, reducing errors, and promoting efficient maintenance and evolution of the system over time.

- **Hide Information**: *Extensibility*: Hiding information promotes Extensibility by encapsulating implementation details, allowing for changes or additions to be made to the system without affecting external interfaces, reducing dependencies and enabling easier modifications or enhancements.

Usability Tactics:

- **Separate UI from the rest of system**: *Maintainability*: Separating the UI from the rest of the system enhances Maintainability by isolating presentation logic, facilitating easier updates or changes to the user interface without affecting underlying functionality, promoting code modularity and simplifying maintenance.

Implementation Patterns

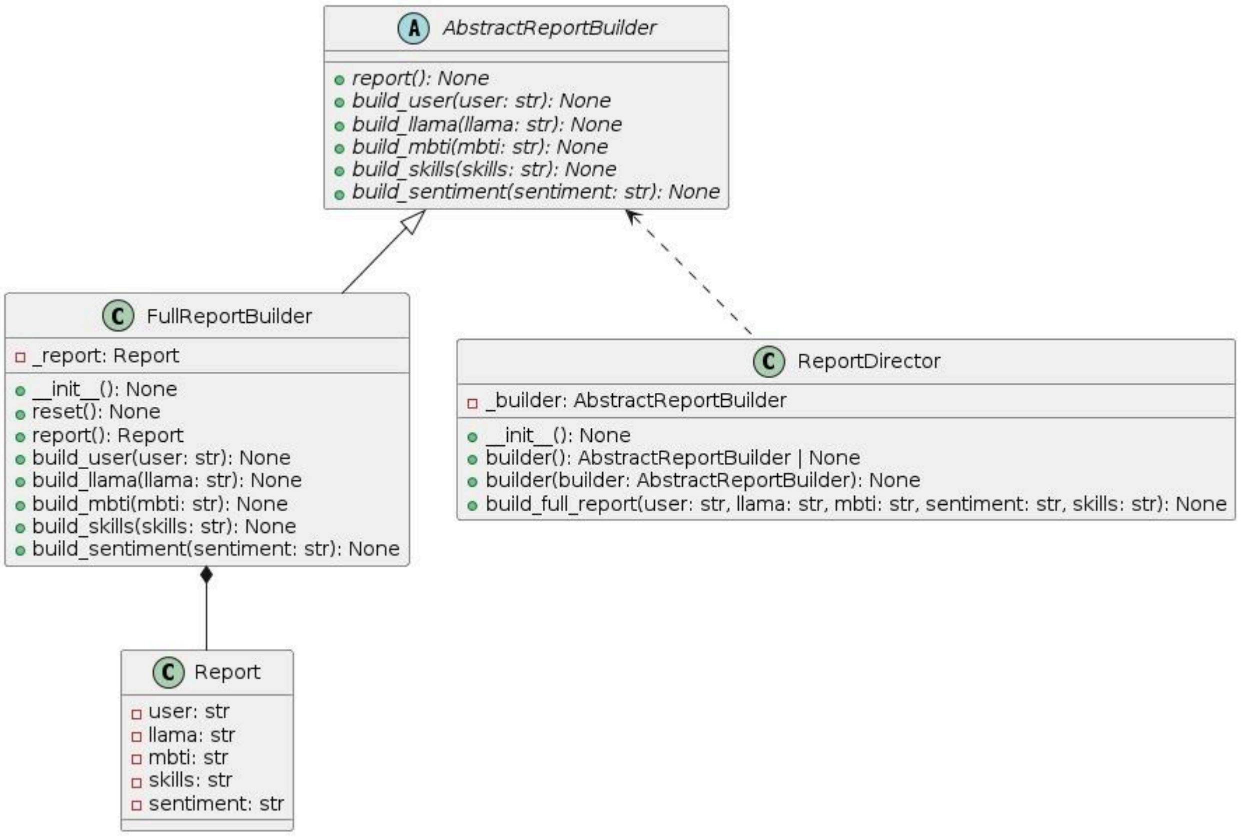
1. **Builder Pattern**:

   The central part of our system is the final report generated by the system, which gives an overview of the candidate's technical skills and personality traits.

   We use several microservices to get different parts of the report. We use the builder pattern to aggregate all the information into a standard format. The rationale behind this is to provide extensibility in adding more types of analysis fields in the report. Since Builder Pattern also provides a director, we can provide various ways of how to build a report, for example:

   - FullReportBuilder() builds a report with all analysis done for a high-level employee.
   - MinimalReportBuilder() only invokes a few analysis services and can be used for an employee not joining a high-level position.

   This would allow the company to avoid wasting resources building complete reports for all job applications.

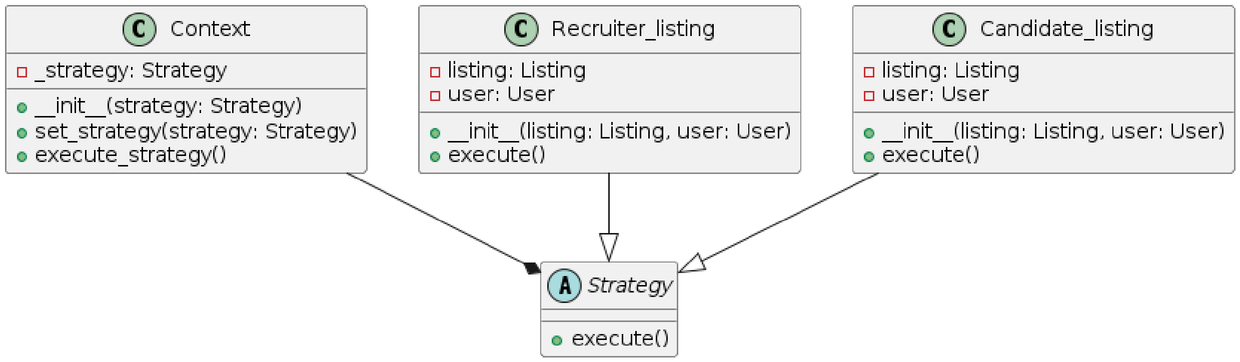


A AbstractReportBuilder
report(): None
build_user(user: str): None
build_llama(llama: str): None
build_mbti(mbti: str): None
build_skills(skills: str): None
build_sentiment(sentiment: str): None
C FullReportBuilder
_report: Report
__init__(): None
reset(): None
report(): Report
build_user(user: str): None
build_llama(llama: str): None
build_mbti(mbti: str): None
build_skills(skills: str): None
build_sentiment(sentiment: str): None
C ReportDirector
_builder: AbstractReportBuilder
__init__(): None
builder(): AbstractReportBuilder | None
builder(builder: AbstractReportBuilder): None
build_full_report(user: str, llama: str, mbti: str, sentiment: str, skills: str): None
C Report
user: str
llama: str
mbti: str
skills: str
sentiment: str

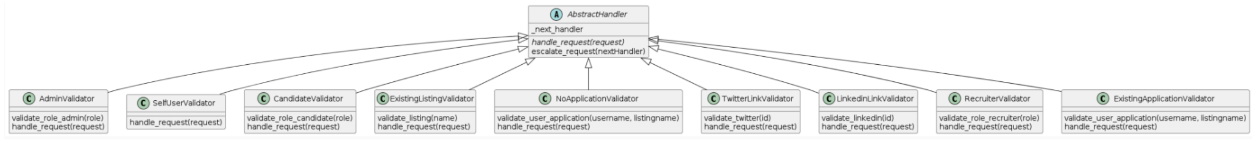
2. **Strategy Pattern**:

   To get user applications, the applications returned depend on the type of user. A recruiter should be able to see all applications for all listings. Meanwhile, the candidate should only be able to see his/her own applications.

   The implementations for both of these functionalities are largely the same, with just minor differences between them in terms of which utility functions are being called. Since the parameters shared between both are same, it was a prime candidate for implementing the strategy pattern to ensure future maintainability and extensibility



3. **Chain of Responsibility**: Chain of Responsibility is applied for validation checks, both when a user is trying to log in or whether a user is trying to apply/perform another applications-related task. This provides authorization capability in our system and extracts it from the main code to a separate implementation.



TASK 4

Prototype Development

Refer to github link

Architecture Analysis

We compare our implemented pattern, i.e. Microservices Pattern against the Monolith Pattern.

Non-functional requirements to be quantified:

		Response Time	Throughput	Latency
Microservices	creating listing	0.0643	348.823	0.061823
	applying listing	0.0635	165.281	0.062714
	getting report	3.6642	1.06647	3.663648
Monolith	creating listing	NA	NA	NA
	applying listing	NA	NA	NA
	getting report	2.7928	1.36564	2.68439

Trade Offs

1. **Scalability**

In the implementation of the monolithic architecture, scaling posed significant challenges. With the need to replicate the entire application for any scalability adjustment, resource allocation often became inefficient, leading to underutilization of resources and increased operational costs. In contrast, the microservice architecture demonstrated superior scalability.

By enabling granular scaling of individual services based on their specific resource requirements and workload, we achieved optimal resource allocation and cost-effectiveness, resulting in improved performance and scalability.

2. **Deployment Agility**

Deploying updates in the monolithic architecture proved to be cumbersome and time-consuming. Modifications to the entire codebase required extensive testing and validation, leading to longer deployment cycles and heightened risks of disruptions.

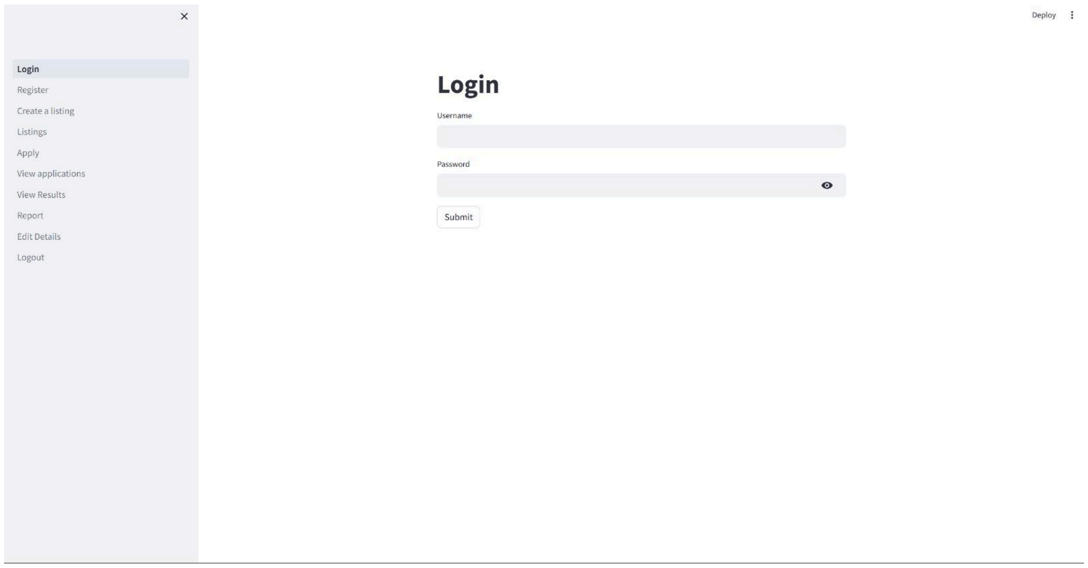
Conversely, the microservice architecture exhibited remarkable deployment agility. With each service operating independently and capable of being developed, tested, and deployed autonomously, we experienced faster iteration cycles and reduced deployment timelines. This agility facilitated the adoption of continuous delivery practices, allowing for rapid and frequent updates with minimal disruptions to other services.

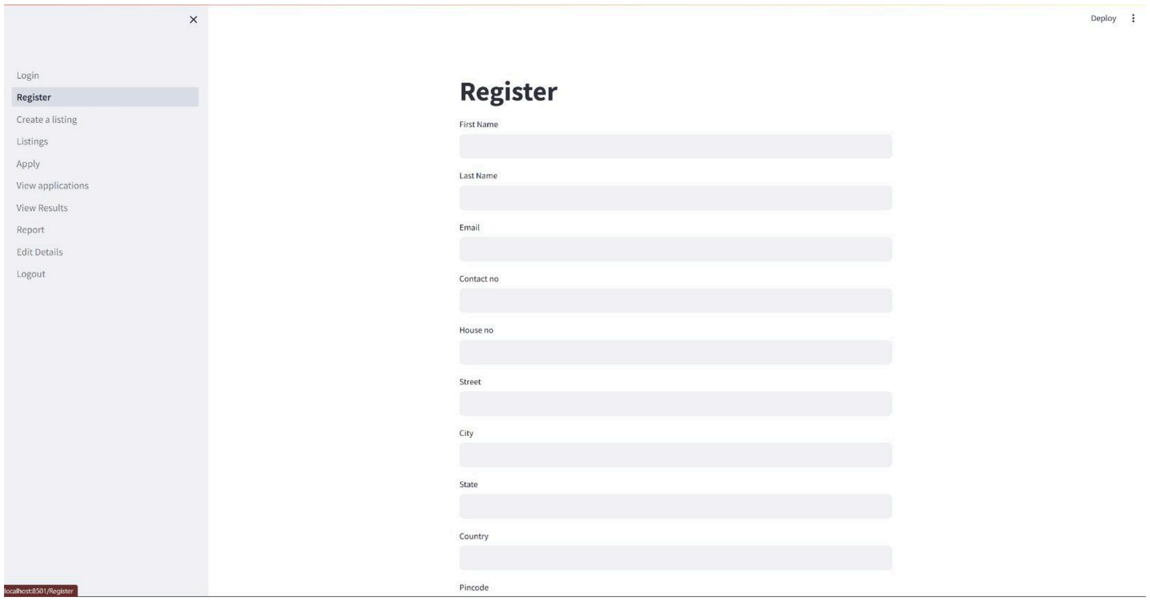
3. **Fault Isolation**

Fault isolation emerged as a critical consideration in our comparison. In the monolithic architecture, identifying and mitigating faults presented significant challenges. With tightly coupled components, failures in one part of the application often propagated throughout the system, resulting in widespread outages and prolonged downtime. In contrast, the microservice architecture demonstrated superior fault isolation capabilities.

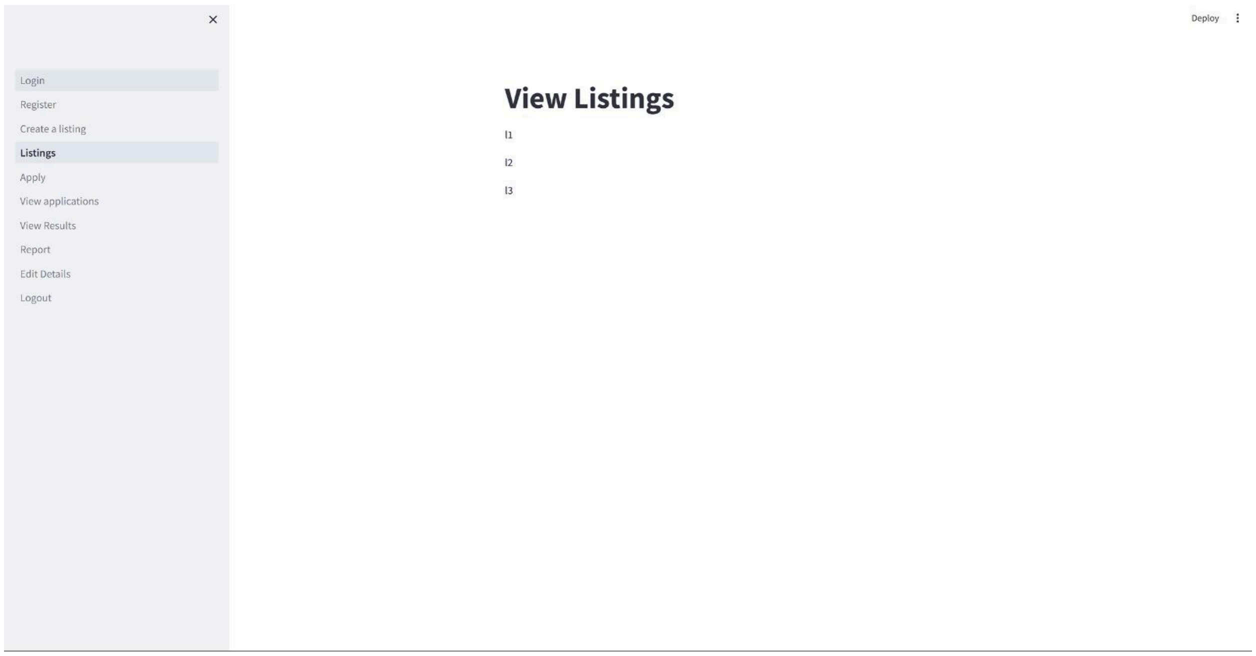
By operating each service independently with its own set of resources and dependencies, we contained failures within individual services, minimising their impact on other components. This enhanced fault isolation improved overall system resilience and availability, as failures in one service did not cascade to other parts of the system.

Prototype preview





Listings Page (to view the listings):



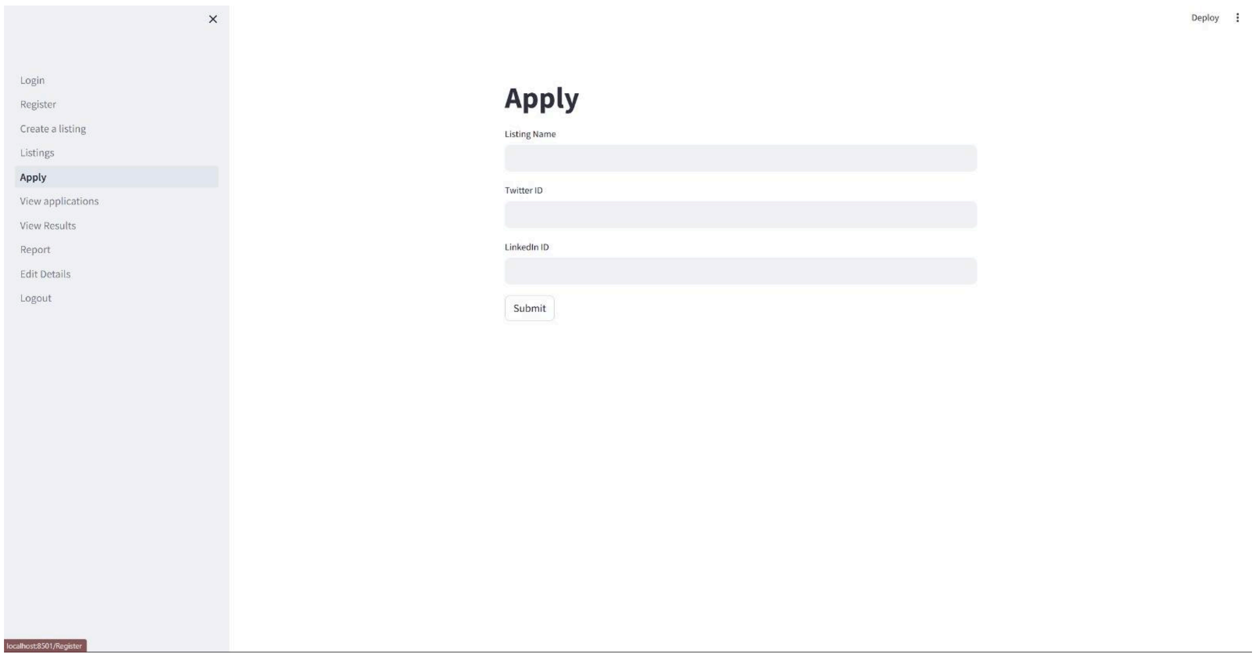
View Listings

l1

l2

l3

Apply Page (to apply to a listing):



Apply

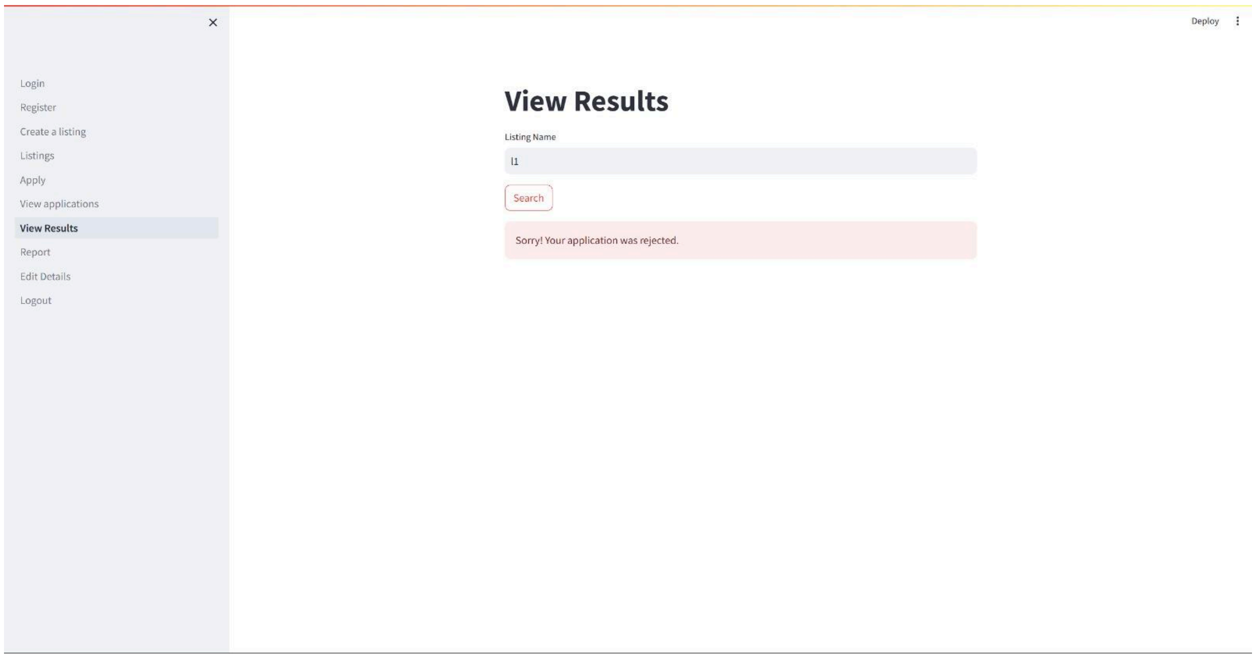
Listing Name

Twitter ID

LinkedIn ID

Submit

View your result (as a candidate):



View Results

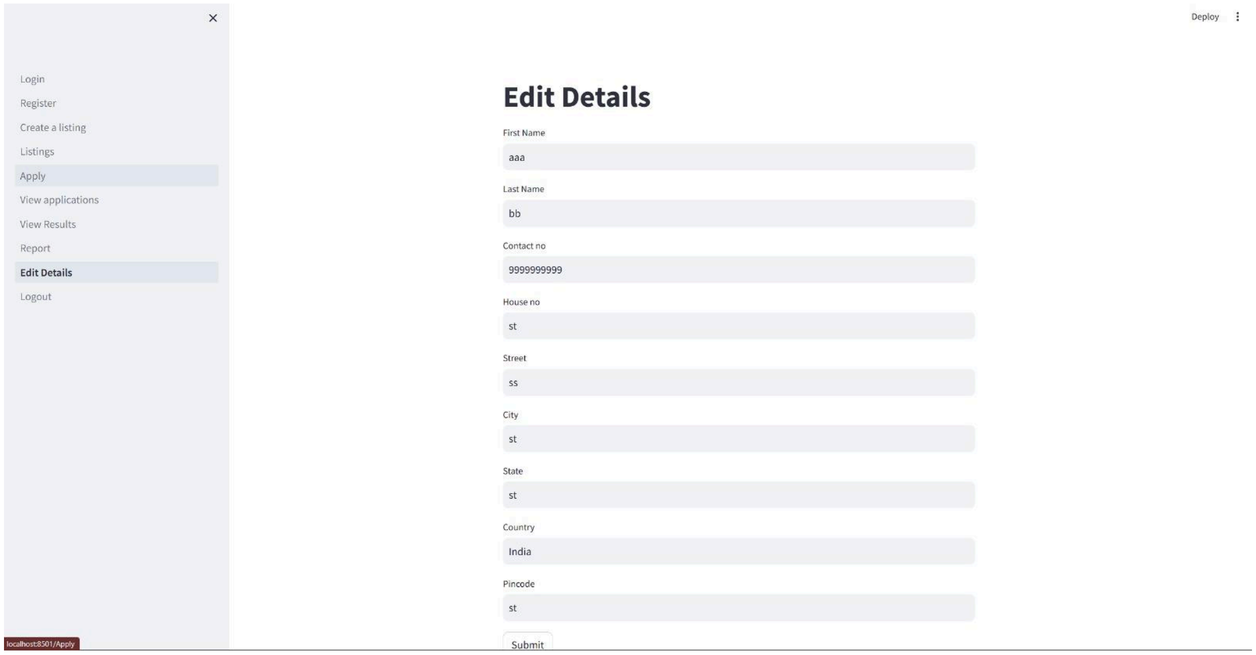
Listing Name

l1

Search

Sorry! Your application was rejected.

Edit Details:



Edit Details

First Name: aaa

Last Name: bb

Contact no: 9999999999

House no: st

Street: ss

City: st

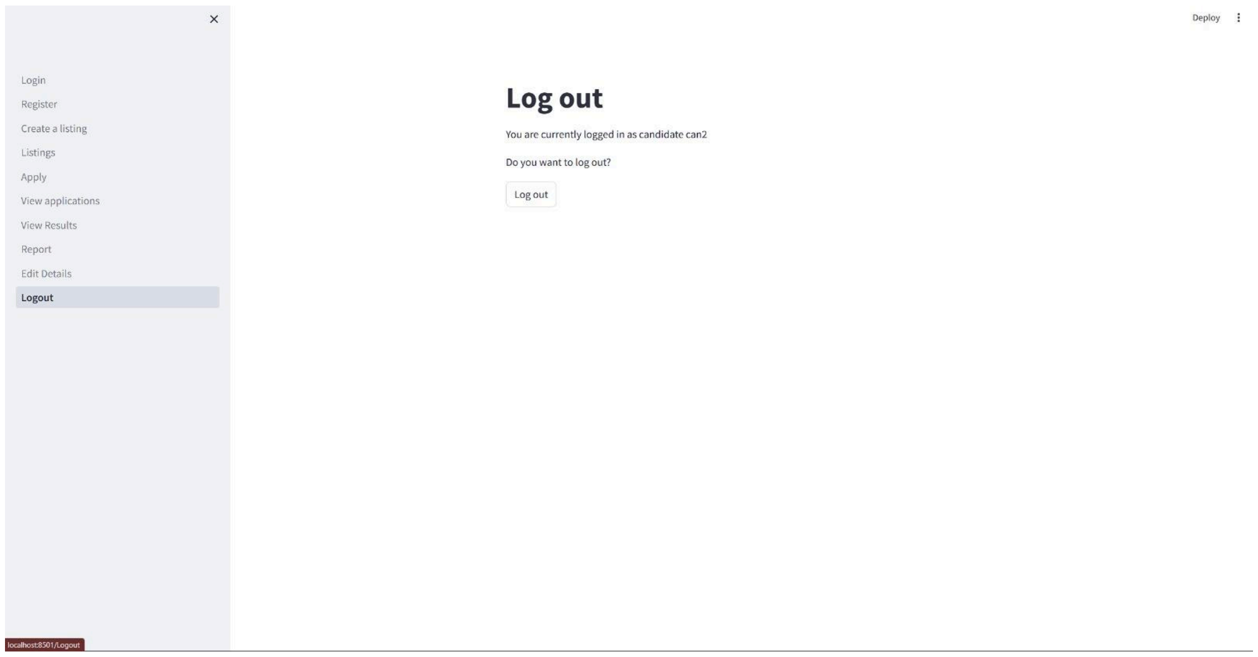
State: st

Country: India

Pincode: st

Submit

LogOut:

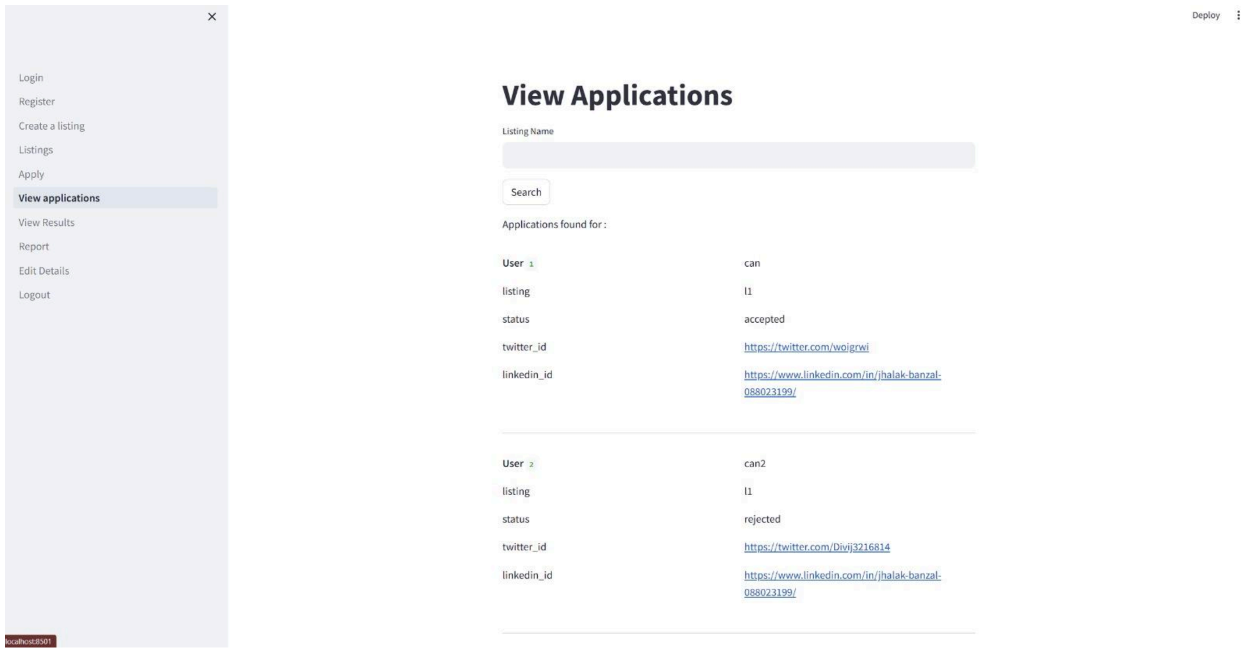


Log out

You are currently logged in as candidate can2

Do you want to log out?

View Applications (to see all the applications) (as a recruiter):



View Applications

Listing Name

Applications found for :

User 1	can
listing	l1
status	accepted
twitter_id	https://twitter.com/woigrwi
linkedin_id	https://www.linkedin.com/in/jhalak-banzal-088023199/
User 2	can2
listing	l1
status	rejected
twitter_id	https://twitter.com/Divij3216814
linkedin_id	https://www.linkedin.com/in/jhalak-banzal-088023199/

individual contributions:

Monolithic: Divij, Bhav
Microservices: Everyone (main part)
Setting up basic microservices: Bhav
Applications microservices: Harshit
User microservices: Bhav
Listings microservices: Divij
AI microservices: Pranav
Frontend: Jhalak
Documentation: Jhalak, Pranav, Harshit

Decision record template for Mocroservices

Introduction

1. Prologue (Summary)
2. Discussion (Context)
3. Solution (Decision)
4. Consequences (Results)

   ## Specifics

5. Prologue: In the context of the system architecture, we face the issue of large load on servers, so, we decided for MicroServices to achive maintanability and scalability accepting higher complexity.
6. Discussion: We face the issue of large load on a single monolithic server due to large AI models for the report generation. Moreover, the failure in one server should not bring down the entire system. We also need to cater with the user need to add more features to the report. Since the users come from diiferent companies and thus the requirements can be different which will need us to extend the services in the future. So, we need easier scalability. Also, failure of one server should not affect the entire system.
7. Solution: We decided to use Microservices Architecture to achieve:
   1. Scalability: Microservices allows individual components of an application to scale independently based on demand.
   2. Flexibility: Each microservice operates independently, enabling us to choose the best technology stack for each component of the AI.
   3. Resilience: Failure in one microservice typically won't bring down the entire system.
   4. Continuous Delivery and Deployment: Microservices promote continuous integration and delivery practices. We can add further festures in the report if the user needs the same.
   5. Easier Maintenance: With smaller, focused services, it's easier to understand, maintain, and debug the system.
8. Consequences: We accept the downsides which come along whith this decision which are:
   1. Complexity: Managing a distributed system with numerous interconnected microservices can be complex.
   2. Data Management Challenges: Microservices often have their own databases or data stores, leading to data duplication and consistency challenges.

Decision record template for MongoDB

Introduction

1. Prologue (Summary)

2. Discussion (Context)

3. Solution (Decision)

4. Consequences (Results)

Specifics

5. Prologue: In the context of the system architecture, we face the issue of large load on servers, so, we decided for MicroServices to achive maintanability and scalability accepting higher complexity.

6. Discussion: A proper SQL would have been very cumbersome to adjust to our data base schema. The usage of the database should also be easy. We also need flexibility in the database.

7. Solution: We decided to use MongoDB to achieve:

    1. Schema flexibility: MongoDB is a NoSQL database, which means it doesn't require a predefined schema. This flexibility allows us to store and manage unstructured or semi-structured data easily, making it ideal for our application.
    2. Scalability: MongoDB is designed to scale out horizontally, meaning we can easily distribute data across multiple servers or clusters to handle increasing loads.
    3. High performance: MongoDB's document-oriented data model and built-in sharding capabilities contribute to its high performance. It supports various types of queries, including complex aggregations, and can efficiently handle read and write operations even at scale.

8. Consequences: We accept the downsides which come along whith this decision which are:

    1. Memory usage: MongoDB can consume significant amounts of memory, especially when working with large datasets or performing complex queries. Proper indexing and schema design can mitigate this to some extent, but it's essential to monitor and manage memory usage, particularly in memory-constrained environments.

Decision record template for flask and Fastapi

Introduction

1. Prologue (Summary)
2. Discussion (Context)
3. Solution (Decision)
4. Consequences (Results)

Specifics

1. Prologue (Summary): For sending API calls to our backend which is rwritten in python, we needed to find a routing framework to handle the incoming requests. We have used FASTAPI for handling most of the calls but have used Flask for the AI models.
2. Discussion (Context): We used FastAPI for most of the endpoints as it is very performant for handling large amounts of data. Moreover, we had experience using FastAPI. Flask was used for calling the AI models as the data transfer

over network between them is not that much. Thus FLask was ideal as it has a very easy to use interface and is easier to maintain and extend.

3. Solution: Our choices for using the appropriate framework at the proper position ensure that the codebase is easier to maintain and extend in the future while still not compromising on performance.
4. Consequences(Results): We found that the api endpoints defined using FastAPI are very performant and reduce latency while api endpoints in Flask are not a bottleneck for our code as majority of the time is taken in generating the response. The code base is also easier to maintain for the AI module

Decision record template for restful

Introduction

1. Prologue (Summary)
2. Discussion (Context)
3. Solution (Decision)
4. Consequences (Results)

Specifics

5. Prologue (Summary): In the context of sending api requests, we needed to select a architecture style between REST and GraphQL. We decided for using RESTful architecture as opposed to GraphQl to achieve simpler and more maintainable codebase. While this allows lesser control over our api requests, we believe it is justified as our requests are not complex.
6. Discussion (Context): Most of the team had little experience with graphQL and were much more comfortable with using REST. Due to the easier ease of implementation as well, it was decided that REST architecture will be much more conducive for smoother and faster development
7. Solution: The decision will help us make API getways for our separate services in a simple and maintainable manner
8. Consequences: We found that the team found no issues while implementing API endpoints in the REST architecture. Our code is also performant and does not suffer from any latencies. The codebase is also much more cleaner than if we had implement GraphQL