CS673 Software Engineering

Team 4 - ResumAI

Software Design Document

Team Member	Role(s)	Signature	Date
Hemant Krishnakumar	Team Lead		10 - 17- 2024
Faizan Ahmad	Design and Implementation Lead		10 - 17- 2024
Tushar	Requirement Lead		10 - 17- 2024
Shubh Gupta	Q/A Lead		10 - 17- 2024
Amruth Reddy	Security Lead		10 - 17- 2024
Jaindra Parvathaneni	Configuration Lead		10 - 17- 2024

Revision history

Version	**Author**	**Date**	**Change**
1	**Faizan Ahmad**	10 - 17- 2024	**New**
1	**Faizan Ahmad**	11 - 07 - 2024	**Added more APIs with descriptions** **Added Class Diagram** **Update Design Patterns**

Introduction

ResumAI is an AI powered resume analysis and matching tool. The problem that we are trying to solve is to allow applicants (job seekers) to analyze their resumes using AI to make them better while also allowing them to compare their resumes to provided Job Descriptions to see how much their resumes fit for the role. The second part is for recruiters where they can upload multiple resumes provided to them along with the job description for which they are hiring so that they are able to shortlist resumes that are more suitable for the role. Lastly, as an additional feature, we also intend to allow this to become an AI powered application tool where Recruiters can create job listings and applicants can apply to those while using our AI capabilities to match with the job.

Our primary design goals are to make the platform **reliable**, **scalable**, **efficient**, **easy-to-use** and **secure**. We intend to use our capabilities to create a software architecture that will meet all those requirements.

Software Architecture

We have designed our software architecture based on **Client-Server** architecture. Our design will include 3 main layers: **Frontend** and **Backend & Database**

Frontend

Our Frontend is a web application built using **ReactJS** framework built using **Typescript.**

- For state management, we are using **React Redux**
- For testing, we are using **JEST** and **React Testing Library**
- For API calls, we are using **Axios (HTTPs)**
- For routing, we will be using **ReactRouter**
- For UI components, we will be using the **Material UI** library
- Developer tools:
    - **VITE** for local development tooling
    - **Prettier** + **Eslint** for formatting and linting

Backend & Database

Our Backend server is built on top of **Fastify** which is a web framework for **Node.JS** using **Typescript**

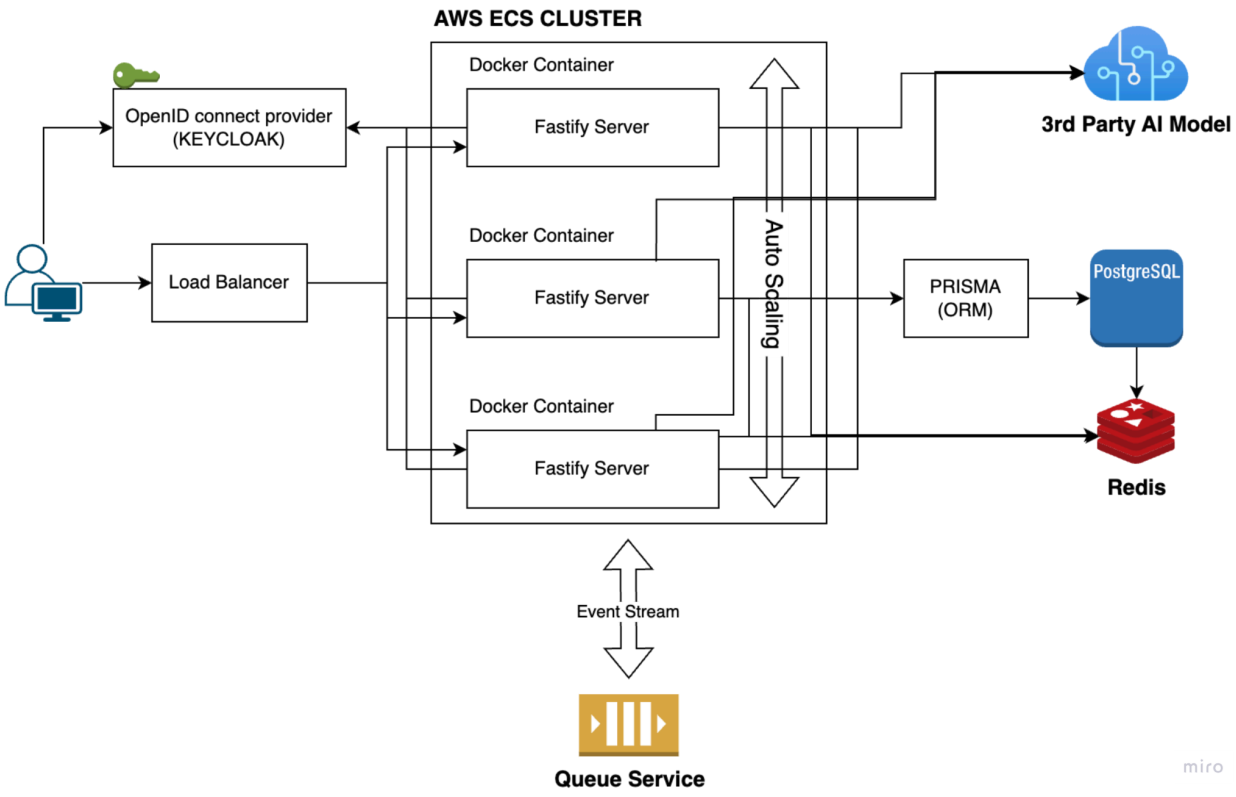
- **Docker** as our containerization tool
- For Database, we are using **PostgreSQL**
- For Identity & Access Management, we are using **Keycloak** and **JWT tokens**
- Our Backend server also incorporate **REDIS** Cache
- We will also be using an Event Streaming service like **SQS/Kafka**
- 3rd Party API calls are handled using **Axios**

- Our APIs are **RESTFul**
- We are also using **Yup** as our schema builder for parsing and validations
- For testing, we are using **Node Tap** library
- For object storage, we are using **Disk Storage** on Fastify using **Multer**

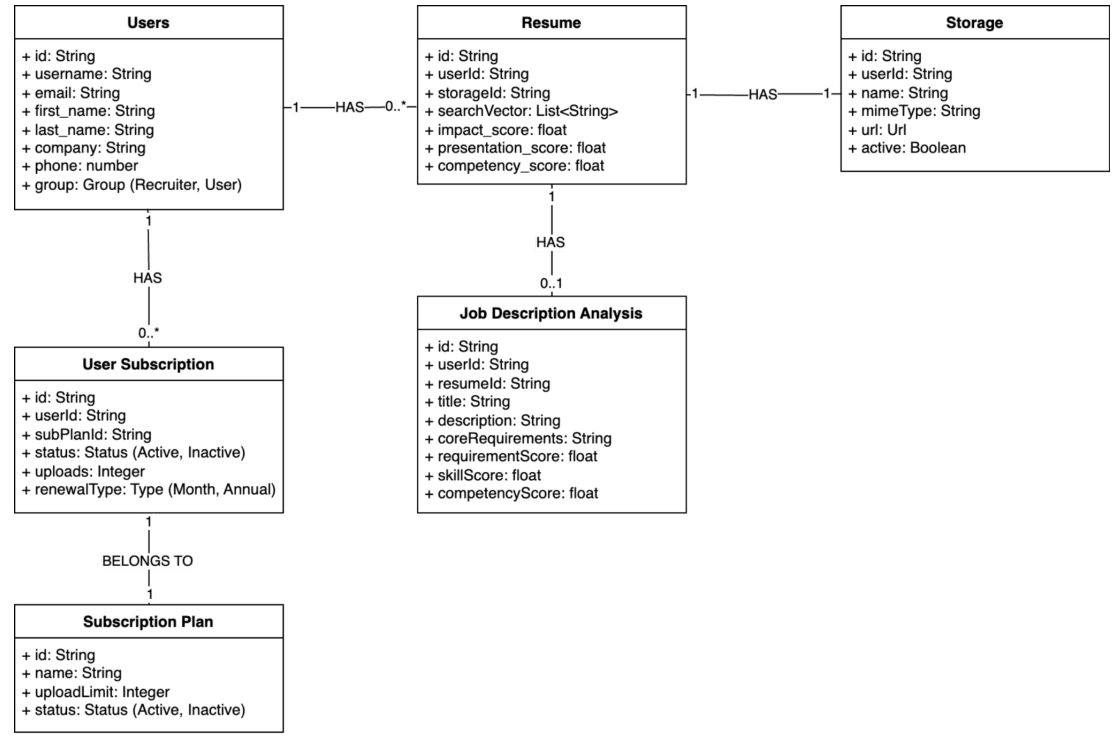
Interaction between Frontend and Backend

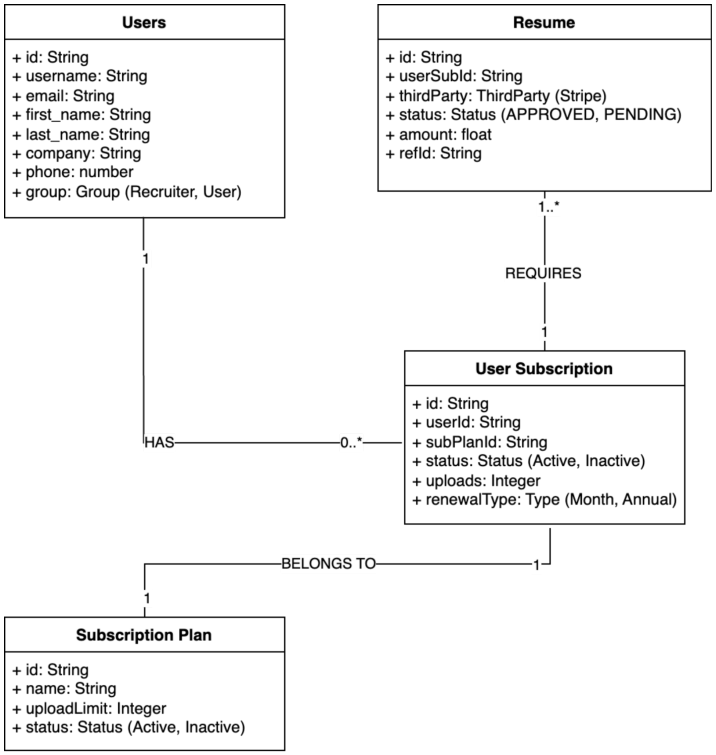
- Frontend interacts with our Authentication service based on Keycloak
- Frontend interacts with our Fastify server APIs for Business logic
- Prisma is our ORM which bridges the gap between APIs and database
- We use Redis for our caching mechanisms
- We use event streams for asynchronous behavior

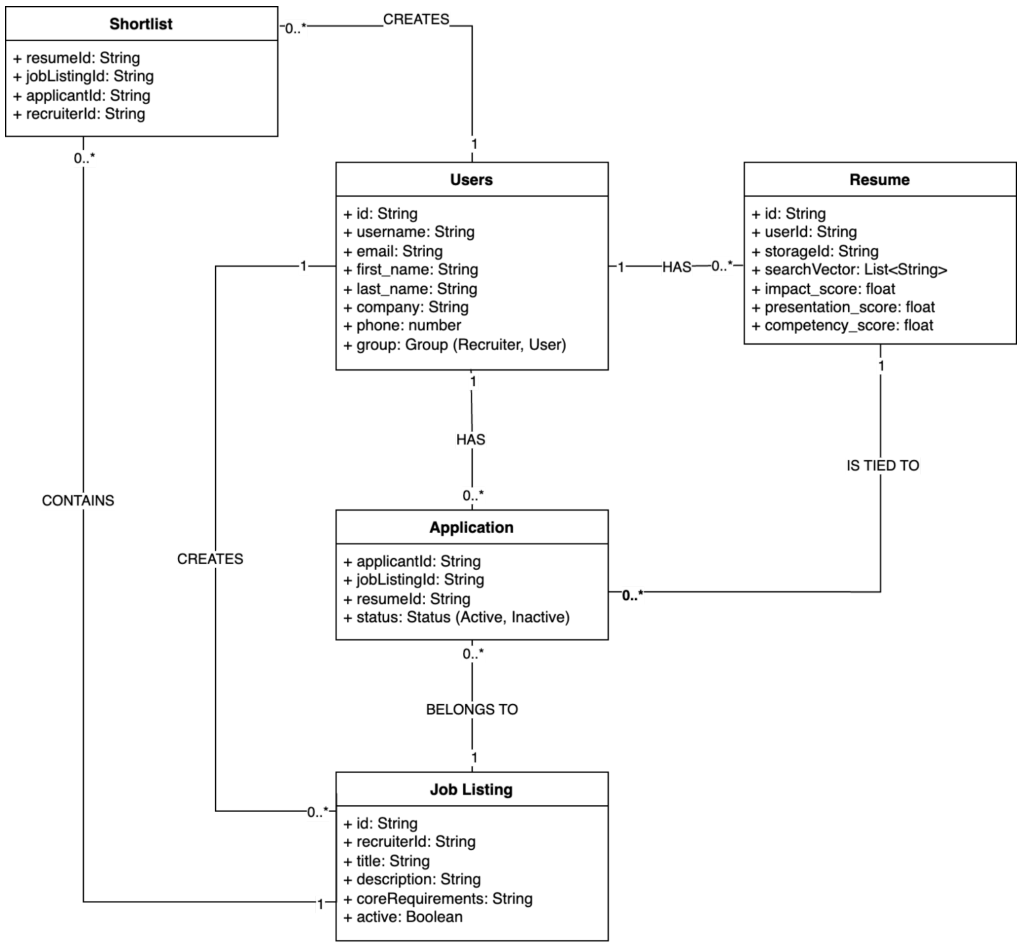
Architectural Diagram



- Class Diagram

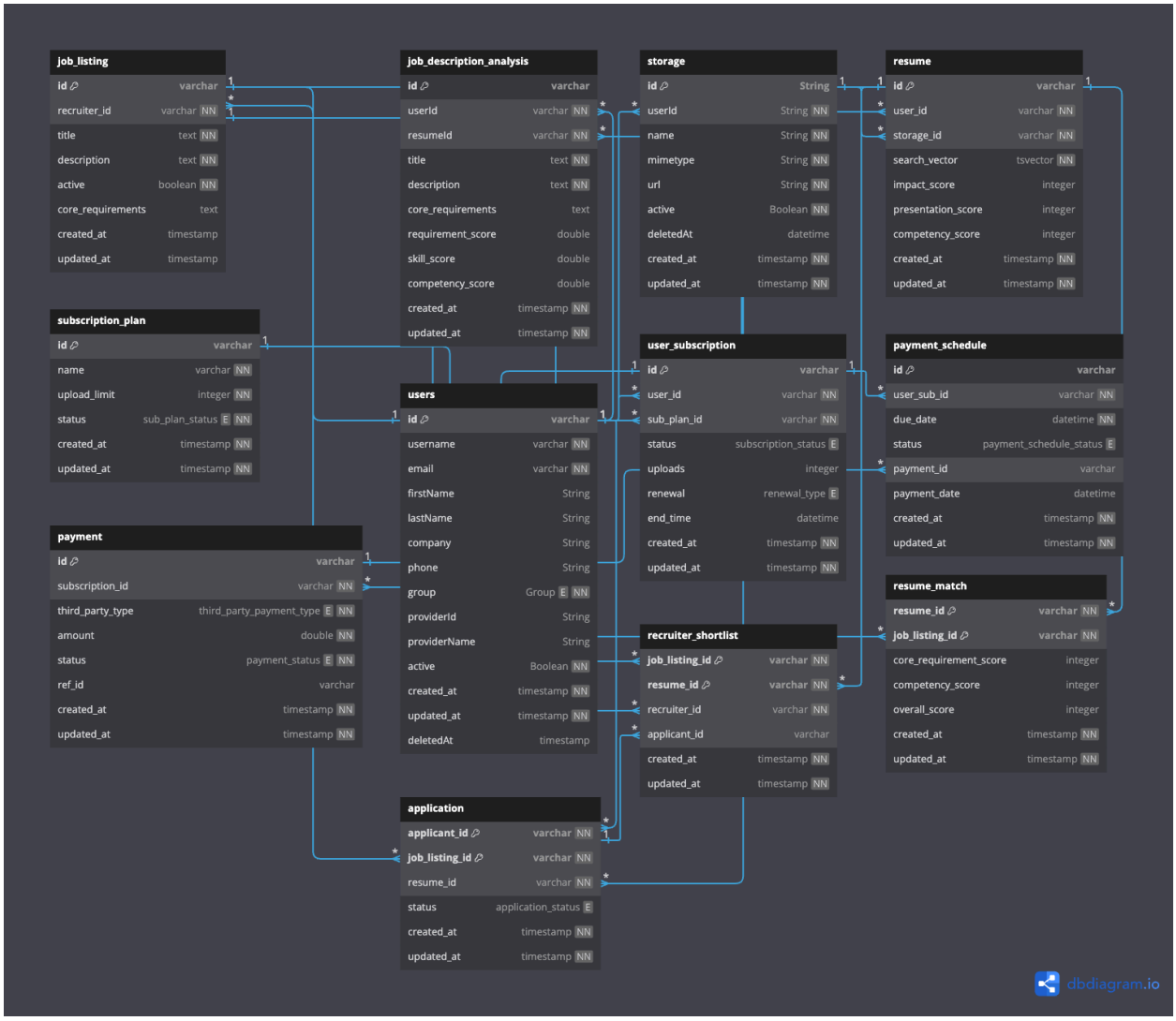






- UI Design (if applicable)
    - Our UI will have initial Authentication pages which will include:
        - Landing Page
        - Login Page
        - Signup Page
        - Forgot Password Page
    - Our UI will also include the following Dashboard pages
        - Analytics page
        - Resume w/ JD upload page
        - Resume w/ JD analysis page
        - Job creation page
        - Job listing page
        - Applications
        - Profile page
        - Subscribe/Unsubscribe pages
        - Payments & Billing History pages

- Database Design (if applicable)



- Security Design

We are building our project with Security in mind. We have incorporated key security features in the following ways:

- We are using Keycloak for Identity Access and Management with authentication expiries to enhance security
- We are also using **JWT Tokens** with an interceptor for our APIs to extract scopes from our JWT Token and use those scopes for API access
- We plan on deploying our applications on a Private Network (AWS VPN) to secure our Backend and use a Proxy for enhanced security
- We have incorporated Prisma which provides intermediate level of security against malicious attacks.

- # Business Logic and/or Key Algorithms
    - **Upload Resume Bulk**: This logic allows a recruiter to upload bulk resumes against a certain job. This triggers an async analysis logic to use AI to analyze resumes against the job description and provide insights. This also involves checking whether the user can exceeded their upload limit based on their tier
    - **Upload Resume (User)**: This logic allows an applicant to upload a resume against a certain job. This triggers an async analysis logic to use AI to analyze the resume against the job description and provide insights. This also involves checking whether the user can exceeded their upload limit based on their tier
    - **Account Creation**: This allows the user to create an account
    - **Subscribe to a plan**: This subscribe a user to a certain plan
    - **Make a payment**: This allows the user to make a payment for their plan
    - **Create job listing**: This allows a recruiter to post a job
    - **Apply to job listings**: This allows the user to apply to a certain job

- # Design Patterns
  We have designed our software architecture based on **Client-Server** architecture
  ## Frontend Design Patterns
    - Container and Presentation Pattern
        - We have created about 20 or so reusable components with containers to provide flexibility and reusability for those components
    - State Management with Reducers
        - We are using Slices and consequently Reducers to manage our state for each of our pages
    - Middleware
        - Redux also works as our middleware where we use AsyncThunk to make API calls and validate our responses. This acts as a middleware to make sure correct data is being shown.
    - Component Compositions with Hooks
        - We use react hooks to build our functional components. For some new functionality that might be reusable and handles state, we create a custom hook and use that
    - Data Management with Providers
        - We are using different types of Providers to handle state changes. For example, we are using Redux Provider to handle, view and update redux state within our components. Our main component is wrapped with Redux Provider. We are also using ThemeProvider from Material UI to manage our theme across our entire dashboard wrapping Dashboard Layout
    - Promise pattern
        - We use this pattern for making asynchronous calls like Data fetching and API calls

Backend Design Patterns

- Middleware pattern
    - We have implemented this pattern to handle our API interceptor which verifies the authentication of the API call and validates incoming body
- Promise pattern
    - We are using this pattern to make calls to our database asynchronously and also return promises in our API call responses.
- Singleton pattern (db conn)
    - We are using this pattern to share our db connections across the entire server. A single instance handles our database connections
- Plugin Pattern
    - We use this pattern to enable different functionalities in our backend server such as authentication plugin, database, redux etc.
- Decorator Pattern
    - We use fastify's decorate library to decorate our API calls and interceptors using preHandler decorator.

Rest APIs

- **Applications**

Contracts

```
type applicationPostBody = {
   jobListingId: string;
   resumeId: string;
 };

enum ApplicationStatus {
   APPLIED = 'APPLIED',
   CANCELLED = 'CANCELLED',
   INTERVIEWING = 'INTERVIEWING',
   REJECTED = 'REJECTED',
   OFFER = 'OFFER',
 }

 type applicationPutBody = {
   status: ApplicationStatus;
 };
```

- **GET Application (/applications/{id})**
    - Returns the application with the id
- **POST Application (/applications)**
    - Creates a new application with a validated request body
- **PUT Application (/applications/{id})**
    - Updates an existing application with id
- **DELETE Application (/applications)**
    - SOFT deletes an existing application with id

- **Jobs**
    - **GET Jobs (/job/{id})**
        - Returns job with the id for recruiter along with applications
    - **GET Jobs (/job/public?searchTerm={searchTerm})**
        - Returns jobs for applicant with searchTerm
    - **POST Jobs (/job)**
        - Creates a new job listing
    - **PUT Jobs (/job/{id})**
        - Updates a specific Job Listing

Contracts:

```
type JobListingsPostBody = {
   title: string;
   description: string;
   coreRequirements: string;
   active?: boolean;
 };
type JobListingsPutBody = {
   title?: string;
   description?: string;
   coreRequirements?: string;
   active?: boolean;
 };
```

- **Resume**

Contract

```
type ResumePostBody = {
   storageId: string;
 };
```

- **GET Resume (/resume/{id})**
    - Fetches a specific resume with ID
- **POST Resume (/resume) (For Applicants)**
    - Adds a resume to the database tables
    - This requires a resume upload functionality first
- **POST Resume (/resume/bulk) (For Recruiters)**
    - This adds multiple resumes to the database tables
    - This requires an already uploaded resume first
- **GET Resume (/resume/{id}/view-analysis)**
    - This shows analysis for a specific resume
- **DELETE Resume (/resume/{id})**
    - This SOFT deletes an existing resume

- **Me (User)**

Contract

```
type MePost = {
   firstName?: string;
   lastName?: string;
   phone?: string;
 };

 const MePostSchema = {
   firstName: String,
   lastName: String,
   phone: String,
 };
```

  - **POST User (/me)**
    - Creates a new User (both applicant and recruiter)
  - **GET User (/me/{id})**
    - This returns user details of a specific user
  - **PUT User (/me/{id})**
    - This updates and existing user

- **Shortlist**

Contract

```
type shortListPostBody = {
   applicationId: string;
   jobListingId: string;
 };
```

  - **GET Shortlist (/shortlists/{id})**
    - This returns a specific shortlist that recruiter has created
  - **POST Shortlist (/shortlists)**
    - This creates a new shortlist for a recruiter with data from body
  - **DELETE Shortlist (/shortlists/{id})**
    - This soft deletes an existing shortlist

- **Storage**
  - **POST Storage (/storage)**
    - This uploads a new file in the backend and returns its id
  - **GET Storage (/storage/{filename})**
    - This returns a specific file with filename

- Any Additional Topics you would like to include.
  - We are planning to deploy our project to AWS for cloud deployment
  - We are planning to deploy our React App over a CDN for better performance
  - For cloud deployment, we are planning to put our Backend components like Database, Object Storage, Event Streams and Redis cache under a Private Network so that only our Fastify servers can interact with database enhancing security.
  - We are also exploring the possibility of including a Load Balancer in the future including a Rate Limiter (Proxy) to prevent malicious attacks such as DoS

- References

https://react.dev/
https://fastify.dev/
https://www.postgresql.org/
https://v3.vitejs.dev/guide/
https://jestjs.io/
https://redux.js.org/