Software Design Document: Recruitment & Interview Manager

Table of Contents

---

1. Introduction

This Software Design Document (SDD) provides a detailed description of the design for the Recruitment & Interview Manager system. This document outlines the system\'s architecture, components, interfaces, and data, serving as a blueprint for the development team. It is intended for use by project managers, software developers, testers, and other stakeholders involved in the project lifecycle.

1.1. Purpose

The primary purpose of this document is to provide a comprehensive architectural and design specification for the Recruitment & Interview Manager system. It aims to clearly define the system\'s structure and behavior, ensuring that all stakeholders have a common understanding of the design. This SDD will guide the implementation, testing, and future maintenance of the system. It details how the system will meet the functional and non-functional requirements identified during the analysis phase, including those specified by the user and derived from the project\'s DeepWiki documentation.

1.2. Scope

The scope of the Recruitment & Interview Manager system, as detailed in this document, encompasses a comprehensive solution for managing the entire recruitment lifecycle. This includes functionalities for Human Resources (HR) personnel to post and manage job openings, and for Applicants to search for jobs and submit their applications. The system will also facilitate the scheduling of interview dates, allow for the updating and tracking of application statuses, and provide HR with a consolidated view of candidates for each job opening. The system is designed as a microservices-based architecture, featuring distinct services for job management, applicant tracking, interview scheduling, service discovery (Eureka Server), and an API Gateway to manage client requests. The design covers the core services, their interactions, the database structure, and high-level considerations for user interface, security, and deployment. This document will not cover detailed project management plans, specific hardware procurement, or low-level implementation details beyond what is necessary to understand the design.

1.3. Definitions, Acronyms, and Abbreviations

- **SDD:** Software Design Document
- **HR:** Human Resources
- **UI:** User Interface
- **API:** Application Programming Interface
- **ERD:** Entity-Relationship Diagram
- **JDBC:** Java Database Connectivity
- **MySQL:** A popular open-source relational database management system.
- **Microservice:** A small, autonomous service that works with other services to provide the functionality of a larger application.
- **Gateway Service:** A microservice that acts as a single entry point for all client requests to the system\'s backend services.
- **Job Service:** A microservice responsible for managing job postings.
- **Applicant Service:** A microservice responsible for managing applicant data and applications.
- **Interview Service:** A microservice responsible for managing interview scheduling and tracking.
- **Eureka Server:** A service discovery tool used in microservices architectures.

1.4. References

- User-provided project overview and requirements.

- Recruitment & Interview Manager Project DeepWiki: https://deepwiki.com/OmarAlhaj10/SwE2project/1-overview (and its sub-pages for System Architecture, Database Design, Core Services, etc.)
- `project_requirements_initial.md` : Internal document capturing initial user requirements.
- `project_requirements_deepwiki_overview.md` : Internal document summarizing DeepWiki overview.
- `project_requirements_deepwiki_architecture.md` : Internal document summarizing DeepWiki system architecture.
- `project_requirements_deepwiki_database.md` : Internal document summarizing DeepWiki database design.
- `project_requirements_deepwiki_core_services.md` : Internal document summarizing DeepWiki core services.
- `sdd_outline.md` : Internal document outlining the structure of this SDD.

1.5. Document Overview

This SDD is organized into several major sections to provide a clear and comprehensive understanding of the Recruitment & Interview Manager system design:

- **Section 1 (Introduction):** Provides the purpose, scope, definitions, references, and an overview of this document.
- **Section 2 (System Overview):** Describes the project, its goals, target users, system context, and major features.
- **Section 3 (System Architecture):** Details the chosen architectural style (microservices), presents a high-level architectural diagram, describes the key components (Gateway, Job, Applicant, Interview, Eureka services), outlines communication mechanisms, and lists the technology stack.
- **Section 4 (Data Design):** Covers the database overview, the data model including ERDs and table definitions, and a data dictionary.
- **Section 5 (Module/Component Detailed Design):** Provides a detailed design for each microservice, including its responsibilities, APIs/endpoints, interactions, and data managed.
- **Section 6 (Security Design):** Outlines authentication, authorization, and data security measures.
- **Section 7 (Deployment Considerations):** Presents a conceptual deployment diagram and environment requirements.
- **Section 8 (Non-Functional Requirements):** Addresses aspects like scalability, resilience, performance, reliability, and maintainability.

- **Section 9 (Glossary):** Provides definitions for terms used throughout the document.
- **Section 10 (References):** Lists all documents, web pages, and other resources referenced during SDD creation.

This structure is designed to guide the reader logically through the various aspects of the system design, from a high-level overview to detailed component specifications.

---

2. System Overview

This section provides a high-level overview of the Recruitment & Interview Manager system. It describes the project, its primary goals and objectives, identifies the target users, outlines the system\'s context within a typical organizational environment, and lists its major functional capabilities. The information presented here is derived from the initial project description provided by the user and the detailed documentation available on the project\'s DeepWiki page.

2.1. Project Description

The Recruitment & Interview Manager is a software system designed to streamline and manage the entire lifecycle of job applications and interview processes within an organization. As indicated in the initial user request and further detailed in the DeepWiki documentation, the system aims to provide a centralized platform for Human Resources (HR) personnel to effectively manage job postings and track candidate progress, and for Applicants to seamlessly apply for open positions and monitor their application status. The system is envisioned as a modern, microservices-based solution, which allows for scalability, resilience, and independent deployment of its various functional components. This architectural choice, highlighted in the DeepWiki\'s System Architecture section, underpins the system\'s capability to handle complex recruitment workflows efficiently.

2.2. System Goals and Objectives

The primary goal of the Recruitment & Interview Manager system is to enhance the efficiency and effectiveness of the recruitment process for both HR staff and job applicants. To achieve this, the system has the following key objectives:

- **Automate Core Recruitment Tasks:** The system aims to automate critical tasks such as posting job openings, receiving and organizing applications, scheduling

interviews, and updating application statuses. This automation is intended to reduce manual effort and minimize errors, as suggested by the core features listed by the user.
- **Improve Applicant Experience:** By providing a clear and accessible platform for job searching, application submission, and status tracking, the system seeks to offer a positive and transparent experience for applicants.
- **Enhance HR Productivity:** The system will provide HR personnel with tools to efficiently manage multiple job openings, filter and review candidate applications, coordinate interview schedules, and maintain a comprehensive record of the recruitment process for each position. The DeepWiki\'s description of the Job Service, Applicant Service, and Interview Service supports this objective.
- **Centralize Recruitment Data:** By consolidating all recruitment-related information into a single system with a centralized database (as noted in the DeepWiki\'s Database Design section), the system aims to provide a single source of truth, facilitating better decision-making and reporting.
- **Ensure Scalability and Maintainability:** The adoption of a microservices architecture is a strategic objective to ensure that the system can scale to meet growing organizational needs and can be easily maintained and updated over time. This is a recurring theme in the DeepWiki documentation.

2.3. Target Users

The Recruitment & Interview Manager system is designed to serve two primary groups of users, as explicitly stated in the initial project overview:

- **Applicant:** Individuals seeking employment opportunities who will use the system to search for available job openings, submit their applications along with necessary documents (e.g., resumes), and track the status of their applications throughout the recruitment process. Their interaction will primarily be with the applicant-facing interfaces of the system, likely facilitated through the Applicant Service.
- **HR (Human Resources):** Personnel within the organization responsible for managing the recruitment process. HR users will utilize the system to create and publish job openings, review submitted applications, manage candidate pools for different jobs, schedule and coordinate interview rounds, update application statuses (e.g., screened, shortlisted, interviewed, offered, rejected), and communicate with applicants. Their functionalities are supported by the Job Service, Applicant Service, and Interview Service, orchestrated via an HR-facing interface.

2.4. System Context

The Recruitment & Interview Manager system will operate as a key component within an organization\'s broader human resources management and talent acquisition ecosystem. It is expected to interact with potential job seekers externally and with HR department personnel internally. While the current scope focuses on the core recruitment and interview management functionalities, in a larger enterprise context, it might eventually need to integrate with other systems such as HR Information Systems (HRIS) for new hire onboarding, corporate websites for job postings, or external job boards. The microservices architecture, particularly the API Gateway, provides a flexible foundation for such future integrations. The system will rely on standard web technologies for user interaction and will be deployed in a server environment capable of hosting its various microservices and the central MySQL database, as detailed in the DeepWiki.

2.5. Major System Features

The Recruitment & Interview Manager system will provide a range of features to support the end-to-end recruitment process. The core features, as identified in the user\'s initial request and corroborated by the DeepWiki\'s service descriptions, are detailed below:

2.5.1. HR Posts Job Openings

HR personnel will have the capability to create new job openings within the system. This includes specifying details such as the job title, description, responsibilities, required qualifications, location, and application deadline. Once created, these job openings can be published, making them visible to potential applicants. This functionality is primarily managed by the Job Service.

2.5.2. Applicants Submit Applications

Applicants will be able to browse or search for published job openings. Upon finding a suitable position, they can submit their application through the system. This process typically involves filling out an application form, uploading relevant documents such as a resume and cover letter, and providing personal contact information. The Applicant Service is responsible for handling these submissions.

2.5.3. Schedule Interview Dates

Once applications are reviewed and candidates are shortlisted, HR personnel can use the system to schedule interview dates and times. This feature will involve coordinating

with interviewers (who may be other employees within the organization, though their direct system interaction is not explicitly detailed in the current scope beyond HR managing schedules) and notifying applicants of their interview appointments. The Interview Service will manage the intricacies of interview scheduling, including different rounds and types of interviews.

2.5.4. Change Application Status

Throughout the recruitment process, the status of each application will change (e.g., from \'Submitted\' to \'Under Review\', \'Shortlisted for Interview\', \'Interview Scheduled\', \'Offer Extended\', \'Hired\', or \'Rejected\'). HR users will be able to update the application status at each stage. Applicants, in turn, should be able to view the current status of their application. Both the Applicant Service and potentially the Interview Service will be involved in managing and reflecting these status changes.

2.5.5. View List of Candidates per Job

HR personnel require the ability to view a comprehensive list of all candidates who have applied for a specific job opening. This view should allow them to easily access applicant details, submitted documents, and the current status of each application, facilitating efficient candidate management and comparison. This functionality draws data primarily from the Applicant Service and Job Service.

---

3. System Architecture

This section delves into the architectural design of the Recruitment & Interview Manager system. It begins by defining the chosen architectural style, followed by a high-level visual representation of the system. Each major component within the architecture is then described, along with the mechanisms for communication between these components. Finally, the underlying technology stack that supports this architecture is outlined. The design choices presented here are heavily influenced by the information provided in the project\"s DeepWiki documentation, particularly the "System Architecture" and "Core Services" pages.

3.1. Architectural Style (Microservices)

The Recruitment & Interview Manager system adopts a **microservices architectural style**. This choice, as highlighted in the DeepWiki documentation, is driven by the desire for scalability, resilience, and the ability to independently develop, deploy, and manage

different parts of the system. In a microservices architecture, the application is structured as a collection of small, autonomous services, each focused on a specific business capability. These services are developed independently, can be deployed in isolation, and communicate with each other over well-defined APIs, typically using lightweight protocols such as HTTP/REST. This approach contrasts with a monolithic architecture where all functionalities are built into a single, large application. The benefits sought through this style include improved modularity, easier maintenance, technology diversity (though the current project seems to lean towards a consistent stack), and better fault isolation, meaning that the failure of one service is less likely to impact the entire system.

3.2. Architectural Diagram (High-Level)

A high-level architectural diagram illustrates the overall structure of the Recruitment & Interview Manager system, showing its key microservices and their primary interactions. (A visual diagram would be embedded here in a formal SDD. The following is a textual description based on the DeepWiki information.)

The diagram would typically depict the following:

- **Client Applications (User Interfaces):** These represent the front-end interfaces through which Applicants and HR personnel interact with the system. These are not microservices themselves but are the entry points for user requests.
- **API Gateway (GatewayService):** Positioned as the single entry point for all incoming client requests. It routes requests to the appropriate backend microservices.
- **Core Microservices:**
    - **Job Service:** Responsible for all functionalities related to job postings.
    - **Applicant Service:** Manages applicant profiles, applications, and related data.
    - **Interview Service:** Handles the scheduling and management of interviews.
- **Supporting Services:**
    - **Eureka Server (Service Discovery):** Enables dynamic registration and discovery of microservice instances, allowing them to locate and communicate with each other without hardcoded addresses.
- **Databases:** While the DeepWiki mentions a centralized MySQL database (`recruitment_db`) accessed by the Job, Applicant, and Interview services, a pure microservices approach often advocates for each service owning its own database to ensure loose coupling. The SDD will reflect the centralized model as per the DeepWiki but might note this as a point of consideration for future evolution if

stricter service independence is desired. The diagram would show these services interacting with this central database.
- **Communication Paths:** Arrows indicating the flow of requests and data between client applications, the API Gateway, the various microservices, the Eureka Server, and the database.

This high-level view provides an immediate understanding of the system\"s decomposition into services and their interconnectedness.

3.3. Component Description

The Recruitment & Interview Manager system is composed of several key microservices, each with distinct responsibilities, as detailed in the DeepWiki\"s "Core Services" and "System Architecture" sections.

3.3.1. Gateway Service

The **Gateway Service** acts as the primary entry point for all client requests originating from user interfaces (Applicant portal, HR portal). Its main responsibility is to route these incoming requests to the appropriate downstream microservice (Job Service, Applicant Service, or Interview Service). Beyond basic routing, the API Gateway can also handle cross-cutting concerns such as request authentication and authorization, SSL termination, rate limiting, request/response transformation, and load balancing across instances of the backend services. This centralizes these common functionalities, simplifying the individual microservices. The DeepWiki identifies this as a key component in managing client interactions.

3.3.2. Job Service

The **Job Service** is dedicated to managing all aspects of job postings. Its core responsibilities include allowing HR users to create new job listings, update existing ones (e.g., change descriptions, extend deadlines), delete postings, and search or filter available jobs. It manages data entities related to jobs, such as job titles, descriptions, requirements, locations, and status (e.g., open, closed). This service exposes APIs for these operations, which are consumed by the API Gateway on behalf of HR users and potentially by applicants (for viewing/searching jobs).

3.3.3. Applicant Service

The **Applicant Service** focuses on managing applicant information and their applications. Its key functions include enabling applicants to create and manage their profiles, submit applications for specific job postings (including uploading documents

like resumes), and track the status of their submitted applications. For HR users, this service provides functionalities to view applicant details, access submitted documents, and manage the pool of candidates for each job. It handles data related to applicant profiles, application forms, and application statuses. The APIs exposed by this service are crucial for both applicant and HR user workflows.

3.3.4. Interview Service

The **Interview Service** is responsible for the coordination and management of the interview process. This includes functionalities for HR users to schedule interviews for shortlisted candidates, define interview rounds (e.g., phone screen, technical interview, HR interview), assign interviewers (though direct interviewer interaction is not detailed as a separate user role in the initial scope), track interview progress, and record interview feedback or outcomes. It manages data pertaining to interview schedules, rounds, feedback, and interview statuses. Its APIs facilitate the smooth progression of candidates through the interview stages.

3.3.5. Eureka Server (Service Discovery)

The **Eureka Server** plays a critical role in the microservices ecosystem by providing service discovery capabilities. Each microservice instance (Job, Applicant, Interview, and potentially the Gateway Service itself) registers with the Eureka Server upon startup, making its network location (IP address and port) known. When one service needs to communicate with another, it queries the Eureka Server to find the available instances of the target service. This allows for dynamic scaling and resilience, as services can be added or removed without requiring manual reconfiguration of clients. The DeepWiki explicitly mentions Eureka Server for service discovery and registration.

3.4. Communication Mechanisms

Communication between the components of the Recruitment & Interview Manager system primarily occurs through synchronous HTTP/REST APIs. Client applications (web browsers used by Applicants and HR) will communicate with the **API Gateway** using HTTPS requests. The API Gateway, in turn, will communicate with the backend microservices (Job Service, Applicant Service, Interview Service) also typically via synchronous RESTful API calls over HTTP/HTTPS. These APIs will use standard HTTP methods (GET, POST, PUT, DELETE) and exchange data in a common format, likely JSON.

Internal communication between the microservices themselves (e.g., if one service needs to query another directly, though this is often mediated by the Gateway or through event-driven patterns in more complex scenarios) would also use REST APIs.

Furthermore, all operational microservices (Job, Applicant, Interview, Gateway) will communicate with the **Eureka Server** for registration and discovery. This involves sending heartbeat signals to Eureka to indicate they are alive and querying Eureka to find other services.

While the current design focuses on synchronous communication, future enhancements could introduce asynchronous communication patterns (e.g., using message queues like RabbitMQ or Kafka) for tasks that can be decoupled, such as sending notifications or processing background tasks, to improve system responsiveness and resilience. However, based on the DeepWiki, the primary mechanism appears to be REST-based synchronous calls.

3.5. Technology Stack

While the DeepWiki provides high-level architectural details, it also gives hints about the technology stack, particularly through references in configuration files or common practices associated with the mentioned components. A likely technology stack, which would be confirmed and detailed during implementation, includes:

- **Backend Development:** Java, likely with the Spring Boot framework, is a common choice for building microservices, especially when using components like Spring Cloud Netflix Eureka for service discovery. The DeepWiki\"s mention of `.idea/compiler.xml` and Maven configuration points towards a Java-based environment.
- **Database:** MySQL is explicitly mentioned in the DeepWiki\"s "Database Design" section as the centralized relational database (`recruitment_db`), accessed via JDBC.
- **API Gateway:** If using Spring Boot, Spring Cloud Gateway is a common option. Other alternatives include Netflix Zuul (though less current) or dedicated gateway solutions.
- **Service Discovery:** Netflix Eureka Server, as explicitly named.
- **Frontend Development:** (Not detailed in DeepWiki\"s backend focus) Standard web technologies such as HTML, CSS, and JavaScript, possibly with a modern framework like React, Angular, or Vue.js, would be used for the client applications (Applicant and HR portals).
- **Build & Dependency Management:** Maven is suggested by the DeepWiki\"s project configuration details.
- **Deployment Environment:** The services would be containerized (e.g., using Docker) and orchestrated (e.g., using Kubernetes) in a typical microservices deployment, although specific deployment details are not fully elaborated in the provided DeepWiki content beyond general architectural patterns.

This technology stack provides a robust and widely adopted foundation for building scalable and maintainable microservices-based applications.

---

4. Data Design

This section elaborates on the data design for the Recruitment & Interview Manager system. It provides an overview of the database strategy, details the proposed data model including an Entity-Relationship Diagram (ERD) concept and table definitions, and outlines the need for a comprehensive data dictionary. The information presented is primarily derived from the "Database Design" section of the project\"s DeepWiki documentation and logical inferences based on the system\"s functional requirements.

4.1. Database Overview

The Recruitment & Interview Manager system will utilize a **centralized relational database** to store and manage all persistent data related to job postings, applicant information, applications, and interview processes. As explicitly stated in the DeepWiki\"s "Database Design" section, the chosen database management system is **MySQL**, and the primary database will be named `recruitment_db`. This database will serve as the single source of truth for the core microservices, namely the Job Service, Applicant Service, and Interview Service, which will interact with it to perform data storage and retrieval operations. The connection to the database will be established using JDBC (Java Database Connectivity), with the connection URL specified as `jdbc:mysql://localhost:3306/recruitment_db` for local development environments, according to the DeepWiki.

While a centralized database is specified, it\"s worth noting that in a pure microservices paradigm, each service often owns its private data store to maximize autonomy and loose coupling. The current design, however, follows the DeepWiki\"s specification of a shared, centralized database. This approach simplifies data consistency management across services but requires careful schema design and coordination to avoid tight coupling between services at the data layer.

4.2. Data Model / Schema

The data model defines the structure of the data to be stored in the `recruitment_db`. This includes the entities (tables), their attributes (columns), and the relationships

between them. A detailed schema is crucial for ensuring data integrity and supporting the system\"s functionalities.

4.2.1. Entity-Relationship Diagram (ERD)

(A visual ERD would be embedded here in a formal SDD. The following describes the key entities and their likely relationships based on the project scope and common recruitment system patterns. The DeepWiki page did not provide a visual ERD, but it\"s a standard component of data design.)

The ERD would visually represent the following key entities and their relationships:

- **Users:** Stores information about system users, distinguishing between HR personnel and Applicants. This entity is fundamental for authentication and authorization.
- **Jobs:** Contains details of job openings posted by HR users.
- **Applicants:** Stores profile information for individuals who apply for jobs. This might be distinct from the `Users` table if applicants can browse jobs without creating a full user account initially, or it could be integrated if all applicants are considered users.
- **Applications:** Represents an instance of an applicant applying for a specific job. This is a central entity linking Jobs and Applicants.
- **ApplicationDocuments:** Could be a separate table to store paths or references to uploaded documents (resumes, cover letters) associated with an application, allowing for multiple documents per application.
- **Interviews:** Manages information about scheduled interviews for an application, including rounds, dates, and feedback.
- **InterviewRounds:** (Potentially) A table to define standard interview round types (e.g., Phone Screen, Technical, HR).

Relationships would include: * One HR User (from `Users`) can post many Jobs. * One Job can have many Applications. * One Applicant (from `Users` or `Applicants` table) can submit many Applications. * One Application belongs to one Job and one Applicant. * One Application can have many ApplicationDocuments. * One Application can have multiple Interviews (representing different rounds or attempts). * One Interview is associated with one Application.

4.2.2. Table Definitions

Based on the inferred ERD and system functionalities, the following table definitions are proposed. These would be refined during detailed design and implementation. Data types are illustrative and would be specific to MySQL.

- **Users Table (`users`)**
    - `user_id` (INT, Primary Key, Auto Increment)
    - `username` (VARCHAR(255), Unique, Not Null)
    - `password_hash` (VARCHAR(255), Not Null)
    - `email` (VARCHAR(255), Unique, Not Null)
    - `first_name` (VARCHAR(100))
    - `last_name` (VARCHAR(100))
    - `role` (ENUM(\'HR\', \'Applicant\'), Not Null)
    - `created_at` (TIMESTAMP, Default CURRENT_TIMESTAMP)
    - `updated_at` (TIMESTAMP, Default CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP)
- **Jobs Table (`jobs`)**
    - `job_id` (INT, Primary Key, Auto Increment)
    - `hr_user_id` (INT, Foreign Key references `users.user_id`)
    - `title` (VARCHAR(255), Not Null)
    - `description` (TEXT, Not Null)
    - `requirements` (TEXT)
    - `location` (VARCHAR(255))
    - `posted_date` (TIMESTAMP, Default CURRENT_TIMESTAMP)
    - `expiry_date` (TIMESTAMP, Nullable)
    - `status` (ENUM(\'Open\', \'Closed\', \'Draft\'), Not Null, Default \'Draft\')
    - `created_at` (TIMESTAMP, Default CURRENT_TIMESTAMP)
    - `updated_at` (TIMESTAMP, Default CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP)
- **Applications Table (`applications`)**
    - `application_id` (INT, Primary Key, Auto Increment)
    - `job_id` (INT, Foreign Key references `jobs.job_id` , Not Null)
    - `applicant_user_id` (INT, Foreign Key references `users.user_id` , Not Null)
    - `submission_date` (TIMESTAMP, Default CURRENT_TIMESTAMP)

    - `status` (ENUM(\'Submitted\', \'Under Review\', \'Shortlisted\', \'Interview Scheduled\', \'Interviewed\', \'Offer Extended\', \'Hired\', \'Rejected\', \'Withdrawn\'), Not Null, Default \'Submitted\')
    - `cover_letter_text` (TEXT, Nullable)
    - `created_at` (TIMESTAMP, Default CURRENT_TIMESTAMP)
    - `updated_at` (TIMESTAMP, Default CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP)

- **ApplicationDocuments Table (`application_documents`)**
    - `document_id` (INT, Primary Key, Auto Increment)
    - `application_id` (INT, Foreign Key references `applications.application_id`, Not Null)
    - `document_name` (VARCHAR(255), Not Null)
    - `document_type` (ENUM(\'Resume\', \'Cover Letter\', \'Portfolio\', \'Other\'), Not Null)
    - `file_path_or_reference` (VARCHAR(1024), Not Null)
    - `uploaded_at` (TIMESTAMP, Default CURRENT_TIMESTAMP)

- **Interviews Table (`interviews`)**
    - `interview_id` (INT, Primary Key, Auto Increment)
    - `application_id` (INT, Foreign Key references `applications.application_id`, Not Null)
    - `interviewer_name` (VARCHAR(255)) / Could be a foreign key to a Users table if interviewers are also users /
    - `interview_round_name` (VARCHAR(100)) / Or FK to an InterviewRounds table /
    - `scheduled_datetime` (TIMESTAMP, Not Null)
    - `location_or_link` (VARCHAR(255)) / For physical location or video call link /
    - `status` (ENUM(\'Scheduled\', \'Completed\', \'Cancelled\', \'Rescheduled\'), Not Null, Default \'Scheduled\')
    - `feedback_notes` (TEXT, Nullable)
    - `outcome` (ENUM(\'Progress\', \'Hold\', \'Reject\'), Nullable)
    - `created_at` (TIMESTAMP, Default CURRENT_TIMESTAMP)
    - `updated_at` (TIMESTAMP, Default CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP)

4.3. Data Dictionary

A comprehensive data dictionary will be created to accompany the database schema. This dictionary will provide detailed descriptions for each table and each column within those tables. For each column, the data dictionary will specify:

- **Column Name:** The logical name of the column.
- **Data Type:** The specific MySQL data type (e.g., VARCHAR(255), INT, TEXT, TIMESTAMP, ENUM).
- **Size/Length:** If applicable (e.g., for VARCHAR).
- **Constraints:** Any constraints applied (e.g., Primary Key, Foreign Key, Not Null, Unique, Default value, Check constraints).
- **Description:** A clear, human-readable explanation of the data element\"s purpose and meaning.
- **Example Values:** Illustrative examples of the data that might be stored in the column.
- **Notes:** Any additional relevant information or considerations.

This data dictionary will serve as a crucial reference for developers, database administrators, and anyone else needing to understand the structure and semantics of the data stored within the Recruitment & Interview Manager system. It ensures consistency in data interpretation and usage across the development team and throughout the system\"s lifecycle.

---

5. Module/Component Detailed Design

This section provides a more granular look at each of the core microservices and supporting components that constitute the Recruitment & Interview Manager system. For each module or component, we will detail its specific responsibilities, the Application Programming Interfaces (APIs) or endpoints it exposes, its primary interactions with other parts of the system, and the key data entities it manages or processes. This detailed design builds upon the high-level architectural overview presented in Section 3 and draws heavily from the service descriptions found in the project\"s DeepWiki documentation, particularly the "Core Services" and "System Architecture" pages, as well as the functional requirements outlined by the user.

The objective is to provide sufficient detail to guide the development team in implementing each service independently while ensuring they integrate seamlessly to deliver the overall system functionality. The design for each service will consider its role in fulfilling the core features of the system, such as HR posting job openings, applicants

submitting applications, scheduling interviews, changing application statuses, and viewing candidate lists.

5.1. Gateway Service

The Gateway Service, as identified in the system architecture, serves as the single, unified entry point for all client requests originating from the user interfaces (e.g., the Applicant portal and the HR administration portal). Its design is critical for managing external communication, ensuring security, and simplifying the interfaces of the backend microservices.

5.1.1. Responsibilities

The primary responsibilities of the Gateway Service are multifaceted and crucial for the robust operation of the microservices ecosystem:

- **Request Routing:** The foremost responsibility is to intelligently route incoming HTTP requests from clients to the appropriate downstream microservice (Job Service, Applicant Service, or Interview Service) based on the request path, headers, or other criteria. This decouples clients from the internal network locations and number of instances of the backend services.
- **API Composition/Aggregation:** In some scenarios, the Gateway might aggregate responses from multiple microservices to fulfill a single client request, thus reducing chattiness between the client and the backend and simplifying the client-side logic.
- **Authentication and Authorization:** The Gateway Service can act as a central point for authenticating users and authorizing their requests before they reach the internal services. This might involve validating tokens (e.g., JWTs) and checking permissions against defined roles (Applicant, HR).
- **SSL Termination:** It can handle incoming HTTPS connections, decrypt requests, and forward them to backend services over HTTP within the trusted internal network, offloading SSL processing from the individual microservices.
- **Rate Limiting and Throttling:** To protect backend services from being overwhelmed by too many requests, the Gateway can implement rate limiting policies per client or per API endpoint.
- **Load Balancing:** While service discovery (Eureka) helps services find each other, the Gateway can also perform load balancing across available instances of a particular microservice, distributing the request load effectively.
- **Caching:** For frequently accessed, non-sensitive data, the Gateway can cache responses to improve performance and reduce load on backend services.

- **Request/Response Transformation:** It may transform request or response payloads to adapt to different client needs or backend service interfaces.

5.1.2. APIs / Endpoints

The Gateway Service itself does not expose new business-specific APIs but rather acts as a reverse proxy, exposing the aggregated and managed APIs of the downstream services. The URL structure exposed by the Gateway will be designed to be user-friendly and resource-oriented. For example:

- `/api/jobs/*` would route to the Job Service.
- `/api/applicants/*` or `/api/applications/*` would route to the Applicant Service.
- `/api/interviews/*` would route to the Interview Service.
- `/api/auth/*` might handle authentication-related requests (login, logout, token refresh) if authentication is centralized at the Gateway.

The specific paths and methods will mirror those of the underlying services but will be presented under a unified domain and base path. The Gateway configuration will maintain the mapping rules for this routing.

5.1.3. Interactions

The Gateway Service interacts with several other components:

- **Client Applications:** Receives all incoming HTTP(S) requests from user-facing applications (web browsers of Applicants and HR personnel).
- **Job Service, Applicant Service, Interview Service:** Forwards processed and authorized requests to these backend microservices and receives responses from them to send back to the client.
- **Eureka Server:** Registers itself with the Eureka Server (if it\"s also a discoverable service) and uses Eureka to discover the network locations of the backend microservices it needs to route requests to. This allows it to adapt to changes in the number or location of service instances.
- **Authentication Service (if separate):** If authentication logic is complex and encapsulated in a dedicated service, the Gateway would interact with it to validate credentials or tokens.

5.1.4. Data Managed

The Gateway Service is typically stateless and does not manage persistent business data itself. Its state, if any, is usually related to configuration (routing rules, security policies)

or short-lived session information for request correlation or caching. The primary data stores reside within the respective backend microservices.

5.2. Job Service

The Job Service is a core microservice within the Recruitment & Interview Manager system, specifically designed to handle all functionalities related to the creation, management, and retrieval of job postings. Its primary users are HR personnel who need to advertise open positions, and applicants who search and view these positions. The design of this service is critical for enabling the initial stages of the recruitment workflow.

5.2.1. Responsibilities

The Job Service has several key responsibilities that align with its domain focus:

- **Job Posting Creation:** It allows authorized HR users to create new job postings. This includes capturing comprehensive details such as job title, a detailed description of responsibilities and qualifications, required skills, experience level, location, department, employment type (e.g., full-time, part-time, contract), salary range (if applicable), and application deadlines.
- **Job Posting Updates:** HR users must be able to modify existing job postings. This could involve correcting typos, updating requirements, extending application deadlines, or changing the status of a job (e.g., from \'Draft\' to \'Open\', or \'Open\' to \'Closed\').
- **Job Posting Deletion/Archival:** The service should support the removal or archival of job postings that are no longer active or were created in error. Archival might be preferred over hard deletion for record-keeping purposes.
- **Job Search and Retrieval:** It provides functionality for both applicants and HR users to search and retrieve job postings. Applicants will search for open positions based on keywords, location, job category, etc. HR users might need to retrieve specific jobs for management purposes.
- **Job Status Management:** The service manages the lifecycle status of a job posting (e.g., Draft, Open, Filled, Expired, Closed, On Hold). This status determines its visibility and whether applications can be submitted.
- **Data Management:** It is responsible for the persistence and integrity of all data related to job postings, interacting with the `jobs` table (and potentially related entities like job categories or departments if the schema becomes more complex) in the `recruitment_db`.

5.2.2. APIs / Endpoints

The Job Service will expose a set of RESTful APIs to enable the functionalities described above. These endpoints will be accessed via the API Gateway. The following are representative examples of the APIs:

- **POST `/api/jobs`**: Allows HR users to create a new job posting. The request body would contain all necessary job details (title, description, requirements, etc.).
  - Request Body Example: `{ "title": "Software Engineer", "description": "...", "requirements": "...", "location": "New York", "expiry_date": "2025-08-31" }`
  - Response: `201 Created` with the newly created job resource, including its `job_id`.
- **GET `/api/jobs`**: Retrieves a list of job postings. This endpoint should support pagination and filtering parameters (e.g., `?status=Open&location=NewYork&keyword=Engineer`).
  - Response: `200 OK` with a list of job objects.
- **GET `/api/jobs/{job_id}`**: Retrieves the details of a specific job posting by its ID.
  - Response: `200 OK` with the job object, or `404 Not Found` if the job does not exist.
- **PUT `/api/jobs/{job_id}`**: Allows HR users to update an existing job posting. The request body would contain the fields to be updated.
  - Request Body Example: `{ "description": "Updated description...", "status": "Open" }`
  - Response: `200 OK` with the updated job resource, or `404 Not Found`.
- **DELETE `/api/jobs/{job_id}`**: Allows HR users to delete (or archive) a job posting.
  - Response: `204 No Content` on successful deletion, or `404 Not Found`.
- **PATCH `/api/jobs/{job_id}/status`**: A more specific endpoint for changing the status of a job (e.g., to \'Closed\').
  - Request Body Example: `{ "status": "Closed" }`
  - Response: `200 OK`.

These APIs will adhere to REST principles, using appropriate HTTP methods, status codes, and JSON for request/response payloads.

5.2.3. Interactions

The Job Service interacts with other components within the system:

- **API Gateway:** All external requests (from client applications used by HR or Applicants) to the Job Service are routed through the API Gateway. The Job Service receives requests from and sends responses back to the Gateway.

- **Database (`recruitment_db`):** The Job Service performs Create, Read, Update, and Delete (CRUD) operations on the `jobs` table (and any other related tables it owns) in the central MySQL database. It is responsible for all data persistence related to job postings.
- **Eureka Server:** The Job Service registers itself with the Eureka Server upon startup and sends regular heartbeats to remain discoverable by the API Gateway and potentially other internal services (though direct inter-service communication might be limited in this specific design if all calls go via the Gateway).
- **Applicant Service (Indirectly):** While not a direct synchronous call, the jobs created and managed by the Job Service are fundamental for the Applicant Service, as applicants apply for these jobs. The `job_id` from the Job Service will be a key foreign key in the `applications` table managed by the Applicant Service.

5.2.4. Data Managed

The primary data entity managed by the Job Service is the **Job Posting**. This corresponds to the `jobs` table in the database schema outlined in Section 4 (Data Design). Key attributes managed include:

- `job_id` (Primary Key)
- `hr_user_id` (Foreign Key to identify the HR user who posted the job)
- `title`
- `description`
- `requirements`
- `location`
- `posted_date`
- `expiry_date`
- `status` (e.g., Open, Closed, Draft)
- Timestamps for creation and updates (`created_at` , `updated_at`)

The service ensures the integrity, consistency, and availability of this job-related data.

5.3. Applicant Service

The Applicant Service is a pivotal microservice in the Recruitment & Interview Manager system, dedicated to managing all aspects of applicant information and their job applications. This service directly supports the core user journey for job seekers and provides HR personnel with the necessary tools to manage candidate data effectively. Its design must ensure a smooth application process for applicants and efficient data retrieval and management for HR.

5.3.1. Responsibilities

The Applicant Service is charged with several critical responsibilities:

- **Applicant Profile Management (Potentially part of User Management):** While the `Users` table (Section 4.2.2) includes an \'Applicant\' role, this service would handle specific profile details relevant to job applications if they extend beyond basic user credentials. This could include contact information, work experience, education, skills, and uploaded resume/CV documents that form a general applicant profile, distinct from individual job applications.
- **Application Submission:** This is a primary function where applicants submit their interest for a specific job posting (identified by `job_id` from the Job Service). The service will capture all application-specific information, which might include answers to pre-screening questions, a tailored cover letter, and references to uploaded documents (like a resume specifically for that application, even if a general one exists on their profile).
- **Application Document Management:** It manages the storage and retrieval of documents submitted with an application, such as resumes, cover letters, and portfolios. This involves storing file references (e.g., paths to files in a designated storage location) as detailed in the `application_documents` table.
- **Application Status Tracking:** The service maintains the status of each application as it progresses through the recruitment pipeline (e.g., Submitted, Under Review, Shortlisted, Interview Scheduled, etc.). It allows HR users to update these statuses and enables applicants to view the current status of their applications.
- **Candidate Data Retrieval for HR:** It provides HR users with the ability to view lists of applicants for specific jobs, access individual applicant profiles and application details, and filter or search through candidate pools based on various criteria.
- **Data Management:** The Applicant Service is responsible for the persistence and integrity of data related to applicants, their applications, and associated documents. This primarily involves CRUD operations on the `applications` and `application_documents` tables, and potentially parts of the `users` table if applicant-specific profile fields are extensive.

5.3.2. APIs / Endpoints

The Applicant Service will expose RESTful APIs to support its functionalities. These endpoints will be accessed via the API Gateway. Examples include:

- **POST `/api/applications`**: Allows an authenticated applicant to submit an application for a specific job.
    - Request Body Example: `{ "job_id": 123, "applicant_user_id": 456, "cover_letter_text": "...", "documents": [{"document_name": "resume.pdf", "document_type": "Resume", "file_path_or_reference": "/uploads/resume_xyz.pdf"}] }`
    - Response: `201 Created` with the newly created application resource, including its `application_id`.
- **GET `/api/applications`**: For HR users, retrieves a list of applications, supporting pagination and filtering (e.g., `?job_id=123&status=Submitted`). For applicants, retrieves a list of their own submitted applications.
    - Response: `200 OK` with a list of application objects.
- **GET `/api/applications/{application_id}`**: Retrieves details of a specific application by its ID. Accessible by the applicant who owns it or by HR users.
    - Response: `200 OK` with the application object, or `404 Not Found`.
- **PUT `/api/applications/{application_id}/status`**: Allows HR users to update the status of an application.
    - Request Body Example: `{ "status": "Shortlisted" }`
    - Response: `200 OK` with the updated application resource.
- **GET `/api/applicants/{applicant_user_id}/profile`**: Retrieves the profile information of a specific applicant (if profile management is distinct and detailed).
    - Response: `200 OK` with applicant profile data.
- **PUT `/api/applicants/{applicant_user_id}/profile`**: Allows an applicant to update their profile information.
    - Response: `200 OK`.
- **POST `/api/applications/{application_id}/documents`**: Allows an applicant to upload a document for a specific application.
    - Request Body Example: `{ "document_name": "portfolio.pdf", "document_type": "Portfolio", "file_path_or_reference": "/uploads/portfolio_abc.pdf" }`
    - Response: `201 Created` with document details.
- **GET `/api/applications/{application_id}/documents/{document_id}`**: Retrieves a specific document associated with an application (e.g., for download).
    - Response: `200 OK` with file stream or reference.

These APIs will use standard HTTP methods, status codes, and JSON payloads.

5.3.3. Interactions

The Applicant Service interacts with several other system components:

- **API Gateway:** All client requests destined for the Applicant Service are routed through the API Gateway. The service receives requests from and sends responses to the Gateway.
- **Database (`recruitment_db`):** Performs CRUD operations on the `applications`, `application_documents`, and potentially `users` tables in the central MySQL database.
- **Eureka Server:** Registers itself with the Eureka Server for discovery by the API Gateway.
- **Job Service (Indirectly):** Relies on `job_id` from the Job Service to associate applications with specific job postings. It does not typically call the Job Service directly for application submission but uses the `job_id` as a foreign key.
- **Interview Service (Indirectly):** Application status changes (e.g., to \'Interview Scheduled\') managed here might trigger or be triggered by actions in the Interview Service. The `application_id` from this service will be a key input for the Interview Service.
- **File Storage System (Implicit):** For managing uploaded documents (resumes, cover letters), the Applicant Service will need to interact with a file storage solution. This could be a local filesystem path (as suggested by `file_path_or_reference`) in a simple setup, or a dedicated object storage service (like AWS S3, MinIO) in a more scalable deployment. The service would store the file and save its path or reference in the database.

5.3.4. Data Managed

The Applicant Service is primarily responsible for managing data related to:

- **Applications:** Corresponds to the `applications` table. Key attributes include `application_id`, `job_id`, `applicant_user_id`, `submission_date`, `status`, and `cover_letter_text`.
- **Application Documents:** Corresponds to the `application_documents` table. Key attributes include `document_id`, `application_id`, `document_name`, `document_type`, and `file_path_or_reference`.
- **Applicant Profiles (Potentially):** If applicant profiles are extensive and managed separately from the basic `users` table, this service would handle that data. Otherwise, it interacts with the `users` table for applicant role users.

The service ensures the accurate storage, retrieval, and status management of all application-related data, forming a critical link between job postings and the interview process.

5.4. Interview Service

The Interview Service is a specialized microservice within the Recruitment & Interview Manager system, responsible for orchestrating and managing all aspects of the candidate interview process. This service is primarily utilized by HR personnel to schedule interviews, track their progress, and record outcomes. It plays a crucial role in moving candidates from the application stage to potential hiring, ensuring a structured and well-documented interview workflow.

5.4.1. Responsibilities

The Interview Service has several key responsibilities related to the interview lifecycle:

- **Interview Scheduling:** It enables HR users to schedule interviews for candidates who have been shortlisted (typically identified by an `application_id` from the Applicant Service). This includes selecting interview types/rounds (e.g., phone screen, technical interview, panel interview, HR round), assigning interviewers (names or user IDs if interviewers are also system users), setting dates, times, and specifying locations (physical or virtual meeting links).
- **Calendar Integration (Future Scope/Optional):** While not explicitly detailed in the initial requirements, a mature interview service might integrate with calendar systems (e.g., Google Calendar, Outlook Calendar) to check interviewer availability and send out meeting invitations. For the current scope, it will manage schedules internally.
- **Interview Status Management:** The service tracks the status of each scheduled interview (e.g., Scheduled, Confirmed, Completed, Cancelled, Rescheduled, No-show). HR users can update these statuses as events unfold.
- **Feedback Collection and Management:** It provides a mechanism for interviewers or HR personnel to record feedback and evaluations for each interview conducted. This feedback is crucial for making informed hiring decisions. The structure of feedback (e.g., ratings, notes) would be defined.
- **Interview Round Management:** The system supports multiple rounds of interviews for a single application. The Interview Service manages the sequence and details of these rounds.
- **Notification Management (Conceptual):** Although not explicitly detailed as a separate notification service, the Interview Service would likely be responsible for

triggering notifications to applicants about scheduled interviews, reminders, or changes in schedule, and potentially to interviewers as well. This might be via email or in-app notifications, potentially through an integrated notification mechanism or by publishing events that another service consumes.
- **Data Management:** It is responsible for the persistence and integrity of all data related to interviews, interacting with the `interviews` table (and potentially related entities like `interview_rounds` or `feedback_templates`) in the `recruitment_db`.

5.4.2. APIs / Endpoints

The Interview Service will expose RESTful APIs to facilitate its functionalities. These endpoints will be accessed via the API Gateway. Representative examples include:

- **POST `/api/interviews`**: Allows HR users to schedule a new interview for a given application.
  - Request Body Example: `{"application_id": 789, "interviewer_name": "Jane Doe", "interview_round_name": "Technical Round 1", "scheduled_datetime": "2025-06-15T14:00:00Z", "location_or_link": "https://meet.example.com/interview123"}`
  - Response: `201 Created` with the newly created interview resource, including its `interview_id`.
- **GET `/api/interviews`**: Retrieves a list of interviews. This endpoint should support pagination and filtering (e.g., `?application_id=789&status=Scheduled&date_from=2025-06-01&date_to=2025-06-30`).
  - Response: `200 OK` with a list of interview objects.
- **GET `/api/interviews/{interview_id}`**: Retrieves the details of a specific interview by its ID.
  - Response: `200 OK` with the interview object, or `404 Not Found`.
- **PUT `/api/interviews/{interview_id}`**: Allows HR users to update an existing interview (e.g., reschedule, change interviewer, update location).
  - Request Body Example: `{"scheduled_datetime": "2025-06-16T10:00:00Z", "status": "Rescheduled"}`
  - Response: `200 OK` with the updated interview resource, or `404 Not Found`.
- **PATCH `/api/interviews/{interview_id}/status`**: Specifically updates the status of an interview.
  - Request Body Example: `{"status": "Completed", "outcome": "Progress", "feedback_notes": "Candidate performed well..."}`
  - Response: `200 OK`.

- **POST `/api/interviews/{interview_id}/feedback`**: Allows recording of feedback for a completed interview (if feedback is managed as a sub-resource or part of the interview update).
    - Request Body Example: `{ "interviewer_id": 101, "rating": 4, "comments": "Strong technical skills demonstrated." }`
    - Response: `200 OK` or `201 Created` if feedback is a separate entity.
- **GET `/api/applications/{application_id}/interviews`**: Retrieves all interviews scheduled for a specific application.
    - Response: `200 OK` with a list of interview objects.

These APIs will adhere to REST principles, using appropriate HTTP methods, status codes, and JSON for request/response payloads.

5.4.3. Interactions

The Interview Service interacts with other components within the system:

- **API Gateway:** All external requests (from client applications used by HR) to the Interview Service are routed through the API Gateway.
- **Database (`recruitment_db`):** Performs CRUD operations on the `interviews` table (and any other related tables it owns, like `interview_feedback`) in the central MySQL database.
- **Eureka Server:** Registers itself with the Eureka Server for discovery by the API Gateway.
- **Applicant Service (Indirectly):** It relies on `application_id` from the Applicant Service to associate interviews with specific candidates and their applications. Updates to interview status (e.g., "Completed", "Offer Stage") might trigger status updates in the `applications` table managed by the Applicant Service, potentially via direct API calls if necessary or through an event-driven mechanism.
- **Notification System (Conceptual):** May interact with a notification system (or trigger events for one) to send alerts to applicants and interviewers regarding interview schedules, changes, or reminders.

5.3.4. Data Managed

The primary data entity managed by the Interview Service is the **Interview**. This corresponds to the `interviews` table in the database schema outlined in Section 4 (Data Design). Key attributes managed include:

- `interview_id` (Primary Key)
- `application_id` (Foreign Key linking to the application)

- `interviewer_name` (or `interviewer_user_id`)
- `interview_round_name` (or type)
- `scheduled_datetime`
- `location_or_link` (for virtual interviews)
- `status` (e.g., Scheduled, Completed, Cancelled)
- `feedback_notes`
- `outcome` (e.g., Progress, Hold, Reject)
- Timestamps for creation and updates (`created_at`, `updated_at`)

The service ensures the accurate scheduling, tracking, and documentation of all interview-related activities, providing a clear history of a candidate\"s progression through the interview stages.

5.5. Eureka Server

The Eureka Server is a critical infrastructure component within the Recruitment & Interview Manager system\'s microservices architecture. Unlike the other services that provide direct business functionalities (Job, Applicant, Interview services), the Eureka Server provides a crucial runtime governance capability: **service discovery**. Its role, as explicitly mentioned in the DeepWiki documentation, is to allow other microservices to register themselves and to discover the network locations (IP addresses and ports) of other registered services dynamically. This is fundamental for enabling resilient and scalable communication between microservices without hardcoding dependencies.

5.5.1. Responsibilities

The Eureka Server has a well-defined set of responsibilities focused on service registration and discovery:

- **Service Registration:** Each microservice instance (e.g., an instance of the Job Service, Applicant Service, Interview Service, or even the API Gateway) acts as a Eureka client. Upon startup, each client instance registers itself with the Eureka Server, providing metadata such as its service name (e.g., "job-service"), host, port, health indicator URL, and home page URL.
- **Service De-registration:** When a service instance shuts down gracefully, it should de-register itself from the Eureka Server. If an instance crashes or becomes unreachable, Eureka Server will eventually de-register it after a configurable period of missed heartbeats.
- **Service Discovery (Registry Lookup):** Other services (typically the API Gateway or other microservices needing to communicate directly) query the Eureka Server to

find the network locations of available instances of a target service. Eureka Server maintains a registry of all active service instances and provides this information to clients.
- **Instance Health Monitoring (via Heartbeats):** Registered service instances periodically send heartbeat signals (pings) to the Eureka Server to indicate that they are alive and healthy. If the Eureka Server does not receive heartbeats from an instance for a configurable timeout period, it removes that instance from its active registry, assuming it is no longer available. This prevents requests from being routed to unhealthy or non-existent instances.
- **Registry Replication (in a Clustered Setup):** In a production environment, Eureka Servers are typically deployed in a cluster for high availability. Each Eureka Server instance replicates its registry information to other peers in the cluster. This ensures that if one Eureka Server instance fails, others can still provide service discovery capabilities. Clients are usually configured to talk to multiple Eureka Server instances.

5.5.2. APIs / Endpoints (Internal)

While end-users or typical client applications do not directly interact with the Eureka Server\'s APIs, Eureka clients (the microservices themselves) use a set of RESTful APIs to communicate with the Eureka Server. These APIs are part of the Eureka protocol. Key interactions include:

- **POST** `/eureka/apps/{appName}`: Used by a service instance to register itself. `{appName}` is the service ID (e.g., `job-service`). The request body contains instance information (hostname, port, health check URL, etc.) in XML or JSON format (depending on client configuration).
- **DELETE** `/eureka/apps/{appName}/{instanceID}`: Used by a service instance to de-register itself.
- **PUT** `/eureka/apps/{appName}/{instanceID}`: Used by a service instance to send heartbeats to renew its lease with the Eureka Server.
- **GET** `/eureka/apps`: Used by clients to fetch the entire registry of all registered applications and their instances.
- **GET** `/eureka/apps/{appName}`: Used by clients to fetch all instances of a specific application/service.
- **GET** `/eureka/apps/{appName}/{instanceID}`: Used to fetch information about a specific instance of an application.

These endpoints are typically consumed by Eureka client libraries (e.g., `spring-cloud-starter-netflix-eureka-client` in a Spring Boot application) rather than being called directly by application code.

5.5.3. Interactions

The Eureka Server primarily interacts with:

- **Job Service, Applicant Service, Interview Service, Gateway Service (as Eureka Clients):** These services register with Eureka, send heartbeats, and query Eureka to discover other services. The API Gateway, for instance, would query Eureka to find active instances of the Job, Applicant, and Interview services to route requests to.
- **Other Eureka Server Instances (in a Cluster):** In a clustered deployment, Eureka Server instances communicate with each other to replicate registry information and maintain consistency across the cluster.

It does not directly interact with the main database (`recruitment_db`) as its registry is typically held in memory (and can be backed up if needed, but its primary nature is dynamic).

5.5.4. Data Managed

The Eureka Server manages a dynamic registry of service instances. This data is primarily held in memory for fast access and includes for each registered instance:

- Service Name (Application Name, e.g., `JOB-SERVICE`)
- Instance ID (a unique identifier for the specific instance)
- Hostname or IP Address
- Port Number
- Status (UP, DOWN, STARTING, OUT_OF_SERVICE, UNKNOWN)
- Health Check URL
- Home Page URL
- Other metadata (e.g., data center information, version)
- Lease information (duration, last renewal timestamp)

This information is volatile in the sense that it reflects the current state of the deployed microservices and changes frequently as instances start, stop, or scale. The reliability of this registry is crucial for the overall stability and resilience of the microservices architecture.

---

6. Security Design

This section details the security design considerations for the Recruitment & Interview Manager system. Ensuring the confidentiality, integrity, and availability of user data and system resources is paramount. The security measures outlined here will address

authentication, authorization, data security (at rest and in transit), and other relevant security concerns pertinent to a web-based, microservices application handling potentially sensitive personal information.

7.1. Authentication Mechanisms

Authentication is the process of verifying the identity of a user or service. Robust authentication mechanisms will be implemented to ensure that only legitimate users and services can access the system.

- **User Authentication:**
    - **Applicant and HR Portals:** Both Applicant and HR users will be required to authenticate themselves before accessing protected resources and functionalities. This will be achieved through a username (or email) and password-based login system. Passwords will be securely stored using strong, one-way hashing algorithms (e.g., bcrypt, scrypt, or Argon2) with unique salts per user to prevent common password attacks like rainbow table attacks.
    - **Session Management/Token-Based Authentication:** Upon successful login, the system will likely employ token-based authentication, such as JSON Web Tokens (JWT). A JWT will be issued to the client (user\"s browser) and must be included in the header of subsequent requests to protected API endpoints. The API Gateway will be responsible for validating these tokens before routing requests to backend microservices. Tokens will have an expiration time to limit their validity period, and mechanisms for token refresh might be implemented for better user experience.
    - **Password Policies:** Strong password policies will be enforced during user registration and password changes (e.g., minimum length, complexity requirements including uppercase, lowercase, numbers, and special characters). Account lockout mechanisms after a certain number of failed login attempts will be implemented to deter brute-force attacks.
    - **Two-Factor Authentication (2FA) (Future Consideration):** For enhanced security, especially for HR users with higher privileges, 2FA could be considered as a future enhancement.
- **Service-to-Service Authentication:**
    - In a microservices architecture, it is also important for services to authenticate each other when they communicate directly (though in this design, most communication is routed via the API Gateway). If direct inter-service communication is implemented, mechanisms like mutual TLS (mTLS)

or OAuth 2.0 client credentials flow could be used to ensure that only authorized services can interact.

7.2. Authorization and Access Control (Role-Based)

Authorization is the process of determining whether an authenticated user or service has the necessary permissions to perform a specific action or access a particular resource. The Recruitment & Interview Manager system will implement Role-Based Access Control (RBAC).

- **User Roles:** The system defines two primary user roles as identified earlier: `Applicant` and `HR`.
  - **Applicant Role:** Users in this role will have permissions to search for jobs, view job details, submit applications, manage their own profiles and applications, and view the status of their applications and interviews.
  - **HR Role:** Users in this role will have broader permissions, including creating and managing job postings, viewing all applications for jobs they manage, updating application statuses, scheduling interviews, managing interview feedback, and potentially managing other HR user accounts (if an admin HR role is defined).
- **Permission Granularity:** Permissions will be defined at a granular level (e.g., create_job, view_application_details, update_application_status, schedule_interview). These permissions will be assigned to roles.
- **Enforcement:** Authorization checks will be performed primarily at the API Gateway before requests are forwarded to backend services. Backend services may also perform secondary checks to ensure defense in depth. The JWT issued during authentication can contain role information, which the Gateway and services can use to make authorization decisions.
- **Data Scoping:** For HR users, access to data might be further scoped (e.g., an HR user might only be able to manage job postings or applications for their specific department or business unit, though this level of detail is not in the initial requirements).

7.3. Data Security

Protecting the data, especially Personally Identifiable Information (PII) of applicants, is a top priority.

- **Data in Transit:**
  - All communication between client applications (browsers) and the API Gateway, as well as communication between the API Gateway and backend

microservices (if over an untrusted network), will be encrypted using HTTPS (TLS/SSL). This ensures that data exchanged, including credentials, PII, and session tokens, is protected from eavesdropping and tampering.
    - Internal communication between microservices within a trusted network might use HTTP for performance, but if the network cannot be fully trusted, mTLS should be used for inter-service communication as well.
- **Data at Rest:**
    - **Database Encryption:** Sensitive data stored in the MySQL database (`recruitment_db`), such as applicant PII (names, contact details, resume content) and user credentials (password hashes), should be protected. This can be achieved through database-level encryption mechanisms (e.g., MySQL Transparent Data Encryption - TDE, if available and appropriate) or application-level encryption for specific sensitive fields before storing them. File systems storing uploaded documents (resumes, cover letters) should also be encrypted.
    - **Secrets Management:** Application secrets such as API keys, database credentials, and signing keys for JWTs must be managed securely. They should not be hardcoded in source code but stored in secure configuration management systems or environment variables, with restricted access (e.g., using tools like HashiCorp Vault, AWS Secrets Manager, or Kubernetes Secrets).
- **Input Validation and Sanitization:**
    - All user inputs received by the system (via API endpoints) will be rigorously validated and sanitized on the server-side to prevent common web vulnerabilities such as Cross-Site Scripting (XSS), SQL Injection (SQLi), and Command Injection. This applies to data entered in forms, URL parameters, and HTTP headers.
    - Parameterized queries or Object-Relational Mappers (ORMs) will be used for database interactions to prevent SQLi.
- **Secure File Uploads:**
    - Mechanisms for uploading documents (resumes, cover letters) will be secured by validating file types, sizes, and potentially scanning for malware before storing them. Files should be stored in a non-web-accessible location, and access should be mediated by the application.

7.4. Other Security Considerations

- **Security Headers:** Appropriate HTTP security headers (e.g., Content-Security-Policy, X-Content-Type-Options, X-Frame-Options, Strict-Transport-Security) will be implemented to protect against various client-side attacks.

- **Logging and Monitoring:** Comprehensive logging of security-relevant events (e.g., login attempts, access denials, significant data changes) will be implemented. These logs will be monitored for suspicious activities and retained according to policy.
- **Regular Security Audits and Penetration Testing:** The system should undergo regular security assessments, code reviews, and penetration testing to identify and remediate vulnerabilities.
- **Dependency Management:** Software dependencies (libraries, frameworks) will be regularly scanned for known vulnerabilities, and outdated or vulnerable dependencies will be updated or patched promptly.
- **Error Handling:** Error messages will be designed to be informative to the user without revealing sensitive system information that could be exploited by attackers.
- **Data Privacy Compliance:** The system design and data handling practices will need to consider relevant data privacy regulations (e.g., GDPR, CCPA, depending on the target users and deployment region). This includes mechanisms for data access requests, data deletion (right to be forgotten), and consent management where applicable.

By implementing these security measures, the Recruitment & Interview Manager system aims to provide a secure environment for its users and protect the sensitive data it handles.

---

7. Deployment Considerations

This section outlines the key considerations for deploying the Recruitment & Interview Manager system. As a microservices-based application, the deployment strategy needs to address how individual services are packaged, deployed, managed, and scaled. While the DeepWiki documentation provides architectural insights, specific deployment environment details are not fully elaborated, so this section will cover common and recommended practices for such systems.

8.1. Deployment Diagram (Conceptual)

(A visual deployment diagram would be embedded here in a formal SDD. The following is a textual description of what such a diagram would typically illustrate for a microservices application.)

The deployment diagram would visually represent the physical or virtual infrastructure where the system components are hosted. Key elements would include:

- **Containerization:** Each microservice (Job Service, Applicant Service, Interview Service, Gateway Service, Eureka Server) will be packaged as a lightweight, portable container (e.g., using Docker). This ensures consistency across different environments (development, testing, production).
- **Container Orchestration:** A container orchestration platform like Kubernetes (K8s) or Docker Swarm will be used to automate the deployment, scaling, and management of these containerized services. The diagram would show a Kubernetes cluster with multiple nodes.
- **Service Instances:** Multiple instances of each microservice (except perhaps Eureka Server in a very small setup, though clustering is recommended for it too) would be deployed across the orchestrator nodes for high availability and load balancing.
- **API Gateway Deployment:** The API Gateway service would be deployed, potentially with multiple instances, and exposed to the internet, often via a load balancer.
- **Database Deployment:** The MySQL database (`recruitment_db`) would be deployed either as a managed cloud database service (e.g., AWS RDS, Google Cloud SQL, Azure Database for MySQL) for ease of management, backups, and scaling, or as a self-managed instance (possibly containerized as well, though stateful services require careful handling in orchestrators).
- **Service Discovery (Eureka Server):** Eureka Server instances would be deployed within the cluster, accessible by other services for registration and discovery.
- **Load Balancers:** External load balancers (e.g., AWS ELB/ALB, Nginx) would distribute incoming traffic from client applications to the API Gateway instances. Internal load balancing might also be handled by the orchestration platform or service mesh.
- **Networking:** Virtual Private Clouds (VPCs) or similar network isolation mechanisms would be used to secure the deployment. Firewalls and network policies would control traffic flow between services and to/from the internet.
- **Persistent Storage:** For the database and any file uploads (resumes, etc.), persistent storage volumes would be required, managed by the orchestration platform or cloud provider.
- **Logging and Monitoring Infrastructure:** Centralized logging (e.g., ELK stack - Elasticsearch, Logstash, Kibana; or EFK stack - Elasticsearch, Fluentd, Kibana) and monitoring/alerting systems (e.g., Prometheus, Grafana, Datadog) would be deployed to collect logs and metrics from all services for operational visibility and troubleshooting.

8.2. Environment Requirements

Successful deployment and operation of the Recruitment & Interview Manager system will depend on several environmental factors:

- **Server Infrastructure:** Sufficient compute resources (CPU, memory, disk space) are needed for the orchestration platform nodes, database server, and any supporting services. This can be on-premises hardware or, more commonly, cloud-based infrastructure (e.g., AWS EC2, Google Compute Engine, Azure Virtual Machines).
- **Operating System:** While containerization abstracts the OS, the underlying host nodes will typically run a stable Linux distribution (e.g., Ubuntu, CentOS, Amazon Linux).
- **Containerization Platform:** Docker runtime will be required on all host nodes.
- **Container Orchestration Platform:** A Kubernetes cluster or Docker Swarm environment needs to be set up and configured.
- **Database Server:** A MySQL server (version compatible with the application) must be available, either self-hosted or as a managed service.
- **Networking Configuration:**
    - Proper network configuration is essential, including DNS resolution for service discovery and external access.
    - Firewall rules to allow traffic on necessary ports (e.g., HTTP/S for the API Gateway, database port for service access, Eureka Server ports).
    - Sufficient network bandwidth for inter-service communication and client traffic.
- **Java Runtime Environment (JRE):** As the services are likely Java-based (Spring Boot), the containers for these services will include the appropriate JRE version.
- **File Storage:** A reliable and scalable file storage solution is needed for applicant documents (resumes, cover letters). This could be a network file system (NFS), a distributed file system, or an object storage service (e.g., AWS S3, Google Cloud Storage).
- **Secrets Management Solution:** A secure way to manage and inject sensitive configuration (database passwords, API keys, JWT signing keys) into the services at runtime.
- **Backup and Recovery Mechanisms:** Regular backups for the database and persistent file storage must be configured, along with a tested recovery plan.

8.3. Deployment Strategy

A phased deployment strategy is recommended:

- **Development Environment:** Each developer might run services locally using Docker Compose or a local Kubernetes setup (e.g., Minikube, Kind, Docker Desktop Kubernetes).
- **Testing/Staging Environment:** A dedicated environment that mirrors production as closely as possible for integration testing, QA, and user acceptance testing (UAT).
- **Production Environment:** The live environment for end-users. Deployments to production should be carefully planned and executed, possibly using strategies like blue/green deployments or canary releases to minimize downtime and risk. Automated CI/CD (Continuous Integration/Continuous Deployment) pipelines will be crucial for managing deployments efficiently and reliably.

8.4. Scalability and Maintenance

- **Scalability:** The microservices architecture and container orchestration allow for independent scaling of services. Individual services (e.g., Applicant Service during peak application periods) can be scaled horizontally by increasing the number of running instances based on load or resource utilization metrics. The database may also need scaling strategies (read replicas, sharding if it becomes a bottleneck, though the current design is a single DB).
- **Maintenance:** Containerization and orchestration simplify updates and rollbacks. Services can be updated independently without affecting others. Centralized logging and monitoring are essential for proactive maintenance and quick issue resolution. Regular patching of underlying OS, JRE, and dependencies is also a key maintenance activity.

---

8. Non-Functional Requirements

This section defines the Non-Functional Requirements (NFRs) for the Recruitment & Interview Manager system. NFRs describe the quality attributes and operational characteristics of the system, such as performance, scalability, reliability, usability, security, and maintainability. These are critical for ensuring the system not only meets its functional requirements but also delivers a satisfactory user experience and is robust and manageable in the long term. The NFRs outlined here are based on the nature of the application, the chosen microservices architecture, and general best practices for web-based systems.

9.1. Performance

Performance requirements address the responsiveness and efficiency of the system under various load conditions.

- **Response Time:**
  - Interactive user operations (e.g., page loads, form submissions, search results display) should complete within 2-3 seconds under normal load conditions (e.g., 95th percentile).
  - API Gateway response times for individual microservice calls should ideally be under 500 milliseconds, excluding network latency between the client and the Gateway.
  - Critical backend operations (e.g., application submission, status updates) should be processed by the respective microservices within 1 second.
- **Throughput:**
  - The system should be able to handle a target number of concurrent users without significant degradation in performance. (Specific numbers, e.g., 100 concurrent HR users and 1000 concurrent applicants, would be defined based on expected load, but are not specified in the initial requirements. For now, the system should be designed with scalability in mind to accommodate growth.)
  - The system should support a target number of job applications submitted per hour/day during peak periods.
- **Load Handling:**
  - The system should remain stable and responsive under peak load conditions. Performance degradation should be graceful, and the system should recover quickly once the peak load subsides.
  - Specific stress tests will be defined to simulate peak loads, such as a large number of applicants applying simultaneously or HR users performing bulk operations.

9.2. Scalability

Scalability refers to the system\"s ability to handle an increasing amount of work or its potential to be enlarged to accommodate that growth.

- **Horizontal Scalability:** Each microservice (Job, Applicant, Interview, Gateway) must be designed to be stateless where possible and allow for horizontal scaling by adding more instances behind a load balancer. The container orchestration platform (e.g., Kubernetes) will manage this.

- **Database Scalability:** The MySQL database (`recruitment_db`) should be configured for scalability. This might involve strategies like read replicas for read-heavy workloads, connection pooling, and optimized queries. If the centralized database becomes a bottleneck, future considerations might include sharding or migrating specific services to their own databases.
- **Eureka Server Scalability:** Eureka Server instances can be clustered to handle a large number of service registrations and lookups.
- **User Load Scalability:** The system should be ableto scale to support a 2x to 5x increase in the number of registered users and concurrent sessions over a 12-month period without requiring major architectural changes.
- **Data Volume Scalability:** The system must efficiently manage and query growing volumes of data (job postings, applicant profiles, applications, documents) over time.

9.3. Reliability and Availability

Reliability ensures the system operates without failure for a specified period, while availability ensures it is operational and accessible when needed.

- **Availability:**
  - The system should aim for high availability, targeting an uptime of at least 99.9% (excluding scheduled maintenance windows).
  - Critical functionalities like job searching, application submission, and HR access to candidate data must be highly available.
- **Fault Tolerance:**
  - The microservices architecture should provide fault isolation. The failure of one non-critical service instance should not cause a cascading failure of the entire system. For example, if the Interview Service has a temporary issue, applicants should still be able to browse jobs and submit applications.
  - The Eureka Server helps in fault tolerance by de-registering unhealthy service instances, preventing traffic from being routed to them.
  - Redundant deployment of services and database (e.g., multiple instances, database replicas) will be implemented to ensure no single point of failure for critical components.
- **Data Integrity and Durability:** All submitted data (applications, resumes, job postings, interview feedback) must be stored durably and protected against loss or corruption. Regular database backups and, if applicable, file storage backups are mandatory.
- **Mean Time Between Failures (MTBF):** The system should aim for a high MTBF for critical components.

- **Mean Time To Recovery (MTTR):** In case of a failure, the system should be recoverable within a defined timeframe (e.g., critical services back online within 30 minutes).

9.4. Usability

Usability refers to the ease with which users can learn, use, and achieve their goals with the system.

- **Learnability:** New users (both Applicants and HR personnel) should be able to learn how to use the system\"s core functionalities with minimal training or documentation. Intuitive navigation and clear labeling are key.
- **Efficiency:** Users should be able to complete common tasks efficiently, with a minimal number of clicks or steps.
- **Memorability:** Occasional users should be able to return to the system after a period of non-use and remember how to use it without having to re-learn everything.
- **Error Prevention and Handling:** The system should be designed to prevent common user errors. When errors do occur, clear, understandable, and actionable error messages should be provided.
- **User Satisfaction:** The overall user experience should be positive, leading to high user satisfaction. Feedback mechanisms might be incorporated to gather user opinions.
- **Accessibility:** The UIs should comply with relevant web accessibility standards (e.g., WCAG 2.1 Level AA) to ensure usability for people with disabilities.

9.5. Maintainability

Maintainability refers to the ease with which the system can be modified, corrected, adapted, and enhanced.

- **Modularity:** The microservices architecture inherently promotes modularity, allowing individual services to be updated, deployed, and maintained independently.
- **Code Quality:** Code should be well-documented, follow consistent coding standards, and be covered by automated tests (unit, integration).
- **Testability:** Services should be designed for testability, with clear APIs and separation of concerns, facilitating automated testing.
- **Deployability:** Automated CI/CD pipelines should be established for building, testing, and deploying services, making the deployment process repeatable and reliable.

- **Configurability:** System parameters and service configurations should be externalized and manageable without requiring code changes.
- **Diagnosability:** Comprehensive logging, monitoring, and tracing capabilities across services are essential for diagnosing and troubleshooting issues quickly.

8.6. Security (Cross-reference)

Security requirements are detailed in Section 6 (Security Design). Key NFR aspects of security include:

- **Confidentiality:** Protecting sensitive data (PII, credentials) from unauthorized disclosure.
- **Integrity:** Ensuring data is accurate and cannot be tampered with by unauthorized parties.
- **Auditability:** The ability to log and audit security-relevant events.

These NFRs will guide the design, development, and testing phases to ensure the Recruitment & Interview Manager system is not only functional but also robust, user-friendly, and secure.

---

9. Glossary

This section provides definitions for key terms, acronyms, and abbreviations used throughout the Software Design Document (SDD) for the Recruitment & Interview Manager system. The aim is to ensure a common understanding of the terminology for all stakeholders reading this document.

- **API (Application Programming Interface):** A set of rules and protocols that allows different software applications to communicate with each other. In this system, RESTful APIs are used for communication between the frontend, API Gateway, and microservices.
- **API Gateway:** A server that acts as a single entry point for all client requests to the backend microservices. It handles tasks like request routing, authentication, SSL termination, and rate limiting. (See Gateway Service)
- **Applicant:** A target user of the system; an individual seeking employment who uses the system to find jobs, submit applications, and track their status.
- **Applicant Service:** A core microservice responsible for managing applicant profiles, submitted applications, and related data.

- **Application (Job Application):** A formal request submitted by an applicant for a specific job opening, typically including a resume and other relevant information.
- **Authentication:** The process of verifying the identity of a user, service, or device, usually by checking credentials like a username and password or a token.
- **Authorization:** The process of determining whether an authenticated entity has the permission to access a specific resource or perform a particular action.
- **Availability:** The percentage of time that a system is operational and accessible to users when needed.
- **CI/CD (Continuous Integration/Continuous Deployment or Delivery):** A set of practices and tools that automate the process of building, testing, and deploying software changes.
- **Containerization:** The process of packaging software code and all its dependencies (libraries, frameworks, configuration files) into a standardized unit called a container (e.g., using Docker). This ensures the application runs consistently across different computing environments.
- **CRUD:** An acronym for Create, Read, Update, and Delete, which are the four basic functions of persistent storage.
- **Database:** An organized collection of structured information, or data, typically stored electronically in a computer system. In this project, a MySQL database named `recruitment_db` is used.
- **Deployment:** The process of making a software system available for use in a specific environment (e.g., development, testing, production).
- **Docker:** A popular open-source platform for developing, shipping, and running applications in containers.
- **ERD (Entity-Relationship Diagram):** A visual representation of the entities (tables) in a database and the relationships between them.
- **Eureka Server:** A service discovery tool from the Netflix OSS suite, used in microservices architectures to enable services to register themselves and discover other services dynamically.
- **Fault Tolerance:** The ability of a system to continue operating, possibly at a reduced level, rather than failing completely when one or more of its components fail.
- **Gateway Service:** (See API Gateway)
- **HR (Human Resources):** A target user of the system; personnel within an organization responsible for managing the recruitment process, including posting jobs, reviewing applications, and scheduling interviews.
- **HTTP (Hypertext Transfer Protocol):** The foundation of data communication for the World Wide Web. HTTPS is its secure version.
- **Instance (Service Instance):** A running copy of a microservice. Multiple instances can be run for scalability and fault tolerance.

- **Interview Service:** A core microservice responsible for managing the scheduling, tracking, and feedback collection for candidate interviews.
- **JDBC (Java Database Connectivity):** An API for the Java programming language that defines how a client may access a database.
- **Job Posting:** An advertisement for an open position within an organization, detailing the role, responsibilities, and qualifications.
- **Job Service:** A core microservice responsible for managing the creation, updating, and retrieval of job postings.
- **JSON (JavaScript Object Notation):** A lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate.
- **JWT (JSON Web Token):** A compact, URL-safe means of representing claims to be transferred between two parties, commonly used for authentication and authorization in web applications.
- **Kubernetes (K8s):** An open-source container orchestration platform for automating the deployment, scaling, and management of containerized applications.
- **Load Balancing:** The process of distributing network traffic or computational workload across multiple servers or resources to optimize resource use, maximize throughput, minimize response time, and avoid overload.
- **Microservice:** An architectural style that structures an application as a collection of small, autonomous, and independently deployable services, each focused on a specific business capability.
- **Module:** A distinct and self-contained unit of a software system that performs a specific function. In this SDD, microservices are often referred to as modules or components.
- **MySQL:** A popular open-source relational database management system (RDBMS).
- **NFR (Non-Functional Requirement):** A requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors (which are functional requirements). Examples include performance, security, usability, and reliability.
- **PII (Personally Identifiable Information):** Any data that could potentially identify a specific individual. Examples include names, addresses, email addresses, and resume details.
- **REST (Representational State Transfer):** An architectural style for designing networked applications, commonly used for building web services (APIs) that communicate over HTTP.
- **RBAC (Role-Based Access Control):** A method of restricting system access to authorized users based on their roles within an organization.

- **Scalability:** The ability of a system to handle a growing amount of work by adding resources, either by scaling up (increasing the capacity of existing resources) or scaling out (adding more resources).
- **SDD (Software Design Document):** This document; a comprehensive description of the design of a software system.
- **Service Discovery:** The process by which services in a microservices architecture dynamically find the network locations (IP address and port) of other services they need to communicate with. (See Eureka Server)
- **Spring Boot:** A popular Java-based framework used to create stand-alone, production-grade Spring-based applications with minimal configuration.
- **SSL (Secure Sockets Layer) / TLS (Transport Layer Security):** Cryptographic protocols designed to provide secure communication over a computer network. HTTPS uses SSL/TLS.
- **Technology Stack:** The set of technologies, software, and tools used to build and run an application.
- **UI (User Interface):** The means by which a user interacts with a computer system or application.
- **UML (Unified Modeling Language):** A standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers specify, visualize, construct, and document the artifacts of software systems.
- **User Story:** A high-level, informal description of a software feature from the perspective of an end-user.
- **WCAG (Web Content Accessibility Guidelines):** A set of guidelines for making web content more accessible to people with disabilities.

---

10. References

This section lists all the documents, web pages, and other resources that were referenced during the creation of this Software Design Document (SDD) for the Recruitment & Interview Manager system. These references provided the foundational information, requirements, and architectural insights that shaped the design detailed herein.

1. **User-Provided Project Overview and Requirements:**
    - Initial message from the user (OmarAlhaj10) received on May 12, 2025, outlining the project overview, target users (Applicant, HR), and core features

(HR posts job openings, Applicants submit applications, Schedule interview dates, Change application status, View list of candidates per job).

2. **Recruitment & Interview Manager Project DeepWiki Documentation:**
    - Main Overview Page: https://deepwiki.com/OmarAlhaj10/SwE2project/1-overview
    - System Architecture Page: (Implicitly) https://deepwiki.com/OmarAlhaj10/SwE2project/2-system-architecture (and its sub-pages like Service Discovery with Eureka, API Gateway Pattern)
    - Database Design Page: (Implicitly) https://deepwiki.com/OmarAlhaj10/SwE2project/2.3-database-design
    - Core Services Page: (Implicitly) https://deepwiki.com/OmarAlhaj10/SwE2project/3-core-services (and its sub-pages for Gateway Service, Job Service, Applicant Service, Interview Service, Eureka Server)
    - Other related pages on the DeepWiki site concerning Project Configuration, Deployment Architecture, etc., were also reviewed for context.
3. **Internal Documentation and Analysis Files (Generated during SDD preparation):**
    - `/home/ubuntu/project_requirements_initial.md` : File capturing the initial project requirements as provided by the user.
    - `/home/ubuntu/project_requirements_deepwiki_overview.md` : File summarizing information extracted from the DeepWiki project overview page.
    - `/home/ubuntu/project_requirements_deepwiki_architecture.md` : File summarizing information extracted from the DeepWiki system architecture page.
    - `/home/ubuntu/project_requirements_deepwiki_database.md` : File summarizing information extracted from the DeepWiki database design page.
    - `/home/ubuntu/project_requirements_deepwiki_core_services.md` : File summarizing information extracted from the DeepWiki core services page.
    - `/home/ubuntu/sdd_outline.md` : File containing the structured outline for this Software Design Document.
    - `/home/ubuntu/todo.md` : Task checklist used to track the progress of SDD creation.
4. **General Software Engineering Principles and Best Practices:**
    - Standard concepts and practices related to microservices architecture design.
    - Common patterns for API design (RESTful principles).
    - Best practices for database design and relational data modeling.

- Industry standards for web application security (e.g., OWASP guidelines).
- Principles of user interface and user experience (UI/UX) design.
- Guidelines for writing effective software documentation.

These references collectively formed the basis for the analysis, architectural decisions, and detailed design specifications presented in this document.
