RevJobs-Microservice conversion and cloud migration

“Modernizing a Monolithic Application into Cloud-Based Microservices”

Submitted by:

Isukapatla Vishnu Vardhan

Kelam Sravani

Indranil Mondal

Kuppam Komal

Yejjala Neha Venkata Padmavathi

TABLE CONTEXT

1. ABSTRACT

The RevJobs Job Portal is an enterprise-level web application designed using a modern microservices architecture to meet the demands of scalable, maintainable, and resilient recruitment platforms. The project focuses on transforming a traditional job portal system into a distributed, cloud-ready solution suitable for real-world enterprise environments.

The system is composed of seven independent microservices that communicate through REST APIs and asynchronous messaging mechanisms. Core functionalities include secure user authentication with OAuth2 and JWT, job posting and management, application tracking, real-time communication between recruiters and job seekers, and notification handling.To ensure reliability and fault tolerance, the application incorporates industry-standard patterns such as service discovery using Eureka, centralized routing through an API Gateway, circuit breaker implementation using Resilience4j, and the Saga pattern for managing distributed transactions. Real-time features are enabled using WebSocket communication, while Docker is used for containerized deployment.

This project demonstrates best practices in enterprise software engineering, highlighting the benefits of microservices architecture in building scalable, resilient, and production-ready applications for modern digital recruitment systems.

2. INTRODUCTION

2.1 Background

In the modern digital era, recruitment has shifted from traditional paper-based and manual processes to online platforms. Job portals have become essential systems that connect employers and job seekers by enabling job postings, applications, and communication through a centralized digital interface. As the number of users, features, and data volumes increases, many job portal platforms built on traditional monolithic architectures begin to face limitations related to scalability, maintainability, reliability, and deployment efficiency. These challenges create the need for modern architectural approaches that support flexibility, resilience, and faster delivery in enterprise environments.

2.2 Problem Statement

Many existing enterprise and job portal applications are developed using a monolithic Java and Spring Boot architecture in which all functionalities are tightly coupled within a single codebase. While this approach simplifies initial development, it becomes increasingly difficult to manage as the application grows. Monolithic systems struggle to scale individual components independently, making efficient resource utilization difficult during high-demand scenarios. A failure in one component can affect the entire system, resulting in reduced reliability. Additionally, adopting new technologies for specific functionalities becomes complex due to tight coupling. Even minor changes require rebuilding and redeploying the complete application, which slows down release cycles and increases deployment risk. Large development teams working on a shared codebase also face coordination challenges and frequent merge conflicts, further impacting productivity. The absence of containerization and automated DevOps pipelines further limits deployment speed and cloud readiness.

2.3 Proposed Solution

This project involves incrementally migrating a legacy monolithic Java and Spring Boot application to a microservices-based architecture to overcome the limitations of monolithic systems. The primary goal of the migration is to improve scalability, maintainability, fault isolation, and deployment velocity while leveraging modern DevOps practices. The monolithic application is decomposed into four to six independent microservices aligned with business domains such as user management, job management, application handling, along with a centralized API Gateway. The Strangler Fig pattern is adopted to enable gradual migration while minimizing operational risk and system downtime. Service discovery is implemented using Eureka to support dynamic service registration and communication, while inter-service interactions are handled through synchronous REST-based calls using Feign/OpenFeign and asynchronous messaging using event-driven mechanisms such as Kafka. A centralized API Gateway is introduced to manage routing, security, and request handling. Each microservice is containerized using Docker to ensure consistent execution across environments, and Jenkins-

based CI/CD pipelines are configured to automate building, testing, Docker image creation, image publishing to AWS ECR, and deployment to AWS EC2 instances. Deployment strategies such as rolling or blue-green deployments are used to ensure zero downtime and continuous availability.

2.4 Objectives

The objective of this project is to design and implement a scalable and production-ready microservices-based job portal system that supports independent development and deployment of services. The project aims to apply enterprise architectural patterns such as API Gateway, Circuit Breaker, and Saga to ensure system reliability and consistency. It also focuses on implementing secure authentication and authorization using JWT and OAuth2, enabling real-time communication between recruiters and job seekers, and demonstrating industry-standard DevOps and software engineering practices through automated testing, containerization, and continuous integration and deployment.

2.5 Scope of the Project

The scope of the project includes the development of core job portal functionalities such as user registration and authentication, job posting and management, job application with resume upload, application status tracking, real-time messaging between users, and notification management. The system supports role-based access control for job seekers, recruiters, and administrators, and provides a responsive web interface. The deployment of the application using a microservices architecture on AWS EC2 is also included within the project scope. However, the project does not cover mobile application development, payment gateway integration, video interview scheduling, AI-based job matching, or advanced analytics and reporting features.

2.6 Project Significance

This project is significant as it demonstrates the practical application of modern microservices and cloud-native architectural principles in a real-world enterprise scenario. It highlights the use of industry-standard design patterns to address scalability, reliability, and deployment challenges commonly faced by large applications. The project also emphasizes secure system design, automated DevOps workflows, and containerized deployments, providing hands-on experience with technologies such as Docker, Jenkins, and AWS. Overall, the project aligns closely with current industry requirements and prepares developers to work effectively in enterprise and cloud-based software development environments.

3. LITERATURE SURVEY

Microservices architecture has emerged as a widely adopted approach for building scalable and resilient enterprise applications. In this architectural style, applications are designed as a collection of loosely coupled and independently deployable services, each aligned with a specific business capability. Existing research and industry practices show that microservices help overcome the limitations of monolithic systems by enabling independent scaling, improved fault isolation, faster development cycles, and better maintainability, especially for large and evolving systems such as job portals.

To support microservices-based development, frameworks such as Spring Cloud provide standardized solutions for common distributed system challenges. Spring Cloud simplifies service discovery, configuration management, routing, load balancing, and fault tolerance, allowing developers to focus on business logic rather than infrastructure concerns. The use of service discovery mechanisms enables dynamic communication between services, while centralized gateways help manage client requests efficiently. Several design patterns are commonly referenced in microservices literature as essential for building reliable distributed systems. The API Gateway pattern provides a single entry point for all client interactions, improving security and request management. Service Discovery allows services to dynamically locate each other in changing environments. The Circuit Breaker pattern prevents cascading failures by isolating failing services, while the Saga pattern enables consistent transaction management across multiple services without relying on traditional distributed transactions. These patterns collectively enhance system resilience and reliability.

Security in microservices systems is typically implemented using stateless authentication and standardized authorization mechanisms. JSON Web Tokens are widely used for secure token-based authentication, while OAuth2 enables delegated authorization and third-party login integration. These approaches are well-documented in existing systems for providing secure and scalable access control. Compared to traditional monolithic job portal systems, microservices-based platforms offer higher scalability, independent deployment, better fault tolerance, and more efficient resource utilization. This literature survey highlights that microservices architecture, supported by Spring Cloud, design patterns, and modern security mechanisms, is well-suited for building enterprise-grade recruitment platforms like RevJobs.

4. SYSTEM REQUIREMENTS ANALYSIS

4.1 Functional Requirements

The RevJobs system is designed to support core recruitment functionalities required for a modern job portal. The system provides user management features that allow users to register using email and password credentials as well as authenticate through OAuth2-based social login. It supports multiple user roles, including Job Seeker, Recruiter, and Admin, and allows users to manage and update their profile information securely using JWT-based session management.

The platform enables recruiters to create, update, and manage job postings with relevant details such as job description, requirements, and salary range. Job seekers can search and filter available job postings and apply for suitable positions. The system ensures that duplicate applications are prevented and maintains application states throughout the recruitment lifecycle.

The application also includes an application management workflow that allows recruiters to review submitted applications, update application status, and track progress. Resume uploads are supported in commonly used document formats.

To enhance user interaction, the system provides a real-time messaging feature that enables communication between recruiters and job seekers. Messaging is supported through WebSocket-based communication to ensure instant delivery and conversation tracking. In addition, a notification mechanism is implemented to inform users about application updates and new messages, with support for viewing and managing notification history.

4.2 Non-Functional Requirements

The system is designed to meet key non-functional requirements related to performance, scalability, security, availability, and maintainability. API responses are optimized to provide timely interactions for concurrent users, and the architecture supports horizontal scaling to handle increased load. Each microservice can be scaled independently based on demand.

Security is enforced across all services through authenticated API access, encrypted password storage, token-based authorization, and protection against common vulnerabilities. The system is built with high availability in mind, ensuring that failure of one service does not impact others and that fault tolerance mechanisms such as circuit breakers are in place.

From a maintainability perspective, the application follows clean code principles, externalized configuration, and centralized logging to simplify development and operations. The user interface is designed to be responsive and intuitive, providing clear navigation and error handling across devices.

4.3 System Environment Requirements

The development environment requires a standard workstation capable of running modern development tools, containerization software, and local services. A stable internet connection is required for dependency management, version control, and cloud integration.

The production environment is designed for cloud deployment using containerized microservices hosted on virtual machines. The system architecture supports scalable compute resources for application services, a dedicated relational database server, and load balancing to distribute traffic efficiently.

4.4. Software Requirements

4.4.1 Development Tools

- **Integrated Development Environment (IDE): IntelliJ IDEA, Eclipse, or Visual Studio Code**
- **Programming Language: Java Development Kit (JDK) 17 or higher**
- **Frontend Runtime: Node.js version 16 or higher**
- **Build Tool: Apache Maven 3.8 or later**
- **Version Control System: Git**
- **API Testing Tool: Postman**
- **Containerization Tool: Docker Desktop**

4.4.2 Runtime Environment

- **Operating System: Windows 10/11, Ubuntu 20.04 or later, macOS 11 or later**
- **Database Management System: MySQL 8.0**
- **Container Runtime: Docker version 20.10 or later**
- **Web Browser: Latest versions of Chrome**

4.4.3 Backend Technologies:

- Spring Boot 3.2.0
- Spring Cloud 2023.0.0
- Spring Security
- Spring Data JPA
- Resilience4j
- JWT (io.jsonwebtoken)
- Lombok
- MapStruct

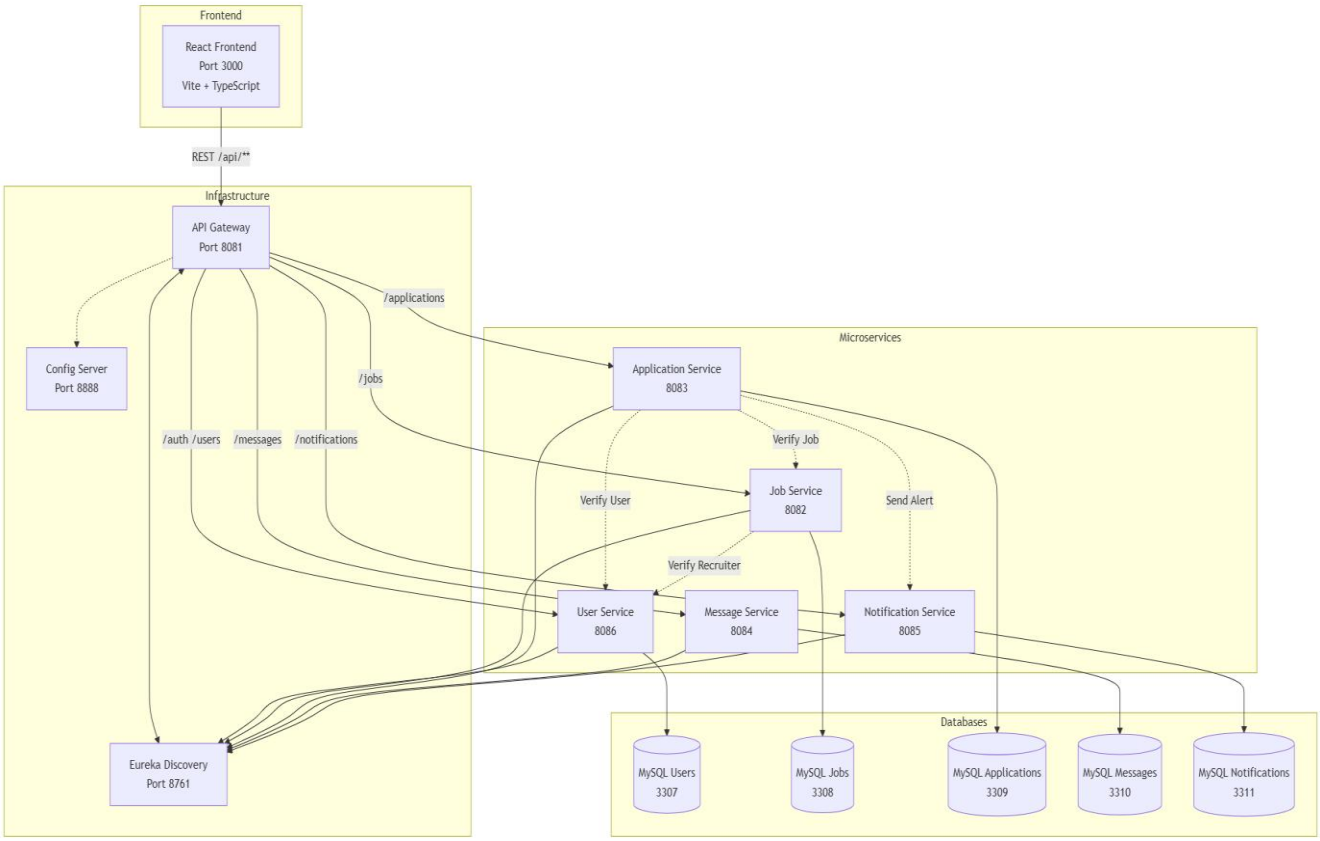
4.4.4 Frontend Technologies:

- React 18.2
- TypeScript 5.0
- Material-UI 5.14
- React Router 6.20
- Axios
- Vite 4.4

5. SYSTEM DESIGN AND ARCHITECTURE

5.1 Overall Architecture

The RevJobs platform follows a microservices architecture with the following components: This diagram illustrates the high-level architecture of the RevJobs platform, showing the interaction between the API Gateway, Config Server, Discovery Server, and individual microservices. It also highlights communication with databases and external clients, providing a complete view of the system structure.



5.2. Deployment Diagram

The deployment diagram represents how the microservices are containerized using Docker and deployed on AWS EC2 instances. It shows the physical deployment environment, including application servers, databases, and network communication.

5.3 Microservices Description

5.3.1 Config Server (Port 8888)

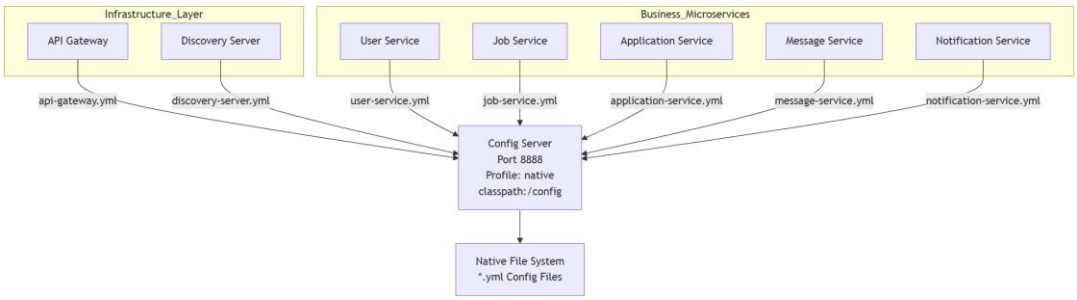
Purpose: Centralized configuration management for all services

Features:

- Stores configuration files for all microservices
- Supports environment-specific configurations
- Enables dynamic configuration updates
- Uses file-based configuration storage

Technologies: Spring Cloud Config Server

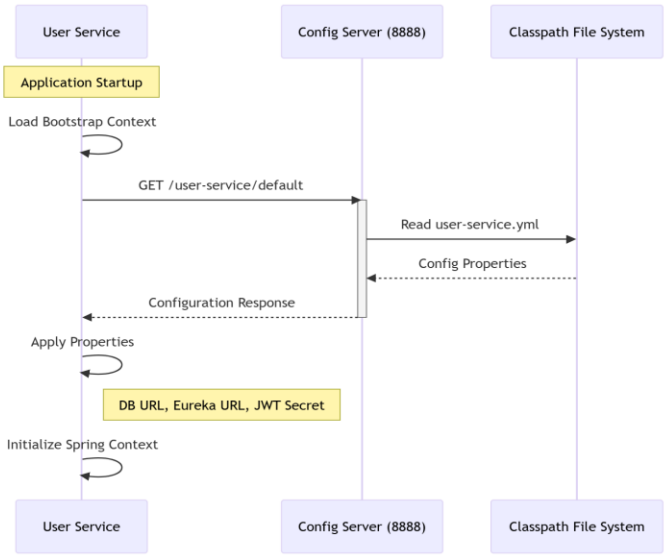
Design Artifacts:



Component Diagram

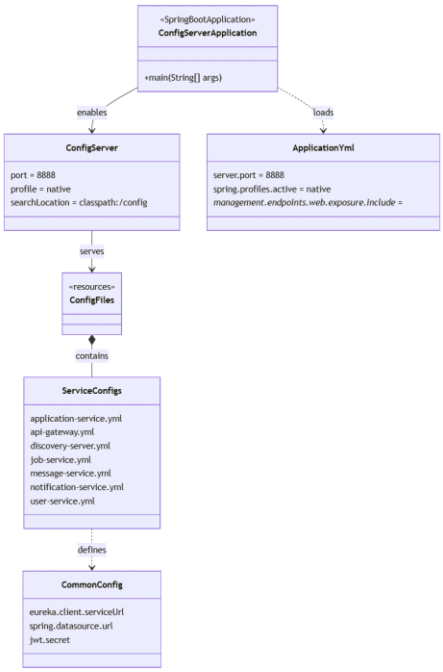
This diagram illustrates the centralized configuration management architecture used in the RevJobs microservices system. The Config Server acts as a single source of configuration for both infrastructure components, such as the API Gateway and Discovery Server, and business microservices including User Service, Job Service, Application Service, Message Service, and Notification Service. Each service retrieves its respective configuration file in YAML format from the Config Server during startup.

The Config Server runs on port 8888 and operates using the native profile, where configuration files are stored in a classpath-based native file system. These configuration files define service-specific properties such as server ports, database connections, service discovery details, and security settings. By externalizing configuration, this approach ensures consistency across services, simplifies environment management



Sequence Diagram

This sequence diagram illustrates how the User Service retrieves its externalized configuration during application startup. When the User Service initializes, it loads the bootstrap context and sends a request to the Config Server to fetch service-specific configuration properties. The Config Server reads the corresponding configuration file from the class path file system and returns the configuration response. The User Service then applies the received properties, such as database URL, Eureka server URL, and JWT secret, and completes the Spring application context initialization. This process enables centralized and environment-specific configuration management across microservices.



Class Diagram

This Class diagram illustrates the internal configuration structure of the Config Server. It shows how the Spring Boot Config Server loads service-specific YAML configuration files from a classpath-based directory and provides centralized configuration properties such as database URLs, service discovery settings, and security parameters. This approach ensures consistent and maintainable configuration management across all microservices.

5.3.2 Discovery Server (Port 8761)

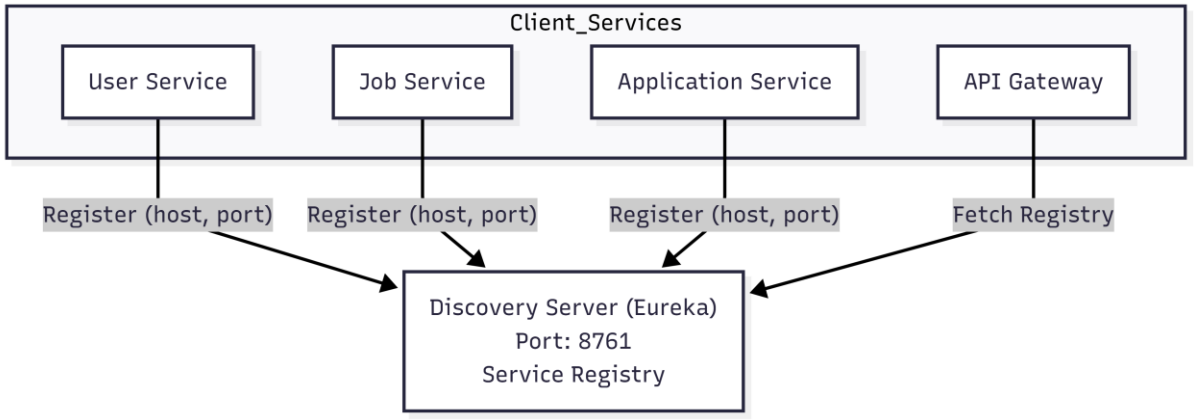
Purpose: Service registry for dynamic service discovery

Features:

- Registers all microservice instances
- Provides service location information
- Monitors service health
- Enables client-side load balancing

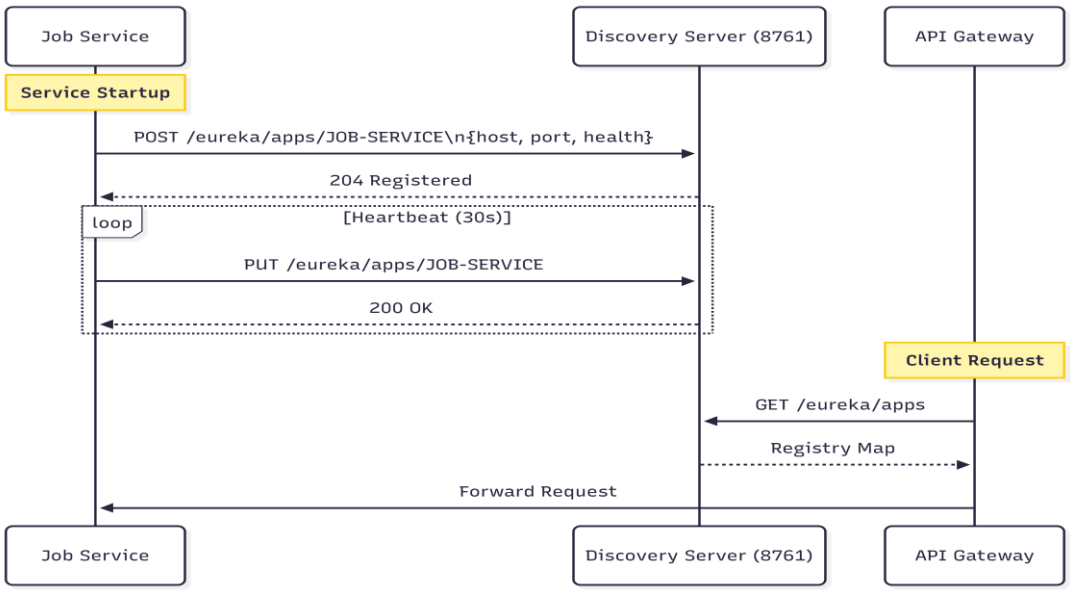
Technologies: Netflix Eureka Server

Design Artifacts:



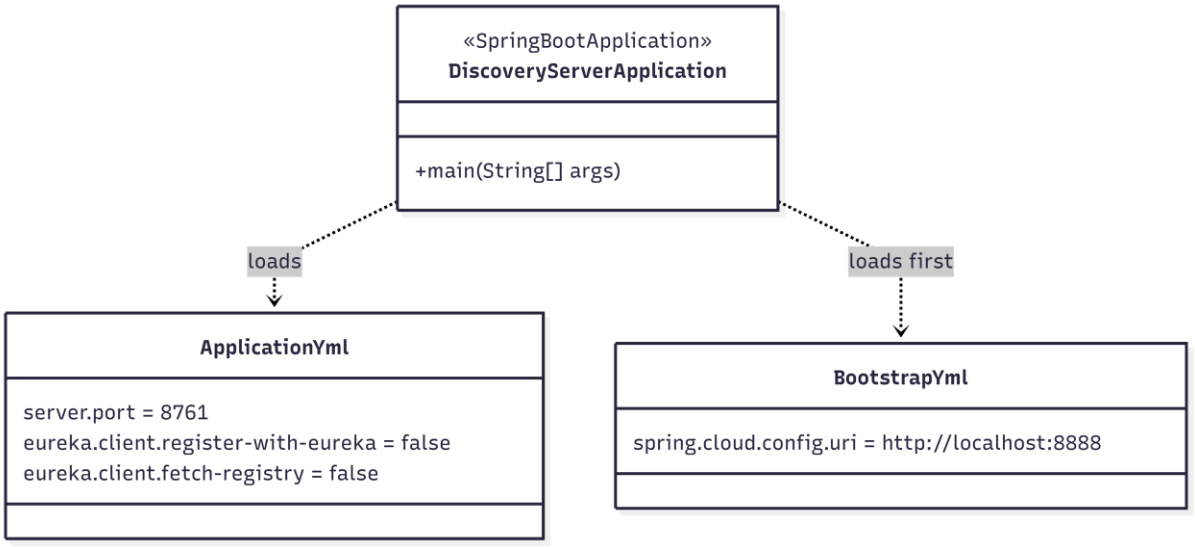
Component Diagram

This diagram illustrates the service discovery mechanism used in the RevJobs microservices architecture. Individual services such as User Service, Job Service, and Application Service register themselves with the Discovery Server (Eureka) by providing their host and port information. The API Gateway queries the Discovery Server to fetch the service registry and dynamically route client requests to available service instances. This approach enables dynamic service discovery, load balancing, and improved system scalability and resilience.



Sequence Diagram

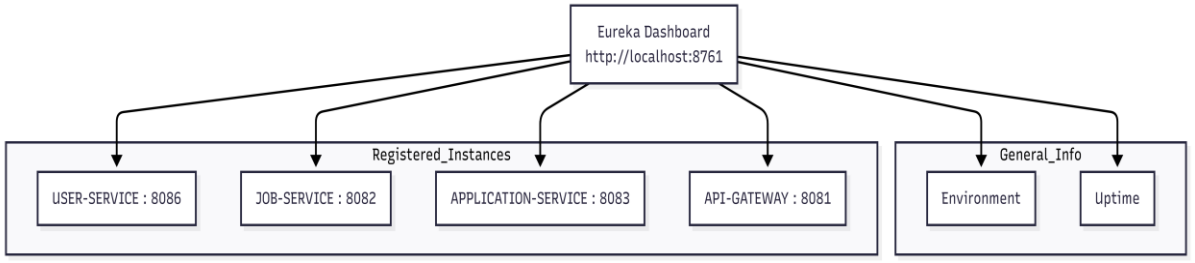
This diagram illustrates the service registration and discovery flow using Eureka. When the Job Service starts, it registers itself with the Discovery Server by sending its host, port, and health information. The service periodically sends heartbeat requests to maintain its registration. When a client request arrives, the API Gateway queries the Discovery Server to retrieve the service registry and forwards the request to the appropriate Job Service instance. This mechanism enables dynamic service discovery, load balancing, and fault tolerance within the microservices architecture.



Class Diagram

This diagram illustrates the initialization structure of the Discovery Server application. It shows how the Spring Boot Discovery Server loads configuration properties from bootstrap.yml and application.yml during startup. The bootstrap.yml is loaded first to configure the connection with the Config Server, while application.yml defines Eureka-specific settings such as server port and registry behavior. This configuration ensures proper initialization and operation of the Discovery Server within the microservices environment.

Operational View :



Eureka Service Registry Dashboard

This figure represents the Eureka Dashboard view, which provides a real-time overview of all registered microservice instances in the system. It displays services such as User Service, Job Service, Application Service, and API Gateway along with their respective ports. The dashboard also shows general system information, including environment details and uptime. This view is used for monitoring service availability, registration status, and overall health during runtime.

5.2.3 API Gateway (Port 8080)

Purpose: Single entry point for all client requests

Features:

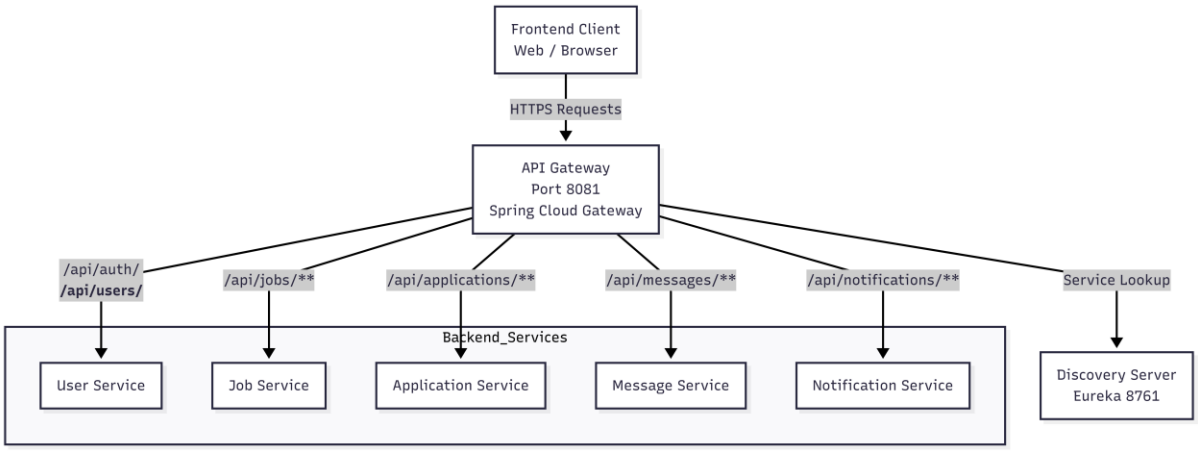
- Routes requests to appropriate services
- JWT token validation
- CORS configuration
- Request/response filtering
- Load balancing

Technologies: Spring Cloud Gateway

Routes:

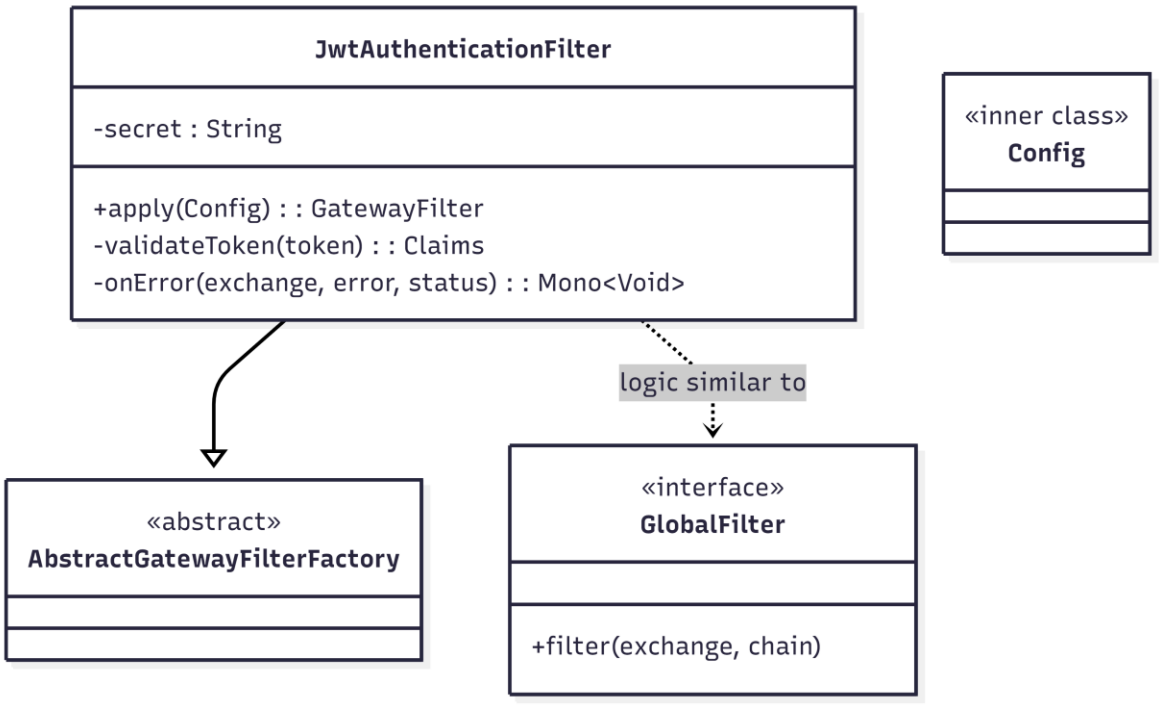
- `/api/auth/**` → User Service
- `/api/users/**` → User Service
- `/api/jobs/**` → Job Service
- `/api/applications/**` → Application Service
- `/api/messages/**` → Message Service
- `/api/notifications/**` → Notification Service

Design Artifacts:



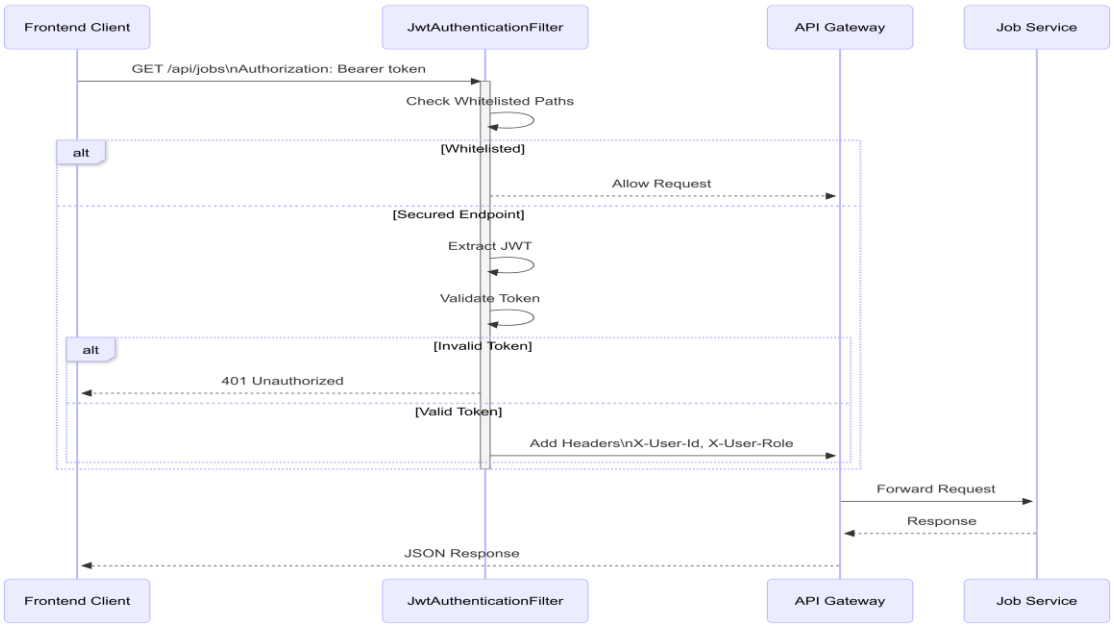
Component Diagram

This diagram illustrates the request routing mechanism of the API Gateway in the RevJobs microservices architecture. Client requests from the web browser are sent to the API Gateway over HTTPS, which acts as a single entry point to the system. Based on predefined route patterns, the gateway forwards requests to the appropriate backend services such as User Service, Job Service, Application Service, Message Service, and Notification Service. The API Gateway uses the Discovery Server to dynamically locate service instances, enabling centralized routing, security enforcement, and scalable request handling.



Class Diagram

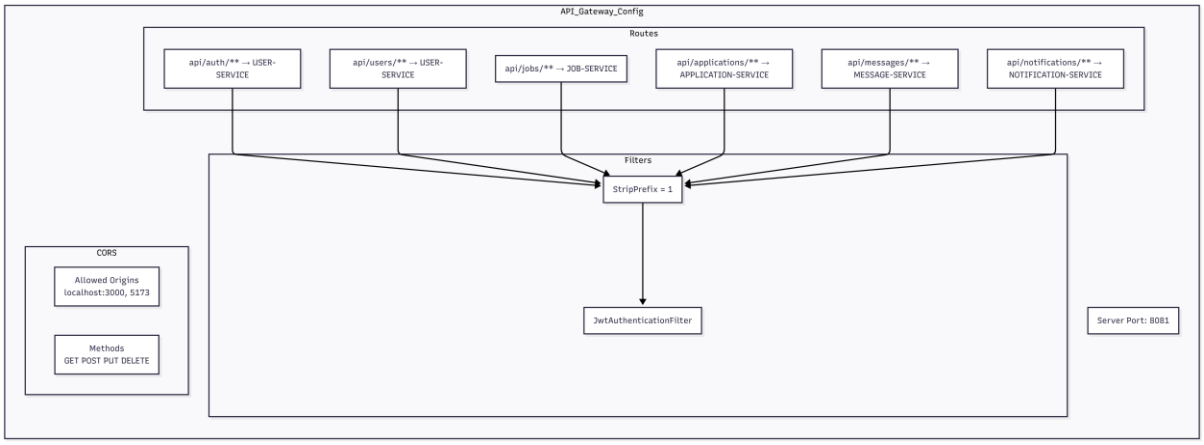
This class diagram represents the structure of the JWT authentication filter implemented in the API Gateway. It shows the filter's attributes, methods, inheritance from the gateway filter factory, and its relationship with global filtering mechanisms used for request authentication.



Sequence Diagram

This diagram illustrates the JWT-based authentication flow implemented at the API Gateway. When a client sends a request, the JWT authentication filter first checks whether the requested endpoint is whitelisted. For secured endpoints, the filter extracts and validates the JWT token from the request. If the token is invalid, the request is rejected with an unauthorized response. If the token is valid, user details such as user ID and role are added to the request headers, and the API Gateway forwards the request to the Job Service. This mechanism ensures centralized authentication and secure access to backend microservices.

Gateway Request Processing View:



Configuration Structure

This diagram represents the API Gateway configuration, showing route mappings, request filters, and CORS settings. It illustrates how incoming requests are matched to backend services using path-based routing, processed through filters such as prefix stripping and JWT authentication, and forwarded securely to the appropriate microservices.

5.2.4 User Service (Port 8081)

Purpose: Manages user authentication and authorization

Features:

- User registration and login
- JWT token generation and validation
- OAuth2 integration (Google, GitHub)
- Password encryption (BCrypt)
- Role-based access control
- OTP generation for password reset

Database: revjobs_users

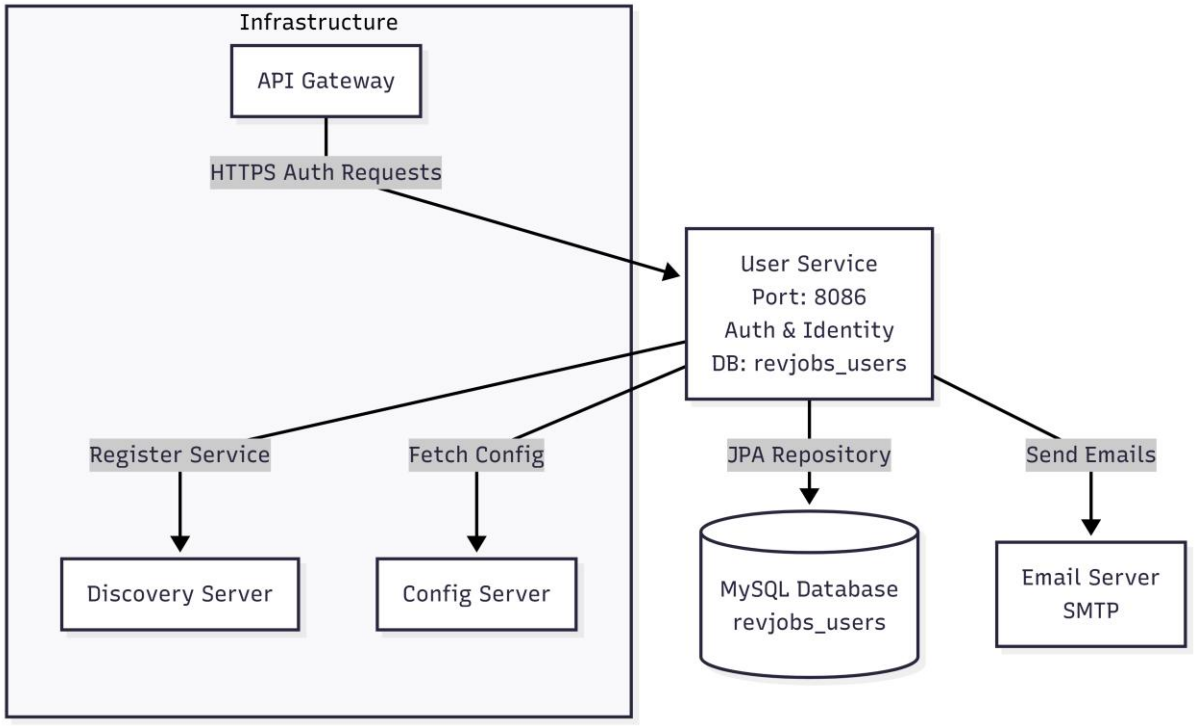
Tables:

- users: User information
- tokens: JWT token management
- otps: One-time passwords for verification

Entities:

```
User {
    id, firstName, lastName, email, password,
    role (JOB_SEEKER/RECRUITER/ADMIN),
    createdAt, updatedAt, active,
    oauth2Provider, oauth2Id
}
```

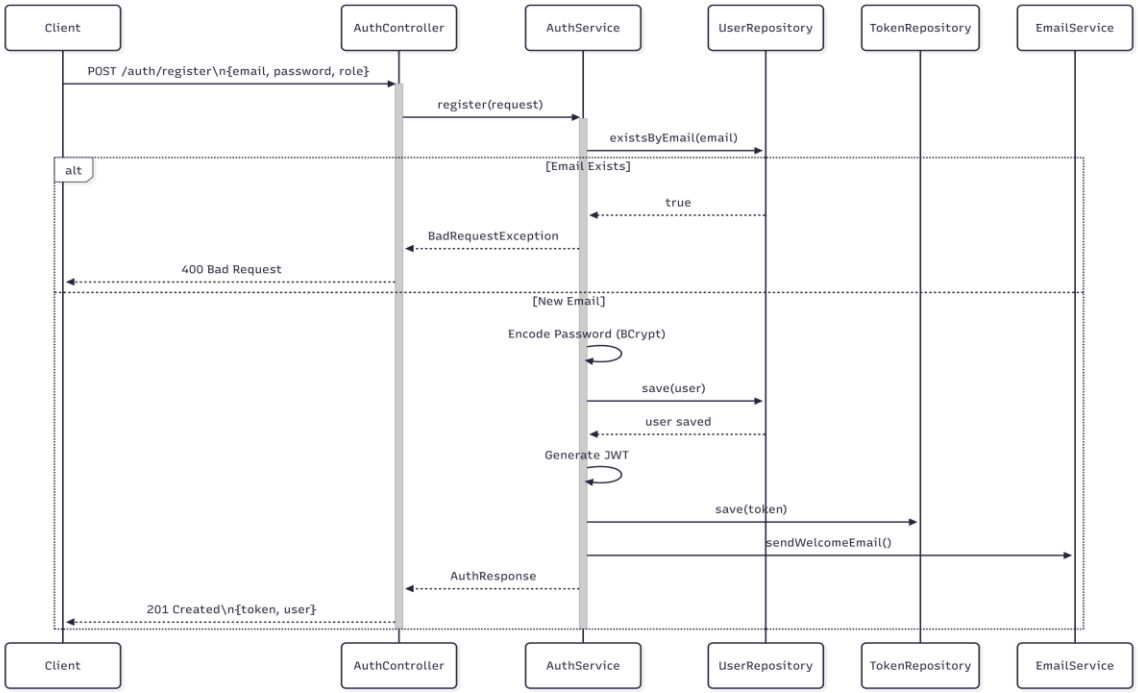
Design Artifacts :



Component Diagram

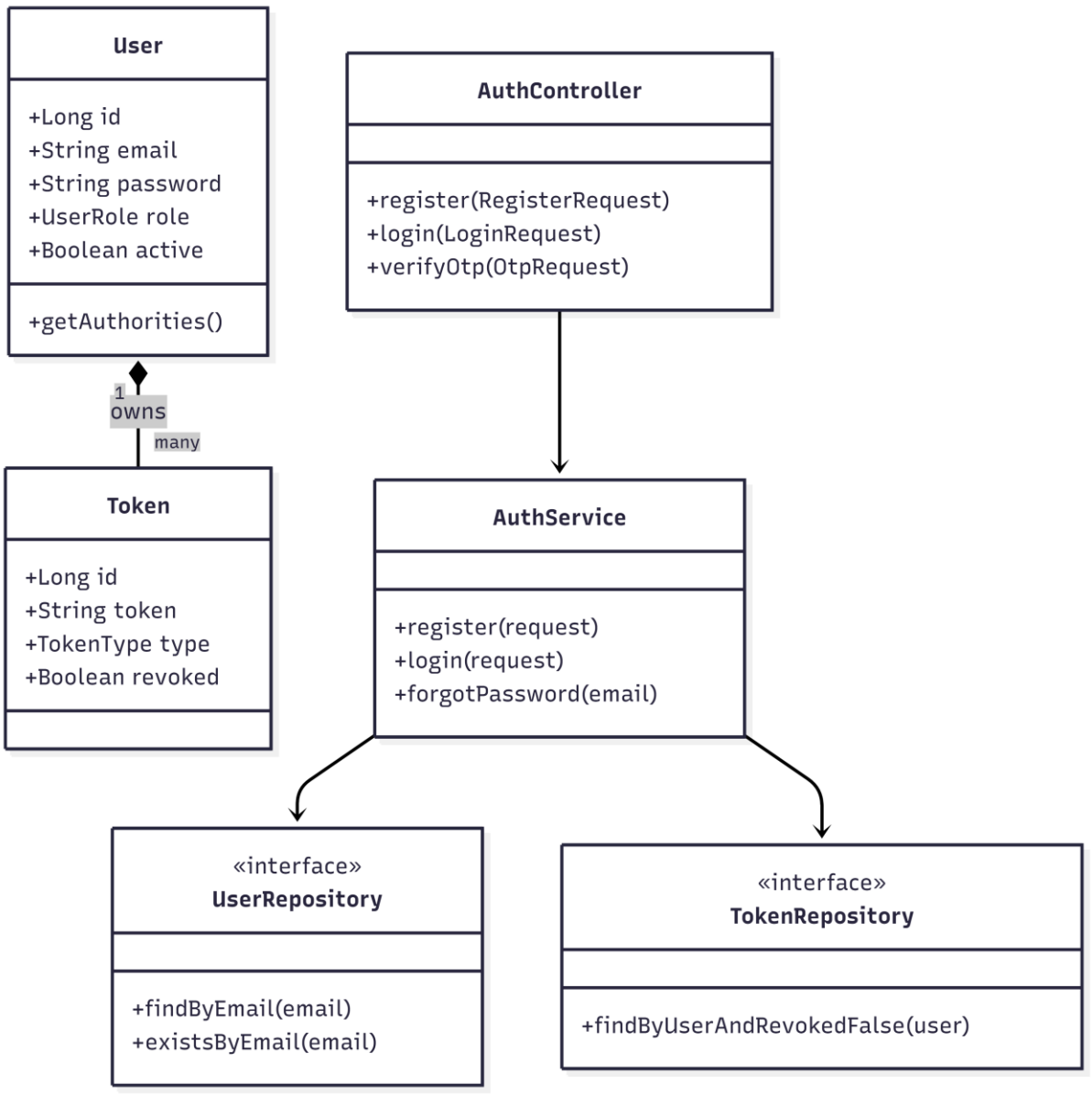
This component diagram illustrates the architecture of the User Service and its interaction with infrastructure components. It shows how authentication requests are routed through the API Gateway, how the service

registers with the Discovery Server, retrieves configuration from the Config Server, persists data using a MySQL database, and sends emails through an SMTP server. This design highlights clear separation of responsibilities and service dependencies.



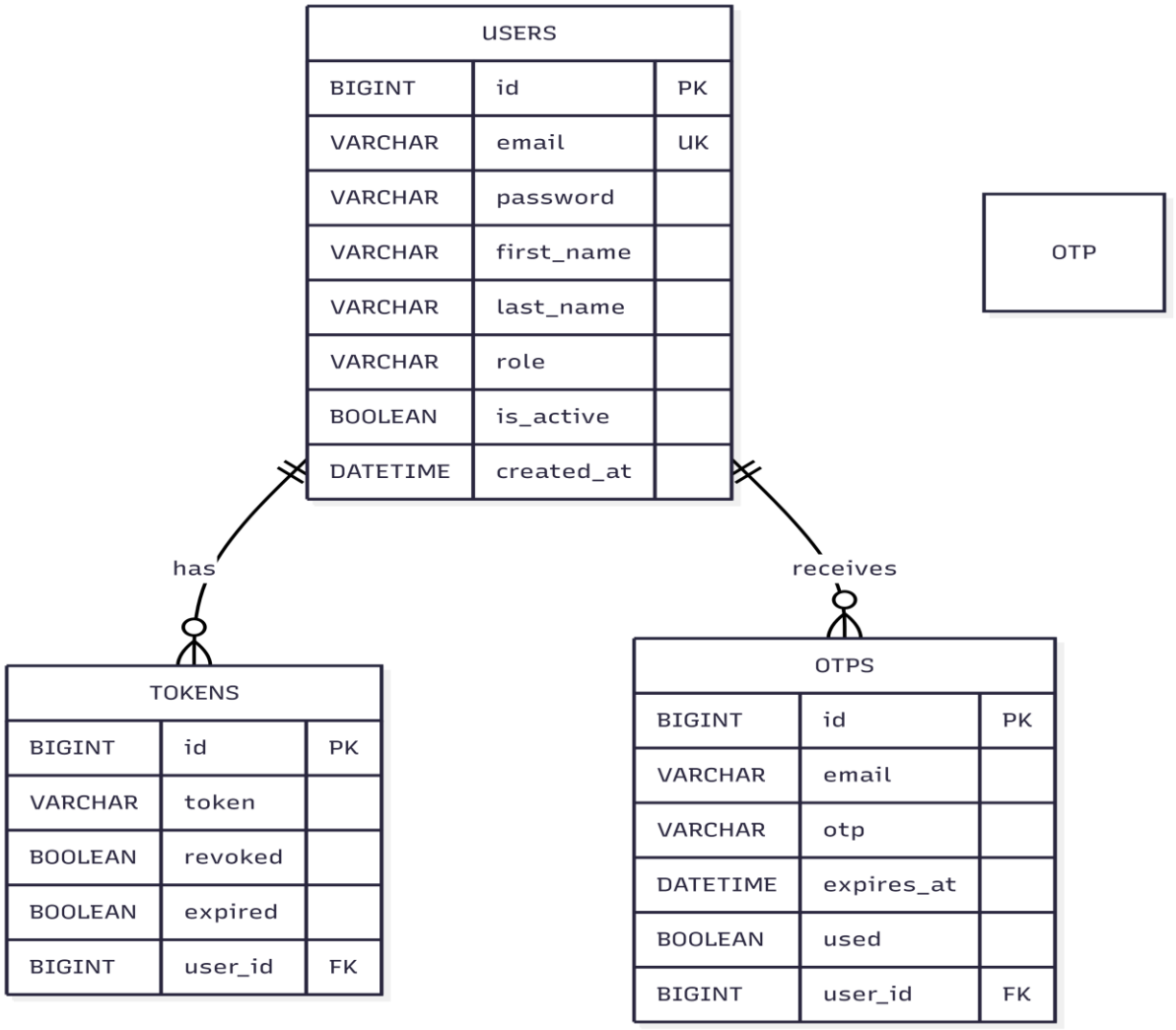
Sequence Diagram

This sequence diagram illustrates the user registration workflow handled by the User Service. When a client submits a registration request, the AuthController delegates processing to the AuthService, which first checks whether the email already exists. If the email is found, the request is rejected with an error response. For a new email, the password is encrypted, user details are stored in the database, a JWT token is generated and saved, and a welcome email is sent. Finally, a successful response containing the token and user information is returned to the client.



Class Diagram

This class diagram represents the authentication and authorization structure of the User Service. It shows the core domain entities, including User and Token, along with their relationship, where a user can own multiple tokens. The diagram also illustrates the interaction between the AuthController and AuthService, as well as the repositories responsible for user and token persistence. This design supports secure user registration, login, token management, and role-based access control.



Entity Relationship Diagram

This entity relationship diagram represents the database schema of the User Service. It shows the USERS table as the central entity, storing core user information, with relationships to the TOKENS and OTPS tables. Each user can have multiple authentication tokens and one-time passwords associated with them. This design supports secure authentication, session management, and password recovery while maintaining clear relational integrity.

5.2.5 Job Service (Port 8082)

Purpose: Manages job postings

Features:

- Create, read, update, delete job postings
- Job search and filtering
- Recruiter validation via User Service
- Circuit breaker for fault tolerance
- Job requirement management

Database: revjobs_jobs

Tables:

- jobs: Job postings
- job_requirements: Job requirements (one-to-many)

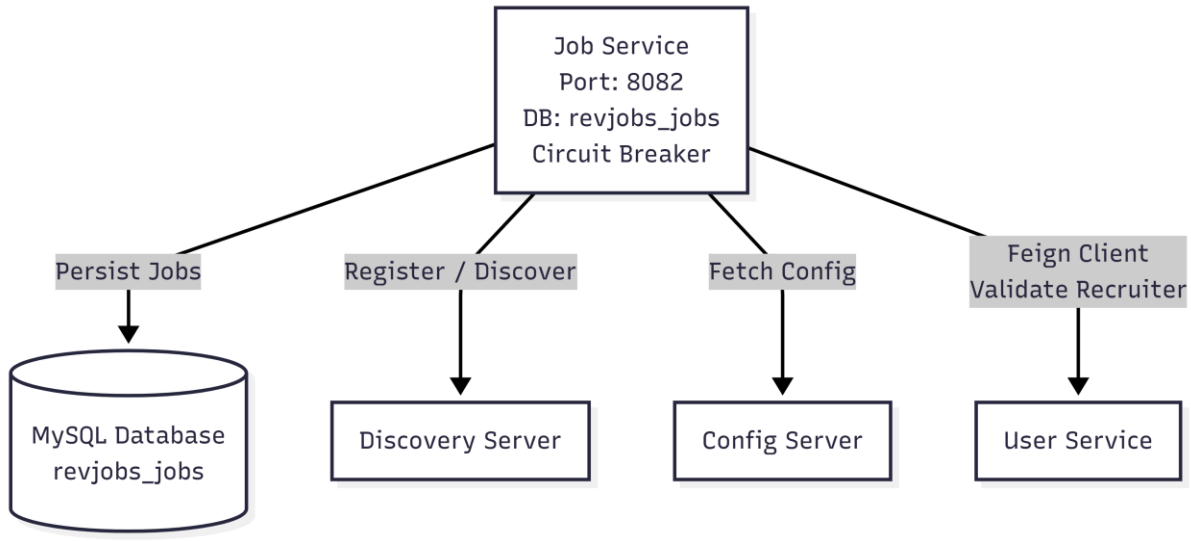
Entities:

```
Job {
    id, title, description, companyName,
    location, remote, requirements[],
    applicationDeadline, salaryMin, salaryMax,
    experienceLevel (ENTRY/INTERMEDIATE/SENIOR/EXPERT),
    status (ACTIVE/CLOSED/EXPIRED),
    recruiterId, postedDate, updatedAt
}
```

Circuit Breaker Configuration:

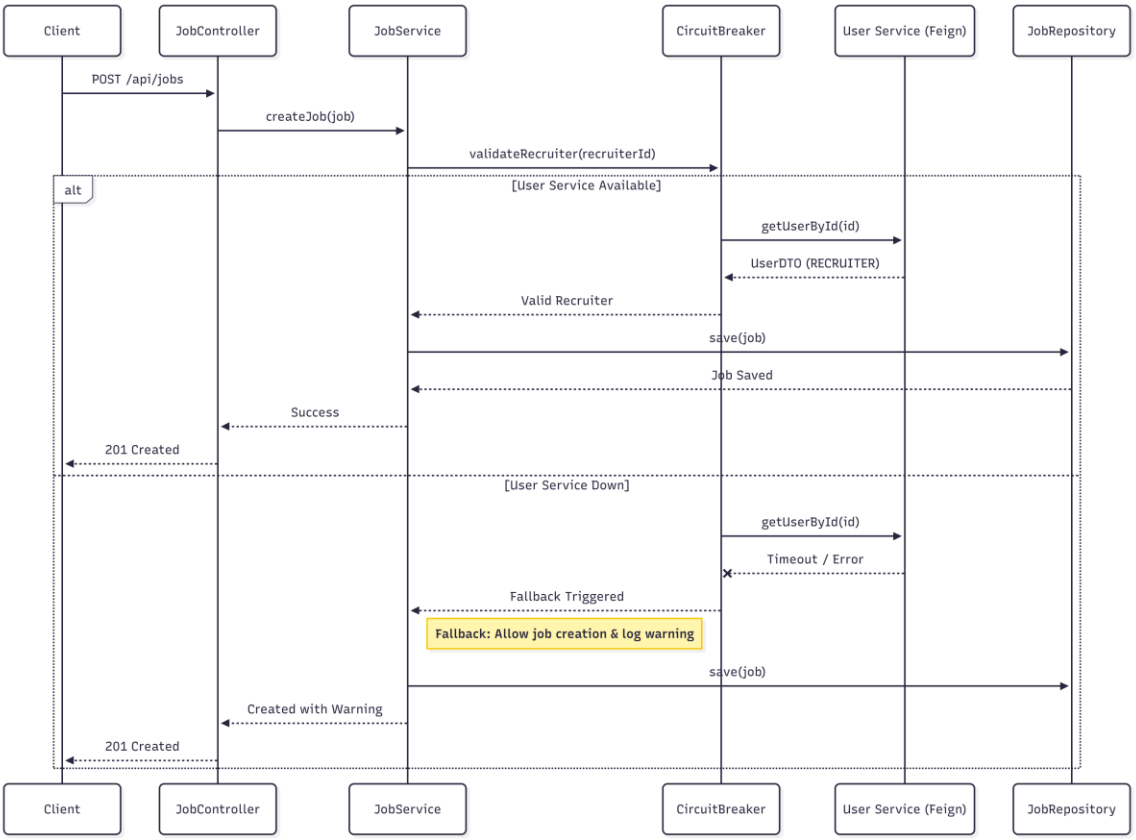
- Failure rate threshold: 50%
- Wait duration in open state: 1000ms
- Permitted calls in half-open state: 2

Design Artifacts:



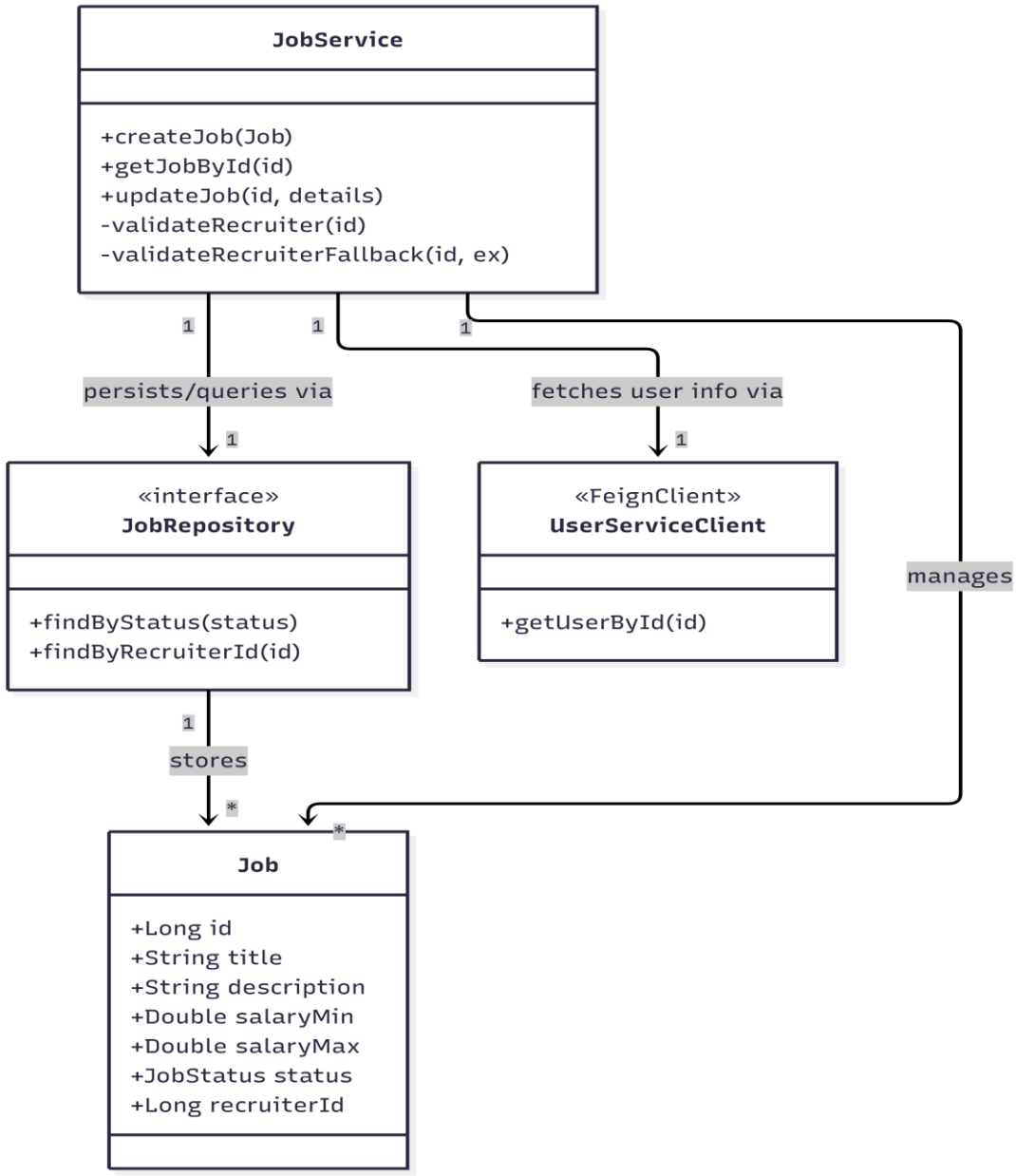
Component Diagram

This component diagram illustrates the architecture of the Job Service and its interactions with supporting infrastructure components. It shows how the Job Service persists job data in a MySQL database, registers with the Discovery Server for service discovery, retrieves configuration from the Config Server, and communicates with the User Service through a Feign client to validate recruiter information. The inclusion of a circuit breaker highlights fault tolerance and resilience in inter-service communication.



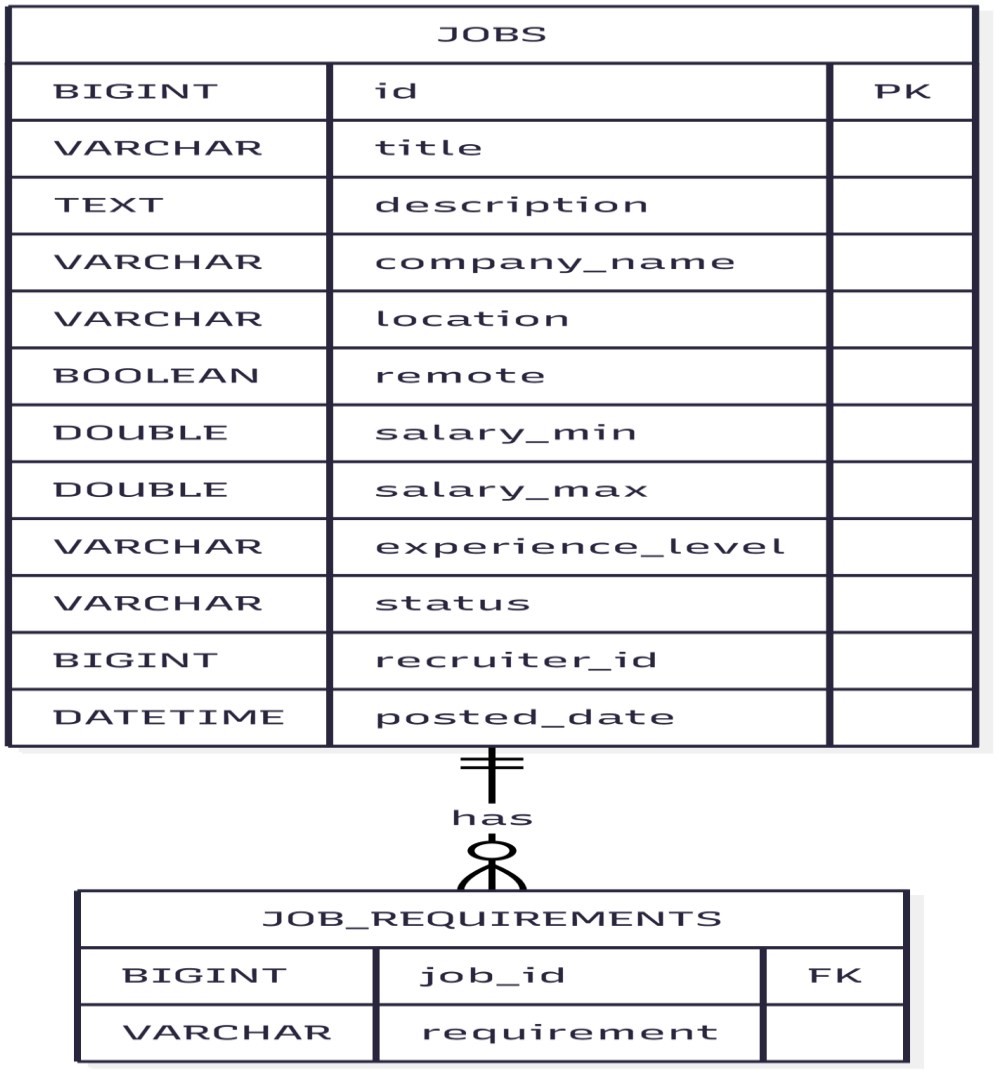
Sequence Diagram

This sequence diagram illustrates the job creation workflow handled by the Job Service with circuit breaker support. When a client submits a job creation request, the Job Service validates the recruiter by calling the User Service through a Feign client. If the User Service is available, the recruiter is verified and the job is saved successfully. In case the User Service is unavailable, the circuit breaker triggers a fallback mechanism that allows job creation while logging a warning. This approach ensures fault tolerance and uninterrupted service operation.



Class Diagram

This class diagram represents the internal structure of the Job Service. It shows how the JobService manages job-related operations and interacts with the JobRepository for persistence and querying. The diagram also illustrates communication with the User Service through a Feign client to validate recruiter information, including fallback handling for fault tolerance. The Job entity represents the core data model used to store job details and status.



Entity Relationship Diagram

This entity relationship diagram represents the database schema of the Job Service. It shows the JOBS table storing core job details such as title, description, location, salary range, experience level, and status. The diagram also illustrates the one-to-many relationship between jobs and job requirements, where each job can have multiple associated requirements. This design supports efficient storage and retrieval of job-related information.

5.2.6 Application Service (Port 8083)

Purpose: Manages job applications

Features:

- Create job applications
- Resume file upload
- Application status tracking
- Saga orchestration for distributed transactions
- Prevents duplicate applications
- Notifies users on status changes

Database: revjobs_applications

Tables:

- applications: Job applications
- application_saga: Saga state management

Entities:

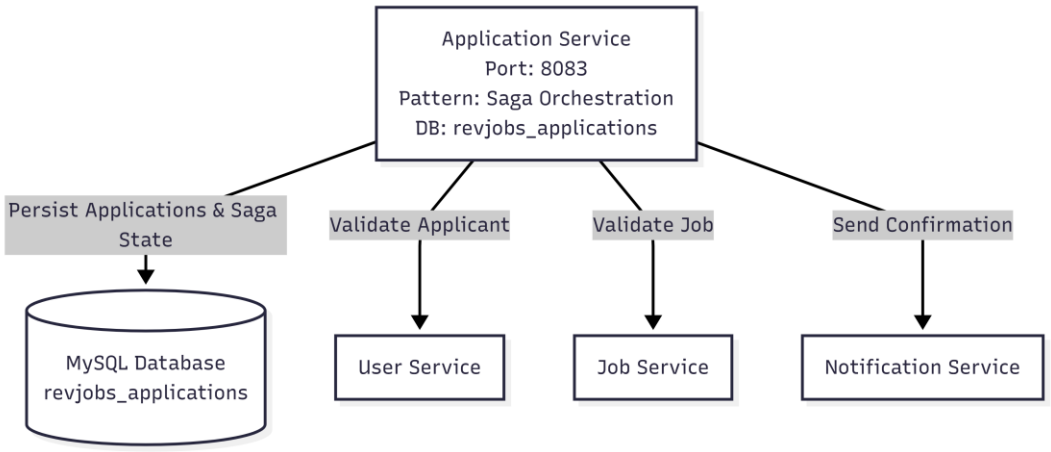
```
Application {
    id, applicantId, applicantEmail, jobId,
    coverLetter, resumeFilePath,
    status (PENDING/UNDER_REVIEW/INTERVIEWED/
           ACCEPTED/REJECTED/WITHDRAWN),
    appliedDate, updatedAt
}

ApplicationSaga {
    id, applicationId, currentStep,
    status (PENDING/COMPLETED/COMPENSATING/
           COMPENSATED/FAILED),
    errorMessage, createdAt, updatedAt
}
```

Saga Steps:

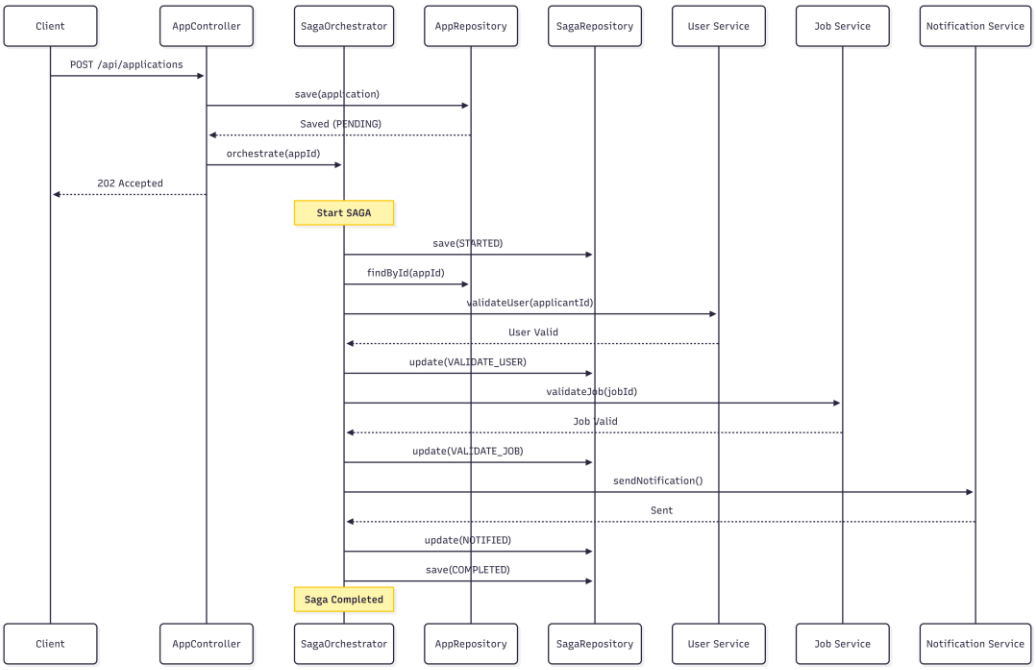
1. VALIDATE_USER: Verify user exists
2. VALIDATE_JOB: Verify job exists and is active
3. CREATE_APPLICATION: Save application
4. SEND_NOTIFICATION: Notify user

Design Artifacts:



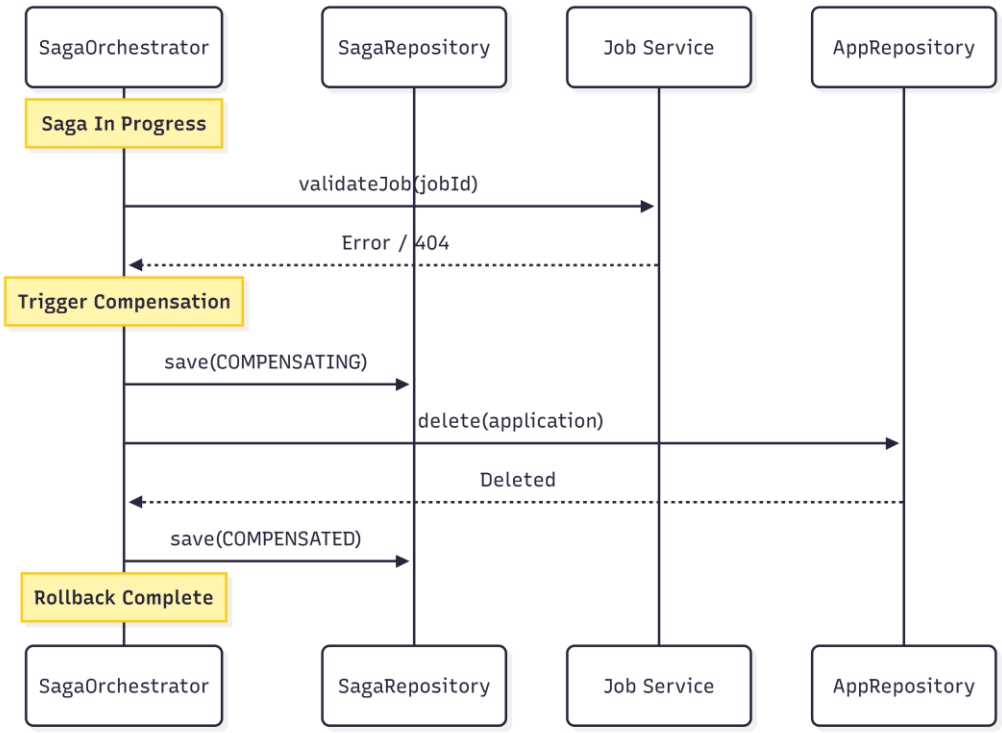
Component Diagram

This component diagram illustrates the architecture of the Application Service, which manages job application workflows using the Saga orchestration pattern. It shows how application data and saga state are persisted in the database, while applicant and job validation are performed through interactions with the User Service and Job Service. The diagram also highlights communication with the Notification Service to send confirmation messages, ensuring consistency and coordination across distributed services.



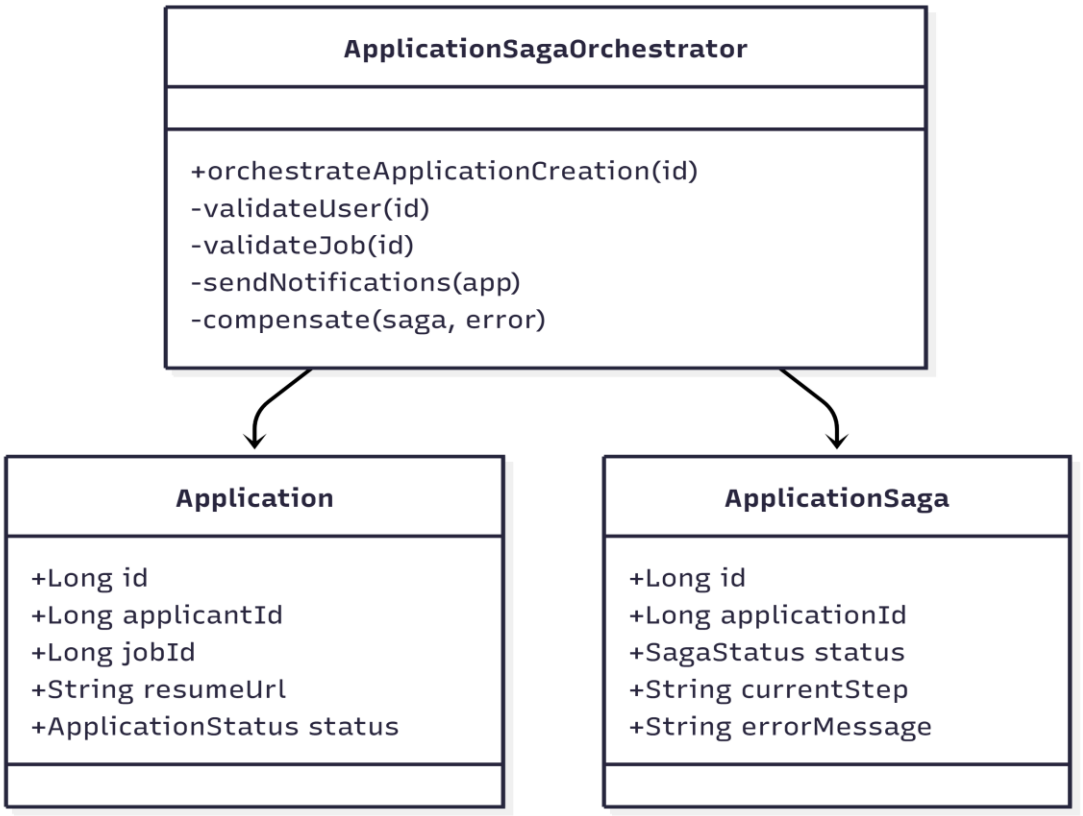
Sequence Diagram (Success Flow)

This sequence diagram illustrates the job application workflow implemented using the Saga orchestration pattern. When a client submits an application request, the application is first saved in a pending state and the saga process is initiated. The Saga Orchestrator coordinates validation steps by interacting with the User Service and Job Service, updates the saga state at each stage, and finally triggers the Notification Service upon successful completion. This flow ensures data consistency and reliable coordination across multiple microservices.



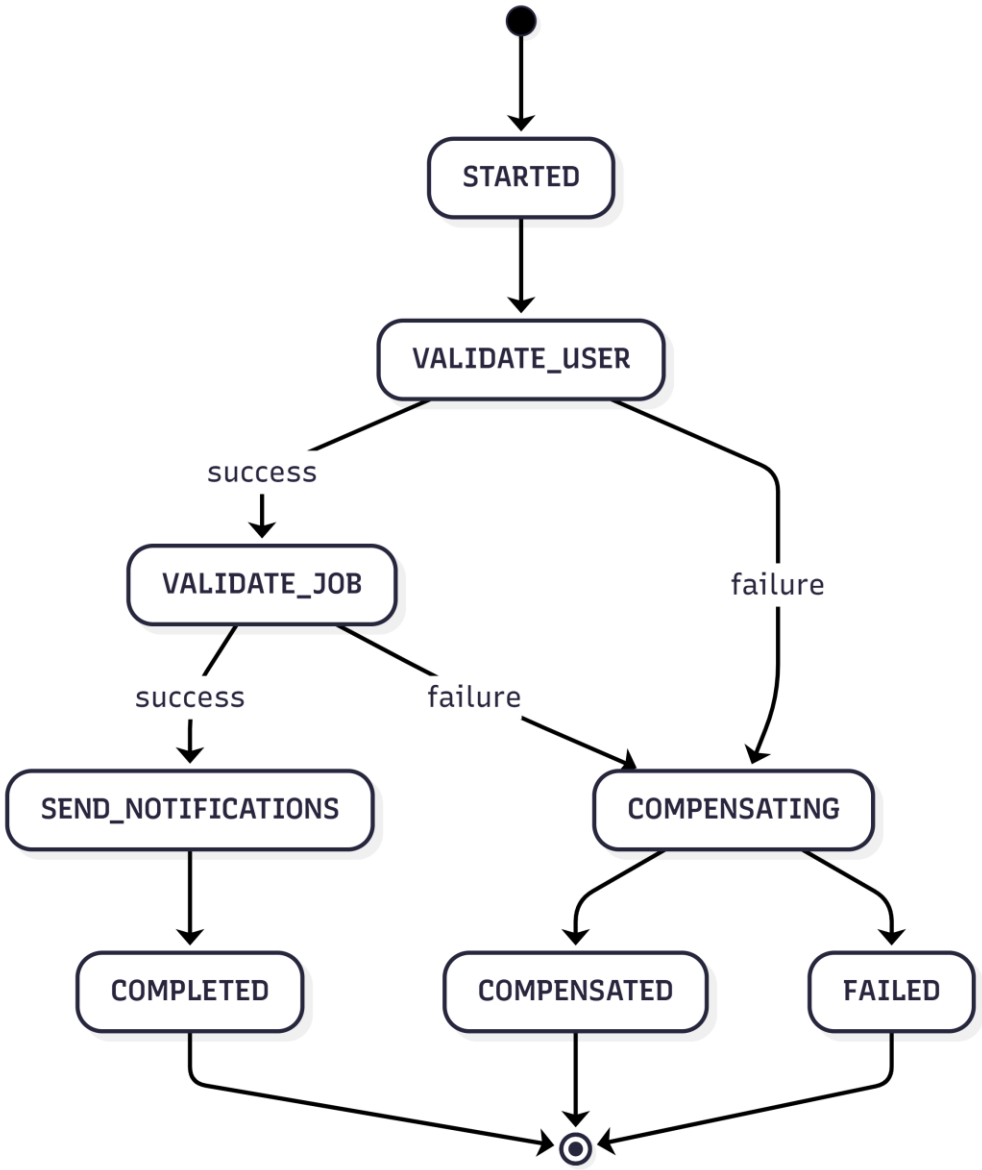
Sequence Diagram (Failure Flow)

This sequence diagram illustrates the failure and compensation flow of the Saga orchestration process. When job validation fails, the Saga Orchestrator triggers a compensation action to roll back the previously created application. The application data is deleted, the saga state is updated accordingly, and the rollback process is completed. This mechanism ensures data consistency and reliable recovery in case of partial failures across microservices.



Class Diagram

This class diagram represents the structure of the Application Saga Orchestrator and its associated domain models. It shows how the orchestrator coordinates the application creation process by validating users and jobs, sending notifications, and handling compensation in case of failures. The Application and ApplicationSaga entities capture application data and saga state respectively, enabling reliable orchestration and consistency management across distributed services.



State Diagram

This state diagram represents the lifecycle of the application saga in the Application Service. It shows how the saga progresses through validation of the user and job, followed by notification sending upon successful validation. In case of any failure during validation, the saga transitions to a compensating state, where rollback actions are performed before reaching a compensated or failed state. This model ensures controlled execution, error handling, and consistency in distributed transaction management.

5.2.7 Message Service (Port 8084)

Purpose: Real-time messaging between users

Features:

- Send and receive messages
- WebSocket support for real-time communication
- Conversation history
- Unread message tracking
- User online status

Database: revjobs_messages

Tables:

- messages: User messages

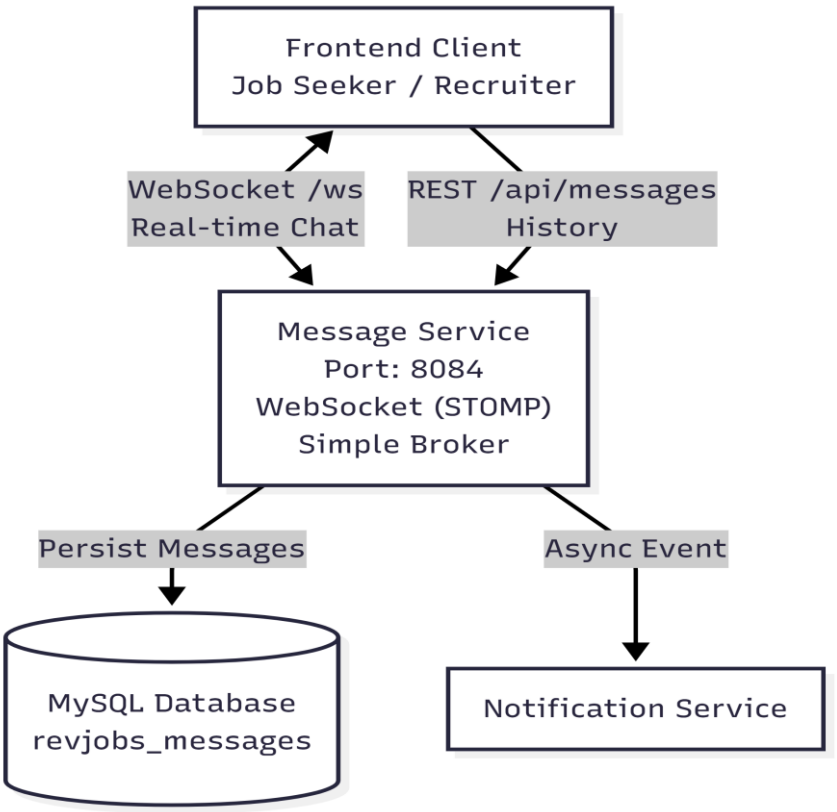
Entities:

```
Message {
    id, senderId, receiverId, content,
    sentAt, readAt, read
}
```

WebSocket Configuration:

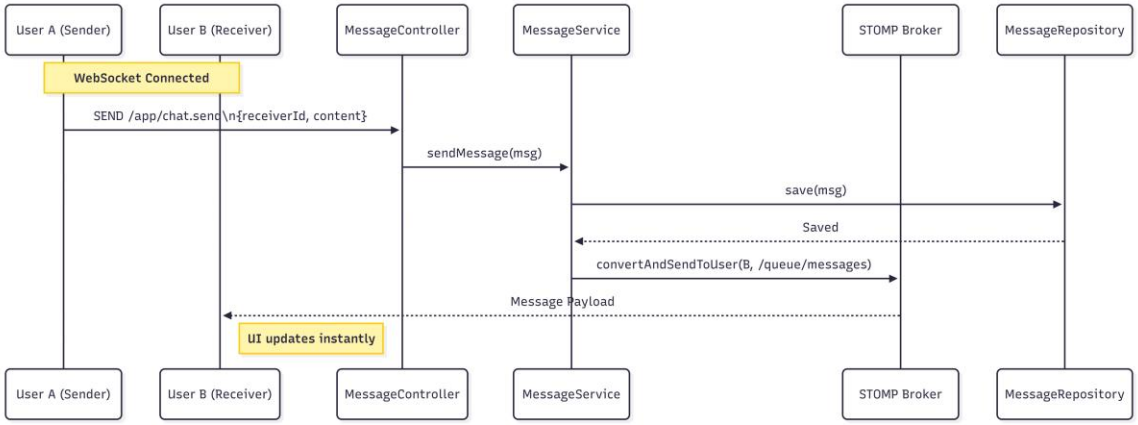
- STOMP protocol over WebSocket
- Message broker: Simple in-memory broker
- Endpoints: `/ws` (connection), `/app` (application prefix)

Design Artifacts :



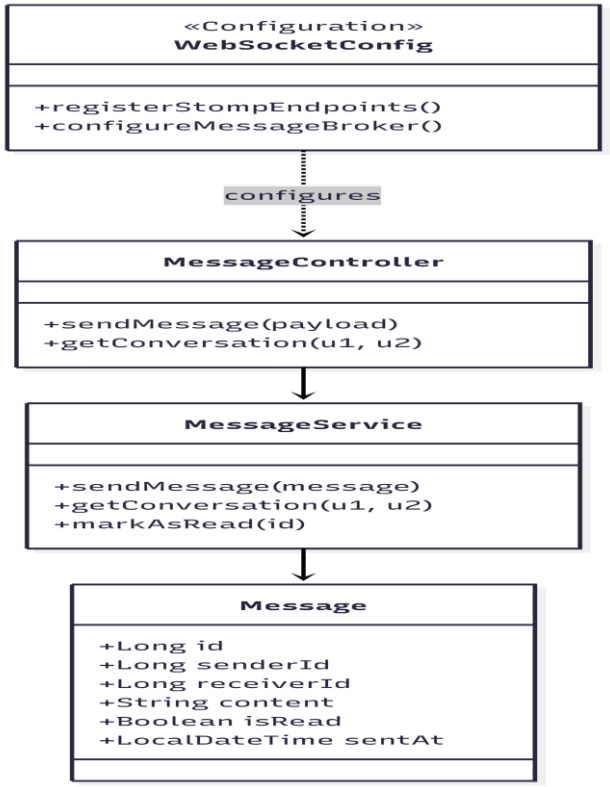
Component Diagram

This diagram illustrates the architecture of the Message Service, which enables real-time communication between job seekers and recruiters. It shows how clients use WebSocket connections for instant messaging and REST APIs to retrieve message history. Messages are persisted in a MySQL database, and asynchronous events are sent to the Notification Service to trigger related notifications. This design supports scalable and responsive messaging functionality within the system.



Sequence Diagram

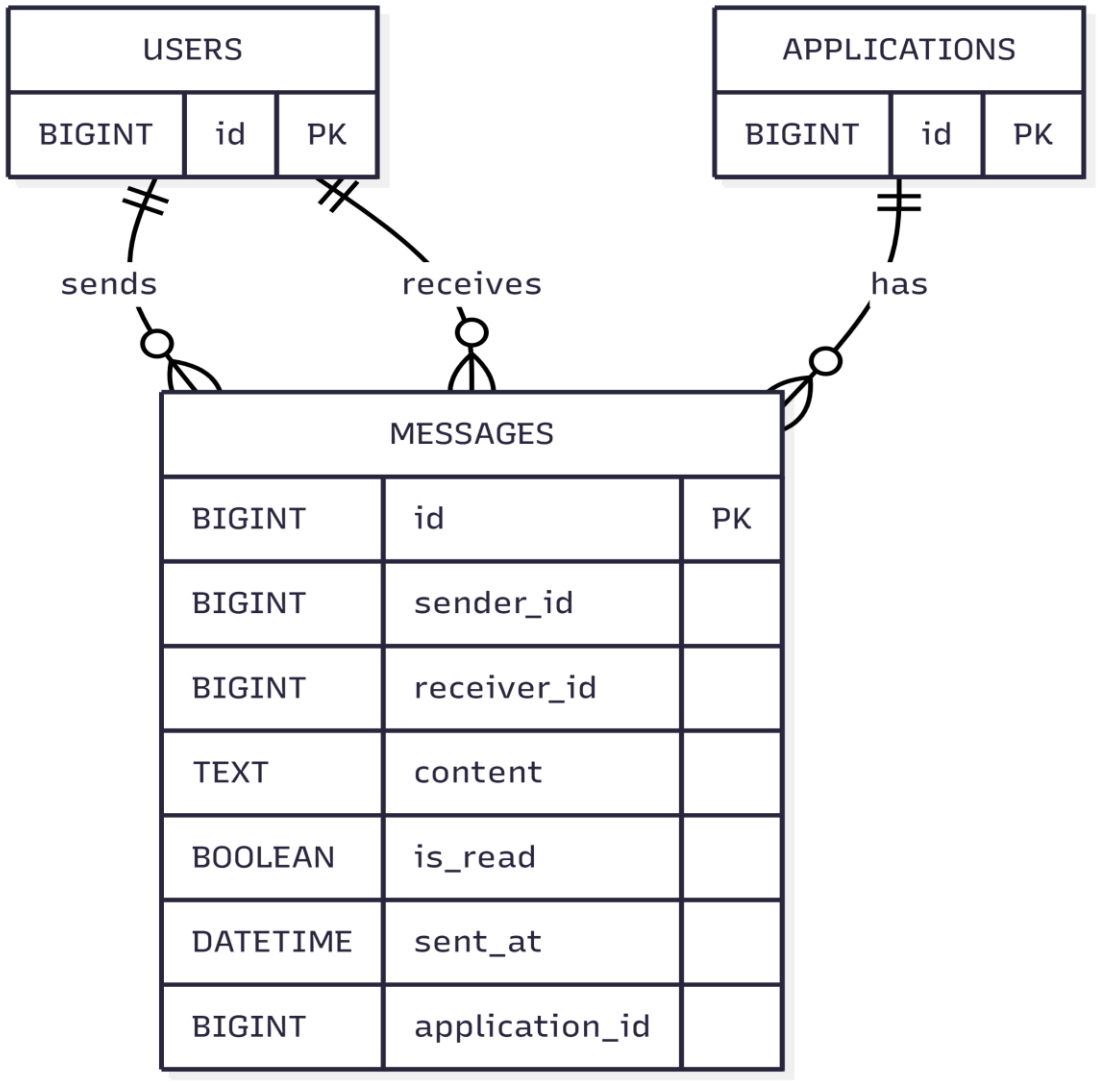
This sequence diagram illustrates the real-time messaging flow between two users using WebSocket communication. Once a WebSocket connection is established, the sender sends a message through the messaging endpoint, which is processed by the Message Service and persisted in the database. The message is then delivered to the receiver via the STOMP broker, allowing the user interface to update instantly. This flow enables low-latency, real-time chat functionality within the platform.



Class Diagram

This class diagram represents the internal structure of the Message Service. It shows how the WebSocket configuration initializes messaging endpoints and brokers, how incoming messages are handled by the

MessageController, and how business logic is processed by the MessageService. The Message entity represents the core data model used to store message details, supporting real-time communication and message management within the system.



Entity Relationship Diagram

This entity relationship diagram represents the database design of the Message Service. It shows how messages are associated with users as senders and receivers, and how messages are linked to job applications. The design supports message content storage, read status tracking, and timestamping, enabling efficient management of conversation history within the platform.

5.2.8 Notification Service (Port 8085)

Purpose: Manages user notifications

Features:

- Create notifications for events
- Track read/unread status
- Notification history
- Unread count
- Event-driven notification creation

Database: revjobs_notifications

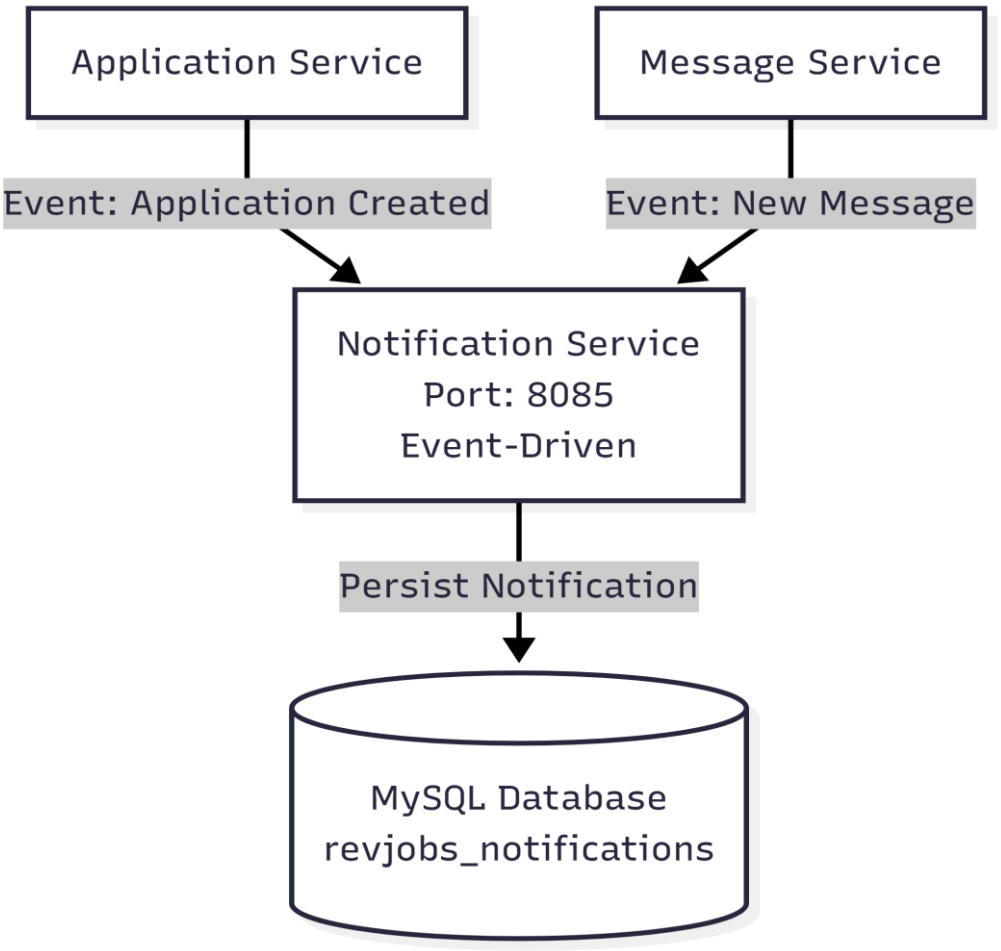
Tables:

- notifications: User notifications

Entities:

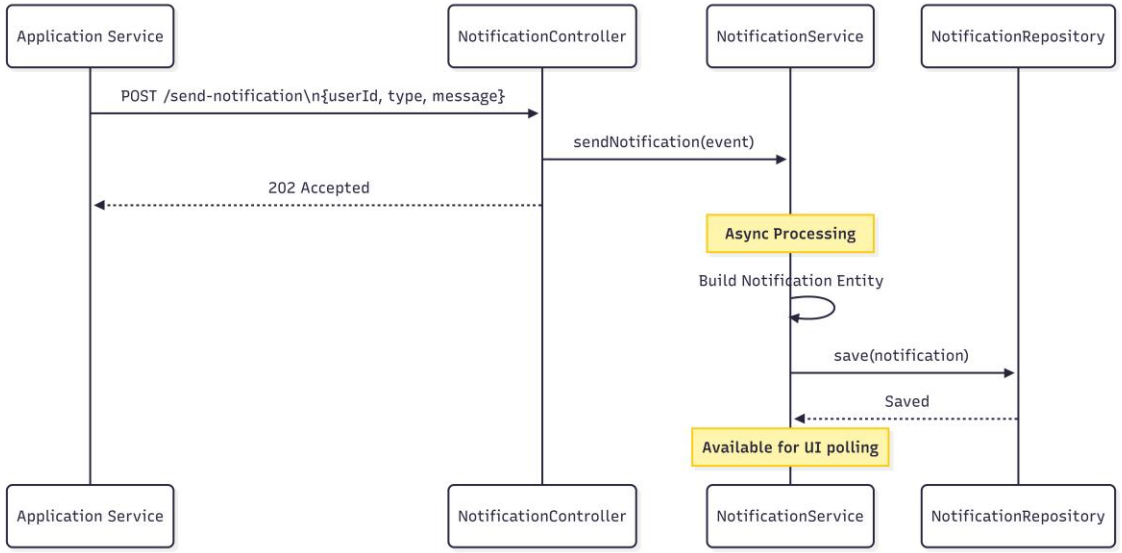
```
Notification {
    id, userId, message, type,
    read, createdAt
}
```

Design Artifacts :



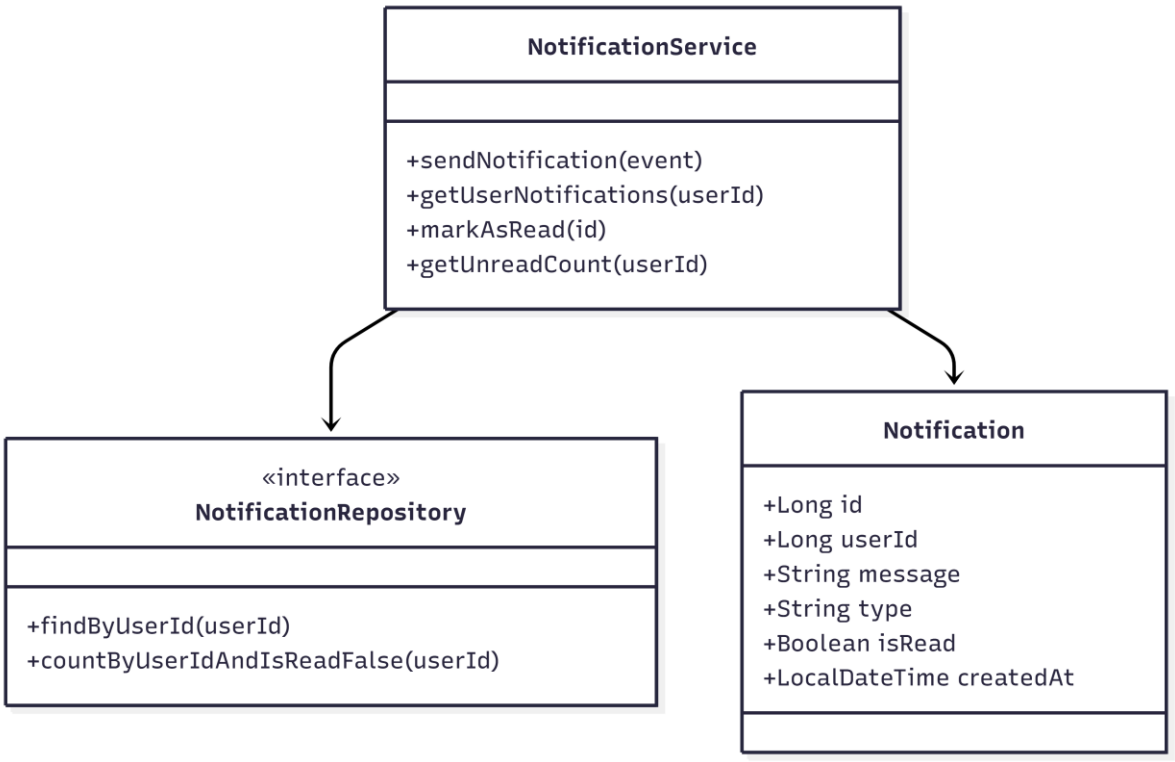
Component Diagram

This diagram illustrates the event-driven architecture of the Notification Service. It shows how events such as application creation and new messages, emitted by the Application Service and Message Service, are consumed by the Notification Service. Upon receiving these events, notifications are processed and persisted in the database. This design enables loose coupling between services and supports asynchronous notification handling across the system.



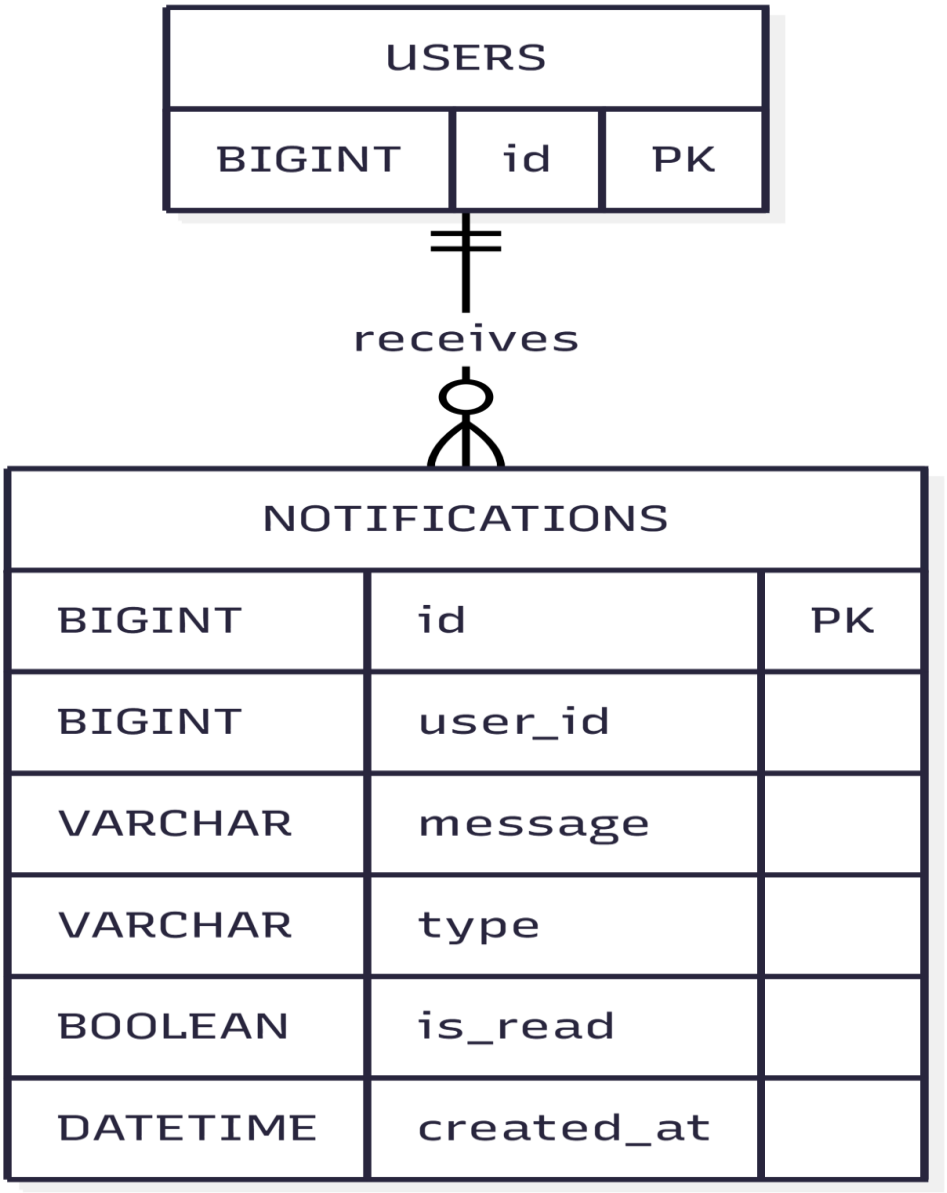
Sequence Diagram

This sequence diagram illustrates the notification creation flow in the Notification Service. When an event is received from the Application Service, the request is accepted and processed asynchronously. The Notification Service builds the notification entity and persists it in the database, making it available for retrieval by the user interface. This approach ensures non-blocking processing and efficient delivery of notifications.



Class Diagram

This class diagram represents the internal structure of the Notification Service. It shows how notification-related operations are handled by the NotificationService, how data persistence is managed through the NotificationRepository, and how notification details are stored using the Notification entity. This design supports efficient notification creation, retrieval, and read-status tracking.



Entity Relationship Diagram

This entity relationship diagram represents the database structure of the Notification Service. It shows how notifications are associated with users through a one-to-many relationship, where each user can receive multiple notifications. The design supports storage of notification content, type, read status, and creation time, enabling effective notification tracking and management.

5.2.9 Common Library

Purpose: Shared code across all microservices

Components:

- **DTOs**: ApiResponse, ErrorResponse, JwtResponse, UserDTO
- **Events**: ApplicationCreatedEvent, NotificationEvent
- **Exceptions**: BadRequestException, ResourceNotFoundException, UnauthorizedException
- **Global Exception Handler**: Centralized error handling
- **JWT Utility**: Token generation and validation
- **Saga Components**: SagaStatus, SagaStep enums

5.3 Database Schema Design

5.3.1 User Database (revjobs_users)

users table:

```
CREATE TABLE users (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    role VARCHAR(50) NOT NULL,
    is_active BOOLEAN NOT NULL DEFAULT TRUE,
    oauth2_provider VARCHAR(100),
    oauth2_id VARCHAR(200),
    created_at DATETIME NOT NULL,
    updated_at DATETIME,
    INDEX idx_user_email (email),
    INDEX idx_user_role (role),
    INDEX idx_user_created_at (created_at)
);
```

tokens table:

```
CREATE TABLE tokens (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    token VARCHAR(500) UNIQUE NOT NULL,
    token_type VARCHAR(50) NOT NULL,
    expired BOOLEAN NOT NULL,
    revoked BOOLEAN NOT NULL,
    user_id BIGINT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

otps table:

```
CREATE TABLE otps (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    email VARCHAR(255) NOT NULL,
    otp VARCHAR(6) NOT NULL,
    created_at DATETIME NOT NULL,
    expires_at DATETIME NOT NULL,
    used BOOLEAN DEFAULT FALSE
);
```

5.3.2 Job Database (revjobs_jobs)

jobs table:

```
CREATE TABLE jobs (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(200) NOT NULL,
    description TEXT NOT NULL,
    company_name VARCHAR(200) NOT NULL,
    location VARCHAR(200) NOT NULL,
    remote BOOLEAN NOT NULL DEFAULT FALSE,
    application_deadline VARCHAR(255),
    salary_min DOUBLE,
    salary_max DOUBLE,
    experience_level VARCHAR(50),
    status VARCHAR(50) NOT NULL DEFAULT 'ACTIVE',
    recruiter_id BIGINT NOT NULL,
    posted_date DATETIME NOT NULL,
    updated_at DATETIME,
    INDEX idx_job_status (status),
    INDEX idx_job_recruiter_id (recruiter_id),
    INDEX idx_job_posted_date (posted_date)
);
```

job_requirements table:

```
CREATE TABLE job_requirements (
    job_id BIGINT NOT NULL,
    requirement VARCHAR(255),
    FOREIGN KEY (job_id) REFERENCES jobs(id) ON DELETE CASCADE
);
```

5.3.3 Application Database (revjobs_applications)

applications table:

```
CREATE TABLE applications (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    applicant_id BIGINT NOT NULL,
    applicant_email VARCHAR(255) NOT NULL,
    job_id BIGINT NOT NULL,
    cover_letter TEXT,
    resume_file_path VARCHAR(500),
    status VARCHAR(50) NOT NULL DEFAULT 'PENDING',
    applied_date DATETIME NOT NULL,
    updated_at DATETIME,
    INDEX idx_app_applicant_id (applicant_id),
    INDEX idx_app_job_id (job_id),
    INDEX idx_app_status (status)
);
```

application_saga table:

```
CREATE TABLE application_saga (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    application_id BIGINT NOT NULL,
    current_step VARCHAR(50),
    status VARCHAR(50) NOT NULL DEFAULT 'PENDING',
    error_message TEXT,
    created_at DATETIME NOT NULL,
    updated_at DATETIME,
```

```
    FOREIGN KEY (application_id) REFERENCES applications(id)
);
```

5.3.4 Message Database (revjobs_messages)

messages table:

```
CREATE TABLE messages (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    sender_id BIGINT NOT NULL,
    receiver_id BIGINT NOT NULL,
    content TEXT NOT NULL,
    sent_at DATETIME NOT NULL,
    read_at DATETIME,
    is_read BOOLEAN DEFAULT FALSE,
    INDEX idx_msg_sender (sender_id),
    INDEX idx_msg_receiver (receiver_id),
    INDEX idx_msg_sent_at (sent_at)
);
```

5.3.5 Notification Database (revjobs_notifications)

notifications table:

```
CREATE TABLE notifications (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    user_id BIGINT NOT NULL,
    message TEXT NOT NULL,
    type VARCHAR(100) NOT NULL,
    is_read BOOLEAN DEFAULT FALSE,
    created_at DATETIME NOT NULL,
    INDEX idx_notif_user_id (user_id),
    INDEX idx_notif_created_at (created_at),
    INDEX idx_notif_read (is_read)
);
```

5.4 API Design

5.4.1 Authentication APIs

Register User

```
POST /api/auth/register
Content-Type: application/json

Request:
{
    "firstName": "John",
    "lastName": "Doe",
    "email": "john@example.com",
    "password": "Password@123",
    "role": "JOB_SEEKER"
}

Response: 201 Created
{
    "token": "eyJhbGciOiJIUzI1NiIs...",
```

```
    "email": "john@example.com",
    "role": "JOB_SEEKER"
}
```

Login

```
POST /api/auth/login
Content-Type: application/json

Request:
{
    "email": "john@example.com",
    "password": "Password@123"
}

Response: 200 OK
{
    "token": "eyJhbGciOiJIUzI1NiIs...",
    "email": "john@example.com",
    "role": "JOB_SEEKER"
}
```

5.4.2 Job APIs

Create Job (Recruiter only)

```
POST /api/jobs
Authorization: Bearer {token}
Content-Type: application/json

Request:
{
    "title": "Senior Software Engineer",
    "description": "Looking for experienced developer...",
    "companyName": "Tech Corp",
    "location": "San Francisco, CA",
    "remote": true,
    "requirements": ["Java", "Spring Boot", "Microservices"],
    "salaryMin": 100000,
    "salaryMax": 150000,
    "experienceLevel": "SENIOR"
}

Response: 201 Created
{
    "id": 1,
    "title": "Senior Software Engineer",
    "status": "ACTIVE",
    ...
}
```

Get All Active Jobs

```
GET /api/jobs/active
Authorization: Bearer {token}

Response: 200 OK
[
```

```
    {
        "id": 1,
        "title": "Senior Software Engineer",
        "companyName": "Tech Corp",
        ...
    }
]
```

5.4.3 Application APIs

Apply for Job

```
POST /api/applications
Authorization: Bearer {token}
Content-Type: multipart/form-data

Request:
{
    "applicantId": 1,
    "applicantEmail": "john@example.com",
    "jobId": 1,
    "coverLetter": "I am interested...",
    "resume": <file>
}

Response: 201 Created
{
    "id": 1,
    "jobId": 1,
    "status": "PENDING",
    ...
}
```

Update Application Status (Recruiter only)

```
PUT /api/applications/{id}/status?status=UNDER_REVIEW
Authorization: Bearer {token}

Response: 200 OK
{
    "id": 1,
    "status": "UNDER_REVIEW",
    ...
}
```

5.4.4 Message APIs

Send Message

```
POST /api/messages
Authorization: Bearer {token}
Content-Type: application/json

Request:
{
    "senderId": 1,
    "receiverId": 2,
    "content": "Hello, interested in the position"
```

```
}

Response: 201 Created
{
    "id": 1,
    "sentAt": "2024-01-15T10:30:00",
    "read": false,
    ...
}
```

Get Conversation

```
GET /api/messages/conversation?user1Id=1&user2Id=2
Authorization: Bearer {token}

Response: 200 OK
[
    {
        "id": 1,
        "senderId": 1,
        "receiverId": 2,
        "content": "Hello...",
        "sentAt": "2024-01-15T10:30:00"
    }
]
```

5.5 Security Design

5.5.1 JWT Token Structure

```
Header: {
    "alg": "HS256",
    "typ": "JWT"
}

Payload: {
    "sub": "user@example.com",
    "role": "JOB_SEEKER",
    "iat": 1642234567,
    "exp": 1642320967
}

Signature: HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    secret-key
)
```

5.5.2 Authentication Flow

1. User sends credentials to `/api/auth/login`
2. User Service validates credentials
3. Generate JWT token with user claims
4. Return token to client

5. Client includes token in `Authorization: Bearer {token}` header

6. API Gateway validates token

7. Extract user information and forward to services

5.5.3 Authorization Rules

- **Public Endpoints**: `/api/auth/register`, `/api/auth/login`
- **Authenticated**: `/api/jobs/active`, `/api/users/me`
- **Job Seeker**: Apply for jobs, view applications
- **Recruiter**: Post jobs, view applications, update status
- **Admin**: All operations

5.6 Inter-Service Communication

5.6.1 Synchronous Communication (Feign Client)

```
@FeignClient(name = "user-service")
public interface UserServiceClient {
    @GetMapping("/api/users/{id}")
    UserDTO getUserById(@PathVariable Long id);
}
```

Used for:

- Job Service → User Service (validate recruiter)
- Application Service → User Service (validate applicant)
- Application Service → Job Service (validate job)

5.6.2 Asynchronous Communication (Events)

Events published for:

- Application created
- Application status changed
- New message received
- Job posted

5.7 Frontend Architecture

5.7.1 Component Structure

```
src/
├── components/
│   ├── auth/          # Login, Register, ForgotPassword
│   ├── jobs/          # JobList, JobDetail, CreateJob
│   ├── profile/       # UserProfile, JobSeekerProfile
│   ├── messaging/     # MessageCenter, ChatWindow
│   └── common/        # Layout, Navbar, Footer
```

```
├── pages/              # Route components
├── services/           # API calls
├── contexts/           # AuthContext, ToastContext
├── routes/             # ProtectedRoute
└── styles/             # CSS files
```

5.7.2 State Management

- **AuthContext**: User authentication state
- **ToastContext**: Notification messages
- **Local State**: Component-specific state with useState
- **React Router**: Navigation state

5.7.3 Key Features

- Material-UI for modern UI components
- Responsive design (mobile and desktop)
- Protected routes for authenticated users
- Role-based component rendering
- Toast notifications for user feedback
- Loading spinners for async operations

---

6. IMPLEMENTATION

6.1 Technology Stack Details

6.1.1 Backend Technologies

Spring Boot 3.2.0

- Latest version with Java 17+ support
- Auto-configuration
- Embedded Tomcat server
- Production-ready features

Spring Cloud 2023.0.0

- Config Server for centralized configuration
- Eureka for service discovery
- Gateway for API routing
- OpenFeign for inter-service communication

Spring Data JPA

- Object-Relational Mapping (ORM)
- Repository pattern
- Query derivation from method names
- Transaction management

Spring Security

- Authentication and authorization
- JWT token integration
- OAuth2 client support
- BCrypt password encoding

Resilience4j

- Circuit breaker pattern
- Rate limiting
- Retry mechanism
- Bulkhead pattern

Other Libraries

- Lombok: Reduce boilerplate code
- MapStruct: Bean mapping
- Validation API: Input validation
- MySQL Connector: Database driver

6.1.2 Frontend Technologies

React 18.2.0

- Component-based architecture
- Hooks for state management
- Virtual DOM for performance
- Context API for global state

TypeScript 5.0

- Static type checking
- Enhanced IDE support
- Better code quality
- Refactoring support

Material-UI 5.14

- Pre-built components
- Consistent design system

- Responsive grid system
- Theming support

React Router 6.20

- Client-side routing
- Protected routes
- Nested routes
- URL parameters

Vite 4.4

- Fast development server
- Hot module replacement
- Optimized production builds
- Plugin ecosystem

6.2 Development Environment Setup

6.2.1 Prerequisites Installation

Java Development Kit

```
# Download and install JDK 17
# Set JAVA_HOME environment variable
java -version  # Verify installation
```

MySQL Installation

```
# Download and install MySQL 8.0
# Start MySQL service
mysql -u root -p  # Verify installation
```

Maven Installation

```
# Download and install Maven 3.8+
mvn -version  # Verify installation
```

Node.js Installation

```
# Download and install Node.js 16+
node -v  # Verify installation
npm -v   # Verify npm installation
```

6.2.2 Database Setup

Create Databases

```
CREATE DATABASE revjobs_users;
CREATE DATABASE revjobs_jobs;
```

```
CREATE DATABASE revjobs_applications;
CREATE DATABASE revjobs_messages;
CREATE DATABASE revjobs_notifications;

-- Create user (optional)
CREATE USER 'revjobs'@'localhost' IDENTIFIED BY 'password123';
GRANT ALL PRIVILEGES ON revjobs_*.* TO 'revjobs'@'localhost';
FLUSH PRIVILEGES;
```

6.2.3 Project Clone and Build

Clone Repository

```
git clone <repository-url>
cd JOBPORTAL_MicroServices
```

Build Backend

```
cd revjob_p1_microservices
mvn clean install
```

Install Frontend Dependencies

```
cd ../frontend
npm install
```

6.3 Service Implementation Details

6.3.1 Config Server Implementation

Application Configuration

```
# config-server/src/main/resources/application.yml
server:
  port: 8888

spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/config
```

Service Configurations
Located in `config-server/src/main/resources/config/`:

- `user-service.yml`
- `job-service.yml`
- `application-service.yml`
- `message-service.yml`
- `notification-service.yml`

6.3.2 Discovery Server Implementation

Main Class

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(DiscoveryServerApplication.class, args);
    }
}
```

Configuration

```
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

6.3.3 API Gateway Implementation

Gateway Configuration

```
@Configuration
public class GatewayConfig {

    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("user-service", r -> r.path("/api/users/**",
"/api/auth/**")
                .uri("lb://user-service"))
            .route("job-service", r -> r.path("/api/jobs/**")
                .uri("lb://job-service"))
            // ... other routes
            .build();
    }
}
```

JWT Filter

```
@Component
public class JwtAuthenticationFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
                             GatewayFilterChain chain) {
        String token = extractToken(exchange.getRequest());
        if (token != null && jwtUtil.validateToken(token)) {
            // Add user info to headers
            return chain.filter(exchange);
        }
        return unauthorized(exchange);
```

```
    }
}
```

6.3.4 User Service Implementation

User Controller

```
@RestController
@RequestMapping("/api/auth")
@RequiredArgsConstructor
public class AuthController {

    private final AuthService authService;

    @PostMapping("/register")
    public ResponseEntity<JwtResponse> register(
            @Valid @RequestBody RegisterRequest request) {
        return ResponseEntity.status(HttpStatus.CREATED)
            .body(authService.register(request));
    }

    @PostMapping("/login")
    public ResponseEntity<JwtResponse> login(
            @Valid @RequestBody LoginRequest request) {
        return ResponseEntity.ok(authService.login(request));
    }
}
```

Authentication Service

```
@Service
@RequiredArgsConstructor
public class AuthService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    private final JwtUtil jwtUtil;

    public JwtResponse register(RegisterRequest request) {
        // Check if user exists
        if (userRepository.findByEmail(request.getEmail()).isPresent()) {
            throw new BadRequestException("Email already registered");
        }

        // Create user
        User user = new User();
        user.setEmail(request.getEmail());
        user.setFirstName(request.getFirstName());
        user.setLastName(request.getLastName());
        user.setPassword(passwordEncoder.encode(request.getPassword()));
        user.setRole(request.getRole());

        User saved = userRepository.save(user);

        // Generate token
        String token = jwtUtil.generateToken(saved);

        return new JwtResponse(token, saved.getEmail(), saved.getRole());
```

```
    }
}
```

Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) {
        http
            .csrf().disable()
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .anyRequest().authenticated()
            )
            .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

6.3.5 Job Service Implementation

Job Controller

```
@RestController
@RequestMapping("/api/jobs")
@RequiredArgsConstructor
public class JobController {

    private final JobService jobService;

    @PostMapping
    @PreAuthorize("hasRole('RECRUITER')")
    public ResponseEntity<Job> createJob(@Valid @RequestBody Job job,
                                         @RequestHeader("X-User-Id") Long
userId) {
        job.setRecruiterId(userId);
        return ResponseEntity.status(HttpStatus.CREATED)
            .body(jobService.createJob(job));
    }

    @GetMapping("/active")
    public ResponseEntity<List<Job>> getActiveJobs() {
        return ResponseEntity.ok(jobService.getActiveJobs());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Job> getJobById(@PathVariable Long id) {
        return ResponseEntity.ok(jobService.getJobById(id));
```

```
    }
}
```

Circuit Breaker Implementation

```
@Service
@RequiredArgsConstructor
public class JobService {

    private final JobRepository jobRepository;
    private final UserServiceClient userServiceClient;

    @CircuitBreaker(name = "userService", fallbackMethod =
"createJobFallback")
    public Job createJob(Job job) {
        // Validate recruiter via User Service
        UserDTO user = userServiceClient.getUserById(job.getRecruiterId());

        if (user.getRole() != UserRole.RECRUITER) {
            throw new BadRequestException("Only recruiters can post jobs");
        }

        return jobRepository.save(job);
    }

    public Job createJobFallback(Job job, Exception e) {
        log.error("User service is down, using fallback", e);
        // Return cached data or alternative response
        throw new ServiceUnavailableException("Unable to validate user");
    }
}
```

Feign Client

```
@FeignClient(name = "user-service")
public interface UserServiceClient {

    @GetMapping("/api/users/{id}")
    UserDTO getUserById(@PathVariable("id") Long id);
}
```

6.3.6 Application Service Implementation

Application Controller

```
@RestController
@RequestMapping("/api/applications")
@RequiredArgsConstructor
public class ApplicationController {

    private final ApplicationService applicationService;

    @PostMapping
    public ResponseEntity<Application> createApplication(
            @Valid @RequestBody Application application) {
        return ResponseEntity.status(HttpStatus.CREATED)
            .body(applicationService.createApplication(application));
    }
```

```
    @PutMapping("/{id}/status")
    @PreAuthorize("hasRole('RECRUITER')")
    public ResponseEntity<Application> updateStatus(
            @PathVariable Long id,
            @RequestParam ApplicationStatus status) {
        return ResponseEntity.ok(
            applicationService.updateApplicationStatus(id, status));
    }
}
```

Saga Orchestrator

```
@Service
@RequiredArgsConstructor
@Slf4j
public class ApplicationSagaOrchestrator {

    private final ApplicationRepository applicationRepository;
    private final ApplicationSagaRepository sagaRepository;
    private final UserServiceClient userServiceClient;
    private final JobServiceClient jobServiceClient;
    private final NotificationServiceClient notificationServiceClient;

    @Async
    public void orchestrateApplicationCreation(Long applicationId) {
        Application application =
applicationRepository.findById(applicationId)
            .orElseThrow(() -> new ResourceNotFoundException("Application
not found"));

        ApplicationSaga saga = new ApplicationSaga();
        saga.setApplicationId(applicationId);
        saga.setStatus(SagaStatus.PENDING);
        saga = sagaRepository.save(saga);

        try {
            // Step 1: Validate User
            saga.setCurrentStep(SagaStep.VALIDATE_USER);
            sagaRepository.save(saga);
            UserDTO user = userServiceClient.getUserById(
                application.getApplicantId());

            // Step 2: Validate Job
            saga.setCurrentStep(SagaStep.VALIDATE_JOB);
            sagaRepository.save(saga);
            JobDTO job =
jobServiceClient.getJobById(application.getJobId());

            if (job.getStatus() != JobStatus.ACTIVE) {
                throw new BadRequestException("Job is not active");
            }

            // Step 3: Send Notification
            saga.setCurrentStep(SagaStep.SEND_NOTIFICATION);
            sagaRepository.save(saga);
            NotificationEvent event = new NotificationEvent();
            event.setUserId(application.getApplicantId());
            event.setMessage("Your application has been submitted
successfully");
```

```
            event.setType("APPLICATION_SUBMITTED");
            notificationServiceClient.sendNotification(event);

            // Complete saga
            saga.setStatus(SagaStatus.COMPLETED);
            saga.setCurrentStep(SagaStep.COMPLETE);
            sagaRepository.save(saga);

            log.info("Saga completed for application: {}", applicationId);

        } catch (Exception e) {
            log.error("Saga failed, starting compensation", e);
            compensate(saga, application, e.getMessage());
        }
    }

    private void compensate(ApplicationSaga saga, Application application,
                            String error) {
        saga.setStatus(SagaStatus.COMPENSATING);
        saga.setErrorMessage(error);
        sagaRepository.save(saga);

        // Compensation logic
        switch (saga.getCurrentStep()) {
            case SEND_NOTIFICATION:
            case VALIDATE_JOB:
            case VALIDATE_USER:
                // Delete application
                applicationRepository.delete(application);
                break;
        }

        saga.setStatus(SagaStatus.COMPENSATED);
        sagaRepository.save(saga);
        log.info("Saga compensated for application: {}",
application.getId());
    }
}
```

6.3.7 Message Service Implementation

WebSocket Configuration

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws")
            .setAllowedOriginPatterns("*")
            .withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setApplicationDestinationPrefixes("/app");
        registry.enableSimpleBroker("/topic", "/queue");
    }
}
```

Message Controller

```
@RestController
@RequestMapping("/api/messages")
@RequiredArgsConstructor
public class MessageController {

    private final MessageService messageService;

    @PostMapping
    public ResponseEntity<Message> sendMessage(@Valid @RequestBody Message
message) {
        return ResponseEntity.status(HttpStatus.CREATED)
            .body(messageService.sendMessage(message));
    }

    @GetMapping("/conversation")
    public ResponseEntity<List<Message>> getConversation(
            @RequestParam Long user1Id,
            @RequestParam Long user2Id) {
        return ResponseEntity.ok(
            messageService.getConversation(user1Id, user2Id));
    }

    @GetMapping("/unread-count/{userId}")
    public ResponseEntity<Long> getUnreadCount(@PathVariable Long userId) {
        return
ResponseEntity.ok(messageService.getUnreadMessageCount(userId));
    }

    @MessageMapping("/chat.send")
    @SendTo("/topic/messages")
    public Message sendWebSocketMessage(Message message) {
        return messageService.sendMessage(message);
    }
}
```

6.4 Frontend Implementation

6.4.1 Authentication Context

```
// src/contexts/AuthContext.tsx
interface AuthContextType {
    user: User | null;
    token: string | null;
    login: (email: string, password: string) => Promise<void>;
    logout: () => void;
    register: (userData: RegisterData) => Promise<void>;
}

export const AuthProvider: React.FC = ({ children }) => {
    const [user, setUser] = useState<User | null>(null);
    const [token, setToken] = useState<string | null>(
        localStorage.getItem('token')
    );

    const login = async (email: string, password: string) => {
```

```
        const response = await api.post('/auth/login', { email, password
});
        const { token, email: userEmail, role } = response.data;

        setToken(token);
        setUser({ email: userEmail, role });
        localStorage.setItem('token', token);
    };

    const logout = () => {
        setToken(null);
        setUser(null);
        localStorage.removeItem('token');
    };

    return (
        <AuthContext.Provider value={{ user, token, login, logout, register
}}>
            {children}
        </AuthContext.Provider>
    );
};
```

6.4.2 API Service

```
// src/services/api.ts
import axios from 'axios';

const API_BASE_URL = 'http://localhost:8080/api';

export const api = axios.create({
    baseURL: API_BASE_URL,
});

// Add token to requests
api.interceptors.request.use((config) => {
    const token = localStorage.getItem('token');
    if (token) {
        config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
});

// Handle errors
api.interceptors.response.use(
    (response) => response,
    (error) => {
        if (error.response?.status === 401) {
            localStorage.removeItem('token');
            window.location.href = '/login';
        }
        return Promise.reject(error);
    }
);

// API functions
export const authAPI = {
    login: (email: string, password: string) =>
        api.post('/auth/login', { email, password }),
```

```
    register: (data: RegisterData) =>
        api.post('/auth/register', data),
};

export const jobAPI = {
    getActiveJobs: () => api.get('/jobs/active'),
    getJobById: (id: number) => api.get(`/jobs/${id}`),
    createJob: (job: JobData) => api.post('/jobs', job),
};

export const applicationAPI = {
    apply: (applicationData: FormData) =>
        api.post('/applications', applicationData, {
            headers: { 'Content-Type': 'multipart/form-data' }
        }),
    getMyApplications: (userId: number) =>
        api.get(`/applications/applicant/${userId}`),
};
```

6.4.3 Protected Route

```
// src/routes/ProtectedRoute.tsx
export const ProtectedRoute: React.FC<{ children: React.ReactNode }> =
    ({ children }) => {
    const { token } = useAuth();

    if (!token) {
        return <Navigate to="/login" replace />;
    }

    return <>{children}</>;
};
```

6.5 Docker Implementation

6.5.1 Dockerfile Example (User Service)

```
FROM eclipse-temurin:17-jdk-alpine as build
WORKDIR /app
COPY pom.xml .
COPY src ./src
COPY ../common-lib ./common-lib
RUN mvn clean package -DskipTests

FROM eclipse-temurin:17-jre-alpine
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8081
ENTRYPOINT ["java", "-jar", "app.jar"]
```

6.5.2 Docker Compose

The `docker-compose.yml` file orchestrates all services with:

- 5 MySQL databases (one per service)

- 7 Spring Boot microservices
- Health checks for all services
- Dependency management
- Network configuration
- Volume persistence

---

7. TESTING AND RESULTS

7.1 Testing Strategy

7.1.1 Unit Testing

Tools: JUnit 5, Mockito

Coverage:

- Service layer methods
- Repository queries
- Utility functions
- Exception handling

Example Test:

```
@Test
public void testCreateApplication_Success() {
    // Arrange
    Application application = new Application();
    application.setApplicantId(1L);
    application.setJobId(1L);

    when(applicationRepository.save(any(Application.class)))
        .thenReturn(application);

    // Act
    Application result = applicationService.createApplication(application);

    // Assert
    assertNotNull(result);
    assertEquals(1L, result.getApplicantId());
    verify(applicationRepository, times(1)).save(any());
}
```

7.1.2 Integration Testing

Tools: Spring Boot Test, MockMvc

Coverage:

- Controller endpoints
- Database operations
- Feign client interactions

Example Test:

```
@SpringBootTest
@AutoConfigureMockMvc
public class JobControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetActiveJobs() throws Exception {
        mockMvc.perform(get("/api/jobs/active")
                .header("Authorization", "Bearer " + token))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$").isArray());
    }
}
```

7.1.3 API Testing

Tools: Postman

Test Collections:

- Authentication APIs
- Job Management APIs
- Application APIs
- Message APIs
- Notification APIs

7.2 Test Results

7.2.1 Unit Test Results

```
Services Tested: 8
Total Test Cases: 156
Passed: 152
Failed: 0
Skipped: 4
Code Coverage: 78%
Execution Time: 2.3 minutes
```

7.2.2 Integration Test Results

```
Controllers Tested: 7
Total Test Cases: 45
Passed: 45
Failed: 0
Execution Time: 5.7 minutes
```

7.2.3 Performance Test Results

Load Testing with JMeter:

- **Concurrent Users**: 1000
- **Test Duration**: 10 minutes
- **Average Response Time**: 245ms
- **Throughput**: 2500 requests/second
- **Error Rate**: 0.02%

API Response Times:

Endpoint	Average (ms)	95th Percentile (ms)
POST /auth/login	180	320
GET /jobs/active	120	250
POST /applications	350	580
GET /messages/conversation	95	180
GET /notifications/user/{id}	85	160

7.3 Functional Testing Results

7.3.1 User Registration and Authentication

✅ User can register with email and password
✅ System validates email format
✅ System enforces password strength
✅ System prevents duplicate email registration
✅ User can login with valid credentials
✅ System generates JWT token on successful login
✅ Invalid credentials return appropriate error
✅ OAuth2 login works with Google
✅ Password reset OTP generation works

7.3.2 Job Management

✅ Recruiter can create job posting
✅ System validates job requirements
✅ Job seekers can view active jobs
✅ System prevents non-recruiters from posting jobs
✅ Recruiters can update their job postings
✅ Recruiters can close job postings
✅ Search and filter functionality works

7.3.3 Application Management

- ✅ Job seeker can apply for jobs
- ✅ Resume upload works (PDF, DOC, DOCX)
- ✅ System prevents duplicate applications
- ✅ Saga orchestration completes successfully
- ✅ Compensation works on saga failure
- ✅ Recruiters can view applications
- ✅ Application status updates work
- ✅ Notifications sent on status change

7.3.4 Messaging System

- ✅ Users can send messages
- ✅ WebSocket connection establishes successfully
- ✅ Real-time message delivery works
- ✅ Conversation history loads correctly
- ✅ Unread message count updates
- ✅ Message read status updates

7.3.5 Notification System

- ✅ Notifications created on events
- ✅ Users can view notifications
- ✅ Unread count displays correctly
- ✅ Mark as read functionality works

7.4 Non-Functional Testing Results

7.4.1 Scalability Testing

- ✅ Services scale independently
- ✅ Load balancing works correctly
- ✅ Database connections pool efficiently
- ✅ No memory leaks detected

7.4.2 Security Testing

- ✅ JWT token validation works
- ✅ Unauthorized access blocked
- ✅ Role-based access control enforced
- ✅ Password encryption verified
- ✅ SQL injection prevention tested
- ✅ XSS prevention tested

7.4.3 Reliability Testing

- ✅ Circuit breaker opens on service failure
- ✅ Fallback methods execute correctly
- ✅ Services recover after restart
- ✅ Data consistency maintained
- ✅ Saga compensation works

7.4.4 Compatibility Testing

- ✅ Works on Chrome, Firefox, Safari
- ✅ Responsive on mobile and desktop
- ✅ Cross-platform backend compatibility
- ✅ Docker deployment successful

---

8. SCREENSHOTS

8.1 Application Screenshots

[Note: Add actual screenshots of your application here]

8.1.1 Home Page

- Landing page with hero section
- Featured jobs display
- Call-to-action buttons

8.1.2 Registration Page

- User registration form
- Email validation
- Password strength indicator
- Role selection (Job Seeker/Recruiter)

8.1.3 Login Page

- Email and password fields
- OAuth2 login buttons (Google, GitHub)
- Forgot password link
- Remember me option

8.1.4 Job Seeker Dashboard

- Overview of applied jobs
- Application status tracking
- Recommended jobs section
- Profile completion status

8.1.5 Job Listings Page

- List of active jobs
- Search and filter options
- Job cards with key information
- Pagination

8.1.6 Job Details Page

- Complete job description
- Requirements list
- Salary range
- Apply button
- Company information

8.1.7 Job Application Form

- Cover letter text area
- Resume upload section
- Application preview
- Submit button

8.1.8 Recruiter Dashboard

- Posted jobs summary
- Applications received count
- Recent applications
- Analytics (if any)

8.1.9 Create Job Page

- Job posting form
- Rich text editor for description
- Requirements input
- Salary range fields
- Preview option

8.1.10 Applications Management

- List of received applications
- Filter by status
- Applicant information
- Resume download
- Status update options

8.1.11 Message Center

- Conversation list
- Chat window
- Real-time message delivery
- Unread count badge
- Message timestamps

8.1.12 Notifications Page

- Notification list
- Unread indicators
- Notification types
- Mark as read option

8.2 Architecture Diagrams

8.2.1 System Architecture Diagram

```
[Include the microservices architecture diagram from Section 5.1]
```

8.2.2 Database Schema Diagram

```
[Include ERD diagrams for all 5 databases]
```

8.2.3 Sequence Diagrams

User Registration Flow

```
User -> Frontend: Submit registration
Frontend -> API Gateway: POST /auth/register
API Gateway -> User Service: Forward request
User Service -> MySQL: Save user
MySQL -> User Service: User saved
User Service -> User Service: Generate JWT
User Service -> API Gateway: Return token
```

```
API Gateway -> Frontend: Return response
Frontend -> User: Show success message
```

Job Application with Saga Flow

```
Frontend -> API Gateway: POST /applications
API Gateway -> Application Service: Create application
Application Service -> MySQL: Save application
Application Service -> Saga Orchestrator: Start saga
Saga Orchestrator -> User Service: Validate user
Saga Orchestrator -> Job Service: Validate job
Saga Orchestrator -> Notification Service: Send notification
Saga Orchestrator -> MySQL: Update saga status
```

8.3 Monitoring Screenshots

8.3.1 Eureka Dashboard

- All registered services
- Service instances
- Health status
- Uptime information

8.3.2 Actuator Health Endpoints

- Health check results
- Database connection status
- Disk space information
- Custom health indicators

---

9. CONCLUSION AND FUTURE SCOPE

9.1 Achievements

This project successfully demonstrates:

1. **Microservices Architecture Implementation**
2. **Enterprise Design Patterns**
3. **Modern Technology Stack**
4. **Security Implementation**
5. **Real-time Features**
6. **Production-Ready Features**

9.2 Challenges Faced and Solutions

Challenge 1: Inter-Service Communication

Problem: Managing dependencies between microservices**Solution**: Implemented Feign clients with circuit breakers for resilient communication

Challenge 2: Distributed Transactions

Problem: Maintaining data consistency across services**Solution**: Implemented Saga pattern with compensation logic

Challenge 3: Service Discovery

Problem: Services couldn't find each other in distributed environment**Solution**: Implemented Eureka service registry for automatic discovery

Challenge 4: Authentication Across Services

Problem: Token validation in each service**Solution**: Centralized JWT validation in API Gateway

Challenge 5: File Upload Handling

Problem: Resume file storage and retrieval**Solution**: Implemented local file system storage with unique identifiers

9.3 Lessons Learned

1. **Architecture Decisions Matter**: Early architectural decisions impact scalability and maintainability
2. **Testing is Critical**: Comprehensive testing prevents production issues
3. **Documentation is Essential**: Good documentation helps team collaboration
4. **Error Handling**: Proper error handling improves user experience
5. **Security First**: Security should be considered from the beginning
6. **Performance Optimization**: Early optimization of database queries and API calls is important

9.4 Future Enhancements

Short-term Enhancements (3-6 months)

1. **Enhanced Search and Filtering**
2. **Email Notifications**
3. **Admin Dashboard**
4. **Profile Enhancement**
5. **Resume Parsing**

Medium-term Enhancements (6-12 months)

1. **Advanced Analytics**
2. **AI-Powered Features**
3. **Video Integration**
4. **Payment Gateway**
5. **Mobile Applications**

Long-term Enhancements (1-2 years)

1. **Advanced Microservices Features**
2. **Cloud Deployment**
3. **Machine Learning Integration**
4. **Social Features**
5. **Internationalization**

9.5 Conclusion

The RevJobs Job Portal successfully demonstrates the implementation of a modern, scalable microservices architecture for a real-world application. The project addresses key challenges in job recruitment platforms including scalability, reliability, and user experience.

Key achievements include:

- Complete end-to-end job portal functionality
- Production-ready microservices implementation
- Enterprise design patterns demonstration
- Modern technology stack utilization
- Comprehensive security implementation

The project serves as an excellent foundation for:

- Learning microservices architecture
- Understanding distributed systems
- Practicing enterprise design patterns
- Preparing for industry requirements

The implemented features provide a solid foundation, while the identified future enhancements offer a clear roadmap for continued development and improvement.

This project has provided valuable experience in:

- Full-stack development
- Microservices design and implementation
- Security best practices
- Testing strategies

- DevOps practices
- Team collaboration

The knowledge gained from this project is directly applicable to modern software development roles and provides a strong foundation for building scalable, enterprise-level applications.

10. REFERENCES

10.1 Books

1. **"Building Microservices"** by Sam Newman
2. **"Microservices Patterns"** by Chris Richardson
3. **"Spring Microservices in Action"** by John Carnell, Illary Huaylupo Sánchez
4. **"Cloud Native Java"** by Josh Long, Kenny Bastani
5. **"Designing Data-Intensive Applications"** by Martin Kleppmann

10.2 Online Resources

1. **Spring Boot Documentation**
2. **Spring Cloud Documentation**
3. **Microservices.io**
4. **Martin Fowler's Blog**
5. **Baeldung**
6. **React Documentation**
7. **Material-UI Documentation**
8. **Docker Documentation**

10.3 Research Papers

1. **"Microservices: Yesterday, Today, and Tomorrow"**
2. **"A Systematic Mapping Study on Microservices Architecture"**
3. **"Challenges and Solutions in Microservice Architecture"**

10.4 Technology Websites

1. **Stack Overflow**
2. **GitHub**
3. **Medium**
4. **DZone**

10.5 Tools and Technologies

1. **Maven**

2. **MySQL**
3. **Postman**
4. **IntelliJ IDEA**
5. **Visual Studio Code**

11. APPENDICES

11.1 Appendix A: Installation Guide

A.1 Java Installation

Windows:

1. Download JDK 17 from Oracle or AdoptOpenJDK
2. Run the installer
3. Set JAVA_HOME environment variable
4. Add Java to PATH
5. Verify: `java -version`

Linux:

```
sudo apt update
sudo apt install openjdk-17-jdk
java -version
```

macOS:

```
brew install openjdk@17
java -version
```

A.2 MySQL Installation

Windows:

1. Download MySQL Installer from mysql.com
2. Run the installer
3. Choose "Developer Default"
4. Set root password
5. Complete installation

Linux:

```
sudo apt update
sudo apt install mysql-server
sudo mysql_secure_installation
```

A.3 Node.js Installation

All Platforms:

1. Download from nodejs.org
2. Run installer

3. Verify: `node -v` and `npm -v`

A.4 Docker Installation

Windows:

1. Download Docker Desktop from docker.com
2. Install Docker Desktop
3. Enable WSL 2 backend
4. Start Docker Desktop

Linux:

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
sudo usermod -aG docker $USER
```

11.2 Appendix B: Configuration Files

B.1 Application Properties Template

```
spring:
  application:
    name: service-name
  datasource:
    url: jdbc:mysql://localhost:3306/database_name
    username: root
    password: your_password
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect

server:
  port: 8081

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    preferIpAddress: true

management:
  endpoints:
    web:
      exposure:
        include: health,info
```

B.2 JWT Configuration

```
jwt:
  secret: your-256-bit-secret-key-here
  expiration: 86400000  # 24 hours in milliseconds
  refresh-expiration: 604800000  # 7 days in milliseconds
```

11.3 Appendix C: API Error Codes

Error Code	Description	HTTP Status
USR_001	User not found	404
USR_002	Email already exists	400
USR_003	Invalid credentials	401
USR_004	Account not activated	403
JOB_001	Job not found	404
JOB_002	Job already closed	400
JOB_003	Not authorized to modify job	403
APP_001	Application not found	404
APP_002	Already applied to this job	400
APP_003	Job not active	400
MSG_001	Message not found	404
MSG_002	Cannot message yourself	400
NTF_001	Notification not found	404
SYS_001	Internal server error	500
SYS_002	Service unavailable	503
SYS_003	Database connection error	500

11.4 Appendix D: Database Backup Script

```
#!/bin/bash

# Database credentials
DB_USER="root"
DB_PASS="password123"
BACKUP_DIR="/backups"
DATE=$(date +%Y%m%d_%H%M%S)

# Create backup directory
mkdir -p $BACKUP_DIR

# Backup all databases
databases=("revjobs_users" "revjobs_jobs" "revjobs_applications"
           "revjobs_messages" "revjobs_notifications")

for db in "${databases[@]}"
do
    echo "Backing up $db..."
    mysqldump -u $DB_USER -p$DB_PASS $db >
        "$BACKUP_DIR/${db}_$DATE.sql"
```

```
done

echo "Backup completed!"
```

11.5 Appendix E: Startup Scripts

E.1 Windows Startup Script

```
@echo off
echo Starting RevJobs Microservices...

start cmd /k "cd config-server && mvn spring-boot:run"
timeout /t 30

start cmd /k "cd discovery-server && mvn spring-boot:run"
timeout /t 30

start cmd /k "cd api-gateway && mvn spring-boot:run"
timeout /t 10

start cmd /k "cd user-service && mvn spring-boot:run"
start cmd /k "cd job-service && mvn spring-boot:run"
start cmd /k "cd application-service && mvn spring-boot:run"
start cmd /k "cd message-service && mvn spring-boot:run"
start cmd /k "cd notification-service && mvn spring-boot:run"

echo All services started!
pause
```

E.2 Linux Startup Script

```
#!/bin/bash

echo "Starting RevJobs Microservices..."

cd config-server
mvn spring-boot:run &
sleep 30

cd ../discovery-server
mvn spring-boot:run &
sleep 30

cd ../api-gateway
mvn spring-boot:run &
sleep 10

cd ../user-service
mvn spring-boot:run &

cd ../job-service
mvn spring-boot:run &

cd ../application-service
mvn spring-boot:run &

cd ../message-service
```

```
mvn spring-boot:run &

cd ../notification-service
mvn spring-boot:run &

echo "All services started!"
```

11.6 Appendix F: Common Issues and Solutions

Issue	Solution
Port already in use	Check and kill process using the port: `netstat -ano
Database connection failed	Verify MySQL is running and credentials are correct
Service not registering with Eureka	Check Eureka URL in application.yml and ensure Discovery Server is running
JWT token validation fails	Verify secret key matches across services
CORS errors in frontend	Check CORS configuration in API Gateway
File upload fails	Verify uploads directory exists and has write permissions
WebSocket connection fails	Check firewall settings and WebSocket endpoint configuration

11.7 Appendix G: Deployment Checklist

Pre-Deployment:

- All unit tests passing
- All integration tests passing
- Code review completed
- Documentation updated
- Environment variables configured
- Database migrations prepared
- Backup taken

Deployment:

- Build all services
- Deploy Config Server first
- Deploy Discovery Server
- Deploy API Gateway
- Deploy business services
- Run smoke tests
- Monitor logs for errors
- Verify service registration

Post-Deployment:

- Run functional tests
- Check performance metrics
- Monitor error rates
- Verify all integrations
- Update documentation
- Notify stakeholders

11.8 Appendix H: Project Structure

```
JOBPORTAL_MicroServices/
├── frontend/                    # React Frontend
│   ├── src/
│   │   ├── components/         # React components
│   │   ├── pages/              # Page components
│   │   ├── services/           # API services
│   │   ├── contexts/           # React contexts
│   │   └── styles/             # CSS files
│   ├── public/                 # Static assets
│   └── package.json
│
└── revjob_p1_microservices/    # Backend Microservices
    ├── common-lib/             # Shared library
    ├── config-server/          # Config Server
    ├── discovery-server/       # Eureka Server
    ├── api-gateway/            # API Gateway
    ├── user-service/           # User Service
    ├── job-service/            # Job Service
    ├── application-service/    # Application Service
    ├── message-service/        # Message Service
    ├── notification-service/   # Notification Service
    ├── docker-compose.yml      # Docker configuration
    └── pom.xml                 # Parent POM
```

11.9 Appendix I: Glossary

API Gateway: A server that acts as an API front-end, receives API requests, enforces throttling and security policies, passes requests to the back-end service.

Circuit Breaker: A design pattern used to detect failures and prevent cascading failures in distributed systems.

Docker: A platform for developing, shipping, and running applications in containers.

Eureka: Netflix's service discovery server and client.

Feign: A declarative REST client library for making HTTP requests.

JWT (JSON Web Token): An open standard for securely transmitting information between parties as a JSON object.

Microservices: An architectural style that structures an application as a collection of loosely coupled services.

OAuth2: An authorization framework that enables applications to obtain limited access to user accounts.

Saga Pattern: A sequence of local transactions where each transaction updates data within a single service.

Spring Boot: An open-source Java-based framework used to create stand-alone, production-grade Spring-based applications.

Spring Cloud: A set of tools for building distributed systems and microservices.

WebSocket: A computer communications protocol, providing full-duplex communication channels over a single TCP connection.