Architecture Documentation

DEVision - Job Manager/Applicant Subsystem

Son Tinh Squad

November 2025

Team Members

Dinh Xuan Minh - s3891847

Huynh Ngoc Duy - s3924704

Nguyen Viet Quan - s3926217

Huynh Nhat Anh - s3924763

Tran Minh Truong - s3891643

Table of Contents

1. Introduction

This document outlines the data model and system architecture for the DevVision Job Applicant (JA) subsystem. It provides a structured overview of the key entities, relationships, and architectural components that support applicant management, job interactions, premium features, and system operations. The purpose of this section is to give stakeholders and future maintainers a clear understanding of how data is organised, how it flows through the subsystem, and how it integrates with external services such as the Job Manager (JM) subsystem, cloud storage, payment services, and event-streaming components.

The JA subsystem delivers several core features specified in the SRS, including user registration and authentication, applicant profile creation, education and work experience tracking, job application submission, job list browsing, subscription handling, notification delivery, and media file management for CVs, cover letters, and portfolio items. Both freemium and premium users can create a single search profile that stores job-related preferences such as desired skills, locations, employment types, and salary expectations. Premium subscribers receive additional capabilities, including real-time job-matching notifications triggered by job-related events published from the Job Manager subsystem. This enhancement allows the system to immediately alert premium users when new or updated job posts align with their saved search profile.

The frontend is implemented using a modularised component-based architecture with React and Headless UI, providing consistent interaction patterns and flexible UI composition. The backend adopts a modular monolith architecture using Spring Boot, grouping functionality into coherent modules such as authentication, profile management, job list retrieval via JM APIs, subscription management, search profile configuration, and notification handling. This structure enables the JA subsystem to operate independently while ensuring seamless integration with external systems such as Kafka, Firebase/Cloudinary, Google OAuth, and the payment gateway.

2. Squad Data Model

2.1. Data Model

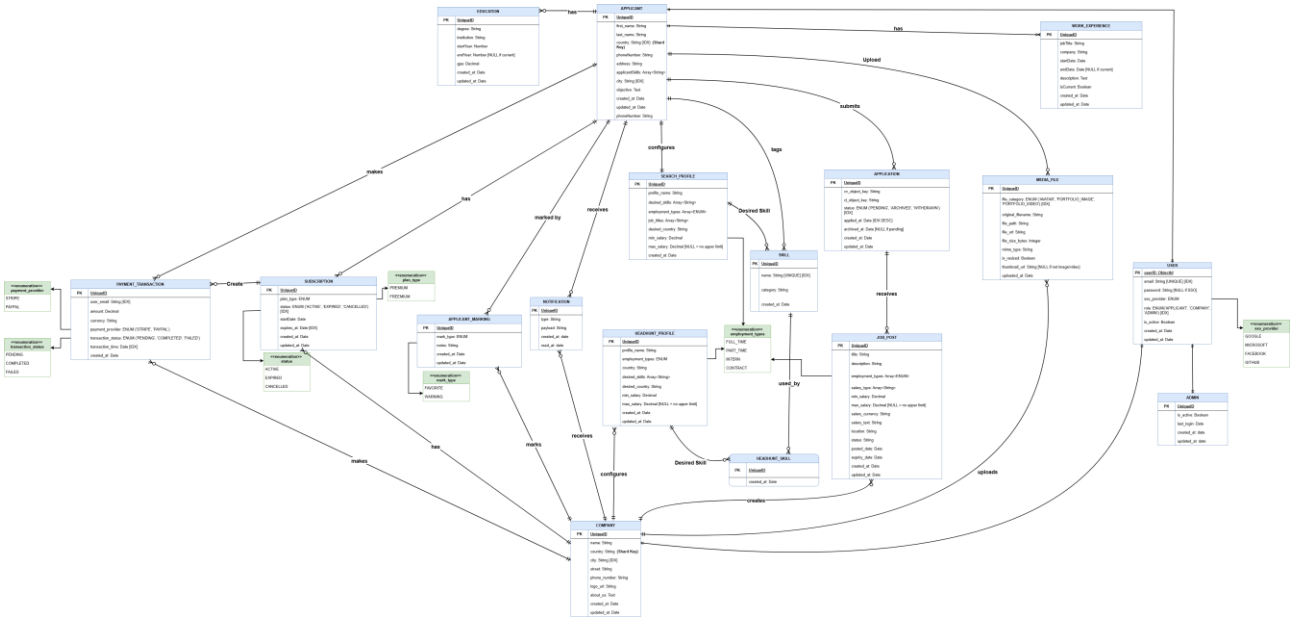

Figure 1: Squad Data Model

Squad Data Model link: [DEVision ERD and C4 - draw.io]

The Job Applicant data model is organised around the main features defined in the SRS rather than individual tables. At its centre is the applicant profile, which stores account and personal details, while supporting tables capture history, preferences, and system interactions.

Applicant registration and authentication

User accounts are stored with basic identity information and are linked to applicant records that hold personal details and high-level profile settings. This separation allows the system to support both local credentials and external SSO providers (e.g., Google) while keeping applicant-specific data independent from authentication data. The JA backend exchanges only minimal identity attributes (name, email, provider ID) with the external auth provider.

Profile, education, experience, and skills

The profile feature is backed by a set of entities for education history, work experience and skills. These structures allow the system to record multiple degrees, jobs and skill tags for each applicant, which are later reused when showing profile details to the user, when generating CV-style views, and when sharing candidate information with the Job Manager (JM) subsystem during application and matching workflows. Only summary attributes (e.g., skill names, job titles, years of experience) need to be sent to JM via REST APIs or events.

Job list browsing and applications

Job-related data in the JA schema is designed primarily as references to job posts owned by the JM subsystem. The JA database stores identifiers and minimal metadata needed to track saved jobs, warnings and applications, while full job details are retrieved from JM over REST. When an applicant applies for a job, the application record ties the applicant to the corresponding external job post and captures status and timestamps. Application details, together with applicant profile data, are then exchanged with JM using the agreed APIs so that JM can handle employer-side processing.

Search profile and premium matching

Applicants can configure a search profile containing job preferences such as desired roles, locations, skills, salary expectations and employment type. These attributes are used by the backend to filter the job list returned from JM and to drive premium matching. For premium subscribers, the same search profile is used as input to the real-time matching pipeline: when JM publishes job post update events to Kafka, the JA backend consumes those events and evaluates them against the stored search profile. Only the job post identifiers and relevant attributes (skills, location, salary range) are needed from JM to perform this matching.

Subscription and payment tracking

Subscription and payment entities record the lifecycle of a premium plan and its payment history. This data enables the system to determine whether an applicant is eligible for premium features such as real-time matching and priority notifications. The payment records also store references to external payment provider transactions, which are exchanged with the payment gateway through REST APIs but not exposed directly to other subsystems.

Notifications and system interactions

Notification data captures alerts sent to applicants, including job-match results, subscription updates and system messages. Each notification stores the type, message content, read status and optional reference to a job post. When a new job matches a premium user's search profile, the matching logic creates a notification entry that is later retrieved by the frontend or pushed through a real-time channel. Only job identifiers and minimal context are required from JM for these notifications.

Media management (CVs, cover letters, portfolios)

The media file structures maintain metadata about uploaded CVs, cover letters, avatars and portfolio files, while the actual binary content is stored in an external cloud storage service. The JA system stores only file URLs and types; these are shared with JM when applications or profile summaries are sent so that employers can access submitted documents.

Overall, the data model focuses on capturing the information necessary to implement the JA features: profile management, job interactions, search preferences, premium subscription, and notifications while keeping job and company details as references to the JM subsystem. This supports clean data ownership boundaries and well-defined data exchange via REST APIs and Kafka events.

2.2.Sharding

The DEVision Job Applicant subsystem implements MongoDB sharding on the APPLICANT collection to satisfy SRS Ultimo requirements 1.3.3 and 4.3.1. This section specifies the shard key, sharded data scope, and backend routing mechanism.

2.2.1. Shard Key: country

The APPLICANT collection uses **country** as the shard key. This field enables MongoDB to partition applicant documents geographically, routing Vietnamese applicants to one shard, Singaporean applicants to another, and so forth.

2.2.2. Sharded Data Scope

Primary Sharded Collection: APPLICANT

The APPLICANT collection is the only collection configured for sharding. This collection contains core user profile data including account information, personal details, and profile metadata.

Related Collections (Not Sharded):

Collections such as EDUCATION, WORK_EXPERIENCE, APPLICATION, SKILL, and NOTIFICATION remain unsharded. These collections reference the APPLICANT collection through applicant_id foreign keys but do not require geographic partitioning as they are queried by applicant_id rather than by country.

2.2.3. Backend Shard Routing Mechanism

The Job Applicant backend does not contain explicit logic to determine which shard to query. MongoDB's query routing infrastructure handles shard selection automatically through the following mechanism:

Connection Architecture:
The backend connects to MongoDB through a mongos router process rather than connecting directly to individual shards. The mongos acts as an intermediary that makes the sharded cluster appear as a single database to the application.

Query Routing Process:

1. **Backend Issues Query:** The Profile Management Module or Job Search Module sends a query to mongos (e.g., searching for applicants in Vietnam with specific skills).
2. **Shard Key Detection:** The mongos router analyzes the query to determine if the shard key (country) is present in the query predicates.
3. **Routing Decision:**
    a. **Targeted Query:** If the query includes country = "Vietnam", mongos routes the request only to the shard responsible for Vietnamese applicants. This results in fast, single-shard execution.
    b. **Broadcast Query:** If the query does not include country, mongos sends the request to all shards, collects results, and merges them before returning to the backend. This scatter-gather approach is slower.
4. **Backend Receives Response:** The backend receives query results without knowing which shard(s) were accessed.

Write Operations:
When the Authentication Module creates a new applicant during registration, it includes the country field in the document. The mongos router examines this field and routes the document to the appropriate shard based on MongoDB's internal chunk distribution.

2.2.4. Shard Distribution Architecture

The MongoDB cluster is configured with a minimum of three shards:

- **Shard 1:** Stores applicants where country = "Vietnam"
- **Shard 2:** Stores applicants where country = "Singapore"
- **Shard 3:** Stores applicants from other Southeast Asian countries

As DEVision expands into new markets (e.g., Thailand, Japan), additional shards can be provisioned. MongoDB automatically assigns country ranges to new shards without requiring backend code changes.

2.2.5. Handling Country Updates (SRS Ultimo 3.3.2)

When an applicant updates their country field (e.g., relocating from Vietnam to Singapore), MongoDB automatically migrates the document between shards:

1. The Profile Management Module issues an update operation changing the country field.
2. MongoDB detects that the document's shard key now maps to a different shard.
3. MongoDB performs an atomic document migration from the source shard to the destination shard.
4. The migration is transparent to the backend, the update operation completes successfully, and subsequent queries automatically route to the correct new shard.

MongoDB guarantees atomicity during migration, ensuring the document exists in exactly one shard throughout the process.

2.2.6. Backend Query Design Optimization

To maximize the benefits of sharding, the backend modules are designed to include the country parameter in applicant queries wherever possible. When users do not specify a location filter, the Job Search Module defaults to country = "Vietnam" (SRS 4.3.1), ensuring most queries execute as fast targeted operations rather than slow broadcast queries.

3. Squad Container Diagram

3.1.Job Applicant Frontend Container Diagram

Technology Stack: React, React Router, Context-based state management, Axios for REST networking, Websocket for live notifications

Architecture Style: Component-Based Architecture integrated with a Headless UI pattern. This separates the visual presentation from the business logic (Hooks & Headless UI components), fulfilling Ultimo extensibility requirements, and ensuring future module expansion without rewriting UI layers.

Core Containers & Responsibilities:

- **Core Containers:**
  - **Browser Router:** Acting as the client-side navigation manager, this container intercepts URL changes to render the appropriate Page components without refreshing the browser. It ensures smooth transition between the Job Search, Profile, and Admin views.
  - **Security Container:** A specialized logic layer that wraps the application. It intercepts outgoing HTTP requests to attach valid JWE/JWS authentication tokens and monitors incoming responses.
  - **Context-based State Management**: This container manages global application data that persists across different pages, such as the current user's profile data, search filter preferences, and active notification counts.
  - **HttpUtil & URL Config:** URL Config centralizes all backend API endpoint definitions (preventing hardcoded strings), while HttpUtil acts as a wrapper around the HTTP client (Axios), providing a standardized method for handling REST requests and error parsing.
  - **Main & Layout:** The root containers responsible for rendering the primary structural frame of the application (headers, sidebars, footers) and injecting the dynamic content from the Page container.
  - **Asset:** A storage container for static resources such as images, fonts, and global style definitions.

- **UI Component Containers:**
  - **Headless UI Components:** Provide functional logic without styling such as form handling, table behavior, pagination, modal state, and feature-specific hooks. These are consumed by multiple modules to maintain consistent behavior.
  - **Reusable Components:** A library of pre-styled UI elements (e.g., Buttons, Form Inputs, Cards). These consume the logic from Headless UI components to render the final interface, ensuring design consistency across the application.
  - **Domain Component Groups:** The application logic is segmented into functional groups:
    - **Auth Components:** Handles login, registration, email verification, SSO, and account recovery for normal users.
    - **Profile Module:** Manages profile viewing/editing, avatar upload, skills, education, experience updates, application history

- **Applicant Components:** Handle CV uploading, displays applicant submission history and detailed application records
- **Job & Subscription Components:** Render job lists, search bars, and pricing/payment modals.
- **Notification Components:** Display real-time alerts and messages.
- **Admin Components:** Offers moderation tools for managing applicants, companies, and job posts through sortable tables

Communication:

- **RESTful API:** The Frontend communicates with the Backend primarily via RESTful HTTP requests for authentication, profile management, job search, application submission, subscription management, and admin actions.
- **WebSocket:** The application maintains a persistent WebSocket connection with the Backend's Notification Module. This channel is used to receive real-time push notifications regarding new job matches.

3.2.Job Manager Frontend Container Diagram

Technology Stack: Node.js (Express.js) for backend web services, MongoDB Atlas for persistence, Redis for caching high-frequency session and token data, and KafkaJS for event streaming integration.

Architecture Style: Modular Monolith Architecture. The system is deployed as a single Node.js artifact but internally structured into clearly separated logical modules that encapsulate specific domains (Authentication, Applicant Profile, Application Processing, Notification, Search, and Subscription).

Each module can be independently migrated into microservices in the future with minimal refactoring, following the same domain-boundaries pattern as the JM system.

Communication:
The Job Manager frontend directly consumes a small set of service APIs produced by the Job Applicant subsystem whenever company recruiters need to view applicant-side data. These REST endpoints are called using Axios from the JM frontend and provide applicant-related information that JM does not store locally.

The JM frontend calls the following JA APIs:

- **JA Authorization API:** Used when JM frontend features require validating JA-issued tokens for cross-system access (e.g., viewing applicant profiles).
- **Applicant Profile API:** Retrieves applicant information such as identity, skills, and contact details when companies inspect applicants who applied to their job posts.
- **Application Data API:** JA provides the list of applicants who applied to a company's job posts ("who applied to this job"). The JM frontend uses this data to populate the application management screens.

3.3.Job Applicant Backend

Technology Stack: Spring Boot (Java) with Spring Security for authentication, MongoDB Atlas for persistence, and Redis for caching.

Architecture Style: Modular Monolith Architecture. The application is deployed as a single artifact but is internally structured into independent logical modules that encapsulate specific business domains, which can easily be restructured to be a Microservice Architecture eventually if needed.

Core Modules (Logical Containers):

- **Gateway & Security Containers:**
  - **Filter & Security Config:** The entry point for all traffic. It intercepts every request to execute the Security Filter Chain, using the Authentication Module logic to validate headers. It directly queries Redis Cache to enforce the "Token Denylist" policy, rejecting revoked tokens before they reach business logic.
  - **Redis Cache:** A high-performance store accessed by the Security Layer for token validation and by the Authentication Module to cache active user session data, reducing database latency.

- **Business Domain Modules:**
  - **Authentication Module:** The authority on user identity. It generates JWE/JWS tokens upon login and updates the Redis Cache during logout events to revoke sessions.
  - **Profile Management Module:** Manages applicant details including any actions with user profile (e.g. updating address, creating new user). It calls the File Storage Module to upload avatar images and publishes "Profile Update" events to the Kafka Broker, enabling the external Job Manager system to upload its “Job and Company Post” asynchronously.
  - **Subscription Module:** Contains the premium feature logic. It consumes "New Job" events from the Kafka Broker, executes the matching algorithm against user profiles, and upon finding a match, invokes the Notification Module to trigger an alert. It also coordinates with the Job Manager System to facilitate payment processing.
  - **Application Module:** Manages the submission workflow. It calls the Job Manager System (via REST) to validate the existence and status of a Job Post before processing an application, and delegates document handling to the File Storage Module to securely upload CVs and Cover Letters.
  - **Job Search Module:** A heavy read module designed for performance. It queries the Job Manager System APIs to aggregate and filter Job Post data for the search interface.
  - **File Storage Module:** An abstraction adapter invoked by the Profile and Application modules. It encapsulates the complexity of streaming files to the external Firebase service, returning secure access URLs to the caller.
  - **Notification Module:** A centralized communication service invoked by the Subscription or Authentication modules. It formats messages and delegates delivery to the external Email Service (for activation links) or pushes real-time alerts to the WebSocket Endpoint for connected Frontend clients.

Communication:

- **Internal:** Communication between modules occurs via direct Interface method calls, ensuring type safety and zero network latency.
- **External:** The Backend exposes REST endpoints and WebSocket channel to the Frontend while acting as a client to the Job Manager System (via HTTP/REST API) and Kafka Broker (via TCP) for ecosystem integration.

3.4. Job Manager Backend

Technology Stack: Node.js (Express.js) for backend web services, MongoDB Atlas for persistence, Redis for caching high-frequency session and token data, and KafkaJS for event-streaming integration. The subsystem uses Jest for automated unit and integration testing, Docker for containerized deployment, and GitHub Actions to automate CI/CD pipelines, run test suites on each commit, and build deployable artifacts.

Architecture Style: Modular Monolith Architecture. The system is deployed as a single Node.js artifact but internally structured into clearly separated logical modules that encapsulate specific domains (Authentication, Applicant Profile, Application Processing, Notification, Search, and Subscription).

Each module can be independently migrated into microservices in the future with minimal refactoring, following the same domain-boundaries pattern as the JM system.

Communication:

- **JM → JA (REST API Producer):** The Job Manager backend produces all Job-related REST APIs consumed by the Job Applicant system, including the Job Listing, Job Detail, Job Search, and Company Information APIs. JM also manages Redis caching for job posts, ensuring high-performance retrieval for all JA requests.
- **JM → JA (Payment API Producer):** The Job Manager backend provides the Payment API that the JA backend calls to initiate and verify payment transactions. JM only handles payment processing; the subscription state and premium logic are maintained entirely within the JA subsystem.
- **JM → JA (Kafka Event Producer):** The Job Manager backend publishes New Job Posted and Job Post Updated Kafka events. The JA backend consumes these events to update its own internal matching processes and trigger premium notifications. Job caching and job storage remain exclusively handled by JM.
- **JM ← JA (REST API Consumer):** The Job Manager backend consumes APIs provided by the Job Applicant subsystem, including JA Authorization, Applicant Profile Data, and Application Data, enabling JM to retrieve applicant information and job application lists for employer-side workflows.

4. Backend Component Diagram

This section presents the component diagrams for all core modules in the Job Applicant and Job Manager subsystems. Each component represents a functional unit within the overall architecture and is responsible for a specific domain capability, such as authentication, applicant profiles, job applications, job listing, or administrative tools. For every component, we provide a brief summary of its role along with the key incoming and outgoing communications that connect it to other modules, databases, Redis caches, or Kafka event streams.

Some modules are marked as optional, meaning they are only used when the corresponding feature is not yet implemented or is delivered later by the other subsystem.

4.1 Authentication Module

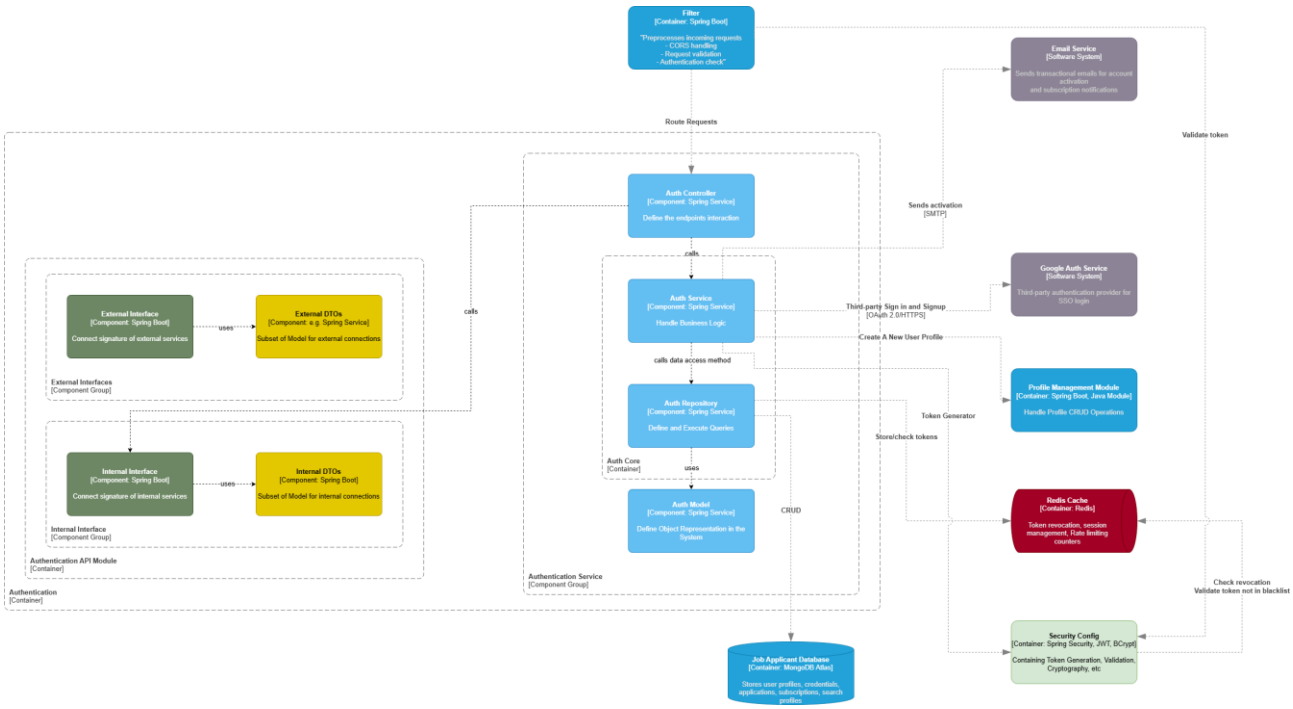

Figure 2: Authentication Module

Purpose:
Handles user registration, login, logout, credential verification, and SSO integration. Generates JWT tokens, validates tokens, and manages security rules for protected endpoints.

Incoming Communication:

- Frontend → /auth/* API calls for login, registration, logout, or SSO initiation.
- Filter Layer → token validation request on each call.

Outgoing Communication:

- Sends JWT tokens to frontend.
- Calls Google OAuth Service for SSO authentication.
- Stores/retrieves revocation tokens in Redis.
- Create init profile after user registration
- Send activation guide through email

4.2 Profile Management Module

Purpose:
Manages applicant profile CRUD operations including personal details, address, contact information, and resume visibility preferences.

Incoming Communication:

- Frontend API calls for viewing or updating profile information.
- JM System requests updated applicant profile summary for application processing.

Outgoing Communication:

- Stores profile information in JA Database.
- Sends updated profile field file values to File Storage Module to saved file in Firebase/Cloudinary (e.g., avatar updates).
- Sends summary profile to JM System when required during applications.

4.3 Application Module

Purpose:
Handles the job application workflow. Receives "apply for job" requests from applicants, saves the application record locally, and provides read APIs for JM to fetch list of applicants for each job post.

Incoming Communication:

- Frontend → /applications request when applicant applies for a job.
- JM System → /applications?jobPostId= to view applicants for a job post.

Outgoing Communication:

- Stores application data in JA Database.
- Forwards resume/cover letter file IDs to File Storage Module.
- Provides application data to JM System via REST API.

4.4 Job Search API Module (Optional)

Purpose:
Provides job listing to the frontend by consuming Job Manager APIs. Does not store job posts locally; acts as a passthrough, adding pagination, filtering, and optional caching.

Incoming Communication:

- Frontend API calls for listing, filtering, or viewing job details.

Outgoing Communication:

- Calls JM Job Post API to retrieve job lists and details.
- Uses Redis for caching job list data.

4.5 Search Profile Module

Purpose:
Manages the applicant's search preferences (skill filters, location, salary range, employment type). Used for filtering job lists and for premium matching logic.

Incoming Communication:

- Frontend → /search-profile/* CRUD operations.
- Kafka Broker → job post update events (used by matching logic).

Outgoing Communication:

- Stores search profile and skills in JA Database.
- Triggers Notification Module when a premium match occurs.
- Stores/retrieves Search Profile data in Redis for matching purposes.

4.6 Subscription Module

Purpose:
Handles premium subscription lifecycle, including plan activation, expiry, renewal, and status checks. Also processes payment requests.

Incoming Communication:

- Frontend → /subscription/* API calls.
- Payment Module → payment result callback (success/failure).

Outgoing Communication:

- Stores subscription state in JA Database.
- Sends payment request to external Payment Gateway (Stripe/PayPal).
- Notifies Notification Module when subscription status changes (e.g., expiry).

4.7 Notification Module

Purpose:
Centralized system for creating and storing notifications for applicants. Handles job-match alerts, subscription reminders, and system messages.

Incoming Communication:

- Search Profile Module → match event triggers.
- Subscription Module → subscription status updates.
- Admin Module → admin-generated notifications.
- Kafka Broker → job post update events that require premium alerts.

Outgoing Communication:

- Sends notifications to frontend via REST API.
- Stores notification data in JA Database.

4.8 File Storage Integration Module

Purpose:
Handles upload and retrieval of files (CVs, cover letters, profile images, portfolios) to cloud storage services such as Firebase or Cloudinary.

Incoming Communication:

- Frontend → file upload requests.
- Application Module → fetch CV/cover letter for job application.
- Profile Module → avatar/profile image updates.

Outgoing Communication:

- Uploads media files to Firebase/Cloudinary
- Saves file URLs and metadata to JA Database.

4.9 Admin Module

Purpose:
Provides functionality for system administrators to manage users, monitor system data, and access audit logs.

Incoming Communication:

- Admin Frontend → /admin/* API calls.

Outgoing Communication:

- CRUD data from account, applicant profile, company and job post.
- Uses Redis for caching account, applicant profile data; company, job post if needed

5. Frontend Component Diagram

5.1. Authentication Module

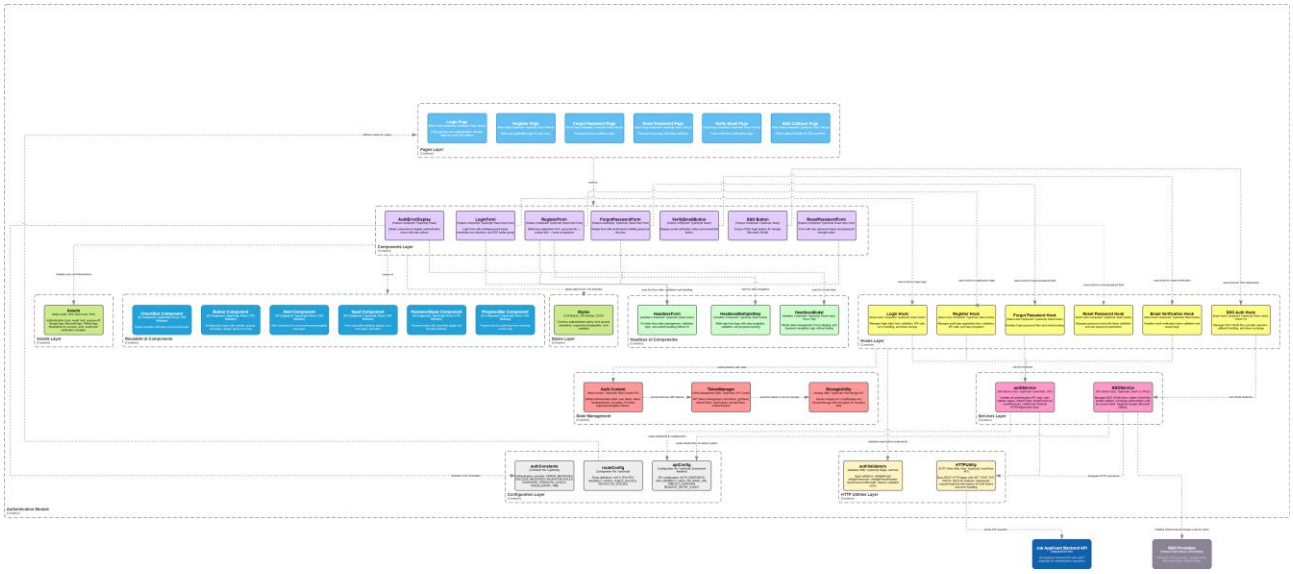

Figure X: Frontend Authentication Module Component

Container Purpose: The Authentication Module handles all user authentication flows including login, registration, SSO (Google/Microsoft/GitHub OAuth), password recovery, email verification, and session management. This module implements JWT-based authentication with refresh token mechanism and secure token storage.

Component Interactions:

- **Pages Layer → Components Layer:** Pages (LoginPage, RegisterPage, ForgotPasswordPage, ResetPasswordPage, VerifyEmailPage, SSO Callback Page) render their respective Components (LoginForm, RegisterForm, ForgotPasswordForm, VerifyEmailButton, SSO Button, ResetPasswordForm) and error display components(AuthErrorDisplay).
- **Components Layer → Hooks Layer:** Each Component uses dedicated Hooks (LoginHook, RegisterHook, ForgotPasswordHook, ResetPasswordHook, EmailVerificationHook, SSOAuthHook) to handle business logic, form state management, and API communication.
- **Hooks Layer → Services Layer:** Hooks call AuthService and SSOService to perform authentication operations, which in turn use HTTPUtil to make REST API calls to the backend.
- **Hooks Layer → State Management:** Hooks read/write authentication state via AuthContext, which manages global user data and session tokens.
- **Hooks Layer → Validators:** All user inputs are validated using AuthValidators (email, password, phone validation) before being sent to the backend.
- **Components Layer → Headless UI:** Forms use HeadlessForm for state management and validation logic. Modal dialogs use HeadlessModal for state and focus management. RegisterForm additionally uses HeadlessMultiStep for multiple step navigation.
- **Components Layer → Reusable UI:** All Feature Components render using Reusable UI Components (Button, Input, Checkbox, PasswordInput, ProgressBar, Alert) for consistent styling.
- **Services → External APIs:** AuthService communicates with Backend API (/auth/*) for login, registration, token refresh. SSOService redirects to SSO Providers (OAuth 2.0) and handles callback.
- **State Management → Storage:** AuthContext persists tokens via TokenManager, which stores data in browser storage using StorageUtility.
- **Components → Assets & Styles:** Components display Icons and Illustrations from Assets, and apply styling from auth.module.css.

5.2. Profile Management Module

Container Purpose: The Profile Module manages all applicant profile functionalities including viewing and editing personal information, avatar management, education records, work experience, technical skills, portfolio uploads, and viewing job application history. This module integrates Headless UI for form and table logic, reusable UI for consistent styling, and structured hooks for profile-related business logic and backend synchronization.

Component Interactions:

- **Pages → Components Layer:** ProfilePage renders all profile-related Components (ProfileView, EducationSection, SkillsListSection, AvatarSection, ExperienceSection, PortfolioSection). ApplicationHistoryPage renders ApplicationHistoryTable for displaying application records.
- **Components Layer → Hooks Layer:** Each profile Component uses dedicated Hooks (ProfileHook, AvatarHook, ApplicationHistoryHook) to handle fetching, updating, form interactions, and business logic specific to each profile domain.
- **Hooks Layer → Services Layer:** Hooks call ProfileService to perform CRUD operations for profile data, avatar uploads, education management, experience records, skills, portfolio items, and job application history. ProfileService uses HTTPUtility to make REST API calls.
- **Hooks Layer → State Management:** Hooks read/write shared profile state via ProfileProfileContext, allowing global access to profile fields, avatar URL, and synchronized updates across pages and components.
- **Hooks Layer→ Validators:** All profile inputs are validated using Validators (email, phone, GPA, dates, skill tag rules) before being submitted to the backend via the service layer.
- **Components Layer → Headless UI:** All edit forms (EducationSection, SkillsListSection, ExperienceSection, AvatarSection, PortfolioSection) use HeadlessForm for form state, validation, and

submission logic. Likewise, ApplicationHistoryTable uses HeadlessTable for sorting, pagination, and data rendering logic.

- **Components Layer → Reusable UI:** Feature Components render using Reusable UI Components (Card, Tag, Input, Button, UploadField, AvatarImage) for consistent styling and layout across the module.
- **Services → External APIs:** ProfileService communicates with Backend API endpoints to fetch and persist all profile-related data.
- **State Management → Storage:** ProfileProfileContext maintains in-memory state for profile data; no browser-persistence layer is used, ensuring all profile values refresh properly on re-authentication

5.3. Job Search Module

Container Purpose: The Job Search module provides comprehensive job search and application functionality. It includes full text search, advanced filtering (location, job type, salary, skills, newbie friendly), infinite lazy loading scroll, job detail view, and a complete application process with the ability to upload a CV and cover letter.

Component Interactions:

- **Search Flow:** JobSearchPage combines SearchBar (quick search), FilterSidebar (advanced filters), and JobList (results) to provide a complete search experience. JobList renders multiple JobCard components with infinite scroll.
- **Detail and Apply Flow:** Clicking a JobCard navigates to JobDetailPage, which displays JobDetailHeader, JobDescription, and CompanyCard. The ApplyButton opens ApplicationModal for submitting applications with files.
- **Search Logic → Hooks:** SearchBar and FilterSidebar both use JobSearch Hook to update search query and filters. JobList uses JobSearch Hook for results and InfiniteScroll Hook for lazy loading pagination.
- **Application Submission:** ApplicationModal uses JobApplication Hook to validate files and submit applications with CV and cover letter to both Job Applicant Backend (save application) and Job Manager Backend (job data).
- **My Applications Tracking:** MyApplicationsPage displays ApplicationCard components filtered by ApplicationFilters, using MyApplications Hook to fetch and manage application history.
- **Headless Integration:** SearchBar uses HeadlessSearch for search state. FilterSidebar uses HeadlessFilter for filter state. ApplicationModal uses HeadlessModal for modal management. JobList uses HeadlessInfiniteScroll and HeadlessPagination for lazy loading logic.
- **Service Communication:** JobSearchService communicates with Job Applicant API for application submissions and tracking. As expected, Job Applicant Backend API communicates with the Job Manager API to retrieve job listings and support job search.
- **Search Optimization:** SearchHelpers build optimized query parameters and parse complex filters. JobHelpers format job data and calculate match scores. FileValidators ensure CV and cover letter meet requirements.

5.4. Premium Subscription Module

Container Purpose: The Subscription Module manages premium subscription features, including plan selection, payment processing through the Job Applicant Backend API using multiple gateways (Stripe, PayPal), search profile configuration for automated job matching notifications, and real time notification delivery through the Kafka message broker. It handles the subscription lifecycle from purchase to cancellation and includes billing history management.

Component Interactions:

- **Subscription Purchase Flow:** SubscriptionPage displays PricingCards with different plans. SubscribeButton triggers navigation to PaymentPage, which renders PaymentForm, PaymentMethodSelector, and OrderSummary for checkout.
- **Payment Processing:** PaymentForm and PaymentMethodSelector use Payment Hook to process payments via PaymentService, integration with Stripe API, PayPal API and Job Manager Payment API is done by Job Applicant Backend API.
- **Subscription Management:** SubscriptionStatusPage shows ActivePlanCard (current plan details), UpgradeDowngradePanel (plan changes), and BillingHistory (past invoices), all using Subscription Hook and BillingHistory Hook.
- **Search Profile Configuration:** SearchProfilePage renders SearchProfileForm allowing users to set notification criteria (technical skills, location, employment status, salary range, frequency). It uses SearchProfile Hook to save criteria through SearchProfileService.
- **Real Time Notifications:** NotificationsPage displays NotificationList with filtering via NotificationFilters. Notifications Hook subscribes to WebSocketClient for real time job match notifications.
- **Payment Validation:** PaymentValidators validate card numbers, CVV, expiry dates. PricingCalculators compute tax, discounts, and prorated amounts for plan changes.
- **Headless Components:** Forms use HeadlessForm, SearchProfileForm uses HeadlessMultiSelect for skills and HeadlessDatePicker for date selection, BillingHistory uses HeadlessTable, NotificationList uses HeadlessNotificationCenter for notification state logic.

5.5. Admin Module

Container Purpose: The Admin Module provides administrators with tools to authenticate into the system, allowing admin users to manage and search for applicants, companies, job posts as well as perform deactivate actions. This module uses Headless UI for search, pagination, modals, and form logic, while Hooks and AdminService handle all admin-side business logic and backend communication.

Component Interactions:

- **Pages → Components Layer:** AdminLoginPage, AdminDashboardPage, AdminApplicantListPage, AdminCompanyListPage, and AdminJobPostListPage render their respective Components such as AdminLoginForm, AdminHeader, AdminSidebar, AdminTable, AdminSearchBar, EntityDetailModal, and ConfirmationModal for viewing, filtering, and managing entitie
- **Components Layer → Hooks Layer:** Components use Hooks (AdminAuthHook, AdminApplicantHook, AdminCompanyHook, AdminJobPostHook, AdminSearchHook) to execute admin business logic, load table data, trigger entity actions, and handle admin actions.
- **Hooks Layer → Services Layer:** Hooks call AdminService to perform administrative operations (fetch lists, search entities, get details, delete/deactivate targets). AdminService uses HTTPUtility to conduct REST API communication with the backend.
- **Hooks Layer → State Management:** Hooks read/write authentication and search state through AdminAuthContext (admin session, role, token) and AdminSearchContext (query, filters, results), enabling consistent state across the Admin interface.
- **Hooks Layer→ Validators:** Inputs such as search queries, identifier fields, and form submissions are validated using AdminValidators before being processed or forwarded to AdminService.
- **Components Layer → Headless UI:** Tables use HeadlessTable for row formatting, sorting, and pagination logic. Search bars use HeadlessSearch for controlled input and debouncing. Paginated lists use HeadlessPagination for page navigation. EntityDetailModal and ConfirmationModal use HeadlessModal for modal state and rendering logic. Any administrative input form uses HeadlessForm for validation and submission workflows.

- **Components Layer → Reusable UI:** Feature Components render using Reusable UI Components (Button, Input, Dropdown, Badge, Card, ModalWrapper, Toolbar) for consistent structure and visual presentation across all admin pages.
- **Services → External APIs:** AdminService communicates with the Backend API to retrieve entities, update statuses, and perform administrative actions.
- **State Management → Storage:** AdminAuthContext stores admin session state in memory during admin usage. AdminSearchContext stores search parameters and results to maintain continuity during navigation.

5.6. Application Module

Container Purpose: The Application Module provides applicants with comprehensive visibility into their job applications. It includes application listing with advanced filtering and sorting, detailed application views with status tracking, timeline visualization of status changes, document preview CV and cover letter, and real time status updates through WebSocket connection. Users can withdraw applications or reapply for rejected positions.

Component Interactions:

- **Application List View:** MyApplicationsPage displays ApplicationList (grid/list toggle) composed of ApplicationCard components. ApplicationStatusFilters (All, Pending, Reviewed, Accepted, Rejected), ApplicationDateFilter, and ApplicationSortOptions control the display. It uses useMyApplications Hook to fetch and filter applications.
- **Application Detail View:** Clicking ApplicationCard navigates to ApplicationDetailPage showing ApplicationDetailHeader (job info + status badge), ApplicationTimeline (status change history with timestamps), SubmittedDocuments (CV/cover letter preview), and ApplicationNotes (user notes section).
- **Status Tracking:** ApplicationStatusPage renders StatusProgressBar (visual progress indicator) and ApplicationTimeline (chronological status history). It uses ApplicationTimeline hook to fetch status change events through ApplicationService.
- **Real-time Updates:** RealTimeStatus Hook subscribes to WebSocket Server for live status updates. When status changes, the UI automatically updates without page refresh through WebSocketClient integration.
- **Application Actions:** WithdrawButton allows cancelling pending applications (opens HeadlessModal for confirmation). ReapplyButton enables reapplying for rejected jobs. Both use useApplicationActions hook calling ApplicationService.
- **Statistics & Analytics:** ApplicationStats component displays total applications, success rate, and average response time. It uses ApplicationStats Hook with StatsCalculators to compute metrics.
- **Data Formatting:** ApplicationHelpers format application data, calculate time elapsed, and group by status. StatusHelpers provide color coding and labels for different statuses. TimelineHelpers sort and format timeline events.
- **Headless Integration:** ApplicationList uses HeadlessTable. Filters use HeadlessFilter and HeadlessDateRangePicker. ApplicationTimeline uses HeadlessTimeline for event logic. StatusProgressBar uses HeadlessStatusTracker for progress calculation. Modals use HeadlessModal.
- **WebSocket Architecture:** WebSocketClient maintains persistent connection to WebSocket Server for real time status notifications, handles disconnections with automatic reconnection, and delivers updates to subscribed components.

6. Architecture Rationale

6.1. Data Model Justification

The DEVision Job Applicant data model employs a normalized relational structure within the subsystem boundary while leveraging MongoDB's document-oriented capabilities for schema flexibility and horizontal scalability. This section analyzes the design decisions governing entity relationships, cross-subsystem integration, and data distribution strategies, evaluating their implications for system scalability, performance, and resilience.

6.1.1 Separate Database Architecture (JA and JM Subsystems)

Design Decision: All our applicant data (APPLICANT, EDUCATION, WORK_EXPERIENCE, APPLICATION, etc.) lives in our own MongoDB database. Within this database, we have proper foreign key relationships, the ERD shows APPLICANT connecting to EDUCATION (one applicant has zero to many education entries), to WORK_EXPERIENCE (one applicant has zero to many jobs experience), and to APPLICATION (one applicant submits zero to many applications). These relationships work exactly like traditional foreign keys.

The challenge is when APPLICATION needs to reference Job Manager's data. Their JOB_POST and COMPANY entities live in a completely separate database, possibly on different servers. So we just store job_post_id and company_id as plain strings, MongoDB can't enforce foreign keys to data that doesn't exist in our database.

Rationale: This pattern is called "database per service" in microservices architecture [1]. It gives both teams complete independence. The JM team can change their job post structure whenever they want without coordinating with us. We deploy separately, they deploy separately. Within our database, all the ERD relationships enforce data integrity properly, MongoDB won't let you create an EDUCATION entry for a non-existent applicant.

Performance and Scalability: Our internal queries are fast, single-digit millisecond response times for profile lookups. Both teams can scale their databases independently. If job search traffic spikes, JM adds more servers without affecting our applicant queries.

On the other hand, cross-database queries are slower. When we need to show an application with full job details, we first query our database (gets APPLICATION), then call JM's API (gets JOB_POST details). That HTTP call adds 10-50ms compared to a database JOIN [3]. More requests means proportionally slower total response time.

Resilience Trade-offs: When JM deletes a job post, our APPLICATION records don't automatically know about it. We end up with applications pointing to jobs that don't exist anymore. Inside our database, referential integrity is solid. But across database, we need scheduled jobs to check with JM's API and flag deleted references. If that job fails, users see broken applications until we fix it. This is the price of database independence, we trade automatic consistency for team autonomy [4].

6.1.2. Country-Based Sharding Strategy

Design Decision and Rationale: The APPLICANT collection employs MongoDB's sharded cluster with country as shard key, partitioning documents across geographic shards. This aligns with predominant query patterns where job searches filter by location, enabling targeted queries rather than scatter-gather operations.

Most job searches are local. Vietnamese people search for jobs in Vietnam, Singaporeans search in Singapore. By grouping users geographically, MongoDB can route these queries to just one shard instead of asking all of

them. When your query includes the shard key (country), MongoDB calls it a "targeted query", it only hits one shard [2]. Without the shard key, MongoDB does "scatter-gather" across all shards, which is much slower.

Performance and Scalability: This scales horizontally. When we launch in Thailand, we just add a Thailand shard. No data migration needed. Studies show targeted queries run 3-10x faster than scatter-gather [5], which is perfect for our main use case.

It also helps with data residency laws. If Vietnam requires data to stay in Vietnam, we physically put the Vietnam shard on Vietnamese servers.

Resilience Limitations: We are assuming users distribute evenly across countries. But in case 70% of our users end up being Vietnamese, that shard handles 70% of traffic while the Singapore shard sits mostly idle. This is called a "hot shard" problem [4], one server gets overloaded while others waste capacity.

Also, when users relocate internationally and update their country, MongoDB moves their document between shards. This operation can fail halfway through if the network drops, leaving the user's data in limbo until we manually fix it.

6.1.3. MongoDB Document Model with Embedded Arrays

Design Rationale: The data model leverages MongoDB's document structure with embedded arrays (APPLICANT.applicantSkills, SEARCH_PROFILE.desired_skills) rather than normalized join tables. This follows MongoDB's design philosophy of "data that is accessed together should be stored together" [6], trading normalization for query performance.

Performance Advantages: Embedding related data eliminates JOIN operations required in relational databases. Retrieving a complete applicant profile (personal info, skills, education) requires a single document fetch from MongoDB's WiredTiger cache, typically completing in single-digit milliseconds. This design particularly benefits mobile clients with high-latency connections, as one network round-trip replaces multiple sequential queries. The approach aligns with SRS performance requirements for "quick, efficient responses".

Limitation: MongoDB has a 16MB document size ceiling [6]. Normal profiles take 2-5 KB, but if a user uploads hundreds of portfolio items and we embed all metadata, we could hit that limit. We need defensive code to paginate large collections or move them to separate documents when they grow too big.

Embedded structures also complicate updates across multiple conceptual entities. Updating skills in APPLICANT while changing SEARCH_PROFILE preferences requires coordinating two document updates, which MongoDB's multi-document transactions handle with lower throughput than single-document operations [2].

6.1.4. External Cloud Storage with Metadata Separation

Design Rationale: CV files, cover letters, and portfolio images live in Firebase or Cloudinary. Our MEDIA_FILE collection only stores metadata, URLs, file sizes, upload timestamps.

Storing a 5MB PDF in MongoDB for 10,000 applicants = 50GB of files bloating the database. MongoDB would waste memory caching file bytes instead of query data. By externalizing files, MongoDB stays focused on structured queries [4]. Cloud storage providers also integrate with CDNs, serving files from edge servers near users, reducing latency from 200ms (origin fetch) to 20-50ms (edge cache) [8].

Performance Benefits: Externalizing files keeps MongoDB focused on structured data queries, reducing database size and improving cache hit rates. With metadata occupying kilobytes versus files occupying megabytes, more applicant profiles fit in WiredTiger's cache, accelerating profile queries. Additionally, cloud storage providers offer CDN integration, employers in Singapore download CVs from regional edge nodes

rather than origin servers in Vietnam, reducing latency from 200ms+ to 20-50ms. This addresses performance requirements for geographically distributed users.

Scalability and Cost: Cloud storage scales elastically without manual capacity planning, Firebase automatically handles replication and load balancing as file volume grows. The pay-per-use pricing model avoids upfront infrastructure costs for storage capacity. However, this introduces vendor lock-in; migrating from Firebase to S3 requires updating all MEDIA_FILE URLs and transferring files between providers, a complex operation without atomic guarantees.

Resilience Vulnerabilities: The two-phase operation (upload file, save metadata) lacks transactional guarantees across systems. If Firebase upload succeeds but MongoDB insert fails, orphaned files accumulate untracked. Conversely, if MongoDB insert succeeds but Firebase upload was incomplete, metadata points to nonexistent files. The system requires scheduled reconciliation jobs to detect and clean orphans, accepting eventual consistency as Newman describes: "you need to be prepared for things to be out of sync" [3].

6.1.5. Normalized Skill Master Table

Design Rationale: We have a SKILL collection with standardized skill names. When applicants add skills, their applicantSkills array stores these standardized names, not random text they type.

Without standardization, applicants would enter "ReactJS", "React.js", "React", "react" as different skills. Employers searching for React developers would miss half the candidates due to spelling variations. The SKILL master table enforces a controlled vocabulary, everyone uses "React" (the canonical version). This is a normalization principle from database theory [7] that prevents data fragmentation.

Performance Trade-off: We store skill names directly in applicantSkills (denormalization) to avoid extra lookup queries when displaying profiles. But if we rename a skill in the SKILL table, we need a migration job to update all applicant documents with the new name. We're trading write complexity for read speed.

6.1.6. Acknowledgment of Design Limitations

The data model exhibits two notable limitations per the rubric's "up to two minor flaws" allowance:

Limitation 1: Cross-database references (APPLICATION → JOB_POST in JM) rely on application code, not database enforcement. Applications can point to deleted jobs until our reconciliation job runs (up to 24 hours). This is the trade-off for subsystem independence.

Limitation 2: Geographic sharding assumes even user distribution. If users concentrate heavily in one country (say 70% Vietnamese), that shard gets overloaded while others sit idle. Fixing this requires either expensive vertical scaling (bigger servers) or risky re-sharding with a compound key like {country: 1, city: 1} requiring live data migration.

In conclusion, the Job Applicant data model achieves high normalization with minimal redundancy through strategic entity separation (ACCOUNT, APPLICANT, SKILL) and external binary storage. The design addresses Simplex, Medium, and Ultimo requirements through country-based sharding, normalized skill management, and distributed file storage. Trade-offs between scalability and consistency reflect intentional architectural choices, accepting eventual consistency for subsystem independence, optimizing for geographic query patterns while risking hot shards, and normalizing skills while incurring join overhead. These decisions position the system for horizontal scaling and independent subsystem evolution while requiring operational mechanisms (reconciliation jobs, monitoring, compensating transactions) to manage the inherent complexities of distributed data architecture.

6.2. Frontend Architecture Justification

Design Rationale

The DEVision Job Applicant frontend implements a component based architecture with React, structured around an N-Tier layered pattern integrated with Headless UI components and Context based State Management. This architectural approach was chosen to separate presentation concerns (visual components), business logic (hooks layer), UI behavior logic (headless components), and data management (services and contexts). The design supports the Ultimo level requirements for modularity, extensibility, and maintainability while delivering a responsive user experience across authentication, profile management, job search, subscription, and administrative features.

By structuring the frontend into distinct functional modules (Authentication, Profile, Job Search, Subscription, Application, Admin), each containing its own pages, components, hooks, and services, the architecture enables teams to work on different features independently while maintaining consistent patterns and interfaces. This modular organization, combined with the Headless UI pattern that decouples behavior from presentation, provides flexibility to redesign visual elements without rewriting core logic, supporting both immediate development needs and long term platform evolution.

The following subsections evaluate the frontend architecture against six key software qualities maintainability, extensibility, resilience, scalability, security, and performance explaining how design decisions benefit the DEVision platform while acknowledging inherent tradeoffs and limitations.

Maintainability

The Advantages:

The frontend's layered component architecture significantly improves maintainability through a clear separation of concerns. Each feature module (Authentication, Profile, Job Search, Subscription, Admin, Application) is self contained with its own pages, components, hooks, services, and styles, adhering to the principle of high cohesion and low coupling. For example, updating password validation rules only requires changes in the Authentication module's AuthValidators utility and Login hook, without impacting other modules. This isolation reduces cognitive load and limits the risk of regressions.

The use of hooks further enhances maintainability by centralizing business logic in reusable, testable units. When job search filtering requirements change, developers only need to update them JobSearch hook and its SearchHelpers utility, with all consuming components automatically reflecting the new behavior.

In addition, TypeScript is applied across components, hooks, and services, offering static type checking and acting as living documentation for API responses, context state, and component props. When backend API contracts change, TypeScript compilation errors indicate precisely which modules require updates, thereby reducing debugging time and preventing subtle runtime errors in production.

The Disadvantages:

The multiple layered architecture introduces initial complexity, as developers must understand the interactions between Pages → Components → Hooks → Services → APIs, as well as the project's specific conventions. This learning curve can slow down onboarding compared to simpler, flatter React structures.

Moreover, maintaining architectural consistency demands discipline and robust code review practices. If developers bypass the established patterns such as calling HTTP utilities directly from components instead of using services, or duplicating validation logic instead of centralizing it in shared validators the long-term maintainability benefits are diminished. Without supporting tools such as architectural linting or automated checks, these inconsistencies can accumulate and gradually make the codebase harder to understand and evolve.

Extensibility

The Advantages:

The use of the Headless UI pattern across the frontend provides strong extensibility by clearly separating behavioral logic from visual presentation. Components such as HeadlessTable, HeadlessForm, and HeadlessModal encapsulate complex state management, keyboard interactions, and accessibility concerns, while allowing visual components (Button, Card) to be freely restyled or replaced.

Context based state management also creates natural extension points for cross cutting concerns. For example, introduce a new ThemeContext for dark mode, only requires defining the context, a Theme hook, and wrapping the application root. Existing components can be migrated gradually, avoiding disruptive “big bang” changes typically associated with tightly coupled, centralized state stores.

The Disadvantages:

The headless approach, while flexible, introduces additional abstraction that can make simple changes more involved. Adjusting basic table styling, for example, may require understanding both the HeadlessTable logic and the styled Table wrapper, spreading what could be a single CSS change across multiple files.

Similarly, the component based, modular architecture depends on well designed boundaries and interfaces. If these are not defined carefully, new requirements may force developers to bypass existing abstractions and introduce technical debt. For example, if the Subscription module needs job details that are not exposed by JobSearchService, the team must either extend JobSearchService in a way that blurs its responsibility or duplicate job fetching logic in SubscriptionService. Both options reduce the long term extensibility that the architecture is intended to provide.

Resilience

The Advantages:

The service layer, implemented via HTTPUtil interceptors, centralizes error handling and retry logic. For transient network issues, the interceptor can retry requests with exponential backoff before surfacing errors to users. It also handles token refresh: when a 401 Unauthorized response is returned, the interceptor automatically attempts to refresh the JWT and retries the original request, preserving user sessions without forcing re-authentication.

State management through React Context and custom hooks further isolates failures. If NotificationContext becomes inconsistent due to a WebSocket error, other contexts (AuthContext, UserContext, JobSearchContext) remain unaffected. Users may temporarily lose real time notifications but can still browse jobs, update profiles, and submit applications, ensuring graceful degradation of non-critical features.

The Disadvantages:

Client-side retry logic in the HTTP interceptor can unintentionally amplify backend overload if it is misconfigured. When the backend is already timing out, aggressive retries from many frontend clients can increase traffic and slow down recovery. Because the frontend has limited visibility into backend health, it must use conservative retry limits and backoff strategies to avoid contributing to cascading failures.

Context based state isolation improves resilience, it also fragments state management. Cross cutting updates such as a successful subscription purchase that should update both SubscriptionContext and NotificationContext are harder to coordinate, increasing the risk of inconsistent state. Debugging such issues becomes more complex, as developers must trace interactions across multiple independent contexts rather than a single centralized store.

Scalability

The Advantages:

The modular frontend architecture supports horizontal scalability by enabling parallel development across feature modules. Dedicated teams can work independently on Authentication, Profile, Job Search, and Subscription without frequent merge conflicts, as each module's components, hooks, and services are isolated.

Scalability at the UI level is also achieved through virtual scrolling and pagination. Components such as HeadlessInfiniteScroll for job lists and HeadlessTable for application tables render only the items visible in the viewport (20 cards instead of 1000), updating as the user scrolls.

The Disadvantages:

Code splitting introduces additional complexity in dependency management and performance tuning. If commonly used primitives (Button, Card) are bundled together with feature specific code instead of being extracted into shared chunks, users may download duplicate code across multiple lazy loaded routes. Maintaining an optimal splitting strategy requires continuous bundle analysis and specialized expertise, increasing operational overhead over time.

Furthermore, while modularization improves team autonomy, it does not remove the need for cross team coordination. Shared infrastructure such as HTTPUtil, shared Context definitions, and headless UI components must be versioned and evolved carefully. For example, if the Profile team changes the validation API of HeadlessForm without aligning with the Authentication and Subscription teams, it can cause regressions. Managing such shared dependencies at scale demands strong technical leadership and robust automated testing, which can become organizational bottlenecks.

Security

The Advantages:

The frontend participates in a defense in-depth authentication design by managing only short lived JWT access tokens in memory through AuthContext, instead of using localStorage, thereby reducing the risk of token theft via XSS. A dedicated TokenManager module handles the access token lifecycle on the client side, it injects tokens into authenticated requests, triggers refresh calls to the backend when access tokens approach expiry, and logs users out or redirects them to the login page when the backend indicates that the refresh token stored in HTTP-only cookies is no longer valid. This collaboration between frontend token management and backend cookie-based refresh endpoints helps balance security (limited exposure of tokens on the client) and usability (longer-lived sessions with minimal explicit re-authentication).

Input validation is implemented on both the client and server. Client-side validators (AuthValidators, ProfileValidators, FileValidators) provide instant feedback, while the backend remains the single source of truth for all security-critical checks. For file uploads (CVs, cover letters, avatars), the frontend enforces constraints on file type (.pdf, .docx), maximum size (5MB), and basic content checks to reduce the attack surface before requests reach the server. React's automatic JSX escaping further mitigates XSS when rendering user-provided content such as profile summaries or job descriptions.

The Disadvantages:

In-memory token storage improves security but introduces usability and implementation complexity. Tokens are lost on page refresh, so the application depends on refresh tokens in HTTP-only cookies and an interceptor that transparently renews access tokens. Any defects in this refresh flow can cause confusing behavior, such as unexpected logouts or repeated login prompts.

Performance

The Advantages:

User input is optimized through debouncing and throttling. The SearchBar debounces text input by 300ms so that typing a query such as "Software Engineer" results in a single API request instead of one per keystroke. Infinite scroll handlers are throttled to limit scroll computations. These techniques both improve perceived responsiveness and reduce backend load.

The Disadvantages:

These optimizations make the code harder to maintain and easier to break. If we set the dependencies for Callback incorrectly, the UI can use old data and not update properly. Caching adds another problem that if the TTL is not tuned well, users may see outdated job data or we make too many requests. Also, lazy loading makes the first page load faster, but can cause a noticeable delay the first time users open less common pages like the Admin panel.

6.3. Backend Architecture Justification

Design Rationale

The DEVision backend adopts a modular monolith architecture where each business capability, such as Authentication, Applicant Profile, Application Processing, and Notification, is isolated into its own module with clear responsibilities and boundaries. This architectural style was selected to balance simplicity with long-term flexibility, allowing the system to be deployed as a single application while still supporting independent evolution of modules.

Evaluating the architecture through key software qualities such as maintainability, extensibility, resilience, scalability, security, and performance helps demonstrate how the chosen design supports both current project requirements and future growth. The following subsections justify how the architecture benefits DEVision in each of these dimensions, while also acknowledging the trade-offs and limitations that come with the design.

Maintainability

The DEVision backend is organised into clear functional modules (Authentication, Applicant Profile, Application Processing, Notification, etc.), each encapsulating its own controllers, services, and persistence logic. This improves maintainability because developers can understand and change a feature in isolation (for

example, adjusting password policies only affects the Authentication module) and can test modules independently with focused unit and integration tests.

However, this structure still requires discipline: if future developers bypass module boundaries (for example, one module directly accessing another module's repositories instead of using defined interfaces), the architecture becomes inconsistent and harder to maintain, increasing onboarding effort and the risk of regression.

Extensibility

The modular architecture of DEVision supports extensibility by allowing new features (such as a Subscription module, activity tracking, or additional admin tools) to be added as separate modules that integrate via existing APIs or domain events, without needing to rewrite the core backend. Stable DTOs and module interfaces mean new endpoints (for example, updating applicant preferences or attaching extra documents to an application) can be introduced with limited impact on existing components.

The downside is that designing and maintaining these extension points adds abstraction overhead, and if backward compatibility is not managed carefully, changes to data contracts or validation rules can break existing clients or internal tools that depend on earlier versions.

Resilience

DEVision improves resilience by separating responsibilities across modules and using infrastructure components such as Redis and asynchronous workers for background tasks like sending notifications. If one part of the system is degraded (for example, the email provider causing delays in the Notification module), core flows like login, profile update, and application submission can continue to operate. Resilience patterns such as retries and fallbacks can also help the system remain functional under transient failures.

On the other hand, adding Redis, queues, and background workers increases the overall number of components that can fail. If Redis becomes unavailable or workers stop processing tasks, token validation or downstream actions may fail or be delayed, and misconfigured retries can worsen incidents by overloading already stressed components.

Scalability

The backend is primarily stateless at the web layer, allowing multiple instances of DEVision to be deployed behind a load balancer and scaled horizontally as applicant traffic increases. Stateful concerns such as applicant data, application records, and session-related information are offloaded to MongoDB Atlas and Redis, enabling the team to scale specific modules (for example, Application Processing or Notification workers) independently based on load.

However, scalability is ultimately constrained by shared resources: if MongoDB collections are not indexed correctly or if many modules rely on the same Redis or database cluster, these can become bottlenecks. Scaling only the application servers without coordinating database and cache capacity may lead to higher contention, slower queries, and degraded performance.

Security

DEVision applies layered security by centralising authentication in the Authentication module, using JWT or session tokens, enforcing role-based access control (for example, Applicant vs Admin), and protecting all public endpoints over HTTPS. Sensitive operations such as login, email verification, and profile updates are guarded by proper authentication and authorization checks, and Redis can be used for fast token revocation to reduce the window of exposure if a token is compromised.

The disadvantage is that strong security often increases system complexity: token lifetimes, refresh flows, and role definitions must be managed carefully to avoid locking out legitimate users or introducing subtle bugs. Misconfiguration of CORS, file upload validation, or admin-only routes can still create security gaps despite a generally secure architecture.

Performance

Performance in DEVision is enhanced by using Redis to cache frequently accessed authentication or profile-related data and by pushing non-critical work (such as sending confirmation emails or notifications) to asynchronous background processes, so user actions like login and application submission can return quickly. Stateless controllers and lightweight DTOs also help the backend handle many concurrent requests efficiently.

Nevertheless, performance optimisations introduce their own trade-offs: poorly designed or invalidated caches can lead to stale data being served, and complex or unindexed queries on applicant and application collections can still be slow regardless of how many backend instances are deployed. Over time, layering multiple performance “patches” without careful design can also reduce code clarity and make future tuning more difficult.stifications specific to the backend

7. Conclusion

System features summary

The DevVision Job Applicant (JA) subsystem serves as a comprehensive platform designed to streamline the candidate experience from registration to employment. The system delivers a robust set of features including:

- **Identity Management:** Secure authentication supporting both local credentials and OAuth SSO providers.
- **Profile & Portfolio:** Detailed candidate profiling that captures education, experience, and skills, supported by cloud-based storage for CVs, cover letters, and portfolio media.
- **Job Interaction:** A high-performance search engine with advanced filtering, coupled with an easy application submission workflow and real-time status tracking.
- **Premium Ecosystem:** A subscription-based model offering automated, real-time job matching via Kafka event streams and instant WebSocket notifications.
- **Administrative Control:** A dedicated module for user management, system monitoring, and content moderation.

Key architectural advantages and limitations

The system architecture uses a Modular Monolith approach, which creates a good balance between ease of development and future scalability. By separating business areas like Authentication and Subscriptions into distinct modules, the code is easier to maintain and test, while still allowing us to migrate to microservices later if needed. To handle growth, we used a country-based sharding strategy in MongoDB, which improves performance for users searching within their local region. We also optimized the system by storing heavy files

in external cloud storage and using Redis to cache frequent data, which keeps the main database fast. Furthermore, using Kafka for asynchronous communication increases resilience, ensuring that the system continues to function even if the connection to the Job Manager subsystem is temporarily delayed.

However, there are some trade-offs in this design. Because we follow the "Database-per-Service" pattern, there is no strict link (foreign key) between the Job Applicant and Job Manager databases. This results in eventual consistency, where the system might briefly show data for a job post that was just deleted. The country-based sharding strategy also carries a risk of "Hot Shards" if the user base is not evenly distributed across regions, which could create performance bottlenecks in popular countries. Finally, because the system interacts with multiple services like MongoDB, Firebase, and Kafka, we cannot guarantee atomic transactions.

8. References

All Diagram Link: https://github.com/ISYS3461-2025C-SonTinh-DEVision/JobApplicant/tree/main/Docs/Diagram

[1] C. Richardson, "Pattern: Database per service," *Microservices.io*, 2018. [Online]. Available: https://microservices.io/patterns/data/database-per-service.html

[2] MongoDB, Inc., "Sharding," *MongoDB Manual*, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/sharding/

[3] S. Newman, "Building Microservices," *O'Reilly*, 2021. [Online]. Available: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/

[4] M. Kleppmann, "Designing Data-Intensive Applications," *O'Reilly*, 2017. [Online]. Available: https://dataintensive.net/

[5] MongoDB, Inc., "Sharding Best Practices," *MongoDB Blog*, 2023. [Online]. Available: https://www.mongodb.com/blog/post/sharding-performance-best-practices

[6] MongoDB, Inc., "Data Model Design," *MongoDB Manual*, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/core/data-model-design/

[7] E. F. Codd, "Relational Model," *ACM*, 1970. [Online]. Available: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

[8] Cloudflare, "What is a CDN?," 2024. [Online]. Available: https://www.cloudflare.com/learning/cdn/what-is-a-cdn/