🏗️ Definitive Architectural Master Plan: job-hunt-api

Strictly Confidential Execution Blueprint

This document consolidates every architectural decision for the greenfield apps/api application. It assumes a **Clean Architecture** approach within a **Modular Monolith**, hosted in a **Single Docker Container** on **Google Cloud Run**.

1. High-Level Project Structure & Monorepo Strategy

Objective: Establish the root-level application apps/api alongside the existing apps/web.

The project shifts from a "Backend-for-Frontend" (Next.js API Routes) to a **Service-Oriented Architecture (SOA)**. While hosted in a single repo and container, the api is a distinct application server.

1.1 Technical Stack

- **Framework:** NestJS v10+ (Node.js 20).
- **Language:** TypeScript (Strict Mode).
- **Architecture:** Modular Monolith with Hexagonal Layers (Domain, Application, Infrastructure).
- **Deployment:** Unified Docker Image (Alpine Edge base). NestJS serves the API *and* the Next.js static export.

1.2 File System Layout

The structure strictly separates the Framework (NestJS) from the Business Domain.

Plaintext

```
repo-root/
├── apps/
│   ├── web/                # Existing Next.js (Refactored to output: export)
│   └── api/                # NEW Greenfield NestJS Application
│       ├── src/
│       │   ├── common/       # Shared Decorators, Filters, Types (DDD Primitives)
│       │   │   ├── filters/      # Global Exception Filters
│       │   │   ├── guards/       # Auth & Role Guards
│       │   │   └── interceptors/ # Response Transformation
│       │   ├── config/       # Environment Validation (Zod)
│       │   ├── core/         # Global Infrastructure (Logger, Database, Config)
│       │   ├── modules/      # Feature Modules (The Business Logic)
│       │   │   ├── auth/     # Session Management & Identity
│       │   │   ├── resume/   # PDF Generation (Tectonic) & Parsing
│       │   │   ├── search/   # Vector Search (Firestore Native)
│       │   │   ├── queue/    # Async Processing (BullMQ)
│       │   │   ├── ai/       # LLM Abstraction (Vertex/OpenAI)
│       │   │   └── cli/      # Operational Commands
│       │   ├── app.module.ts # Root Module
│       │   ├── main.ts       # HTTP Entry Point
│       │   └── cli.ts        # CLI Entry Point (Distinct from main)
│       ├── Dockerfile        # Multi-stage build definition
│       └── package.json
└── docker-compose.yml        # For local dev (Redis emulator, etc.)
```

---

2. Module 1: The Core Infrastructure

Objective: Establish the "plumbing" required for the app to boot securely and observably. This module is imported *only* by the root AppModule.

2.1 Configuration (Fail Fast Strategy)

We utilize @nestjs/config paired with zod to enforce strict environment validation. The app must **crash immediately** on startup if variables are missing.

- **File:** src/config/env.schema.ts
- **Schema:**
  - PORT: Number (Default 8080).
  - GCP_PROJECT_ID: String (Required).
  - FIREBASE_CREDENTIALS: JSON String or File Path (Required).
  - REDIS_URL: String (Required for BullMQ).
  - TECTONIC_BIN_PATH: String (Path to binary in Docker).
- **Behavior:** configuration.ts parses process.env through the Zod schema. If validation fails, NestJS aborts the bootstrap process, printing a clear error to stderr.

2.2 Structured Logging (Native Observability)

We adhere to the "Logs are a Stream" philosophy (12-Factor App). Logs are written purely to stdout in JSON format for Google Cloud Logging ingestion.

- **Library:** nestjs-pino, pino-http.
- **GCP Integration:**
  - **Severity Mapping:** We map Pino integer levels (30, 40) to GCP Severity strings (INFO, WARNING).
  - **Trace Correlation:** We extract the X-Cloud-Trace-Context header from incoming HTTP requests and inject it into the log object. This allows clicking a log in GCP Console to see the full request trace.
- **File:** src/core/logging/gcp-pino.config.ts

2.3 Database Layer (Firestore Abstracted)

- **Library:** firebase-admin (Initialized once via a Custom Provider).
- **Pattern: Abstract Repository**. We do not use the Firestore SDK directly in Controllers.
- **Files:**
  - firestore.provider.ts: Exports the firebase-admin instance.
  - base.repository.ts: A generic abstract class BaseRepository<T>. It handles standard CRUD and the automatic conversion of Firestore Timestamp <-> JavaScript Date.

---

3. Module 2: Authentication & Security

Objective: Replace legacy next-auth with a secure, backend-managed Session Cookie flow.

3.1 Strategy: Firebase Session Cookies

We eliminate the use of short-lived Bearer ID_TOKENs (1 hour) for session management. Instead, we exchange the ID Token for a long-lived **HttpOnly Session Cookie** (2 weeks), managed by Firebase Admin.

3.2 Infrastructure & Flow

- **Adapter:** FirebaseAuthAdapter wraps firebase-admin.auth().
- **Controller:** AuthController
  - **Endpoint:** POST /auth/login
  - **Input:** { idToken } (from client-side Google Sign-In).
  - **Logic:** Verifies token signature -> Calls createSessionCookie -> Sets Set-Cookie header.
  - **Security:** Cookies are HttpOnly, Secure, and SameSite: Lax.
- **Guard:** SessionAuthGuard
  - Intercepts requests.
  - Reads session cookie.
  - Verifies via admin.auth().verifySessionCookie().
  - Attaches the decoded user to req.user.
- **Decorator:** @CurrentUser() extracts the user object for clean injection into Controllers.

---

4. Module 3: Resume Domain (The PDF Engine)

Objective: Implement the "Smart Resizing" loop using **Tectonic**. This is the core domain

logic.

4.1 Architecture: The Feedback Loop

This logic is isolated in the Domain layer, decoupled from HTTP.

- **Service:** TectonicPdfService implements IPdfGenerator.
- **Dependencies:** handlebars (templating), execa (binary execution).

The Algorithm (Smart Resizing):

1. **Render 1:** Compile Handlebars to .tex with default settings (e.g., 10pt font).
    - **Marker Injection:** The template must include \zsaveposy{start} and \zsaveposy{end}.
2. **Execution:** Execute Tectonic via execa.
    - **Command:** tectonic -X build --keep-intermediates ...
    - **Constraint:** --keep-intermediates is mandatory to preserve the .aux file.
3. **Analysis:** The AuxParserUtil reads the .aux file and parses the zposy coordinates.
    - Calculates ContentHeight = StartY - EndY.
4. **Decision Matrix:**
    - **Overflow:** If ContentHeight > PageHeight (but < 110%): **Shrink**. Reduce \linespread or font size. Re-run Step 2.
    - **Underflow:** If ContentHeight < 50%: **Optimize**.
    - **Fit:** If 90-100%: **Success**.

4.2 Legacy Migration: Heuristic Parser

Objective: Port the exact regex logic from open-resume.

- **Service:** OpenResumeParserAdapter.
- **Logic:**
    - Uses pdf-parse to extract text.
    - Applies regex heuristics (copied from legacy lib/resume-parser) to detect Emails, Phone Numbers, and Section Headers ("EXPERIENCE", "EDUCATION").
    - **Constraint:** Do not use AI here. Use the legacy regex to ensure deterministic behavior matching the current production app.

---

5. Module 4: Search Module (Stateless Vector Store)

Objective: Replace local JSON/Pinecone with **Firestore Native Vector Search**.

5.1 Infrastructure

- **Interface:** IVectorStore.
- **Implementation:** FirestoreVectorStore.
- **Data Structure:**
  - Collection: resumes/{userId}/embeddings
  - Field: embedding_vector (Vector<768>).
  - Metric: **Cosine Similarity**.

5.2 Composite Indexing

Firestore requires a specialized index for vector queries.
Command:

Bash

```
gcloud firestore indexes composite create \
--collection-group=resumes \
--query-scope=COLLECTION \
--field-config field-path=embedding,vector-config='{"dimension":"768", "flat": "{}"}'
```

5.3 Query Logic

The repository utilizes the findNearest method in the Firestore Node.js SDK (v7+).

TypeScript

```
const vectorQuery = collectionRef.findNearest('embedding', queryVector, {
  limit: 10,
  distanceMeasure: 'COSINE'
});
```

---

6. Module 5: Asynchronous Queues

Objective: Offload CPU-intensive Tectonic operations to prevent blocking the Node.js Event Loop.

6.1 Setup

- **Library:** @nestjs/bullmq.
- **Backing:** Redis.

6.2 Sandboxed Processors

Constraint: Tectonic involves spawn (child process). We must not run this in the main API thread.

- **Strategy:** Use BullMQ **Sandboxed Processors**.
- **Implementation:** The processor is defined in a separate file (e.g., pdf.processor.ts). BullMQ runs this file in a **forked Node.js process**, completely isolating the CPU load from the API's HTTP handling.

6.3 Queues defined

1. pdf-generation-queue: Handles the Tectonic resize loop (High CPU).
2. vector-embedding-queue: Handles Vertex AI API calls (High Latency).

---

7. Module 6: CLI Module (Operations)

Objective: Replace ad-hoc scripts with a structured command interface.

7.1 Structure

- **Library:** nest-commander.
- **Entry Point:** src/cli.ts (Distinct from main.ts).
- **Bootstrap:** CommandFactory.run(AppModule).

7.2 Commands

1. **seed**: Imports test data into Firestore.
2. **reindex**: The **Backfill Strategy**.
    - Iterates all resumes in Firestore.
    - Checks if embedding is null.
    - Generates embedding via AiModule.
    - Updates the document.
3. **render-debug <resumeId>**: Runs Tectonic locally for a specific resume and saves the PDF to disk (bypassing HTTP/Queue).

---

8. Module 7: Frontend Integration (Serving Strategy)

Objective: Implement the "Single Container" requirement by serving the Next.js export via NestJS.

8.1 The Setup

The Next.js app (apps/web) is built as a static site. The NestJS app (apps/api) acts as the file server.

8.2 Implementation

- **Module:** ServeStaticModule.
- **File:** src/app.module.ts.
- **Configuration:**
  TypeScript
  ```
  ServeStaticModule.forRoot({
    rootPath: join(__dirname, '..', 'client'), // Path to Next.js output inside Docker
    exclude: ['/api/(.*)'], // CRITICAL: Do not intercept API calls
    serveStaticOptions: {
      fallthrough: true, // Allows Next.js client-side routing (SPA fallback)
    }
  })
  ```

- **Logic:** Requests starting with /api are handled by NestJS Controllers. All other requests look for a file in rootPath. If not found, they serve index.html (for Client-Side Routing).

---

9. Module 8: AI Module (Intelligent Providers)

Objective: Abstraction layer for LLM interactions (distinct from Vector Search).

9.1 Architecture

- **Pattern:** Factory Pattern.
- **Interface:** ILlmProvider (Methods: generateText, embed).
- **Components:**

    - VertexAiProvider: Implementation using @google-cloud/aiplatform.
    - OpenAiProvider: (Optional) Implementation for future swap.
    - AiModule: Uses useFactory to return the correct provider class based on process.env.AI_PROVIDER.
- **Singleton:** The Provider is a singleton, but Context (User data) is passed per method call.

9.2 Usage

This module is injected into the **Search Module** (for embedding generation) and potentially the **Resume Module** (for content improvement suggestions), maintaining separation of concerns.

---

10. Deployment & Build Strategy (The Dockerfile)

Objective: Create the unified artifact. This is the most complex infrastructure component.

10.1 The Multi-Stage Dockerfile

We must merge Node.js (App) and Alpine/Rust (Tectonic).

- **Stage 1: Tectonic Base** (alpine:edge)
    - **Action:** Enable community repo. Install tectonic and fontconfig.
- **Stage 2: Frontend Builder** (node:20-alpine)
    - **Action:** Build apps/web with output: 'export'.
    - **Result:** apps/web/out.
- **Stage 3: Backend Builder** (node:20-alpine)
    - **Action:** Build apps/api.
    - **Result:** dist/apps/api.
- **Stage 4: Final Runtime** (node:20-alpine)
    - **Install:** Runtime deps: tectonic, fontconfig, libssl3, dumb-init.
    - **Copy:**
        - apps/web/out -> /app/client
        - dist/apps/api -> /app/dist
        - node_modules (Production only).

- **Env:** NODE_ENV=production, PORT=8080.
- **Entrypoint:** ["/usr/bin/dumb-init", "--"]
- **CMD:** ["node", "dist/main.js"]

Implementation Roadmap (Agent Instructions)

1. **Scaffold:** Initialize NestJS, ConfigModule (Zod), and Logger (Pino/GCP).
2. **Auth:** Implement FirebaseAuthAdapter, AuthController (Session Cookies), and SessionGuard.
3. **Resume:** Create TectonicService (execa), SmartResizer (aux parsing), and OpenResumeAdapter (regex).
4. **Search & AI:** Create AiModule (Factory), FirestoreVectorStore (Native Vectors), and composite index commands.
5. **Queues:** Setup BullMQ with Sandboxed Processors for PDF generation.
6. **CLI:** Implement reindex and seed commands.
7. **Final Assembly:** Configure ServeStaticModule and the Multi-Stage Dockerfile.