**Career Nexus – End-to-End Project Road**

**PHASE 1 – Project Setup (Completed)**

**Step 1: Monorepo & Structure Setup**

* Created root folder career-nexus/
* Inside it:

common-lib/

user-service/

resume-service/

job-matcher-service/

learning-suggester-service/

gateway-service/

docker-compose.yml

.env

* Each microservice = independent **Spring Boot** project
* common-lib used for shared DTOs, exceptions, and utilities

**Step 2: Maven Configuration**

* Created **pom.xml** in each microservice and one **parent/root pom.xml**
* Defined:
  + groupId: com.careernexus
  + artifactId: career-nexus
  + Parent-child Maven structure
  + Dependencies (Spring Boot Starter, JPA, Web, MySQL)
  + Linked common-lib as dependency in all services

**Step 3: IntelliJ Setup**

* Opened root folder as a **multi-module Maven project**
* Synced Maven (Reload All Maven Projects)
* Created **Run Configurations** for each microservice
* Created one **Compound Configuration** → runs all services together

**Step 4: Database Integration**

* Added MySQL configuration in each service’s application.properties
* Verified DB connection established successfully
* All microservices connected to the same database career\_nexus

**Step 5: Environment Management**

* Added a **.env file** in root:

MYSQL\_ROOT\_PASSWORD=root

MYSQL\_DATABASE=career\_nexus

MYSQL\_USER=nexus\_user

MYSQL\_PASSWORD=nexus\_pass

* Added a **.gitignore** file to exclude:

target/

.idea/

.env

\*.log

* Docker Compose created but not yet used (post-development step)

### ****PHASE 2 – Core Backend Development (Next)****

Now the real application logic begins.

#### Step 6: Common Library (Shared Code)

* Add:
  + DTOs → common data transfer models (UserDTO, JobDTO, etc.)
  + Exceptions → CustomException, GlobalExceptionHandler
  + Utility classes → date formatters, response builders, etc.
* Build and install common-lib locally:

mvn clean install -pl common-lib

(This makes it usable in all services)

#### Step 7: User Service

* **Purpose:** Authentication, authorization, user profiles
* Implement:
  + Models → User, Role, etc.
  + Repository → UserRepository
  + Service → UserService, UserServiceImpl
  + Controller → /api/users, /api/auth
* Add **JWT Authentication** (Spring Security + JWT)
* Expose endpoints for:
  + Register
  + Login
  + Profile management

#### Step 8: Resume Service

* **Purpose:** Manage resumes uploaded by users
* Implement:
  + Models → Resume, Skill, Experience
  + File upload (PDF/DOC)
  + Parse resume for skills (can integrate later with ML/NLP)
  + Endpoints:
    - /api/resumes/upload
    - /api/resumes/{userId}
  + Store file metadata + S3 (optional later)

#### Step 9: Job Matcher Service

* **Purpose:** Recommend suitable jobs based on resume/skills
* Initially, hardcode a few job postings
* Implement:
  + Models → Job, SkillMatch
  + Service → match logic comparing skills to job requirements
  + Controller → /api/jobs/match/{userId}
* Later phase: connect to external APIs (e.g., LinkedIn Jobs, RapidAPI)

#### Step 10: Learning Suggester Service

* **Purpose:** Suggest courses or skills to improve employability
* Implement:
  + Models → Course, SkillGap
  + Logic: find missing skills from resume → suggest learning paths
  + Controller → /api/learning/suggestions/{userId}
* Later: connect to external APIs (Coursera, Udemy, etc.)

### ****Step 11: Notification Service****

* **Purpose:** Centralized service for handling all user notifications — both **in-app** and **email-based**.
* **Implements:**
* Models → Notification, EmailTemplate, UserPreference
* Repository → NotificationRepository
* Service → NotificationService, EmailService
* Controller → /api/notifications, /api/emails

#### Step 12: Gateway Service

* **Purpose:** API Gateway for routing and load balancing
* Use **Spring Cloud Gateway**
* Define routes:
  + /api/users/\*\* → User Service
  + /api/resumes/\*\* → Resume Service
  + /api/jobs/\*\* → Job Matcher Service
  + /api/learning/\*\* → Learning Suggester Service
* Add centralized authentication filter (JWT verification here)

### ****PHASE 3 – Testing & Validation****

#### Step 12: Unit Testing

* Use **JUnit + Mockito**
* Write tests for:
  + Services
  + Controllers
  + Repositories
* Test in isolation (each microservice separately)
* Optional: integration testing with **Spring Boot TestContainers**

#### Step 13: API Testing

* Use **Postman / Thunder Client**
* Verify all routes work end-to-end via the Gateway

### ****PHASE 4 – Docker & Deployment****

#### Step 14: Dockerization

* Write Dockerfile for each service
* Use .env + docker-compose.yml
* Test with:

docker-compose up --build

* Verify all containers (MySQL + all microservices) run properly

#### Step 15: Deployment (optional, if time allows)

* Deploy to AWS:
  + ECS (Fargate)
  + RDS for MySQL
  + S3 for resume storage
* Or deploy locally using Docker Desktop

**PHASE 5 – (Optional) Advanced Features**

| **Feature** | **Description** |
| --- | --- |
| **GraphQL Gateway** | Replace REST aggregation with GraphQL for combined queries |
| **Kafka / RabbitMQ** | Event-driven messaging between services |
| **Redis** | Caching for job recommendations |
| **Service Discovery** | Add Eureka / Consul for dynamic routing |
| **API Docs** | Swagger/OpenAPI for all services |
| **CI/CD** | GitHub Actions for build + test + deploy |