

Repository Layout for Africa Global Talent Platform

To keep the project manageable while supporting future growth, we recommend a **monorepo**. A single repository allows tight coupling between frontend and backend during MVP development, ensures atomic commits across services, and simplifies CI/CD pipelines. The structure below segregates concerns cleanly so that parts can later be extracted into separate microservices or packages.

```
/ (repo root)
├── backend/                # Node.js/Express API and business logic
│   ├── src/
│   │   ├── controllers/   # Route handlers (e.g. authController,
│   │   │   jobController)
│   │   ├── routes/        # API route definitions grouped by domain
│   │   ├── models/        # ORM models or database query modules (e.g.
│   │   │   UserModel, JobModel)
│   │   ├── services/      # Reusable business logic (e.g. emailService,
│   │   │   searchService)
│   │   ├── middleware/    # Authentication, validation, error handling
│   │   ├── utils/         # Helpers and shared utilities
│   │   └── index.js        # Entry point creating the Express app
│   └── migrations/        # SQL migration files managed by a tool like Knex
│       or Flyway
│   ├── tests/             # Unit and integration tests (e.g. using Jest or
│   │   Mocha)
│   ├── package.json
│   ├── tsconfig.json      # If TypeScript is used
│   ├── .eslintrc.js
│   ├── .prettierrc
│   └── README.md
├── frontend/              # React/Next.js web application
│   ├── src/
│   │   ├── pages/         # Next.js pages (e.g. index.js, jobs/[id].js)
│   │   ├── components/    # Reusable UI components (e.g. JobCard,
│   │   │   ProfileForm)
│   │   ├── hooks/         # Custom React hooks (e.g. useAuth, useSearch)
│   │   ├── lib/           # API client utilities and type definitions
│   │   ├── styles/        # CSS/SCSS modules or styled-components
│   │   └── index.tsx
│   ├── public/            # Static assets such as images and favicon
│   └── tests/             # Frontend unit and integration tests (e.g. using
```

```

React Testing Library)
├── package.json
├── tsconfig.json
├── .eslintrc.js
├── .prettierrc
├── README.md
├── infra/                                # Infrastructure as code (optional at MVP)
│   ├── docker/                          # Dockerfiles for backend and frontend
│   │   ├── Dockerfile.backend
│   │   └── Dockerfile.frontend
│   ├── kubernetes/                      # Kubernetes manifests for deployment and services
│   │   ├── deployment-backend.yaml
│   │   ├── deployment-frontend.yaml
│   │   └── ingress.yaml
│   ├── terraform/                       # Terraform modules for cloud infrastructure
│   └── README.md
├── scripts/                             # Helper scripts (e.g. for seeding data, running
tests locally)
├── .env.example                          # Sample environment variables; real secrets are
not committed
├── .gitignore
├── docker-compose.yml                   # For local development: orchestrates frontend,
backend, and database
└── README.md

```

Environment Separation

- **Configuration files:** Each app reads its environment from files such as `.env.development`, `.env.staging`, and `.env.production`. These files are loaded by a configuration library (e.g. `dotenv`) at runtime. The `.env.example` in the repo documents required variables. Real `.env.*` files reside outside version control.
- **Per-environment deployments:** CI/CD pipelines deploy to separate namespaces or stacks (e.g., staging and production clusters). The `infra/` directory contains environment-specific variables or overlays (e.g., `kubernetes/overlays/` for `dev`, `staging`, `prod`).

Secrets Handling

- Never commit secrets to the repository. Use environment variables or secret managers (AWS Secrets Manager, HashiCorp Vault, Kubernetes Secrets) to inject sensitive data (DB credentials, JWT keys, email API keys) at runtime.
- The local `.env.development` file contains dummy credentials for development only. CI/CD pipelines retrieve secrets from the appropriate secret store.

Linting & Formatting

- Adopt **ESLint** for JavaScript/TypeScript and **Prettier** for consistent code style across `frontend/` and `backend/`. Linting rules are defined in `.eslintrc.js` and formatting rules in `.prettierrc`. Run linters in CI to enforce quality.
- Use **Husky** and **lint-staged** to run linters and tests on pre-commit.

Testing Structure

- Place tests alongside source code or in dedicated `tests/` directories within `frontend/` and `backend/`. Use Jest or Mocha for backend tests and React Testing Library for frontend tests.
- Support integration tests that spin up the backend and a test database (e.g., using Docker) via `docker-compose`.

Scalability Considerations

- **Domain isolation:** Grouping backend code by domain (`controllers/`, `services/`, `models/`) makes it easier to extract domains into separate microservices later. When job ingestion or AI services grow, they can be moved into their own packages without disrupting other modules.
- **Modular infrastructure:** The `infra/` directory uses Terraform and Kubernetes manifests that can be composed and reused for new services. Each environment can override variables or resources without duplicating code.
- **CI/CD pipeline:** A single pipeline can build, test, and deploy both frontend and backend. As the platform evolves, workflows can branch (e.g., separate pipelines for AI components) but still share common configuration.

This structure keeps the codebase organised, supports multiple teams working concurrently, and provides a clear path from monolithic MVP to a more modular architecture.
