# Current State & Next steps of the Application

Your project is currently a **robust and production-ready backend foundation**.

**Core Backend:** A Java 21 Spring Boot application that serves as the "engineering backbone." It handles all business logic and data persistence.

**Database:** A PostgreSQL database managed by Docker Compose.

**User Profile Service:** A RESTful API that allows for the creation and management of user profiles.

**Testing:** A comprehensive testing pipeline is in place. Unit tests use **Mockito**, while integration tests use **Testcontainers** to spin up a real PostgreSQL database for a reliable and isolated testing environment.

**Code Coverage:** **JaCoCo** is configured to generate code coverage reports, ensuring the quality of your test suite.

**Build Automation:** **Gradle** manages dependencies, builds the project, and runs tests.

### The Backend Next steps

## Security

The goal is to secure the API endpoints, verify user identity, and protect credentials.

| Checklist Item | Option 1 (Industry Standard) | Option 2 (Modern/OAuth 2.0 Focused) | Option 3 (Simplicity/Spring Native) |
| --- | --- | --- | --- |
| **Implement JWT Auth with Spring Security** | Use **Spring Security's Filter Chain** with a **Stateless Session Policy** and a **OncePerRequestFilter** to process the JWT token on every request. | Integrate with an **OAuth 2.0 Provider** like **Keycloak** or **Auth0**. Spring Security can then validate the OAuth tokens (which are often JWTs) directly. | Use **Spring Security 6's Lambda DSL** for concise configuration, leveraging BearerTokenAuthenticationFilter to streamline JWT processing. |
| **Add Password Hashing** | Use **BCrypt** (BCryptPasswordEncoder). This is the standard, secure, and widely accepted choice in the Spring ecosystem. | Use **Argon2** (via a custom PasswordEncoder). It's considered computationally superior and more resistant to GPU-based attacks than BCrypt, though it requires an extra dependency. | Use **Scrypt** (via SCryptPasswordEncoder). It uses more memory, making it expensive for attackers to run on parallel hardware. |
| **Configure Role-Based Authorization** | Use **@PreAuthorize** annotations directly on controller methods (e.g., @PreAuthorize("hasRole('ADMIN')")). | Implement **Voter-based Authorization** for more complex, dynamic rules that require checking multiple conditions (e.g., checking user role AND resource ownership). | Utilize **Method Security Expression Language (SpEL)** within the configuration to secure entire controller classes or endpoint patt |

## Observability

The goal is to provide deep insight into the application's health, performance, and behavior in a production environment.

| Checklist Item | Option 1 (Spring Standard) | Option 2 (Open Standard/Cloud Native) | Option 3 (Log Aggregation Focus) |
| --- | --- | --- | --- |
| **Implement Structured Logging** | Use **SLF4J with Logback** and a **JSON-formatted appender** (like **Logstash Logback Encoder** or Logback−json−log−format). This is the simplest way to get machine-readable logs. | Use **Log4j2** (an alternative logging framework) with a **JSON layout** to structure logs. | Use **Spring Cloud Sleuth** (or its successor, Micrometer Tracing) to automatically add **Trace IDs and Span IDs** to every log entry, linking requests across your services. |
| **Add Metrics Collection** | Use **Spring Boot Actuator** and **Micrometer** (the standard for metrics instrumentation). Micrometer includes a **Prometheus registry** for easy scraping. | Use **OpenTelemetry (OTEL)**. It's an open standard that combines metrics, traces, and logs, offering a more future-proof and vendor-neutral solution. | Use **Dropwizard Metrics** (a classic library) with a **Prometheus reporter** if you want more fine-grained control outside of the Micrometer ecosystem. |
| **Create a Grafana Dashboard** | Configure **Grafana** to use **Prometheus** as a data source. Import a **pre-built JVM/Spring Boot dashboard template** and customize it for your specific metrics (e.g., latency of the /users endpoint). | Configure **Grafana Loki** as a data source alongside Prometheus. Loki is optimized for storing and querying **structured logs**, allowing you to link performance drops (from Prometheus) directly to error logs (from Loki). | Use **New Relic** or **Datadog** (Commercial APM tools). They provide fully automated dashboards and tracing out-of-the-box, significantly reducing setup time, though at a cost. |

Export to Sheets

## API Documentation

The goal is to automatically generate up-to-date and interactive documentation for your API.

| Checklist Item | Option 1 (Modern Standard) | Option 2 (Lightweight/Code-Focused) | Option 3 (Cloud-Native/Async) |
| --- | --- | --- | --- |
| **Add OpenAPI/Swagger** | Use **SpringDoc OpenAPI**. This is the recommended, modern library for Spring Boot. It uses annotations to automatically generate the OpenAPI JSON/YAML spec and serves the **Swagger UI** for an interactive sandbox. | Use **Microspec** or **Springfox**. Springfox is older and less maintained but was a long-time standard. Microspec is newer and focuses on minimal dependencies. | Use **AsciiDoc/Asciidoctor**. Write your documentation in AsciiDoc and use a tool like **Spring REST Docs** during testing to automatically generate code snippets and documentation. This method ensures documentation is always accurate because it's tied to passing tests. |