Monolith Application using Spring Boot (Java21)

Banking Application developed using Spring Boot (java 21) & MySQL (DB).

Spring Initializr (Website – to build Spring boot based application) – <https://start.spring.io>

Spring boot framework supports 3 JVM based lang (JAVA, Kotlin, Groovy)

Banking Application.

Monolith – Single Deployable unit consist of all the modules in your application.

1. Customers / Users
2. Account
3. Transaction

It has 3 layers.

1. Front End ( HTML/CSS/JS, React/Angular/Vue/ Spring Web based application) [Presentation/View Layer]
2. Back End ( All business logics ) [Business Layer]
3. DB layer (Persistence Layer) [h2- in memory database, SQL /No-SQL DB] – Oracle/Postgres/SQL Server/My SQL – Redis/MongoDB

Must Have Packages

Basepacakage 🡪 Starter Class (A class with main method) – Entry point of your application

1. Model/entity/beans -- This is the Java version of DB Tables. – JPA annotations here [@Entity, @Table, @Column, @Id, @GeneratedValue] – spring-data-jpa (dependency) -- validation – NotNull, Unique, Size, pattern (Reg-ex)
2. Repository – DAO (Data Access Object) – JpaRespository (Interface) – Five Basic CRUD operation – findAll(), findById(id), save(), deleteById()
3. Service – All the business logic.
4. Controller – RestController – [GET,POST,PUT & DELETE]

Good to Have packages

1. Config – security config, swagger config
2. Util – Utility classes
3. Exception – Custom Exception
4. DTO – Data Transfer Object (Temporary way of sending data – it’s not persistent data)

Dependencies added in Spring Boot project

1. Spring Web – Creating REST based web services (http) – Embedded Tomcat server (Web Server) – 8080 [port] server.port=8082 - @RestController
2. SpringBoot DevTools –
3. Lombok – Helps to avoid adding boiler plate codes manually. -- @Data, @AllArgsConstructor, @NoArgsConstructor
4. Spring-data-jpa [JPA – implementation using Hibernate] – JDBC [Java Persistence API – Specification – Hibernate, EclipseLink, MyBatis] – ORM framework [Object Relational Mapping]
5. H2/MySQL – RDBMS to use for Storing the Data [2mb/Community version installed locally]

External Dependency – Swagger – API Documentation.

Maven - POM.xml (Project Object Model) - <project> - backbone of any maven based project.

Project Mgmt tool . – To install, Manage, Build, Test, Deploy, run, compile, install

Mvnw – mvnwrapper

“mvn spring-boot:run” -- Maven command to run spring-boot based application

Audit Fields/Audit Columns – CreatedAt, CreatedBy, UpdatedAt, UpdatedBy.

Single Spring Boot Application – (All the modules Account, Customer, Transaction) – Tightly coupled. Only one DB. All the tables are present.

1. We will divide the functionality based on Domain Objects
2. Separate spring boot application for each service [customer-service, account-service, transaction-service] – 3 Different spring boot project.
3. Spring cloud discover / service discovery [Design Pattern] – Where all the service Register to a centralized server – Load Balancing, Routing. – Eureka Discovery server.

Create a Spring boot application as Eureka Server – Cloud Discovery/ Service Discovery.

(Eureka/Zoo Keeper/Consul) – Service Discovery – Netflix (OSS – Open Source Specification)

1. Open STS /Spring Initializr
2. Add all the details (java version, packaging, maven, com.example, eureka-server, com.example.eureka, -- Eureka Server)
3. Add @EnableEurekaServer annotation to the starter class.

Add following properties in application.properties file eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

server.port=8761

1. Run the application.

In this example there are 2 DBs are used. (h2 – Embedded in-memory, MySQL – RDBMS server)

Datasource :

RequestMapping (Generic Mapping)

Monolithic Application – Banking App (Account, Customer, Transaction)

Netflix OSS --

# Building Microservices with Spring Boot

## 1. Introduction to Microservices & Spring Boot

* **What are Microservices?**
  + Definition, benefits: scalability, independent deployment, resilience.
  + Real-world example: E-commerce system divided into order, user, payment, product, and notification services.
* **Why Spring Boot for Microservices?**
  + Fast setup, embedded servers, opinionated defaults, production readiness, integration with Spring Cloud.

## 2. Setting Up Your First Spring Boot Microservice

* **Prerequisites**
  + Java 17+ (or 21+), Maven or Gradle, Spring Initializr.
* **Step-by-step:**
  + Use [Spring Initializr](https://start.spring.io/) to generate a new Spring Boot project.
    - Choose dependencies: Spring Web, Spring Boot Actuator.
  + Project structure and key files explained (pom.xml, src directory, Main Application class).
  + Example: Implement a simple Product Service REST API.
* **Running and Testing**
  + How to run with mvn spring-boot:run or through IDE.
  + Access endpoints using Postman/cURL/Swagger.
* **Packaging & Deployment**
  + Build a JAR, First step for Docker containerization.

Spring Cloud: Service Discovery & Config

## 3. Service Discovery with Spring Cloud Eureka

* **Why Service Discovery?**
  + No hardcoded addresses, support for scaling and high availability.
* **Eureka Overview and Concepts**
  + Eureka Server, Eureka Client, Registry, Heartbeats, Health Checks, Load Balancing.
* **Hands-on: Setting Up Eureka**
  + Create a Eureka Server (add spring-cloud-starter-netflix-eureka-server).
  + Enable with @EnableEurekaServer.
  + Run Eureka at port 8761.
* **Registering Microservices with Eureka**
  + Add spring-cloud-starter-netflix-eureka-client to other microservices.
  + Configurations in application.yml or application.properties.
  + Explorer Eureka dashboard.
* **Discussion: Eureka Deprecation and Alternatives**
  + Present status in 2025: Eureka not actively maintained—consider Consul or Kubernetes for cloud-native scenarios.

## 4. Centralized Configuration with Spring Cloud Config Server

* **Why Centralized Configuration?**
  + Manage all configs in one place, easier environment management.
* **Setting Up Config Server**
  + spring-cloud-config-server dependency.
  + Enable with @EnableConfigServer.
  + Store configs in a Git repo, file system, or database.
* **Connecting Clients**
  + Add client dependencies in microservices.
  + Retrieve config on startup and refresh it at runtime.
  + Example: Move database URLs, feature toggles, secrets out of code.

Inter-Service Communication

## 5. Synchronous Communication: REST & gRPC

* **When to use Synchronous?**
  + Real-time responses needed (e.g., order service needs user details).
* **Techniques**
  + RESTful HTTP: Standard, easy to use.
  + gRPC: High performance, binary protocol, schema-driven.
* **Hands-on: REST API Communication**
  + Use Spring’s RestTemplate or, preferably, Feign for declarative REST clients:
    - Example: Annotate Java interface with @FeignClient, simple method signatures.
* **Demo: Service-to-Service REST call via Feign**.

## 6. Asynchronous Communication: Messaging & Event Streams

* **When to use Asynchronous?**
  + Decoupling, scalability, resilience (e.g., notification service reacts to events).
* **Tools and Patterns**
  + Message Brokers: RabbitMQ, Apache Kafka.
  + Event-driven communication: Publication/subscription, message queues.
* **Hands-on: Sending Events (Kafka Example)**
  + Add Kafka or messaging dependencies.
  + Write code to publish and consume events.

# Implementing Fault Tolerance

## 7. Resilience Patterns in Spring Boot

* **Why Fault Tolerance?**
  + Avoid cascading failures, ensure robust service communication.
* **Circuit Breaker Pattern**
  + Concept and state transitions: Closed, Open, Half-Open.
  + Benefits: Prevent system overload, fallback logic.
* **Implementing Circuit Breaker**
  + Legacy: Hystrix (no longer maintained).
  + Modern: Resilience4j—lightweight, Java 8+, seamless Spring Boot integration.
* **Hands-on: Resilience4j Example**
  + Add resilience4j-spring-boot2 dependency.
  + Annotate methods with @CircuitBreaker, configure failover and thresholds.
  + Monitor circuit breaker metrics.
* **Other Patterns**
  + Retry: Automatically re-invoke calls if they fail.
  + Bulkhead: Limit concurrent calls, isolate failures.
  + Timeouts: Avoid slow dependencies holding up requests.
* **Demo**
  + Example with Resilience4j for Circuit Breaker, Retry, Bulkhead, and TimeLimiter.

Additional Sections

## 8. Best Practices & Recommendations

* Proper microservice design: DDD, clear boundaries, small and single-responsibility.
* Secure communication: OAuth2, JWT for authentication.
* Monitoring and alerting: Spring Boot Actuator, Prometheus, Grafana.
* Automated deployment, CI/CD introduction.