# 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.
* **Packaging & Deployment**
  + Build a JAR, steps 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.