Spring Boot

- **Spring Boot Basics**
- **Spring Boot Core Features**
- **Spring Boot REST Development**
- **Spring Boot with Databases**
- **Spring Boot Security**
- **Spring Boot Microservices**
- **Spring Boot Messaging & Event-Driven Systems**
- **Spring Boot Advanced APIs**
- **Spring Boot Testing**
- **Performance Tuning & Monitoring**
- **Advanced Topics & Deployment**
- **Projects**

Spring Boot Basics

Spring Boot is a framework built on top of the Spring Framework to simplify application development. Traditional Spring applications required lots of XML configuration, manual server deployment, and careful dependency management. Spring Boot changes that by offering opinionated defaults, starter dependencies, and an embedded server, so you can focus on writing business logic instead of infrastructure code.

Spring Boot is designed with three main goals:

1. **Faster Development** – create production-ready apps quickly.
2. **Minimal Configuration** – rely on sensible defaults.
3. **Embedded Servers** – run applications as standalone executables without external servers.

Difference between Spring and Spring Boot

Feature	**Spring**	**Spring Boot**
Configuration	Requires XML or Java-based configuration	Uses auto-configuration, minimal setup
Dependency Management	Manual, prone to version conflicts	Starter dependencies manage versions
Server Deployment	Needs external server like Tomcat/Jetty	Embedded servers (Tomcat, Jetty, Undertow)
Project Setup	Time-consuming	Easy with Spring Initializr
Focus	Infrastructure + Business Logic	Purely Business Logic (infra handled automatically)

Starter Dependencies

One of Spring Boot's best features is its starter dependencies. A starter is a curated set of dependencies for a particular functionality. Instead of manually adding individual libraries, you just add one starter and Spring Boot brings in everything you need.

Examples:

- spring-boot-starter-web → for building web apps / REST APIs (includes Spring MVC, Tomcat, JSON).
- spring-boot-starter-data-jpa → for database access with JPA + Hibernate.
- spring-boot-starter-security → for authentication and authorization.
- spring-boot-starter-test → for testing (JUnit, Mockito, Spring Test).

This avoids dependency hell and ensures version compatibility.

Auto Configuration & Annotations

Spring Boot's magic lies in **auto-configuration**. Based on the dependencies you add, it configures beans automatically. For example, if you include spring-boot-starter-web, it auto-configures a DispatcherServlet and Tomcat for you.

The key annotation is **@SpringBootApplication**, which is a combination of:

- **@Configuration** – marks this class as a configuration source.
- **@EnableAutoConfiguration** – enables auto-configuration.
- **@ComponentScan** – scans the package for Spring components.

Example main application:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
}
```

Project Structure

A typical Spring Boot project (Maven-based) looks like this:

```
demo-project/
 ├── src/
 │   ├── main/
 │   │   ├── java/
 │   │   │   └── com/example/demo/
 │   │   │       ├── DemoApplication.java   // Main class
 │   │   │       └── controller/
 │   │   │           └── HelloController.java
 │   │   └── resources/
 │   │       ├── application.properties     // Configurations
 │   │       └── static/                    // Static resources (html,
css, js)
 │   └── test/
 │       └── java/                          // Unit & integration tests
 └── pom.xml                                // Dependencies
```

- **src/main/java** → business logic, controllers, services, repositories.
- **src/main/resources** → config and static assets.
- **application.properties** → key-value configuration (server port, DB connection, etc.).

- **pom.xml** → manages dependencies and plugins.

Mini Example

Create a Spring Boot project with spring-boot-starter-web using Spring Initializr. Then add this controller:

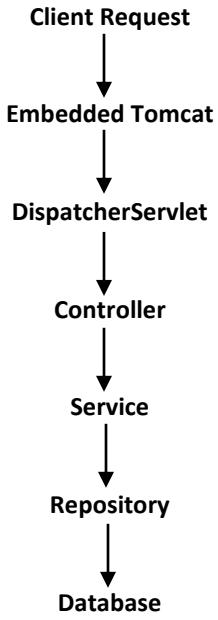
```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
   @GetMapping("/hello")
   public String sayHello() {
      return "Hello, Spring Boot!";
   }
}
```

Run the application (DemoApplication.main). Visit http://localhost:8080/hello in a browser. You'll see: Hello, Spring Boot!

That's a complete running Spring Boot app with just one dependency and almost zero configuration.

Flow Diagram – How a Spring Boot Request Works



Spring Boot auto-configures the dispatcher, controller mapping, and server, so you only focus on writing business code.

Spring Boot Core Features

Spring Boot provides a set of core features that make application development smooth and customizable. These include flexible configuration, environment-specific profiles, powerful logging support, and utilities like CommandLineRunner and ApplicationRunner to run code during application startup.

Configuration (application.properties / application.yml)

Spring Boot applications are driven by configuration files. By default, these files are located in src/main/resources.

Two formats are supported:

1. **application.properties** – simple key-value pairs.
2. **application.yml** – structured format, easier to manage hierarchical data.

Example using application.properties:

```
server.port=8081
spring.application.name=demo-app
```

The same in application.yml:

```
server:
  port: 8081
spring:
  application:
    name: demo-app
```

When you run the application, it will start on port 8081 instead of the default 8080.

Profiles (dev, test, prod)

Spring Boot supports **profiles** to separate environment-specific configurations. This avoids hardcoding values for development, testing, or production.

Example – in application-dev.properties:

```
server.port=8081
spring.datasource.url=jdbc:h2:mem:testdb
```

In application-prod.properties:

```
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/proddb
spring.datasource.username=root
spring.datasource.password=secret
```

To activate a profile, specify it in the main application.properties:

```
spring.profiles.active=dev
```

Or pass it as a command-line argument:

```
java -jar demo-app.jar --spring.profiles.active=prod
```

This allows the same app to run in multiple environments without changing code.

Logging in Spring Boot

Logging is built-in using **SLF4J** with **Logback** as the default implementation. You can configure the log level in application.properties:

```
logging.level.root=INFO
logging.level.com.example.demo=DEBUG
logging.file.name=app.log
```

This means:

- Root logging level is INFO.
- For the package com.example.demo, the level is DEBUG.
- Logs are written to app.log.

Mini example:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class LogController {
   private static final Logger logger = LoggerFactory.getLogger(LogController.class);
   @GetMapping("/log")
   public String logExample() {
      logger.info("INFO log message");
      logger.debug("DEBUG log message");
      return "Check logs for output!";
   }
}
```

CommandLineRunner & ApplicationRunner

Both interfaces let you run code **after the application starts**. They're often used for initializing data or running startup checks.

Mini example with CommandLineRunner:

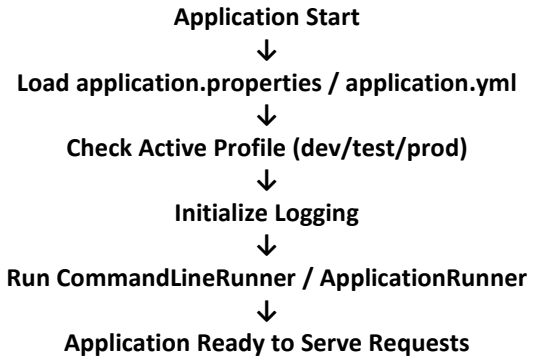
```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
@Component
public class StartupRunner implements CommandLineRunner {
   @Override
   public void run(String... args) throws Exception {
      System.out.println("Application started! Running startup logic...");
   }
}
```

Mini example with ApplicationRunner:

```
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;
@Component
public class AppRunner implements ApplicationRunner {
   @Override
   public void run(ApplicationArguments args) {
      System.out.println("ApplicationRunner executed with arguments: " + args);
   }
}
```

When you start the app, these will execute automatically after the Spring context is fully loaded.

Diagram – Startup Flow with Core Features



Spring Boot REST Development

Spring Boot makes it very simple to build REST APIs. By using the spring-boot-starter-web dependency, you get Spring MVC with an embedded Tomcat server ready to expose endpoints. REST (Representational State Transfer) is the standard architecture for web services where communication happens via HTTP methods like GET, POST, PUT, and DELETE.

Building REST APIs with Spring Boot

A REST controller is created by annotating a class with @RestController. Each endpoint is mapped using **@GetMapping, @PostMapping, @PutMapping, or @DeleteMapping.**

Mini example:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloRestController {
   @GetMapping("/api/hello")
   public String sayHello() {
      return "Hello from REST API!";
```

```
    }
}
```

When you run the application, visiting http://localhost:8080/api/hello will return a plain text response.

Request Mapping & CRUD Operations

Let's say we want to manage a list of students. A typical CRUD API would look like this:

```
import org.springframework.web.bind.annotation.*;
import java.util.*;
@RestController
@RequestMapping("/api/students")
public class StudentController {
    private Map<Integer, String> students = new HashMap<>();

    @GetMapping
    public Collection<String> getAllStudents() {
        return students.values();
    }

    @GetMapping("/{id}")
    public String getStudent(@PathVariable int id) {
        return students.getOrDefault(id, "Student not found");
    }

    @PostMapping
    public String addStudent(@RequestParam int id, @RequestParam String name) {
        students.put(id, name);
        return "Student added successfully!";
    }

    @PutMapping("/{id}")
    public String updateStudent(@PathVariable int id, @RequestParam String name) {
        students.put(id, name);
        return "Student updated successfully!";
    }

    @DeleteMapping("/{id}")
    public String deleteStudent(@PathVariable int id) {
        students.remove(id);
        return "Student deleted successfully!";
    }
}
```

This example covers GET (all and by ID), POST (create), PUT (update), and DELETE (remove).

Exception Handling

Spring Boot provides global exception handling using @ControllerAdvice and @ExceptionHandler.

Example:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
@ControllerAdvice
public class GlobalExceptionHandler {
   @ExceptionHandler(Exception.class)
   public ResponseEntity<String> handleException(Exception ex) {
      return new ResponseEntity<>("Error occurred: " + ex.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
   }
}
```

Now, if any exception occurs in the controller, this handler will send a structured response instead of a stack trace.

Validation

Spring Boot supports bean validation with the **javax.validation** API (spring-boot-starter-validation dependency).

Example with @Valid and constraints:

```
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
public class Student {
   @NotBlank(message = "Name is mandatory")
   @Size(min = 2, max = 20, message = "Name must be 2–20 characters")
   private String name;

   // getters & setters
}
```

Controller:

```
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/validate")
public class ValidationController {
   @PostMapping
   public String validateStudent(@RequestBody @Validated Student student) {
      return "Valid student: " + student.getName();
   }
}
```

If invalid data is sent, Spring Boot automatically returns a 400 Bad Request with error details.

Versioning REST APIs

Sometimes APIs evolve over time, and you may need multiple versions. Common strategies:

1. **URI Versioning**

```
@GetMapping("/api/v1/greet")
public String greetV1() { return "Hello v1"; }

@GetMapping("/api/v2/greet")
public String greetV2() { return "Hello v2"; }
```

2. **Request Parameter Versioning**

```
@GetMapping(value = "/api/greet", params = "version=1")
public String greetParamV1() { return "Hello v1"; }
@GetMapping(value = "/api/greet", params = "version=2")
public String greetParamV2() { return "Hello v2"; }
```

3. **Header Versioning**

```
@GetMapping(value = "/api/greet", headers = "X-API-VERSION=1")
public String greetHeaderV1() { return "Hello v1"; }
@GetMapping(value = "/api/greet", headers = "X-API-VERSION=2")
public String greetHeaderV2() { return "Hello v2"; }
```

Flow Diagram – REST API Lifecycle

Client Request → DispatcherServlet → Controller (mapped method)

↓

Validation → Service Layer → Repository/DB

↓

Response returned to client

Spring Boot with Databases

Databases are at the heart of most applications. Spring Boot provides seamless integration with relational databases through **Spring Data JPA**, JDBC, and other connectors.

Spring Data JPA with Spring Boot

- **Spring Data JPA** is a layer over **Hibernate** (JPA implementation).
- It reduces boilerplate code by providing **Repository interfaces** with ready-made CRUD operations.

Example: Defining an Entity

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
@Entity
public class Student {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
    private String name;
    private String email;

    // Getters and setters
}
```

Repository Interface:

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface StudentRepository extends JpaRepository<Student, Long> {
    Student findByEmail(String email);
}
```

Now you can perform CRUD without writing SQL — thanks to Spring Data JPA.

H2 Database (in-memory DB for testing)

- H2 is a lightweight, in-memory database.
- Useful for development and unit testing.

Add dependency in pom.xml:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

application.properties:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
```

Access H2 console at: http://localhost:8080/h2-console

MySQL/PostgreSQL Integration

For production, you typically use relational DBs like **MySQL** or **PostgreSQL**.

Dependency for MySQL:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

application.properties (MySQL):

```
spring.datasource.url=jdbc:mysql://localhost:3306/springdb
spring.datasource.username=root
```

```
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Dependency for PostgreSQL:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

application.properties (Postgres):

```
spring.datasource.url=jdbc:postgresql://localhost:5432/springdb
spring.datasource.username=postgres
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Pagination & Sorting

Spring Data JPA provides built-in support for **paging** and **sorting**.

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/students")
public class StudentController {
  private final StudentRepository studentRepository;
  public StudentController(StudentRepository studentRepository) {
    this.studentRepository = studentRepository;
  }

  @GetMapping("/paginated")
  public Page<Student> getStudents(@RequestParam int page, @RequestParam int size) {
    Pageable pageable = PageRequest.of(page, size, Sort.by("name").ascending());
    return studentRepository.findAll(pageable);
  }
}
```

- Request: /api/students/paginated?page=0&size=5
- Returns first 5 students sorted by name.

Flyway / Liquibase for Database Migration

When applications grow, managing database schema manually is difficult. **Flyway** and **Liquibase** automate schema versioning.

Flyway Setup

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
</dependency>
```

- Place SQL migration files in src/main/resources/db/migration.
- Naming format: V1__create_student_table.sql, V2__add_email_column.sql

Liquibase Setup

```
<dependency>
  <groupId>org.liquibase</groupId>
  <artifactId>liquibase-core</artifactId>
</dependency>
```

- Configured via db.changelog-master.yaml (or XML/JSON).
- Example YAML:

```
databaseChangeLog:
 - changeSet:
    id: 1
    author: mazid
    changes:
     - createTable:
        tableName: student
        columns:
         - column:
            name: id
            type: BIGINT
            autoIncrement: true
            constraints:
              primaryKey: true
         - column:
            name: name
            type: VARCHAR(255)
```

Flow Diagram – Database Layer

Controller → Service Layer → Repository (Spring Data JPA)
↓
Database (H2/MySQL/PostgreSQL)
↓
Flyway / Liquibase manages schema evolution

Spring Boot Security

Spring Security provides a comprehensive security framework for authentication, authorization, and protection against common vulnerabilities. Spring Boot makes it easier by providing **auto-configuration** and starter dependencies.

Spring Security with Spring Boot

Dependency in pom.xml:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

- By default, Spring Security applies **basic authentication**.
- A default user is created with username: user and a random password (printed in logs).

You can set your own credentials in application.properties:

```
spring.security.user.name=admin
spring.security.user.password=admin123
```

Now any request to your app will require authentication.

Authentication & Authorization

- **Authentication** → Verifying who you are.
- **Authorization** → Verifying what you can do.

Custom Authentication Example (In-memory users):

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    UserDetails user1 = User.withUsername("user")
        .password("{noop}password") // {noop} means no password encoder
        .roles("USER")
        .build();

    UserDetails admin = User.withUsername("admin")
        .password("{noop}admin123")
        .roles("ADMIN")
        .build();

    return new InMemoryUserDetailsManager(user1, admin);
  }
```

```
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeHttpRequests()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/user/**").hasAnyRole("USER","ADMIN")
                .anyRequest().authenticated()
            .and()
            .httpBasic();

        return http.build();
    }
}
```

- /admin/** → Only accessible by users with ADMIN role.
- /user/** → Accessible by both USER and ADMIN.

Role-based Security

Spring Security uses **roles and authorities** to restrict access.

Example:

```
@GetMapping("/admin/dashboard")

public String adminDashboard() {
    return "Welcome Admin!";
}

@GetMapping("/user/profile")

public String userProfile() {
    return "Welcome User!";
}
```

- If logged in as admin, you can access both /admin/dashboard and /user/profile.
- If logged in as user, you can only access /user/profile.

JWT / OAuth2 Implementation

For stateless authentication in REST APIs, we use **JWT (JSON Web Tokens)**.

Flow of JWT Authentication:

1. User sends credentials → /login
2. Server validates and generates JWT token
3. Client stores token (in localStorage or cookies)
4. Each request includes Authorization: Bearer <token> header
5. Server validates JWT before processing the request

JWT Dependency:

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
```

JWT Utility Example:

```
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import java.util.Date;

public class JwtUtil {
  private static final String SECRET_KEY = "secret123";

  public static String generateToken(String username) {
    return Jwts.builder()
        .setSubject(username)
        .setIssuedAt(new Date())
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60)) // 1 hour
        .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
        .compact();
  }
}
```

Use Case:

- On successful login → return JWT token.
- Every secured API → validate JWT.

OAuth2 (Optional)

For social logins (Google, Facebook, GitHub, etc.), Spring Boot integrates with **OAuth2**.

Dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

Configuration is added in application.yml for providers like Google.

Security Flow Diagram

Client → Login Request → Spring Security Auth Manager → UserDetailsService
↓
JWT Generated (if valid) → Returned to Client
↓
Client uses JWT in Authorization header → Accesses Protected APIs

Spring Boot Microservices

Microservices is an architecture style where applications are built as a set of small, independent services that communicate over the network (often REST APIs). Spring Boot, along with Spring Cloud, provides a rich ecosystem for building these systems.

REST Communication between Microservices

Microservices talk to each other using **REST APIs**.

Example Setup:

- **Order Service** → manages orders
- **Payment Service** → handles payments

REST call using RestTemplate:

```
import org.springframework.web.client.RestTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class OrderController {
   private final RestTemplate restTemplate = new RestTemplate();
   @GetMapping("/order")
   public String placeOrder() {
      String paymentResponse = restTemplate.getForObject("http://localhost:8082/payment", String.class);
      return "Order placed. Payment Status: " + paymentResponse;
   }
}
```

REST call using WebClient (Reactive):

```
import org.springframework.web.reactive.function.client.WebClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class OrderController {
   private final WebClient webClient = WebClient.create("http://localhost:8082");
   @GetMapping("/order")
   public String placeOrder() {
      String response = webClient.get()
```

```
                .uri("/payment")
                .retrieve()
                .bodyToMono(String.class)
                .block();
        return "Order placed. Payment Status: " + response;
    }
}
```

Spring Cloud Config

Manages **centralized configuration** for all microservices.

Steps:

1. Create a Spring Cloud Config Server

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

2. Enable server in main class:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApp {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApp.class, args);
    }
}
```

3. Store configurations in a Git repo or local folder.
4. Clients fetch configs via application.properties:
5. spring.config.import=optional:configserver:http://localhost:8888

Eureka Server (Service Registry)

Microservices register themselves in **Eureka** so others can discover them.

Eureka Server Setup:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApp {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApp.class, args);
    }
}
```

Client (microservice) dependency:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
spring.application.name=order-service
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

Load Balancing with Ribbon

Ribbon provides **client-side load balancing** across multiple instances of a service.

Example:

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
  return new RestTemplate();
}
```

Now when you call http://payment-service/payment, Ribbon automatically balances between multiple instances of payment-service.

API Gateway (Spring Cloud Gateway)

Gateway acts as a **single entry point** for all clients.

Dependency:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Config in application.yml:

```
spring:
 cloud:
  gateway:
   routes:
    - id: order-service
      uri: http://localhost:8081
      predicates:
       - Path=/order/**
    - id: payment-service
      uri: http://localhost:8082
      predicates:
       - Path=/payment/**
```

Now requests to /order/** and /payment/** are routed via the gateway.

Resilience4j / Circuit Breaker

Protects microservices from **failures** like service downtime or slow responses.

Dependency:

```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

Usage Example:

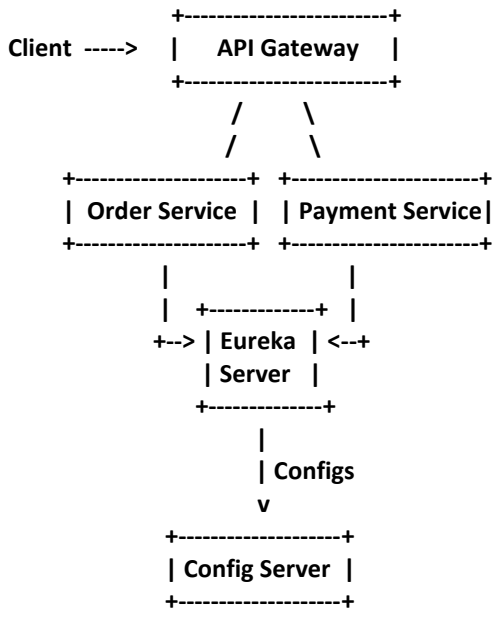
```
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

  @GetMapping("/order")
  @CircuitBreaker(name = "paymentService", fallbackMethod = "paymentFallback")
  public String placeOrder() {
    // Simulate call to Payment Service
    throw new RuntimeException("Payment Service Down!");
  }

  public String paymentFallback(Exception ex) {
    return "Payment service is unavailable. Please try later.";
  }
}
```

Microservices Architecture Diagram (Simplified)



Spring Boot Messaging & Event-Driven Systems

Modern applications often rely on **messaging systems** to handle large volumes of events, decouple services, and improve resilience. Spring Boot provides first-class support for popular brokers like **Kafka** and **RabbitMQ**.

Spring Boot with Kafka

Kafka is a distributed event-streaming platform. In Spring Boot, we use **spring-kafka** starter.

Dependency:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

Producer Example:

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/kafka")
public class KafkaController {
  private final KafkaTemplate<String, String> kafkaTemplate;

  public KafkaController(KafkaTemplate<String, String> kafkaTemplate) {
    this.kafkaTemplate = kafkaTemplate;
  }

  @PostMapping("/publish/{message}")
  public String publish(@PathVariable String message) {
    kafkaTemplate.send("my-topic", message);
    return "Message published: " + message;
  }
}
```

Consumer Example:

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaConsumer {

  @KafkaListener(topics = "my-topic", groupId = "group_id")
  public void consume(String message) {
    System.out.println("Received: " + message);
  }
}
```

Spring Boot with RabbitMQ

RabbitMQ is a lightweight message broker based on **AMQP protocol**.

Dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Producer Example:

```
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/rabbit")
public class RabbitController {
  private final RabbitTemplate rabbitTemplate;

  public RabbitController(RabbitTemplate rabbitTemplate) {
    this.rabbitTemplate = rabbitTemplate;
  }

  @PostMapping("/publish/{message}")
  public String publish(@PathVariable String message) {
    rabbitTemplate.convertAndSend("my-exchange", "my-routing-key", message);
    return "Message sent: " + message;
  }
}
```

Consumer Example:

```
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;
@Service
public class RabbitConsumer {
  @RabbitListener(queues = "my-queue")
 public void receive(String message) {
    System.out.println("Received: " + message);
  }
}
```

Event-Driven Microservices

Instead of synchronous REST calls, microservices can communicate via events.

Example flow:

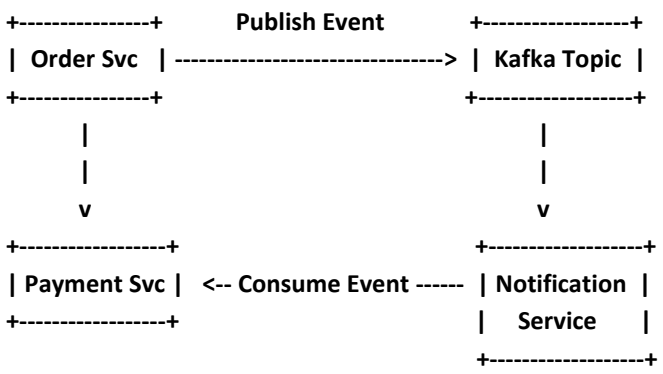
- **Order Service** publishes an event "OrderCreated".
- **Payment Service** subscribes to the event and processes the payment.
- **Notification Service** subscribes and sends confirmation to the user.

This allows services to be **loosely coupled**.

Async Messaging Patterns

1. **Publish/Subscribe** – multiple consumers listen to the same event (fan-out).
2. **Point-to-Point** – one producer sends a message to exactly one consumer (queue).
3. **Event Sourcing** – storing system state as a series of events.
4. **CQRS (Command Query Responsibility Segregation)** – separate models for reads and writes, often backed by events.

Diagram: Event-Driven Microservices



Spring Boot Advanced APIs

Spring Boot with GraphQL

GraphQL is a query language for APIs that allows clients to request **exactly the data they need**, reducing over-fetching and under-fetching.

Dependency:

```
<dependency>
  <groupId>com.graphql-java-kickstart</groupId>
  <artifactId>graphql-spring-boot-starter</artifactId>
  <version>11.1.0</version>
</dependency>
```

Schema file (resources/graphql/schema.graphqls):

```
type Query {
  hello: String
  bookById(id: ID): Book
}

type Book {
  id: ID
  title: String
  author: String
}
```

Resolver Example:

```
import org.springframework.stereotype.Component;
import com.coxautodev.graphql.tools.GraphQLQueryResolver;

@Component
public class BookQueryResolver implements GraphQLQueryResolver {
   public String hello() {
      return "Hello from GraphQL!";
   }

   public Book bookById(String id) {
      return new Book(id, "Spring Boot in Action", "Craig Walls");
   }
}
```

Query Example (via GraphQL Playground):

```
{
  hello
  bookById(id: "1") {
    title
    author
  }
}
```

Spring Boot with gRPC

gRPC is a high-performance, contract-first communication protocol using **Protocol Buffers**. It's commonly used in **microservices** for fast, binary communication.

Dependency:

```
<dependency>
   <groupId>net.devh</groupId>
   <artifactId>grpc-spring-boot-starter</artifactId>
   <version>2.14.0.RELEASE</version>
</dependency>
```

Proto File (src/main/proto/hello.proto):

```
syntax = "proto3";

service HelloService {
  rpc sayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string name = 1;
}

message HelloResponse {
  string message = 1;
}
```

Service Implementation:

```
import net.devh.boot.grpc.server.service.GrpcService;
import io.grpc.stub.StreamObserver;

@GrpcService
public class HelloServiceImpl extends HelloServiceGrpc.HelloServiceImplBase {
  @Override
  public void sayHello(HelloRequest request, StreamObserver<HelloResponse> responseObserver) {
    HelloResponse response = HelloResponse.newBuilder()
        .setMessage("Hello, " + request.getName())
        .build();
    responseObserver.onNext(response);
    responseObserver.onCompleted();
  }
}
```

Client Call Example:

```
ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 9090)
    .usePlaintext()
    .build();
HelloServiceGrpc.HelloServiceBlockingStub stub = HelloServiceGrpc.newBlockingStub(channel);
HelloResponse response = stub.sayHello(HelloRequest.newBuilder().setName("John").build());
System.out.println(response.getMessage());
```

Combining REST + GraphQL in the Same Project

In real-world applications, you might want both REST (for simple CRUD, integrations) and GraphQL (for flexible queries). Spring Boot supports running them **side by side**.

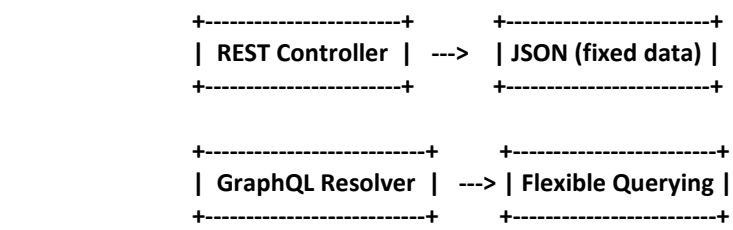
Approach:

1. Keep your REST controllers (`@RestController`) for existing endpoints.
2. Add GraphQL resolvers (`GraphQLQueryResolver`) for flexible queries.
3. Use GraphQL only for endpoints where clients need **nested/filtered queries**, while REST continues for **standard APIs**.

Example Project Structure:

```
src/main/java/com/example/demo
 ├── controller
 │    └── BookRestController.java   (REST API)
 ├── graphql
 │    └── BookQueryResolver.java    (GraphQL API)
 ├── model
 │    └── Book.java
 └── DemoApplication.java
resources/graphql/schema.graphqls
```

Diagram: REST + GraphQL Together



Spring Boot Testing

Unit Testing with JUnit & Mockito

Unit tests verify individual components (like services, repositories, or utils) **in isolation**.

Dependencies: (already included in Spring Boot Starter Test)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Service Example:

```
@Service
public class CalculatorService {
  public int add(int a, int b) {
    return a + b;
  }
}
```

Unit Test with JUnit:

```
@SpringBootTest
class CalculatorServiceTest {

  @Autowired
  private CalculatorService calculatorService;

  @Test
  void testAdd() {
    assertEquals(5, calculatorService.add(2, 3));
  }
}
```

Mockito Example (Mocking Repository):

```
@ExtendWith(MockitoExtension.class)
class UserServiceTest {

  @Mock
  private UserRepository userRepository;
```

```
    @InjectMocks
    private UserService userService;

    @Test
    void testFindUser() {
        User mockUser = new User(1L, "Mazid");
        when(userRepository.findById(1L)).thenReturn(Optional.of(mockUser));

        User result = userService.getUserById(1L);

        assertEquals("Mazid", result.getName());
        verify(userRepository, times(1)).findById(1L);
    }
}
```

Integration Testing with Spring Boot Test

Integration tests validate how **multiple layers (controller, service, repository)** work together.

Example:

```
@SpringBootTest
@AutoConfigureMockMvc
class UserIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testGetUser() throws Exception {
        mockMvc.perform(get("/users/1"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.name").value("Mazid"));
    }
}
```

This test loads the **application context**, starts a test web environment, and verifies the endpoint.

Testing REST APIs

Spring provides **MockMvc** and **TestRestTemplate** for testing REST APIs.

MockMvc Example (fast, no server needed):

```
@SpringBootTest
@AutoConfigureMockMvc
class ApiTest {

    @Autowired
    private MockMvc mockMvc;
    @Test
    void testCreateUser() throws Exception {
```

```
        mockMvc.perform(post("/users")
            .contentType(MediaType.APPLICATION_JSON)
            .content("{\"name\":\"Mazid\"}"))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.name").value("Mazid"));
    }
}
```

TestRestTemplate Example (real HTTP call):

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class ApiRestTemplateTest {

    @Autowired
    private TestRestTemplate restTemplate;
    @Test
    void testGetUser() {
        ResponseEntity<User> response =
            restTemplate.getForEntity("/users/1", User.class);

        assertEquals(HttpStatus.OK, response.getStatusCode());
        assertEquals("Mazid", response.getBody().getName());
    }
}
```

Key Points of Testing Tools:

- **JUnit 5** → Unit testing framework
- **Mockito** → Mock dependencies
- **SpringBootTest + MockMvc** → Integration & REST testing
- **TestRestTemplate** → Real REST endpoint testing

Performance Tuning & Monitoring

Profiling Spring Boot Applications

Profiling helps identify **slow methods, memory leaks, and bottlenecks**.

Tools to use:

- **Spring Boot Actuator** – Exposes endpoints (/actuator/health, /actuator/metrics, etc.)
- **JVisualVM / JProfiler / YourKit** – For JVM profiling
- **Micrometer + Prometheus + Grafana** – Metrics collection & visualization

Example: Enable Actuator in pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Enable Endpoints in application.properties:

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

Caching Strategies (EhCache, Redis)

Caching reduces **duplicate DB calls** by storing frequently accessed results.

EhCache Setup:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
  <groupId>org.ehcache</groupId>
  <artifactId>ehcache</artifactId>
</dependency>
```

Enable Caching:

```
@SpringBootApplication
@EnableCaching
public class DemoApplication {}
```

Cache Example:

```
@Service
public class UserService {
  @Cacheable("users")
  public User getUserById(Long id) {
    simulateSlowService(); // Mocking a slow DB call
    return new User(id, "Mazid");
  }
}
```

First call → executes DB query
Subsequent calls → returns from cache instantly

Redis Caching (better for distributed apps):

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Optimizing Database Calls

Database is usually the **biggest bottleneck** in large apps.

Best Practices:

1. Use **Pagination & Sorting** instead of fetching all records.

2. Page<User> findAll(Pageable pageable);
3. Optimize queries with **JPQL / Native Queries** only when needed.
4. Use **Connection Pooling** (HikariCP is default in Spring Boot).
5. Enable **Lazy Loading** for associations (avoid N+1 queries).
6. Profile queries with **Hibernate Statistics**.

Enable SQL Logging:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.generate_statistics=true
```

Thread Management & Async Tasks

Using **multithreading & async execution** improves performance for heavy tasks.

Enable Async in Spring Boot:

```
@SpringBootApplication
@EnableAsync
public class DemoApplication {}
```

Async Service Example:

```
@Service
public class EmailService {

   @Async
   public void sendEmail(String recipient) {
      System.out.println("Sending email to " + recipient + " - " + Thread.currentThread().getName());
   }
}
```

Controller Example:

```
@RestController
@RequestMapping("/mail")
public class MailController {

   @Autowired
   private EmailService emailService;
   @GetMapping("/send")
   public String sendMail() {
      emailService.sendEmail("mazid@example.com");
      return "Email sent async!";
   }
}
```

his will **not block the main thread**.

Thread Pool Configuration (application.properties):

```
spring.task.execution.pool.core-size=5
spring.task.execution.pool.max-size=10
spring.task.execution.pool.queue-capacity=50
```

Key Points:

- Use **Actuator + Profiling Tools** for monitoring
- Apply **Caching (EhCache / Redis)** for performance boost
- Optimize DB queries with **pagination, pooling, lazy loading**
- Use **@Async + Thread Pooling** for concurrency

Advanced Topics & Deployment

Spring Boot Actuator (Monitoring & Metrics)

Actuator provides **production-ready endpoints** to monitor applications.

Dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Enable Endpoints in application.properties:

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

Example Endpoints:

- /actuator/health → Application health
- /actuator/metrics → Performance metrics
- /actuator/httptrace → Last HTTP requests

Integration with Micrometer:

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

This allows metrics to be scraped by **Prometheus** and visualized in **Grafana**.

Spring Boot Admin

Spring Boot Admin provides a **dashboard to monitor multiple Spring Boot apps**.

Setup:

1. Add dependency:

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-server</artifactId>
```

```
    <version>2.7.0</version>
</dependency>
```

2. Enable server in main class:

```
@SpringBootApplication

@EnableAdminServer

public class AdminServerApplication {

 public static void main(String[] args) {
      SpringApplication.run(AdminServerApplication.class, args);
   }
}
```

3. Clients (applications to monitor) use spring-boot-admin-starter-client.

Dockerizing Spring Boot Application

Docker makes deployment **portable and consistent**.

Dockerfile Example:

```
FROM openjdk:17-jdk-alpine
VOLUME /tmp

COPY target/demo-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Build & Run:

```
docker build -t springboot-app .
docker run -p 8080:8080 springboot-app
```

Deploying on Cloud (AWS / Azure / GCP)

1. **AWS** → Elastic Beanstalk, EC2, or EKS (Kubernetes)
2. **Azure** → App Service, AKS (Kubernetes)
3. **GCP** → App Engine, GKE (Kubernetes)

Deployment Steps (AWS Elastic Beanstalk example):

- Package app: mvn clean package
- Install EB CLI and initialize project: eb init
- Create environment & deploy: eb create springboot-env

CI/CD with GitHub Actions / Jenkins (Optional)

Automate builds, tests, and deployment using pipelines.

GitHub Actions Example: .github/workflows/main.yml

```
name: Spring Boot CI/CD
on:
  push:
    branches: [ main ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Set up JDK
      uses: actions/setup-java@v3
      with:
        java-version: 17
    - name: Build with Maven
      run: mvn clean package
    - name: Docker Build & Push
      run: |
        docker build -t your-dockerhub-username/springboot-app .
        echo ${{ secrets.DOCKER_HUB_PASSWORD }} | docker login -u ${{ secrets.DOCKER_HUB_USERNAME
}} --password-stdin
        docker push your-dockerhub-username/springboot-app
```

Key Points:

- **Actuator + Spring Boot Admin** → Monitor apps in production
- **Docker** → Containerize apps for consistent deployment
- **Cloud Deployment** → AWS, Azure, GCP options
- **CI/CD** → Automate build, test, and deployment

Projects:

Beginner Projects (Core Spring Boot & REST APIs)

1. **Simple To-Do Application**
    - **Features:** CRUD tasks, REST APIs
    - **Concepts Covered:** Spring Boot basics, Spring MVC, Spring Data JPA, H2 Database, Validation
    - **Optional:** Unit testing with JUnit & Mockito
2. **Student Management System**
    - **Features:** Manage students, courses, basic reports
    - **Concepts Covered:** REST API development, CRUD with JPA, DTOs, Exception Handling, Validation
    - **Optional:** Simple Spring Security for admin access
3. **Library Book Tracker**
    - **Features:** Track borrowed books, due dates, search by title/author
    - **Concepts Covered:** Spring MVC, JPA relationships (One-to-Many, Many-to-One), Request Parameters & Path Variables

Intermediate Projects (Security, Messaging, Databases)

1. **Employee Leave Management System**
    - **Features:** Employees request leave, managers approve/reject
    - **Concepts Covered:** Spring Security (Role-based), REST APIs, DTOs, Spring Data JPA, Pagination & Sorting, Testing
2. **E-commerce Product Catalog**
    - **Features:** Product CRUD, category filtering, search, REST APIs
    - **Concepts Covered:** Spring Boot REST APIs, Spring Data JPA, Validation, Exception Handling, H2/MySQL integration, Swagger for API docs
3. **Real-Time Chat Application**
    - **Features:** Chat between users, real-time updates
    - **Concepts Covered:** Spring WebSockets / Messaging, Spring MVC, Security for authentication, Frontend integration (basic HTML/JS or React)
4. **Event Notification System with Kafka/RabbitMQ**
    - **Features:** Publish events (e.g., user registered, order created) and consume them
    - **Concepts Covered:** Spring Boot Messaging, Event-driven microservices, Async tasks, Logging & Monitoring

Advanced Projects (Microservices, GraphQL, gRPC, Cloud)

1. **Online Shopping Platform (Microservices)**
    - **Features:** Services for catalog, orders, payments, notifications
    - **Concepts Covered:** REST APIs, Spring Cloud Config, Eureka Server, Ribbon, API Gateway, Resilience4j, Distributed transactions
2. **GraphQL + REST Hybrid API for Bookstore**
    - **Features:** Clients can query books via REST or GraphQL
    - **Concepts Covered:** Spring Boot REST, GraphQL integration, DTOs, Pagination & Sorting, Security
3. **gRPC-based Inventory Management**
    - **Features:** High-performance inventory services communicating via gRPC
    - **Concepts Covered:** Spring Boot gRPC, Protocol Buffers, Async messaging patterns, Unit & Integration Testing
4. **Monitoring Dashboard with Actuator + Prometheus + Grafana**
    - **Features:** Monitor multiple Spring Boot apps in real-time, alerting on thresholds
    - **Concepts Covered:** Spring Boot Actuator, Micrometer, Prometheus/Grafana integration, Docker deployment
5. **CI/CD Enabled Cloud-Deployed Application**
    - **Features:** Any app with automated build, test, Dockerization, and deployment to AWS/Azure/GCP
    - **Concepts Covered:** GitHub Actions / Jenkins, Docker, Cloud deployment, Monitoring, Spring Boot Actuator