Spring Boot Basics Tutorial

Master the Fundamentals

A Comprehensive Guide to Spring Boot Core Concepts

By: Wildan Anugrah

Agenda

Github

https://github.com/wildananugrah/spring-learning

By: Wildan Anugrah

Part 1: Singleton Pattern

Understanding Object Lifecycle in Spring

What is Singleton Pattern?

> **Definition:** A design pattern that ensures a class has only ONE instance throughout the application lifecycle.

Real-World Analogy

Think of a **country's president** - there can only be one at a time.

Use Cases

- 🗄️ Database connections
- 📋 Configuration managers
- 💾 Caching mechanisms
- 📝 Logging services
- 🔧 Thread pools

Traditional vs Spring Singleton

Aspect	Traditional Singleton	Spring Singleton
Implementation	Manual (private constructor)	Automatic (`@Service`, `@Component`)
Thread Safety	You implement	Spring handles
Lifecycle	You manage	Container manages
Testing	❌ Difficult to mock	✅ Easy with DI
Scope	JVM-wide	Context-wide

Traditional Singleton - Example

```java
public class DatabaseConnection {
    private static DatabaseConnection instance;

    private DatabaseConnection() { }  // Private constructor

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            synchronized (DatabaseConnection.class) {
                if (instance == null) {
                    instance = new DatabaseConnection();
                }
            }
        }
        return instance;
    }
}
```

Issues: Complex, error-prone, hard to test

Spring Singleton - Example

```java
@Service  // That's it! Spring handles everything
public class SingletonService {

    public String processData(String data) {
        return "Processed: " + data;
    }
}
```

Benefits:

- ✅ Simple
- ✅ Thread-safe automatically
- ✅ Easy to test
- ✅ Managed lifecycle

Best Practices - Singleton

✅ DO

- Use Spring-managed singletons (default scope)
- Let Spring handle lifecycle and thread safety
- Keep singletons stateless when possible

❌ DON'T

- Don't use traditional singleton in Spring apps
- Avoid storing mutable global state
- Don't manually manage lifecycle

Part 2: Spring Beans

The Heart of Spring IoC Container

What is a Spring Bean?

> **Definition**: An object that is instantiated, assembled, and managed by the Spring IoC (Inversion of Control) container.

Think of it as:

Spring IoC Container = **Factory** 🏭
Spring Beans = **Products** 📦

The factory creates, configures, and manages the lifecycle of products.

Spring IoC Container

> IoC (Inversion of Control) Container is the core of the Spring Framework. Here's what you need to know:

Definition

The Spring IoC Container is a framework component that manages the lifecycle and configuration of application objects (called beans). It "inverts" the control of object creation from your code to the Spring framework.

What Does It Do?

Creates Objects - Instantiates beans based on configuration
Manages Lifecycle - Handles initialization and destruction
Injects Dependencies - Automatically wires beans together
Manages Scope - Controls bean lifetime (singleton, prototype, etc.)

How It Works

Without IoC Container:

> You → new Object() → You manage everything ❌

With IoC Container:

> You → Define beans → Container creates & manages → You just use them ✅

Bean Creation - Method 1: Component Scanning

```
@Component  // Generic bean
public class GenericComponent { }

@Service    // Business logic layer
public class UserService { }

@Repository // Data access layer
public class UserRepository { }

@Controller // MVC controller
public class WebController { }

@RestController // REST API
public class ApiController { }
```

Spring automatically discovers and registers these beans!

Bean Creation - Method 2: Java Configuration

```
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:postgresql://localhost:5432/mydb");
        ds.setUsername("user");
        return ds;
    }

    @Bean
    public EmailService emailService() {
        return new EmailService(dataSource());
    }
}
```

Use when you need more control over bean creation

Bean Scopes

Scope	Instances	Use Case	Example
singleton	1 per container	Services, repos	`UserService`
prototype	New each time	Stateful objects	`ShoppingCart`
request	1 per HTTP request	Web data	`RequestContext`
session	1 per HTTP session	User session	`UserSession`
application	1 per ServletContext	App-wide	`AppCache`

Default: `singleton`

Bean Lifecycle

```
┌─────────────────────────┐
│  Container Started      │
└─────────────────────────┘
            │
            ↓
┌─────────────────────────┐
│  Bean Instantiation     │
└─────────────────────────┘
            │
            ↓
┌─────────────────────────┐
│  Dependency Injection   │
└─────────────────────────┘
            │
            ↓
┌─────────────────────────┐
│  @PostConstruct         │
└─────────────────────────┘
            │
            ↓
┌─────────────────────────┐
│  Bean Ready for Use     │  ← Your application runs here
└─────────────────────────┘
            │
            ↓
┌─────────────────────────┐
│  @PreDestroy            │
└─────────────────────────┘
            │
            ↓
┌─────────────────────────┐
│  Container Shutdown     │
└─────────────────────────┘
```

Bean Lifecycle - Code Example

```java
@Service
public class UserService {

    @PostConstruct
    public void init() {
        // Called after all dependencies are injected
        System.out.println("UserService initialized!");
    }

    public void processUser(String username) {
        // Business logic here
    }

    @PreDestroy
    public void cleanup() {
        // Called before bean is destroyed
        System.out.println("UserService shutting down!");
    }
}
```

Part 2.5: Bean Lifecycle in Detail

Understanding Bean Creation and Destruction

Bean Lifecycle - Complete Picture

```
Application Startup
        ↓
┌─────────────────────────────────────┐
│ 1. Constructor Called               │
│    - Bean instance created          │
└──────────┬──────────────────────────┘
           ↓
┌─────────────────────────────────────┐
│ 2. Dependencies Injected            │
│    - @Autowired fields/setters      │
│    - Constructor parameters         │
└──────────┬──────────────────────────┘
           ↓
┌─────────────────────────────────────┐
│ 3. @PostConstruct                   │ ⭐ RECOMMENDED
│    - Custom initialization          │
│    - Load resources                 │
└──────────┬──────────────────────────┘
           ↓
┌─────────────────────────────────────┐
│ 4. InitializingBean.afterPropertiesSet() │
│    - Alternative to @PostConstruct  │
└──────────┬──────────────────────────┘
           ↓
┌─────────────────────────────────────┐
│ 5. Custom init method               │
│    - Via @Bean(initMethod)          │
└──────────┬──────────────────────────┘
           ↓
┌─────────────────────────────────────┐
│ ✅ Bean Ready for Use                │
│    - Business logic executes here   │
└──────────┬──────────────────────────┘
           ↓
   Application Running...
           ↓
   Shutdown Signal
           ↓
┌─────────────────────────────────────┐
│ 8. @PreDestroy                      │ ⭐ RECOMMENDED
│    - Cleanup resources              │
│    - Close connections              │
└──────────┬──────────────────────────┘
           ↓
┌─────────────────────────────────────┐
│ 9. DisposableBean.destroy()         │
│    - Alternative to @PreDestroy     │
└──────────┬──────────────────────────┘
           ↓
┌─────────────────────────────────────┐
│ 10. Custom destroy method           │
│     - Via @Bean(destroyMethod)      │
└──────────┬──────────────────────────┘
           ↓
Application Shutdown
```

Lifecycle Callback Methods - Comparison

Approach	Type	Standard	Coupling	Recommendation
@PostConstruct @PreDestroy	Annotation	✅ JSR-250	❌ None	⭐ **RECOMMENDED**
InitializingBean DisposableBean	Interface	❌ Spring-only	⚠️ Spring	Use if needed
@Bean(initMethod) @Bean(destroyMethod)	Configuration	❌ Spring-only	❌ None	Third-party classes

Method 1: @PostConstruct and @PreDestroy ⭐

```java
@Service
public class CacheService {

    private Map<String, Object> cache;

    @PostConstruct  // Called after dependency injection
    public void initialize() {
        System.out.println("Initializing cache...");
        cache = new ConcurrentHashMap<>();
        loadInitialData();
        System.out.println("Cache initialized with " + cache.size() + " items");
    }

    public void put(String key, Object value) {
        cache.put(key, value);
    }

    @PreDestroy  // Called before bean destruction
    public void cleanup() {
        System.out.println("Cleaning up cache...");
        cache.clear();
        System.out.println("Cache cleared");
    }
}
```

✅ **Recommended because:**

- Standard JSR-250 annotations
- Framework independent
- Clear intent

Method 2: InitializingBean & DisposableBean

```
@Service
public class DatabaseService implements InitializingBean, DisposableBean {

    private DataSource dataSource;

    @Override
    public void afterPropertiesSet() {
        // Called after all properties are set
        System.out.println("Connecting to database...");
        initializeConnectionPool();
    }

    public void query(String sql) {
        // Business logic
    }

    @Override
    public void destroy() {
        // Called before bean destruction
        System.out.println("Closing database connections...");
        closeConnectionPool();
    }
}
```

Use when:

- You prefer interface-based approach
- Spring-coupling is acceptable
- You want type-safe callbacks

Method 3: Custom Init/Destroy Methods

```java
// Third-party class you can't modify
public class ExternalLibraryService {

    public void startup() {
        System.out.println("External service starting...");
    }

    public void doWork() {
        // Business logic
    }

    public void shutdown() {
        System.out.println("External service shutting down...");
    }
}

@Configuration
public class AppConfig {

    @Bean(initMethod = "startup", destroyMethod = "shutdown")
    public ExternalLibraryService externalService() {
        return new ExternalLibraryService();
    }
}
```

Use for:

- Third-party classes
- When you can't add annotations
- Legacy code integration

Real-World Example: Connection Pool

```
@Component
public class DatabaseConnectionPool {

    @Value("${db.url}")
    private String dbUrl;

    @Value("${db.max-connections:10}")
    private int maxConnections;

    private List<Connection> connections;

    @PostConstruct
    public void initializePool() {
        System.out.println("Creating connection pool...");
        connections = new ArrayList<>();

        for (int i = 0; i < maxConnections; i++) {
            Connection conn = createConnection(dbUrl);
            connections.add(conn);
        }

        System.out.println("Pool initialized with " +
                         maxConnections + " connections");
    }

    public Connection getConnection() {
        // Return available connection
    }

    @PreDestroy
    public void closePool() {
        System.out.println("Closing all connections...");

        for (Connection conn : connections) {
            closeConnection(conn);
        }

        connections.clear();
        System.out.println("Connection pool closed");
    }
}
```

When to Use Lifecycle Methods

@PostConstruct - Use for:

- 🔧 **Initialize resources** (thread pools, caches)
- 📁 **Load configuration** from files
- 🔌 **Establish connections** (database, message queue)
- ✅ **Validate configuration** at startup
- 📊 **Pre-load data** into cache
- 🚀 **Start background tasks**

@PreDestroy - Use for:

- 🔌 **Close connections** (database, network)
- 🧹 **Release resources** (file handles, threads)
- 💾 **Flush caches** to persistent storage
- 📝 **Write logs** or final state
- 🛑 **Stop background tasks**
- 🗑️ **Delete temporary files**

Lifecycle Events - Execution Order

```java
@Component
public class LifecycleDemo implements InitializingBean, DisposableBean {

    public LifecycleDemo() {
        System.out.println("1. Constructor called");
    }

    @PostConstruct
    public void postConstruct() {
        System.out.println("2. @PostConstruct called");
    }

    @Override
    public void afterPropertiesSet() {
        System.out.println("3. afterPropertiesSet called");
    }

    // Bean is ready for use

    @PreDestroy
    public void preDestroy() {
        System.out.println("4. @PreDestroy called");
    }

    @Override
    public void destroy() {
        System.out.println("5. destroy called");
    }
}
```

Console Output:

```
1. Constructor called
2. @PostConstruct called
```

Lifecycle Best Practices

✅ DO

1. **Use @PostConstruct/@PreDestroy** (standard approach)
2. **Keep initialization fast** (don't block startup)
3. **Make cleanup idempotent** (safe to call multiple times)
4. **Log lifecycle events** for debugging
5. **Handle exceptions gracefully** in cleanup
6. **Validate configuration** in @PostConstruct

❌ DON'T

1. **Don't perform heavy I/O** in constructor
2. **Don't access other beans** in constructor
3. **Don't ignore cleanup** (always release resources)
4. **Don't throw exceptions** from @PreDestroy
5. **Don't rely on execution order** between beans
6. **Don't perform business logic** in lifecycle methods

Common Pitfalls

❌ Accessing Beans in Constructor

```java
@Service
public class BadService {
    private final OtherService otherService;

    public BadService(OtherService otherService) {
        this.otherService = otherService;
        // ❌ BAD: Other bean might not be fully initialized
        otherService.doSomething();
    }
}
```

✅ Use @PostConstruct Instead

```java
@Service
public class GoodService {
    private final OtherService otherService;

    public GoodService(OtherService otherService) {
        this.otherService = otherService;
    }

    @PostConstruct
    public void init() {
        // ✅ GOOD: All beans are fully initialized
        otherService.doSomething();
    }
}
```

Lifecycle in Different Scopes

Scope	@PostConstruct	@PreDestroy
singleton	Once at startup	Once at shutdown
prototype	Every instance	❌ Never called!
request	Per HTTP request	End of request
session	Per HTTP session	Session timeout

⚠️ **Important:** @PreDestroy is **NOT called** for prototype beans!

```
@Component
@Scope("prototype")
public class PrototypeBean {
    @PreDestroy
    public void cleanup() {
        // ❌ This will NEVER be called!
        // You must manage cleanup manually
    }
}
```

Testing Lifecycle Beans

```java
@SpringBootTest
class CacheServiceTest {

    @Autowired
    private CacheService cacheService;

    @Test
    void testCacheInitialized() {
        // @PostConstruct was called before this test
        assertNotNull(cacheService.getCache());
        assertTrue(cacheService.getCache().size() > 0);
    }

    @Test
    void testCacheOperations() {
        cacheService.put("key", "value");
        assertEquals("value", cacheService.get("key"));
    }

    // @PreDestroy will be called after all tests finish
}
```

Part 3: Dependency Injection

The Foundation of Loose Coupling

What is Dependency Injection?

> **Definition:** A design pattern where objects receive their dependencies from external sources rather than creating them internally.

Without DI (Tight Coupling)

```java
public class OrderService {
    private EmailService emailService = new EmailService(); // ❌ Tight coupling
}
```

With DI (Loose Coupling)

```java
@Service
public class OrderService {
    private final EmailService emailService; // ✅ Injected

    public OrderService(EmailService emailService) {
        this.emailService = emailService;
    }
}
```

Why Dependency Injection?

Benefits

✅ **Loose Coupling**

Classes don't depend on concrete implementations

✅ **Testability**

Easy to mock dependencies for unit testing

✅ **Maintainability**

Changes in one class don't cascade to others

✅ **Flexibility**

Easy to swap implementations without changing code

DI Type 1: Constructor Injection ⭐

```
@Service
@RequiredArgsConstructor  // Lombok generates constructor
public class NotificationService {
    private final MessageService messageService;  // final = immutable
    private final UserRepository userRepository;

    public void notify(String username, String message) {
        User user = userRepository.findByUsername(username);
        messageService.send(user.getEmail(), message);
    }
}
```

Pros:

- ✅ Immutable dependencies (final fields)
- ✅ Required dependencies are explicit
- ✅ Easy to test
- ✅ Thread-safe

This is the RECOMMENDED approach! ⭐

DI Type 2: Setter Injection

```
@Service
public class OrderService {
    private MessageService messageService;

    @Autowired
    public void setMessageService(MessageService messageService) {
        this.messageService = messageService;
    }
}
```

When to use:

- ⚠️ **Optional dependencies only**
- Configuration that can change at runtime
- Circular dependencies (not recommended)

Cons:

- ❌ Not immutable
- ❌ Dependencies not explicit

DI Type 3: Field Injection ⚠️

```
@RestController
public class UserController {
    @Autowired
    private UserService userService;  // ❌ DON'T DO THIS!
}
```

Why AVOID?

- ❌ Cannot make fields final (not immutable)
- ❌ Hard to test (requires Spring context)
- ❌ Hides dependencies (no constructor signature)
- ❌ Cannot enforce required dependencies

Only use for quick prototypes or demos!

Handling Multiple Implementations

The Problem

```java
public interface PaymentProcessor { }

@Service
public class CreditCardProcessor implements PaymentProcessor { }

@Service
public class PayPalProcessor implements PaymentProcessor { }

// Which one will Spring inject? 🤔
```

Solution 1: @Primary

```java
@Service
@Primary  // This will be injected by default
public class CreditCardProcessor implements PaymentProcessor { }

@Service
public class PayPalProcessor implements PaymentProcessor { }

// Spring will inject CreditCardProcessor
@RequiredArgsConstructor
public class CheckoutService {
    private final PaymentProcessor processor;  // CreditCard
}
```

Solution 2: @Qualifier

```java
@Service
public class CreditCardProcessor implements PaymentProcessor { }

@Service
public class PayPalProcessor implements PaymentProcessor { }

// Explicitly choose which implementation
@RequiredArgsConstructor
public class CheckoutService {
    @Qualifier("payPalProcessor")
    private final PaymentProcessor processor;  // PayPal
}
```

DI - Best Practices

✅ DO

1. **Use constructor injection** (with `@RequiredArgsConstructor`)
2. **Make dependencies final**
3. **Inject interfaces, not implementations**
4. **Keep constructors simple** (no business logic)

❌ DON'T

1. **Avoid field injection** in production code
2. **Don't create circular dependencies**
3. **Don't inject too many dependencies** (> 5 = code smell)
4. **Don't use `@Autowired` on fields**

Part 4: Annotations

Spring's Metadata Magic

What are Annotations?

> **Definition:** Metadata that provides information about the code to the Spring Framework.

Think of annotations as:

Labels 🏷️ that tell Spring:

- How to create beans
- How to wire dependencies
- How to handle requests
- How to configure components

Stereotype Annotations

Purpose: Component Scanning & Bean Registration

Annotation	Layer	Purpose
`@Component`	Generic	Any Spring-managed component
`@Service`	Business	Business logic, services
`@Repository`	Data	Data access, DAOs
`@Controller`	Web	MVC controllers (views)
`@RestController`	Web	REST APIs (JSON/XML)

All of these create beans automatically via component scanning!

Stereotype Annotations - Examples

```java
@Component  // Generic component
public class UtilityHelper { }

@Service  // Business logic
public class UserService { }

@Repository  // Data access
public class UserRepository extends JpaRepository<User, Long> { }

@Controller  // MVC web controller
public class WebController { }

@RestController  // REST API
public class ApiController { }
```

Configuration Annotations

```java
@Configuration  // Declares this class contains bean definitions
public class AppConfig {

    @Bean  // Method return value is registered as a bean
    public DataSource dataSource() {
        return new HikariDataSource();
    }

    @Bean
    public EmailService emailService(DataSource ds) {
        return new EmailService(ds);
    }
}
```

Use `@Configuration` + `@Bean` when:

- Third-party classes (can't add `@Component`)
- Complex initialization logic
- Multiple beans of same type

Property Injection Annotations

@Value - Simple Property Injection

```java
@Service
public class AppService {

    @Value("${app.name}")  // From application.properties
    private String appName;

    @Value("${app.max-connections:100}")  // With default value
    private int maxConnections;

    @Value("#{${app.timeout} * 1000}")  // SpEL expression
    private long timeoutMs;
}
```

@ConfigurationProperties - Type-Safe Config

```java
@Configuration
@ConfigurationProperties(prefix = "app")
@Data
public class AppProperties {
    private String name;
    private String version;
    private int maxConnections;
    private Email email;

    @Data
    public static class Email {
        private String host;
        private int port;
        private String username;
    }
}
```

```properties
# application.properties
app.name=My App
app.version=1.0.0
app.max-connections=100
app.email.host=smtp.gmail.com
app.email.port=587
```

Dependency Injection Annotations

```java
@Service
public class OrderService {

    // Constructor injection (RECOMMENDED)
    @Autowired  // Optional in Spring 4.3+ if single constructor
    public OrderService(PaymentService paymentService) { }

    // Setter injection
    @Autowired
    public void setEmailService(EmailService emailService) { }

    // Field injection (NOT RECOMMENDED)
    @Autowired
    private NotificationService notificationService;
}
```

@Qualifier and @Primary

```
// Multiple implementations
public interface MessageService { }

@Service
@Primary  // Default choice
public class EmailMessageService implements MessageService { }

@Service
public class SmsMessageService implements MessageService { }

// Using @Qualifier
@Service
public class NotificationService {

    @Autowired
    @Qualifier("smsMessageService")  // Explicitly choose SMS
    private MessageService messageService;
}
```

Lifecycle Annotations

```java
@Service
public class CacheService {

    @PostConstruct  // Called AFTER dependency injection
    public void initialize() {
        System.out.println("Loading cache data...");
        loadCache();
    }

    public void doWork() {
        // Normal business logic
    }

    @PreDestroy  // Called BEFORE bean destruction
    public void cleanup() {
        System.out.println("Clearing cache...");
        clearCache();
    }
}
```

Common Annotations - Quick Reference

Category	Annotations
Stereotypes	`@Component`, `@Service`, `@Repository`, `@Controller`, `@RestController`
Configuration	`@Configuration`, `@Bean`, `@ComponentScan`
DI	`@Autowired`, `@Qualifier`, `@Primary`, `@Value`
Properties	`@Value`, `@ConfigurationProperties`, `@PropertySource`
Lifecycle	`@PostConstruct`, `@PreDestroy`
Web	`@RequestMapping`, `@GetMapping`, `@PostMapping`, `@PathVariable`, `@RequestBody`
Scope	`@Scope`, `@RequestScope`, `@SessionScope`

Part 5: Configuration

Externalizing Application Settings

Why Configuration Files?

❌ Without external config:

```java
public class DatabaseService {
    private String url = "jdbc:postgresql://localhost:5432/mydb";  // Hard-coded!
    private String username = "admin";  // Hard-coded!
    private String password = "password123";  // Security issue!
}
```

Problems:

- Can't change without recompiling
- Different for dev/staging/prod
- Passwords in source code!

✅ With external config:

```
# application.properties
db.url=jdbc:postgresql://localhost:5432/mydb
db.username=admin
db.password=${DB_PASSWORD}  # From environment variable
```

```
@Service
public class DatabaseService {
    @Value("${db.url}")
    private String url;

    @Value("${db.username}")
    private String username;

    @Value("${db.password}")
    private String password;
}
```

Benefits: Flexible, secure, environment-specific

application.properties - Structure

```
# Application Info
spring.application.name=my-app
server.port=8080

# Database
spring.datasource.url=jdbc:postgresql://localhost:5432/mydb
spring.datasource.username=postgres
spring.datasource.password=${DB_PASSWORD}

# Logging
logging.level.root=INFO
logging.level.com.myapp=DEBUG

# Custom Properties
app.name=My Application
app.version=1.0.0
app.max-upload-size=10485760
```

Property Placeholders

```
# Simple value
app.name=Spring Boot App

# Default value (if property doesn't exist)
app.timeout=${custom.timeout:30000}

# Reference another property
app.full-name=${app.name} version ${app.version}

# Random values
app.uuid=${random.uuid}
app.secret=${random.value}
app.port=${random.int[1024,65535]}
```

Spring Expression Language (SpEL)

```
app.max-connections=100
app.timeout=30
app.name=myapp
```

```
// Mathematical operations
@Value("#{${app.max-connections} * 2}")
private int maxPool;  // 200

// String operations
@Value("#{'${app.name}'.toUpperCase()}")
private String upperName;  // MYAPP

// Ternary operator
@Value("#{${app.timeout} > 60 ? 'slow' : 'fast'}")
private String speed;  // fast

// System properties
@Value("#{systemProperties['user.home']}")
private String userHome;
```

Profile-Specific Properties

```
application.properties              # Default for all profiles
application-development.properties # Only for development
application-staging.properties     # Only for staging
application-production.properties  # Only for production
```

```
# application.properties
spring.profiles.active=development

# application-development.properties
db.url=jdbc:postgresql://localhost:5432/devdb
logging.level.root=DEBUG

# application-production.properties
db.url=jdbc:postgresql://prod-server:5432/proddb
logging.level.root=WARN
```

Activate: `java -jar app.jar --spring.profiles.active=production`

@ConfigurationProperties vs @Value

Aspect	@Value	@ConfigurationProperties
Type Safety	❌ No	✅ Yes
Validation	❌ Limited	✅ Full support
Grouping	❌ Individual	✅ Grouped in class
IDE Support	❌ Limited	✅ Autocomplete
Relaxed Binding	❌ No	✅ Yes (kebab-case, camelCase)
Use Case	Single properties	Complex configurations

Recommendation: Use `@ConfigurationProperties` for grouped settings!

Configuration - Best Practices

✅ DO

1. **Use profiles** for environment-specific configs
2. **Externalize secrets** to environment variables
3. **Group related properties** with `@ConfigurationProperties`
4. **Provide default values** where appropriate
5. **Validate configuration** at startup

❌ DON'T

1. **Don't hard-code** values in Java code
2. **Don't commit secrets** to version control
3. **Don't mix concerns** (group logically)
4. **Don't forget documentation** for custom properties

Part 6: Environment Variables

Production-Ready Secret Management

Why Environment Variables?

Security Comparison

Method	Security	Flexibility	Cloud-Ready
Hard-coded	❌ Very Poor	❌ No	❌ No
application.properties	⚠️ Medium	⚠️ Limited	⚠️ Partial
Environment Variables	✅ **Best**	✅ **High**	✅ **Yes**

Environment variables = Industry standard for secrets management

The Problem with Hard-Coded Values

```
// ❌ NEVER DO THIS!
public class EmailService {
    private static final String API_KEY = "sk_live_abc123xyz";  // Exposed!
    private static final String PASSWORD = "mypassword123";      // In Git!
}
```

Risks:

- 🔓 Passwords in source code
- 📦 Committed to version control
- 🌍 Same credentials for all environments
- 🔍 Visible in Git history forever

✅ The Solution: Environment Variables

```
# Set environment variables
export EMAIL_API_KEY=sk_live_abc123xyz
export DB_PASSWORD=super_secret_password
```

```
// Access in Spring Boot
@Service
public class EmailService {

    @Value("${EMAIL_API_KEY}")  // From environment
    private String apiKey;

    @Value("${DB_PASSWORD}")
    private String dbPassword;
}
```

Benefits: Secure, flexible, cloud-native ✅

Setting Environment Variables

Linux / Mac

```
export DB_PASSWORD=mysecretpassword
export API_KEY=abc123xyz
echo $DB_PASSWORD  # Verify
```

Windows (Command Prompt)

```
set DB_PASSWORD=mysecretpassword
set API_KEY=abc123xyz
echo %DB_PASSWORD%
```

Windows (PowerShell)

```
$env:DB_PASSWORD = "mysecretpassword"
$env:API_KEY = "abc123xyz"
echo $env:DB_PASSWORD
```

Setting Env Vars in Docker

```
# docker-compose.yml
version: '3.8'
services:
  app:
    image: myapp:latest
    environment:
      - DB_HOST=postgres
      - DB_PORT=5432
      - DB_PASSWORD=mysecret
      - API_KEY=abc123
    env_file:
      - .env  # Load from file
```

```
# Dockerfile
ENV JAVA_OPTS="-Xmx512m"
ENV SPRING_PROFILES_ACTIVE=production
```

Using .env Files (Development)

```
# .env file (NEVER commit to Git!)
DB_HOST=localhost
DB_PORT=5432
DB_PASSWORD=mysecretpassword
API_KEY=abc123xyz
AWS_ACCESS_KEY_ID=AKIA...
AWS_SECRET_ACCESS_KEY=secret...
```

```
# .gitignore (ALWAYS add this!)
.env
.env.local
.env.*.local
*.env
```

```
# .env.example (This you CAN commit)
DB_HOST=localhost
DB_PORT=5432
DB_PASSWORD=your_password_here
API_KEY=your_api_key_here
```

Accessing Environment Variables in Spring

Method 1: @Value

```java
@Service
public class DatabaseService {
    @Value("${DB_HOST}")
    private String host;

    @Value("${DB_PASSWORD}")
    private String password;
}
```

Method 2: Environment Object

```java
@Service
public class ConfigService {
    @Autowired
    private Environment env;

    public String getDbHost() {
        return env.getProperty("DB_HOST", "localhost");  // With default
    }
}
```

Accessing Environment Variables (cont.)

Method 3: System.getenv()

```
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        String host = System.getenv("DB_HOST");
        String password = System.getenv("DB_PASSWORD");

        if (password == null) {
            throw new IllegalStateException("DB_PASSWORD not set!");
        }

        // Configure datasource...
    }
}
```

Environment Variable Naming

✅ Good Names

```
DB_HOST                 # Clear, uppercase, underscores
DB_PORT
API_KEY
AWS_ACCESS_KEY_ID
SMTP_USERNAME
MAX_UPLOAD_SIZE_MB
```

❌ Bad Names

```
dbHost                  # Not uppercase
db-port                 # Hyphens instead of underscores
apikey                  # Not descriptive enough
password                # Too generic
user                    # Too generic
```

Convention: `UPPER_SNAKE_CASE`

Security Best Practices

1️⃣ Never Commit Secrets

```
# .gitignore
.env
.env.local
.env.*.local
credentials.json
secrets.yml
```

2️⃣ Use Different Values per Environment

```
# Development
DB_PASSWORD=dev_password_123

# Staging
DB_PASSWORD=staging_password_xyz

# Production
DB_PASSWORD=super_secure_prod_password_abc
```

Security Best Practices (cont.)

3️⃣ Validate Required Variables at Startup

```java
@Configuration
public class EnvironmentValidator {

    @PostConstruct
    public void validateEnvironment() {
        String[] required = {"DB_PASSWORD", "API_KEY", "JWT_SECRET"};

        for (String var : required) {
            if (System.getenv(var) == null) {
                throw new IllegalStateException(
                    "Required environment variable not set: " + var
                );
            }
        }
    }
}
```

Security Best Practices (cont.)

4️⃣ Mask Sensitive Values in Logs

```java
@Service
public class SecureLogger {

    public void logConfig(String password) {
        // ❌ BAD
        log.info("Password: {}", password);

        // ✅ GOOD
        log.info("Password: {}",  maskSecret(password));
    }

    private String maskSecret(String value) {
        if (value == null || value.length() < 4) return "****";
        return value.substring(0, 2) + "****" +
               value.substring(value.length() - 2);
    }
}
```

Output: `Password: ab****yz`

Cloud-Native: 12-Factor App

Environment Variable Principles

III. Config

> Store config in the environment

Why?

- ✅ Separates config from code
- ✅ Different config per environment
- ✅ No config changes require code deploy
- ✅ Reduces risk of accidentally committing secrets

Used by: Heroku, AWS, Google Cloud, Kubernetes, Docker

Secret Management Tools

Production-Grade Solutions

Tool	Use Case	Cloud
AWS Secrets Manager	AWS applications	AWS
HashiCorp Vault	Enterprise, multi-cloud	Any
Azure Key Vault	Azure applications	Azure
GCP Secret Manager	Google Cloud apps	GCP
Kubernetes Secrets	K8s deployments	Any
Docker Secrets	Docker Swarm	Any

For serious production apps, use these instead of .env files!

Hands-On Practice

Let's Build Something!

Practice Exercise 1: User Service

Task: Create a UserService with dependency injection

```
// 1. Create interface
public interface UserRepository {
    User findById(Long id);
    void save(User user);
}

// 2. Create implementation with @Repository

// 3. Create UserService with @Service and constructor injection

// 4. Create REST controller to test

// Try it yourself! 💪
```

Concepts: DI, Stereotypes, Constructor Injection

Practice Exercise 2: Configuration

Task: Configure email service using properties

```
# application.properties
email.host=smtp.gmail.com
email.port=587
email.username=${EMAIL_USERNAME}
email.password=${EMAIL_PASSWORD}
email.from=noreply@myapp.com
```

```
// Create EmailConfig class with @ConfigurationProperties
// Inject and use in EmailService
```

Concepts: @ConfigurationProperties, Environment Variables

Practice Exercise 3: Multiple Implementations

Task: Create payment system with multiple processors

```
// 1. Create PaymentProcessor interface
// 2. Create CreditCardProcessor (@Primary)
// 3. Create PayPalProcessor
// 4. Create BitcoinProcessor
// 5. Create CheckoutService that can use any processor
// 6. Use @Qualifier to select specific processor
```

Concepts: @Primary, @Qualifier, Interface-based design

Part 7: Exception Handling

Building Robust Applications

Why Exception Handling Matters

Without Proper Exception Handling ❌

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) {
    return userRepository.findById(id).get();  // Throws ugly exception!
}
```

What user sees:

```
Whitelabel Error Page
This application has no explicit mapping for /error...
java.util.NoSuchElementException: No value present
    at java.base/java.util.Optional.get(Optional.java:143)
```

Problems:

- 💥 Exposes internal implementation
- 🔍 Shows stack traces to users
- ❌ No consistent error format
- 🚫 Poor user experience

With Proper Exception Handling ✅

```java
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) {
    return userRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("User not found"));
}
```

What user sees:

```json
{
  "success": false,
  "message": "User not found",
  "data": null
}
```

Benefits:

- ✅ Clean, consistent error messages
- ✅ Proper HTTP status codes
- ✅ No stack traces exposed
- ✅ Professional API responses

Exception Handling in Spring Boot

Three Key Components

1. **Custom Exceptions** - Business-specific errors
2. **@RestControllerAdvice** - Centralized exception handling
3. **@SneakyThrows** - Lombok magic for checked exceptions

Component 1: Custom Exceptions

Create meaningful exceptions for your domain.

```java
// Generic - Resource not found
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}

// Business logic - Insufficient funds
public class InsufficientBalanceException extends RuntimeException {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}

// Security - Invalid login
public class InvalidCredentialsException extends RuntimeException {
    public InvalidCredentialsException(String message) {
        super(message);
    }
}

// Data integrity - Duplicate entry
public class DuplicateResourceException extends RuntimeException {
    public DuplicateResourceException(String message) {
        super(message);
    }
}
```

Why extend RuntimeException? Unchecked exceptions = no need for try-catch everywhere!

Using Custom Exceptions

```java
@Service
@RequiredArgsConstructor
public class AccountService {
    private final AccountRepository accountRepository;

    public Account getAccount(String accountNumber) {
        return accountRepository.findByAccountNumber(accountNumber)
            .orElseThrow(() -> new ResourceNotFoundException(
                "Account not found: " + accountNumber
            ));
    }

    public void withdraw(String accountNumber, BigDecimal amount) {
        Account account = getAccount(accountNumber);

        if (account.getBalance().compareTo(amount) < 0) {
            throw new InsufficientBalanceException(
                "Insufficient balance. Available: " + account.getBalance()
            );
        }

        account.setBalance(account.getBalance().subtract(amount));
        accountRepository.save(account);
    }
}
```

Component 2: @RestControllerAdvice

Centralized exception handling for all controllers.

```java
@RestControllerAdvice  // Global exception handler
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ApiResponse<Void>> handleResourceNotFound(
            ResourceNotFoundException ex,
            HttpServletRequest request) {

        log.error("Resource not found: {} - URI: {}",
                 ex.getMessage(), request.getRequestURI());

        return ResponseEntity
            .status(HttpStatus.NOT_FOUND)  // 404
            .body(ApiResponse.error(ex.getMessage()));
    }

    @ExceptionHandler(InsufficientBalanceException.class)
    public ResponseEntity<ApiResponse<Void>> handleInsufficientBalance(
            InsufficientBalanceException ex) {

        return ResponseEntity
            .status(HttpStatus.BAD_REQUEST)  // 400
            .body(ApiResponse.error(ex.getMessage()));
    }
}
```

@RestControllerAdvice - More Handlers

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

    // Handle validation errors
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ApiResponse<Map<String, String>>>
            handleValidationErrors(MethodArgumentNotValidException ex) {

        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach(error -> {
            String fieldName = ((FieldError) error).getField();
            String message = error.getDefaultMessage();
            errors.put(fieldName, message);
        });

        return ResponseEntity
            .status(HttpStatus.BAD_REQUEST)
            .body(ApiResponse.<Map<String, String>>builder()
                .success(false)
                .message("Validation failed")
                .data(errors)
                .build());
    }

    // Catch-all for unexpected errors
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ApiResponse<Void>> handleGenericException(
            Exception ex) {

        log.error("Unexpected error occurred", ex);

        return ResponseEntity
            .status(HttpStatus.INTERNAL_SERVER_ERROR)  // 500
            .body(ApiResponse.error("An unexpected error occurred"));
    }
}
```

How @RestControllerAdvice Works

```
┌──────────────────────────────────────────────┐
│           Client Request                     │
└──────────────────────────────────────────────┘
                   │
                   ↓
┌──────────────────────────────────────────────┐
│         @RestController                      │
│   public User getUser(Long id) {             │
│     throw new ResourceNotFoundException()    │  ← Exception thrown
│   }                                          │
└──────────────────────────────────────────────┘
                   │
                   ↓
┌──────────────────────────────────────────────┐
│      @RestControllerAdvice                   │
│   Catches exception automatically!           │
│   @ExceptionHandler(ResourceNotFound...)     │
│   Returns formatted error response           │
└──────────────────────────────────────────────┘
                   │
                   ↓
┌──────────────────────────────────────────────┐
│   Client receives clean error response       │
│   { "success": false,                        │
│     "message": "User not found" }            │
└──────────────────────────────────────────────┘
```

Spring automatically routes exceptions to matching handlers!

Component 3: @SneakyThrows (Lombok)

The Problem: Checked Exceptions

```java
// Without @SneakyThrows - Messy! ❌
public String processFile() {
    try {
        return Files.readString(Path.of("file.txt"));
    } catch (IOException e) {
        throw new RuntimeException(e);  // Boilerplate wrapping
    }
}

public User parseUser(String json) {
    try {
        return objectMapper.readValue(json, User.class);
    } catch (JsonProcessingException e) {
        throw new RuntimeException(e);  // More boilerplate!
    }
}
```

Problem: Checked exceptions force you to handle or declare them.

@SneakyThrows - Clean Solution ✅

```java
@SneakyThrows  // Lombok magic - no try-catch needed!
public String processFile() {
    return Files.readString(Path.of("file.txt"));
}

@SneakyThrows
public User parseUser(String json) {
    return objectMapper.readValue(json, User.class);
}

@SneakyThrows
public AccountResponse createAccount(CreateAccountRequest request) {
    User user = userRepository.findByUsername(username)
        .orElseThrow(() -> new ResourceNotFoundException("User not found"));

    // Complex logic here...
    return mapToResponse(account);
}
```

What @SneakyThrows does:

- Tricks the compiler into not checking for exceptions
- Exceptions still thrown, but no try-catch required
- Code is cleaner and more readable

@SneakyThrows - How It Works

Traditional Java

```java
public void method1() throws IOException {
    method2();
}

public void method2() throws IOException {
    throw new IOException("File error");
}
```

With @SneakyThrows

```java
@SneakyThrows  // No "throws" declaration needed!
public void method1() {
    method2();
}

@SneakyThrows
public void method2() {
    throw new IOException("File error");  // Compiles without try-catch!
}
```

Behind the scenes: Lombok uses bytecode manipulation to bypass Java's checked exception rules.

Real-World Example: Banking App

```java
@Service
@RequiredArgsConstructor
public class TransactionLogic {

    private final AccountRepository accountRepository;
    private final TransactionRepository transactionRepository;

    @Transactional
    @SneakyThrows  // Clean code - no try-catch clutter!
    public TransactionResponse transfer(TransferRequest request) {
        // Find source account
        Account fromAccount = accountRepository
            .findByAccountNumber(request.getFromAccount())
            .orElseThrow(() -> new ResourceNotFoundException(
                "Source account not found"
            ));

        // Find destination account
        Account toAccount = accountRepository
            .findByAccountNumber(request.getToAccount())
            .orElseThrow(() -> new ResourceNotFoundException(
                "Destination account not found"
            ));

        // Validate balance
        if (fromAccount.getBalance().compareTo(request.getAmount()) < 0) {
            throw new InsufficientBalanceException(
                "Insufficient balance for transfer"
            );
        }

        // Perform transfer
        fromAccount.setBalance(fromAccount.getBalance()
            .subtract(request.getAmount()));
        toAccount.setBalance(toAccount.getBalance()
            .add(request.getAmount()));

        // Save and return
        accountRepository.saveAll(List.of(fromAccount, toAccount));
        return createTransactionRecord(fromAccount, toAccount, request);
    }
}
```

Real-World Example: Exception Handler

```
@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ApiResponse<Void>> handleResourceNotFound(
            ResourceNotFoundException ex,
            HttpServletRequest request) {

        // Structured logging
        log.error("""
            ==================== EXCEPTION ====================
            Timestamp: {}
            URI: {}
            Status Code: 404
            Error: {}
            ===================================================
            """,
            LocalDateTime.now(),
            request.getRequestURI(),
            ex.getMessage()
        );

        return ResponseEntity
            .status(HttpStatus.NOT_FOUND)
            .body(ApiResponse.error(ex.getMessage()));
    }

    @ExceptionHandler(DuplicateResourceException.class)
    public ResponseEntity<ApiResponse<Void>> handleDuplicateResource(
            DuplicateResourceException ex) {

        return ResponseEntity
            .status(HttpStatus.CONFLICT)  // 409
            .body(ApiResponse.error(ex.getMessage()));
    }
}
```

Exception Handling Flow

```
User Registration Flow:
    ↓
Client: POST /api/auth/register
    ↓
Controller receives request
    ↓
Service: Check if username exists
    ↓
┌─────────────────────────────────┐
│ Username "john" already exists? │
│ → YES                           │
└─────────────────────────────────┘
           │
           ↓
throw new DuplicateResourceException("Username already exists")
           ↓
@RestControllerAdvice catches exception
           ↓
@ExceptionHandler(DuplicateResourceException.class)
    - Returns 409 Conflict
    - Response: {"success": false, "message": "Username already exists"}
           ↓
Client receives clean error message ✅
```

Exception Handling - Best Practices

✅ DO

1. **Create specific exceptions** for different error types
2. **Use @RestControllerAdvice** for centralized handling
3. **Return appropriate HTTP status codes** (404, 400, 409, 500)
4. **Log exceptions** with context (user, timestamp, URI)
5. **Use @SneakyThrows** to clean up checked exceptions
6. **Provide meaningful error messages** to clients
7. **Hide stack traces** from API responses (security)

Exception Handling - Best Practices (cont.)

❌ DON'T

1. **Don't expose stack traces** to users
2. **Don't return 200 OK** for errors
3. **Don't use generic Exception** for everything
4. **Don't swallow exceptions** (always log them)
5. **Don't put business logic** in exception handlers
6. **Don't forget to log** exception context
7. **Don't return internal error messages** (DB errors, etc.)

HTTP Status Codes for Exceptions

Exception Type	HTTP Status	Code	When to Use
ResourceNotFoundException	Not Found	404	Resource doesn't exist
InvalidCredentialsException	Unauthorized	401	Login failed
InsufficientBalanceException	Bad Request	400	Business rule violation
DuplicateResourceException	Conflict	409	Duplicate entry (unique constraint)
MethodArgumentNotValidException	Bad Request	400	Validation failed
Exception (generic)	Internal Server Error	500	Unexpected errors

Validation with Bean Validation

```java
// DTO with validation
public class CreateAccountRequest {

    @NotBlank(message = "Account name is required")
    @Size(min = 3, max = 50, message = "Account name must be 3-50 characters")
    private String accountName;

    @NotNull(message = "Initial balance is required")
    @DecimalMin(value = "0.0", message = "Balance cannot be negative")
    private BigDecimal initialBalance;
}

// Controller
@PostMapping("/accounts")
public ResponseEntity<AccountResponse> createAccount(
        @Valid @RequestBody CreateAccountRequest request) {  // @Valid triggers validation

    // If validation fails, MethodArgumentNotValidException is thrown
    // @RestControllerAdvice handles it automatically!
    return ResponseEntity.ok(accountService.createAccount(request));
}
```

Exception Handling Summary

Key Concepts

✅ **Custom Exceptions** - Domain-specific errors (ResourceNotFound, InsufficientBalance)
✅ **@RestControllerAdvice** - Centralized exception handling for all controllers
✅ **@ExceptionHandler** - Method-level handlers for specific exceptions
✅ **@SneakyThrows** - Lombok annotation to bypass checked exceptions
✅ **HTTP Status Codes** - 404, 400, 409, 401, 500
✅ **Bean Validation** - @Valid, @NotBlank, @Size, etc.

Architecture

```
Controller → throws exception → @RestControllerAdvice → @ExceptionHandler → Clean JSON response
```

Result: Professional, consistent, secure error handling! ✅

Part 8: HTTP Fundamentals

Understanding Web Communication

What is HTTP?

> **HTTP** (HyperText Transfer Protocol) = The language of the web 🌐

Definition

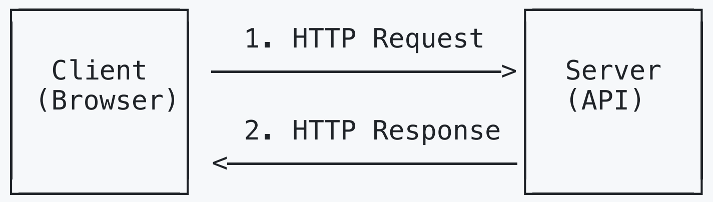
HTTP is a **request-response protocol** used for communication between clients and servers over the internet.

Real-World Analogy

Think of HTTP like **ordering at a restaurant**:

- 👤 **You (Client)**: Make a request ("I want a burger")
- 🍔 **Kitchen (Server)**: Processes request and sends response ("Here's your burger")

HTTP Request-Response Cycle



Example Flow:

1. **Client sends**: `GET /api/users/123`
2. **Server processes**: Fetch user with ID 123 from database
3. **Server responds**: `200 OK` + User data (JSON)

HTTP Request Structure

```
GET /api/users/123 HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOiJIUzI1...
User-Agent: Mozilla/5.0

{optional request body for POST/PUT}
```

Components:

1. **Request Line**: Method + URL + HTTP Version
2. **Headers**: Metadata (content type, auth, etc.)
3. **Body**: Data payload (for POST/PUT/PATCH)

HTTP Methods (Verbs)

Method	Purpose	Has Body?	Idempotent?	Example
GET	Retrieve data	❌ No	✅ Yes	Get user list
POST	Create new resource	✅ Yes	❌ No	Create user
PUT	Update/Replace	✅ Yes	✅ Yes	Update entire user
PATCH	Partial update	✅ Yes	❌ No	Update user email
DELETE	Remove resource	❌ No	✅ Yes	Delete user
HEAD	Get headers only	❌ No	✅ Yes	Check if exists
OPTIONS	Get allowed methods	❌ No	✅ Yes	CORS preflight

Idempotent = Same request repeated multiple times has same effect

HTTP Status Codes

1️⃣ Informational (100-199)

Code	Meaning	Use Case
100	Continue	Client should continue request

2️⃣ Success (200-299) ✅

Code	Meaning	Use Case
200	OK	Successful GET, PUT, PATCH
201	Created	Successful POST (new resource)
204	No Content	Successful DELETE

HTTP Status Codes (cont.)

3️⃣ Redirection (300-399)

Code	Meaning	Use Case
301	Moved Permanently	Resource moved to new URL
302	Found (Temporary)	Temporary redirect
304	Not Modified	Cached version is still valid

4️⃣ Client Errors (400-499) ❌

Code	Meaning	Use Case
400	Bad Request	Invalid syntax/data
401	Unauthorized	Authentication required
403	Forbidden	No permission
404	Not Found	Resource doesn't exist
409	Conflict	Resource conflict (duplicate)
422	Unprocessable	Validation failed

HTTP Status Codes (cont.)

5️⃣ Server Errors (500-599) 💥

Code	Meaning	Use Case
500	Internal Server Error	Unhandled server exception
502	Bad Gateway	Invalid response from upstream
503	Service Unavailable	Server overloaded/maintenance
504	Gateway Timeout	Upstream server timeout

HTTP Headers - Common Examples

Request Headers

```
Host: api.example.com              # Target server
Content-Type: application/json      # Request body format
Authorization: Bearer <token>       # Authentication
Accept: application/json            # Expected response format
User-Agent: Mozilla/5.0             # Client info
Cookie: session=abc123              # Session data
```

Response Headers

```
Content-Type: application/json      # Response body format
Content-Length: 1234                # Body size in bytes
Cache-Control: max-age=3600         # Caching rules
Set-Cookie: session=xyz789          # Set client cookie
Location: /api/users/123            # Resource location (201)
```

REST API Example - Complete Flow

1. Create User (POST)

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "name": "John Doe",
  "email": "john@example.com"
}
```

Response:

```
HTTP/1.1 201 Created
Location: /api/users/123
Content-Type: application/json

{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com",
  "createdAt": "2025-01-15T10:30:00Z"
}
```

REST API Example (cont.)

2. Get User (GET)

```
GET /api/users/123 HTTP/1.1
Host: example.com
Accept: application/json
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com"
}
```

REST API Example (cont.)

3. Update User (PUT)

```
PUT /api/users/123 HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "name": "John Smith",
  "email": "john.smith@example.com"
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 123,
  "name": "John Smith",
  "email": "john.smith@example.com"
}
```

REST API Example (cont.)

4. Delete User (DELETE)

```
DELETE /api/users/123 HTTP/1.1
Host: example.com
```

Response:

```
HTTP/1.1 204 No Content
```

HTTP in Spring Boot - @RestController

```java
@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping("/{id}")  // GET /api/users/123
    public ResponseEntity<User> getUser(@PathVariable Long id) {
        User user = userService.findById(id);
        return ResponseEntity.ok(user);  // 200 OK
    }

    @PostMapping  // POST /api/users
    public ResponseEntity<User> createUser(@RequestBody User user) {
        User created = userService.save(user);
        return ResponseEntity.status(201).body(created);  // 201 Created
    }

    @DeleteMapping("/{id}")  // DELETE /api/users/123
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userService.delete(id);
        return ResponseEntity.noContent().build();  // 204 No Content
    }
}
```

Spring Boot HTTP Annotations

Annotation	HTTP Method	Purpose
`@GetMapping`	GET	Retrieve data
`@PostMapping`	POST	Create new resource
`@PutMapping`	PUT	Update/replace resource
`@PatchMapping`	PATCH	Partial update
`@DeleteMapping`	DELETE	Remove resource
`@RequestMapping`	Any	Generic mapping (can specify method)

Parameter Annotations

Annotation	Source	Example
`@PathVariable`	URL path	`/users/{id}`
`@RequestParam`	Query string	`/users?page=1`
`@RequestBody`	Request body	JSON payload
`@RequestHeader`	Headers	`Authorization` header

URL Components Explained

```
https://api.example.com:8080/api/users/123?active=true&sort=name#section1
└─┬─┘  └──────┬──────┘ └┬─┘ └────┬────┘└┬┘ └──────┬──────┘ └───┬───┘
scheme      host      port   path  id    query params      fragment
```

Breakdown:

- **Scheme**: `https` (protocol)
- **Host**: `api.example.com` (domain)
- **Port**: `8080` (default: 80 for HTTP, 443 for HTTPS)
- **Path**: `/api/users/123` (resource location)
- **Query Params**: `?active=true&sort=name` (filters/options)
- **Fragment**: `#section1` (client-side reference)

Path Variables vs Query Parameters

Path Variables (@PathVariable)

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) { }

// URL: /api/users/123
```

Use for: Resource identification (required)

Query Parameters (@RequestParam)

```
@GetMapping("/users")
public List<User> searchUsers(
    @RequestParam(required = false) String name,
    @RequestParam(defaultValue = "0") int page
) { }

// URL: /api/users?name=John&page=2
```

Use for: Filtering, sorting, pagination (optional)

Request/Response Body Examples

@RequestBody - Receive JSON

```java
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    // Spring automatically converts JSON to User object
    return userService.save(user);
}
```

Client sends:

```json
{
  "name": "John",
  "email": "john@example.com"
}
```

ResponseEntity - Send Response

```java
@GetMapping("/users/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {
    return userService.findById(id)
        .map(ResponseEntity::ok)              // 200 OK
        .orElse(ResponseEntity.notFound().build());  // 404 Not Found
}
```

HTTP Headers in Spring Boot

Reading Headers

```
@GetMapping("/secure")
public String secureEndpoint(
    @RequestHeader("Authorization") String auth,
    @RequestHeader(value = "User-Agent", required = false) String userAgent
) {
    return "Auth: " + auth;
}
```

Setting Response Headers

```
@GetMapping("/download")
public ResponseEntity<byte[]> downloadFile() {
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_PDF);
    headers.setContentDispositionFormData("attachment", "file.pdf");

    return ResponseEntity.ok()
        .headers(headers)
        .body(fileBytes);
}
```

HTTP Best Practices

✅ DO

1. **Use correct HTTP methods** (GET for read, POST for create, etc.)
2. **Return appropriate status codes** (200, 201, 404, 500, etc.)
3. **Use RESTful URL patterns** (`/api/users/123` not `/getUser?id=123`)
4. **Include proper headers** (Content-Type, Authorization, etc.)
5. **Version your API** (`/api/v1/users`)
6. **Use HTTPS** in production (secure communication)

❌ DON'T

1. **Don't use GET for state-changing operations**
2. **Don't expose sensitive data** in URLs
3. **Don't return 200 for errors** (use 4xx/5xx)
4. **Don't forget error handling**

HTTPS vs HTTP

Aspect	HTTP	HTTPS
Security	❌ Plain text	✅ Encrypted (TLS/SSL)
Port	80	443
Speed	Slightly faster	Slightly slower (encryption)
Use Case	Development only	**Production (ALWAYS)**
Data Safety	❌ Can be intercepted	✅ Protected

HTTPS Handshake

```
Client                              Server
  │                                   │
  │──── 1. Client Hello ─────────────>│
  │<─── 2. Server Certificate ────────│
  │──── 3. Verify & Exchange ────────>│
  │<─── 4. Encrypted Connection ──────│
```

Content Negotiation

Client Specifies Desired Format

```
GET /api/users/123 HTTP/1.1
Accept: application/json        # Client wants JSON
```

```
GET /api/users/123 HTTP/1.1
Accept: application/xml         # Client wants XML
```

Spring Boot Handles It

```
@GetMapping(value = "/users/{id}",
            produces = {MediaType.APPLICATION_JSON_VALUE,
                       MediaType.APPLICATION_XML_VALUE})
public User getUser(@PathVariable Long id) {
    return userService.findById(id);
    // Spring converts to JSON or XML based on Accept header
}
```

CORS (Cross-Origin Resource Sharing)

The Problem

```
https://frontend.com  ——X——>  https://api.backend.com
         (Browser blocks requests to different origin)
```

The Solution - Enable CORS

```java
@RestController
@CrossOrigin(origins = "https://frontend.com")  // Allow this origin
public class UserController {

    @GetMapping("/users")
    public List<User> getUsers() {
        return userService.findAll();
    }
}
```

Global CORS Configuration

```java
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins("https://frontend.com")
                .allowedMethods("GET", "POST", "PUT", "DELETE");
    }
}
```

HTTP Caching

Cache-Control Header

```
@GetMapping("/users/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {
    User user = userService.findById(id);

    return ResponseEntity.ok()
        .cacheControl(CacheControl.maxAge(1, TimeUnit.HOURS))
        .body(user);
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: max-age=3600
Content-Type: application/json

{user data}
```

Browser caches this response for 1 hour!

HTTP/2 vs HTTP/1.1

Feature	HTTP/1.1	HTTP/2
Binary Protocol	❌ Text	✅ Binary
Multiplexing	❌ One request/connection	✅ Multiple requests/connection
Header Compression	❌ No	✅ Yes (HPACK)
Server Push	❌ No	✅ Yes
Performance	Slower	⚡ Faster

Spring Boot supports HTTP/2 out of the box:

```
# application.properties
server.http2.enabled=true
```

Testing HTTP Endpoints

Using cURL

```
# GET request
curl http://localhost:8080/api/users/123

# POST request with JSON
curl -X POST http://localhost:8080/api/users \
  -H "Content-Type: application/json" \
  -d '{"name":"John","email":"john@example.com"}'

# With authentication
curl -H "Authorization: Bearer <token>" \
  http://localhost:8080/api/users
```

Using Postman

1. Create collection
2. Set method (GET/POST/etc.)
3. Add URL
4. Add headers/body
5. Send request
6. View response

HTTP Summary

Key Concepts

✅ **HTTP** = Request-Response protocol
✅ **Methods** = GET, POST, PUT, DELETE, PATCH
✅ **Status Codes** = 2xx success, 4xx client error, 5xx server error
✅ **Headers** = Metadata (auth, content type, etc.)
✅ **REST** = Architectural style for APIs
✅ **HTTPS** = Secure HTTP (use in production!)

In Spring Boot

- `@RestController` for HTTP APIs
- `@GetMapping`, `@PostMapping`, etc. for methods
- `@PathVariable`, `@RequestParam`, `@RequestBody` for data
- `ResponseEntity` for complete control over response

API Gateway vs Backend App

Understanding Microservices Architecture

What is a Backend App?

> **Backend Application**: A service that contains business logic and directly handles data operations.

Characteristics

- 🎯 **Single Responsibility**: Focused on specific business domain
- 💾 **Database Access**: Directly connects to database
- 🔧 **Business Logic**: Contains core application logic
- 📦 **Self-Contained**: Can run independently

Example

```
User Service (Backend App)
├── User registration
├── User authentication
├── Profile management
└── Password reset
```

What is an API Gateway?

> **API Gateway**: A server that acts as a single entry point for all client requests, routing them to appropriate backend services.

Think of it as:

🏨 **Hotel Reception Desk**

- Clients don't go directly to rooms
- Reception routes you to the right department
- Handles check-in, security, information

Key Role

Single Entry Point → Routes to multiple backend services

Architecture Comparison

Without API Gateway ❌

```
                  ┌──→ ┌─────────────────┐
                  │    │  User Service   │
                  │    └─────────────────┘
                  │    ┌─────────────────┐
Client (Mobile) ──┼──→ │  Order Service  │
                  │    └─────────────────┘
                  │    ┌─────────────────┐
                  └──→ │ Payment Service │
                       └─────────────────┘
```

Problems:

- Client knows all service URLs
- Client handles multiple connections
- No centralized security
- Cross-cutting concerns duplicated

Architecture Comparison (cont.)

With API Gateway ✅

```
                ┌─────────────┐     ┌───────────────────┐
                │             │───→ │  User Service     │
                │             │     └───────────────────┘
                │             │     ┌───────────────────┐
Client (Mobile)→│ API Gateway │───→ │  Order Service    │
                │             │     └───────────────────┘
                │             │     ┌───────────────────┐
                │             │───→ │ Payment Service   │
                └─────────────┘     └───────────────────┘
```

Benefits:

- Single entry point
- Centralized security
- Request routing
- Protocol translation

Key Differences

Aspect	Backend App	API Gateway
Purpose	Business logic	Request routing
Database	✅ Direct access	❌ No database
Business Logic	✅ Contains logic	❌ Minimal logic
Authentication	Validates tokens	✅ Issues/validates tokens
Responsibility	Single domain	Cross-cutting concerns
Scalability	Scale independently	Scale for traffic
Examples	User Service, Order Service	Netflix Zuul, Kong, AWS API Gateway

API Gateway Functions

1️⃣ Request Routing

Routes requests to appropriate backend services based on URL patterns.

```
// API Gateway routing
GET /api/users/*     → User Service
GET /api/orders/*    → Order Service
GET /api/products/*  → Product Service
GET /api/payments/*  → Payment Service
```

Client only knows: `https://api.company.com`
Gateway knows: All internal service URLs

API Gateway Functions (cont.)

2️⃣ Authentication & Authorization

```
// Client request
GET /api/orders/123
Authorization: Bearer eyJhbGc...

// API Gateway validates token
if (token.isValid()) {
    route to Order Service
} else {
    return 401 Unauthorized
}
```

Benefits:

- Centralized security
- Backend services trust gateway
- Reduce duplicate auth code

API Gateway Functions (cont.)

3️⃣ Rate Limiting & Throttling

Protect backend services from overload.

```
// API Gateway configuration
Rate Limit: 1000 requests/minute per user
Throttle: 100 requests/second per IP

// If exceeded
HTTP 429 Too Many Requests
Retry-After: 60
```

Prevents:

- DDoS attacks
- Service overload
- Abuse

API Gateway Functions (cont.)

4️⃣ Load Balancing

Distribute traffic across multiple instances.

```
                     ┌─────────────────┐
                 ┌──→│ User Service #1 │
API Gateway ─────┼──→│ User Service #2 │
                 └──→│ User Service #3 │
                     └─────────────────┘
```

Strategies:

- Round Robin
- Least Connections
- IP Hash
- Weighted

API Gateway Functions (cont.)

5️⃣ Request/Response Transformation

Modify requests and responses between client and services.

```
// Client sends
GET /api/v2/users/123

// Gateway transforms to
GET /internal/user-service/v1/user?id=123

// Backend responds: XML
<user><name>John</name></user>

// Gateway transforms to: JSON
{"name": "John"}
```

API Gateway Functions (cont.)

6️⃣ Protocol Translation

Convert between different protocols.

```
Client (HTTP/REST) → API Gateway → Backend (gRPC)
Client (WebSocket) → API Gateway → Backend (HTTP)
Client (GraphQL)   → API Gateway → Backend (REST)
```

Example:

- Mobile app uses REST
- Backend uses gRPC for performance
- Gateway translates between them

API Gateway Functions (cont.)

7️⃣ Caching

Cache responses to reduce backend load.

```
// First request
Client → Gateway → Backend Service
         ↓ (cache response)
       Cache

// Subsequent requests (within TTL)
Client → Gateway → Return from Cache (fast!)
         ↑
       Cache
```

Benefits:

- Reduced latency
- Lower backend load
- Cost savings

API Gateway Functions (cont.)

8️⃣ Logging & Monitoring

Centralized logging and metrics collection.

```
// Gateway logs all requests
[2025-01-15 10:30:00] POST /api/orders
  User: user@example.com
  IP: 192.168.1.100
  Response: 201 Created
  Latency: 250ms
  Backend: order-service-2

// Metrics
- Total requests: 1,500,000/day
- Average latency: 150ms
- Error rate: 0.05%
- Top endpoints: /api/users, /api/orders
```

API Gateway Functions (cont.)

9️⃣ Service Discovery

Automatically discover and route to available services.

```
API Gateway ↔️ Service Registry (Eureka, Consul)
                      ↓
                ┌─────────────┐
                │ Services:   │
                │ user-svc:   │
                │   - host-1  │
                │   - host-2  │
                │ order-svc:  │
                │   - host-3  │
                └─────────────┘
```

Dynamic Routing: Gateway automatically adapts to service changes

API Gateway Functions Summary

Function	Benefit
Routing	Single entry point
Authentication	Centralized security
Rate Limiting	Protect from overload
Load Balancing	High availability
Transformation	Client/server decoupling
Protocol Translation	Technology flexibility
Caching	Performance boost
Logging	Centralized monitoring
Service Discovery	Dynamic scaling

Why Do We Need an API Gateway?

1️⃣ Simplified Client Experience

Without Gateway:

```
// Client must know all service URLs
const users = await fetch('https://user-svc.com/users');
const orders = await fetch('https://order-svc.com/orders');
const payments = await fetch('https://pay-svc.com/payments');
```

With Gateway:

```
// Client knows only one URL
const users = await fetch('https://api.company.com/users');
const orders = await fetch('https://api.company.com/orders');
const payments = await fetch('https://api.company.com/payments');
```

Why Do We Need an API Gateway? (cont.)

2️⃣ Centralized Security

Without Gateway:

```java
// Every service duplicates auth logic
@Service
public class UserService {
    public void validateToken(String token) { /* duplicate */ }
}

@Service
public class OrderService {
    public void validateToken(String token) { /* duplicate */ }
}
```

With Gateway:

```java
// Auth logic in one place
@Component
public class GatewayAuthFilter {
    public void validateToken(String token) { /* once */ }
    // All services trust gateway
}
```

Why Do We Need an API Gateway? (cont.)

3️⃣ Backend Service Independence

Services can change without affecting clients.

```
Client → API Gateway → /users → User Service v1

// Backend upgrade (no client changes needed!)
Client → API Gateway → /users → User Service v2
```

Benefits:

- Deploy services independently
- Technology stack flexibility
- Gradual migrations

Why Do We Need an API Gateway? (cont.)

4️⃣ Reduced Latency (Request Aggregation)

Without Gateway:

```
// Client makes 3 separate requests
const user = await fetch('/api/users/123');        // 100ms
const orders = await fetch('/api/orders?user=123'); // 150ms
const profile = await fetch('/api/profile/123');   // 80ms
// Total: 330ms
```

With Gateway (BFF Pattern):

```
// Gateway aggregates internally
const data = await fetch('/api/user-dashboard/123');
// Gateway fetches all in parallel
// Total: 150ms (max of 3)
```

Why Do We Need an API Gateway? (cont.)

5️⃣ Cross-Cutting Concerns

Handle common functionality once, not in every service.

```
┌──────────────────────────────────────┐
│         API Gateway                  │
│  ✅ Authentication                    │
│  ✅ Rate Limiting                     │
│  ✅ Logging                           │
│  ✅ CORS                              │
│  ✅ Compression                       │
│  ✅ SSL Termination                   │
└──────────────────────────────────────┘
           ↓         ↓         ↓
    ┌──────────┬──────────┬──────────┐
    │  User    │  Order   │ Payment  │
    │ Service  │ Service  │ Service  │
    │          │          │          │
    │ (Focus   │ (Focus   │ (Focus   │
    │  on      │  on      │  on      │
    │ business)│ business)│ business)│
    └──────────┴──────────┴──────────┘
```

Real-World Use Cases

Use Case 1: E-Commerce Platform

```
Mobile App → API Gateway → ┌─ User Service (authentication)
                           ├─ Product Service (catalog)
                           ├─ Cart Service (shopping cart)
                           ├─ Order Service (checkout)
                           ├─ Payment Service (processing)
                           ├─ Shipping Service (delivery)
                           └─ Notification Service (emails)
```

Gateway handles:

- Single URL for mobile app
- JWT authentication
- Request routing
- Response aggregation (product + reviews + stock)

Real-World Use Cases (cont.)

Use Case 2: Multi-Platform Support

```
┌─────────────┐
│ Web App     │───┐
└─────────────┘   │
┌─────────────┐   │     ┌─────────────┐
│ Mobile App  │───┼────→│ API Gateway │ → Backend Services
└─────────────┘   │     └─────────────┘
┌─────────────┐   │
│ Partner API │───┘
└─────────────┘
```

Gateway provides:

- Web: Full HTML responses
- Mobile: Optimized JSON (smaller payload)
- Partner: Rate-limited, documented API

Real-World Use Cases (cont.)

Use Case 3: Microservices Migration

Strangler Fig Pattern - Gradually migrate from monolith to microservices.

```
          API Gateway
               ↓
      ┌────────┴────────┐
      ↓                 ↓
┌─────────┐      ┌──────────────┐
│ Old     │      │ New Services │
│Monolith │      ├──────────────┤
│         │      │ User Service │
│ (90%)   │      │ Auth Service │
│         │      │ (10%)        │
└─────────┘      └──────────────┘
```

Gateway routes:

- `/api/users/*` → New User Service
- `/api/*` → Old Monolith (everything else)

Real-World Use Cases (cont.)

Use Case 4: Third-Party API Integration

Provide unified interface to multiple external APIs.

```
Client → API Gateway → ┌─ Payment: Stripe
                       ├─ Payment: PayPal
                       ├─ Email: SendGrid
                       ├─ SMS: Twilio
                       └─ Maps: Google Maps
```

Gateway benefits:

- Client doesn't know which payment provider
- Easy to switch providers
- Consistent error handling
- Unified authentication

Real-World Use Cases (cont.)

Use Case 5: API Versioning

Support multiple API versions simultaneously.

```
API Gateway routes:

/api/v1/users → User Service v1 (deprecated)
/api/v2/users → User Service v2 (current)
/api/v3/users → User Service v3 (beta)
```

Benefits:

- Old clients keep working
- New clients get new features
- Gradual deprecation
- A/B testing

Popular API Gateway Solutions

Cloud-Managed

Provider	Product	Best For
AWS	API Gateway	AWS ecosystem
Google Cloud	Apigee, Cloud Endpoints	Enterprise, GCP
Azure	API Management	Microsoft stack

Self-Hosted / Open Source

Tool	Language	Best For
Kong	Lua/Nginx	High performance
Spring Cloud Gateway	Java	Spring ecosystem
Netflix Zuul	Java	Netflix stack
Traefik	Go	Docker/Kubernetes
NGINX	C	Simple, fast

Spring Cloud Gateway Example

```java
@Configuration
public class GatewayConfig {

    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            // Route to User Service
            .route("user-service", r -> r
                .path("/api/users/**")
                .filters(f -> f
                    .stripPrefix(1)
                    .addRequestHeader("X-Gateway", "Spring-Gateway")
                )
                .uri("lb://USER-SERVICE"))

            // Route to Order Service with rate limiting
            .route("order-service", r -> r
                .path("/api/orders/**")
                .filters(f -> f
                    .stripPrefix(1)
                    .requestRateLimiter(c -> c
                        .setRateLimiter(redisRateLimiter()))
                )
                .uri("lb://ORDER-SERVICE"))

            .build();
    }
}
```

API Gateway Best Practices

✅ DO

1. **Keep it stateless** - Don't store session data
2. **Use circuit breakers** - Prevent cascade failures
3. **Implement health checks** - Monitor backend services
4. **Cache aggressively** - Reduce backend load
5. **Log everything** - Centralized observability
6. **Version your APIs** - Support backward compatibility
7. **Use service discovery** - Dynamic routing

API Gateway Best Practices (cont.)

❌ DON'T

1. **Don't put business logic** in the gateway
2. **Don't become a bottleneck** - Scale the gateway
3. **Don't skip monitoring** - Gateway is critical
4. **Don't expose internal URLs** - Abstract backend services
5. **Don't forget timeouts** - Prevent hanging requests
6. **Don't ignore security** - Gateway is entry point
7. **Don't couple gateway to services** - Loose coupling

When to Use API Gateway

✅ Use API Gateway When:

- Building microservices architecture
- Multiple client types (web, mobile, IoT)
- Need centralized authentication
- Want to protect backend services
- Migrating from monolith
- Need request aggregation
- Multiple backend services (> 3)

⚠️ May Not Need When:

- Simple monolith application
- Single backend service
- Internal-only API
- Very small team/project

API Gateway Summary

Key Takeaways

✅ **API Gateway** = Single entry point for all clients
✅ **Backend App** = Business logic and data operations
✅ **Main Functions:** Routing, auth, rate limiting, load balancing
✅ **Why Needed**: Simplify clients, centralize concerns, protect backends
✅ **Use Cases**: E-commerce, microservices, multi-platform, migrations

Architecture Pattern

```
Clients → API Gateway → Backend Services → Databases
```

Gateway handles cross-cutting concerns
Services handle business logic

Part 9: Spring Cloud Gateway

Building an API Gateway with Spring

What is Spring Cloud Gateway?

> **Spring Cloud Gateway** is a library for building API Gateways on top of Spring Boot and Spring WebFlux.

Key Features

- ✅ Built on **Spring Boot 3.x**
- ✅ **Reactive** (non-blocking, high performance)
- ✅ **Route configuration** (YAML or Java)
- ✅ **Predicates** (route matching)
- ✅ **Filters** (request/response manipulation)
- ✅ **Integration** with Spring ecosystem
- ✅ **Production-ready** (circuit breakers, rate limiting, etc.)

Think of it as: Netflix Zuul's successor, but reactive and more powerful!

Spring Cloud Gateway Architecture

```
┌───────────────────────────────────────────────────────────┐
│                    Client Request                         │
└───────────────────────────────────────────────────────────┘
                       │
                       ↓
┌───────────────────────────────────────────────────────────┐
│              Spring Cloud Gateway                         │
│                                                         │ │
│  ┌─────────────────────────────────────────────────┐    │
│  │ 1. Route Predicates (Path, Method, Header...)   │    │
│  └─────────────────────────────────────────────────┘    │ │
│                      │                                  │
│                      ↓                                  │
│  ┌─────────────────────────────────────────────────┐    │
│  │ 2. Pre-Filters (Add headers, Rate limit...)     │    │
│  └─────────────────────────────────────────────────┘    │ │
│                      │                                  │
│                      ↓                                  │
│  ┌─────────────────────────────────────────────────┐    │
│  │ 3. Forward to Backend Service                   │    │
│  └─────────────────────────────────────────────────┘    │ │
│                      │                                  │
│                      ↓                                  │
│  ┌─────────────────────────────────────────────────┐    │
│  │ 4. Post-Filters (Modify response, Add headers)  │    │
│  └─────────────────────────────────────────────────┘    │
└───────────────────────────────────────────────────────────┘
                       │
                       ↓
┌───────────────────────────────────────────────────────────┐
│                 Backend Service (Port 8000)               │
│          User Account Banking Application                 │
└───────────────────────────────────────────────────────────┘
```

Project Setup

1. Dependencies (pom.xml)

```xml
<dependencies>
    <!-- Spring Cloud Gateway -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>

    <!-- Actuator for monitoring -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>2024.0.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Route Configuration - YAML

```yaml
# application.yml
server:
  port: 9090  # Gateway port

spring:
  application:
    name: spring-gateway
  cloud:
    gateway:
      routes:
        # Route 1: Authentication endpoints
        - id: auth_route
          uri: http://localhost:8000
          predicates:
            - Path=/api/auth/**
          filters:
            - AddRequestHeader=X-Gateway, SpringCloudGateway
            - AddResponseHeader=X-Routed-By, Gateway

        # Route 2: Account management
        - id: account_route
          uri: http://localhost:8000
          predicates:
            - Path=/api/accounts/**
          filters:
            - AddRequestHeader=X-Gateway, SpringCloudGateway

        # Route 3: Transactions
        - id: transaction_route
          uri: http://localhost:8000
          predicates:
            - Path=/api/transactions/**
          filters:
            - AddRequestHeader=X-Gateway, SpringCloudGateway
```

Route Configuration - Java

```java
@Configuration
public class GatewayConfig {

    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            // Route to authentication
            .route("auth_route", r -> r
                .path("/api/auth/**")
                .filters(f -> f
                    .addRequestHeader("X-Gateway", "SpringCloudGateway")
                    .addResponseHeader("X-Routed-By", "Gateway"))
                .uri("http://localhost:8000"))

            // Route to accounts with rate limiting
            .route("account_route", r -> r
                .path("/api/accounts/**")
                .filters(f -> f
                    .addRequestHeader("X-Gateway", "SpringCloudGateway")
                    .requestRateLimiter(c -> c
                        .setRateLimiter(redisRateLimiter())
                        .setKeyResolver(userKeyResolver())))
                .uri("http://localhost:8000"))

            .build();
    }
}
```

Predicates - Route Matching

Predicates determine if a route matches a request.

Built-in Predicates

```yaml
routes:
  - id: path_route
    uri: http://localhost:8000
    predicates:
      - Path=/api/users/**         # Match path pattern

  - id: method_route
    predicates:
      - Path=/api/orders/**
      - Method=GET,POST            # Only GET and POST

  - id: header_route
    predicates:
      - Header=X-Request-Type, mobile  # Match header

  - id: query_route
    predicates:
      - Query=premium, true       # Match query parameter

  - id: host_route
    predicates:
      - Host=**.example.com      # Match host pattern
```

Filters - Request/Response Manipulation

Built-in Filters

```yaml
routes:
  - id: filter_demo
    uri: http://localhost:8000
    filters:
      # Add headers
      - AddRequestHeader=X-Custom-Header, CustomValue
      - AddResponseHeader=X-Response-Time, ${responseTime}

      # Remove prefix from path
      - StripPrefix=1  # /api/users/123 → /users/123

      # Rewrite path
      - RewritePath=/api/(?<segment>.*), /${segment}

      # Redirect
      - RedirectTo=302, https://www.example.com

      # Set status code
      - SetStatus=404

      # Add request parameter
      - AddRequestParameter=source, gateway

      # Circuit breaker
      - CircuitBreaker=backendCircuitBreaker
```

Custom Global Filter

Apply logic to all routes.

```java
@Component
@Slf4j
public class CustomGlobalFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
                             GatewayFilterChain chain) {
        // Pre-processing
        ServerHttpRequest request = exchange.getRequest();
        log.info("Request: {} {}", request.getMethod(), request.getURI());

        // Add request timestamp
        exchange.getAttributes().put("requestTime",
            System.currentTimeMillis());

        // Continue filter chain
        return chain.filter(exchange).then(Mono.fromRunnable(() -> {
            // Post-processing
            Long requestTime = exchange.getAttribute("requestTime");
            Long duration = System.currentTimeMillis() - requestTime;

            ServerHttpResponse response = exchange.getResponse();
            log.info("Response: {} - Duration: {}ms",
                response.getStatusCode(), duration);

            // Add custom response header
            response.getHeaders().add("X-Response-Time",
                duration + "ms");
        }));
    }

    @Override
    public int getOrder() {
        return -1;  // High priority (runs first)
    }
}
```

Custom Gateway Filter Factory

Create reusable filters.

```java
@Component
public class LoggingGatewayFilterFactory extends
        AbstractGatewayFilterFactory<LoggingGatewayFilterFactory.Config> {

    public LoggingGatewayFilterFactory() {
        super(Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        return (exchange, chain) -> {
            if (config.isPreLogger()) {
                log.info("Pre-filter: {} {}",
                    exchange.getRequest().getMethod(),
                    exchange.getRequest().getURI());
            }

            return chain.filter(exchange).then(Mono.fromRunnable(() -> {
                if (config.isPostLogger()) {
                    log.info("Post-filter: Status {}",
                        exchange.getResponse().getStatusCode());
                }
            }));
        };
    }

    @Data
    public static class Config {
        private boolean preLogger = true;
        private boolean postLogger = true;
    }
}
```

Using Custom Filter

```yaml
routes:
  - id: custom_filter_route
    uri: http://localhost:8000
    predicates:
      - Path=/api/users/**
    filters:
      - name: Logging
        args:
          preLogger: true
          postLogger: true
```

Rate Limiting

In-Memory Rate Limiter (Development)

```java
@Component
@Primary
public class InMemoryRateLimiter extends AbstractRateLimiter<Config> {

    private final Map<String, TokenBucket> buckets =
        new ConcurrentHashMap<>();

    @Override
    public Mono<Response> isAllowed(String routeId, String key) {
        Config config = getConfig().get(routeId);

        final Config finalConfig = (config != null) ? config :
            new Config().setReplenishRate(10).setBurstCapacity(20);

        TokenBucket bucket = buckets.computeIfAbsent(key, k ->
            new TokenBucket(finalConfig.getBurstCapacity(),
                           finalConfig.getReplenishRate()));

        boolean allowed = bucket.tryConsume(
            finalConfig.getRequestedTokens());

        return Mono.just(new Response(allowed,
            getHeaders(finalConfig, bucket)));
    }

    // Token bucket implementation...
}
```

Rate Limiting Configuration

```
spring:
  cloud:
    gateway:
      routes:
        - id: rate_limited_route
          uri: http://localhost:8000
          predicates:
            - Path=/api/transactions/**
          filters:
            - name: RequestRateLimiter
              args:
                rate-limiter: "#{@inMemoryRateLimiter}"
                key-resolver: "#{@userKeyResolver}"
```

Result:

```
HTTP/1.1 429 Too Many Requests
X-RateLimit-Remaining: 0
X-RateLimit-Burst-Capacity: 20
X-RateLimit-Replenish-Rate: 10
Retry-After: 5
```

Complete Example: Banking App Gateway

Architecture

```
Client (Browser/Mobile)
        ↓
Spring Cloud Gateway (Port 9090)
        ↓
┌───────────────────────────────────┐
│ Routes:                           │
│ • /api/auth/**    → User Auth     │
│ • /api/accounts/** → Accounts     │
│ • /api/transactions/** → Txns     │
└───────────────┬───────────────────┘
                │
                ↓
User Account Banking App (Port 8000)
                ↓
        PostgreSQL Database
```

Testing the Gateway

1. Start Backend Service

```
cd user-account-app
./mvnw spring-boot:run
# Running on http://localhost:8000
```

2. Start Gateway

```
cd spring-gateway
./mvnw spring-boot:run
# Running on http://localhost:9090
```

3. Test via Gateway

```
# Register via gateway
curl -X POST http://localhost:9090/api/auth/register \
  -H "Content-Type: application/json" \
  -d '{"username":"test","email":"test@example.com",
       "password":"pass123","fullName":"Test User"}'

# Login via gateway
curl -X POST http://localhost:9090/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"username":"test","password":"pass123"}'
```

Monitoring with Actuator

```
# Enable gateway actuator endpoints
management:
  endpoints:
    web:
      exposure:
        include: gateway, health, info, metrics
```

View All Routes

```
curl http://localhost:9090/actuator/gateway/routes | jq
```

Response:

```
[
  {
    "route_id": "auth_route",
    "uri": "http://localhost:8000",
    "order": 0,
    "predicate": "Paths: [/api/auth/**], match trailing slash: true",
    "filters": [
      "[[AddRequestHeader X-Gateway = 'SpringCloudGateway'], order = 1]"
    ]
  }
]
```

Spring Cloud Gateway vs Traditional Gateway

Feature	Spring Cloud Gateway	Traditional Gateway (NGINX/Kong)
Language	Java	C/Lua
Ecosystem	Spring Boot	Standalone
Configuration	YAML/Java	Config files
Customization	Easy (Java code)	Plugin system
Reactive	✅ Yes (WebFlux)	Varies
Spring Integration	✅ Native	❌ No
Learning Curve	Low (if know Spring)	Medium

Spring Cloud Gateway Best Practices

✅ DO

1. **Use predicates** for precise routing
2. **Apply filters** for cross-cutting concerns
3. **Enable actuator** for monitoring
4. **Implement rate limiting** for protection
5. **Use global filters** for common logic
6. **Log all requests** with timestamps
7. **Handle errors** gracefully (custom error handlers)
8. **Scale horizontally** for high availability

Spring Cloud Gateway Best Practices (cont.)

❌ DON'T

1. **Don't put business logic** in filters
2. **Don't block threads** (use reactive patterns)
3. **Don't skip error handling**
4. **Don't hardcode URIs** (use service discovery)
5. **Don't forget timeouts** (configure response timeouts)
6. **Don't expose internal services** (use gateway as facade)

Real-World Use Case: Banking Gateway

```
Mobile App → Spring Cloud Gateway (Port 9090)
                    ↓
    ┌───────────────────────────────────────┐
    │ Global Filters:                       │
    │ • Logging (all requests)              │
    │ • CORS (mobile app origin)            │
    │ • Custom headers (X-Gateway-Version)  │
    └───────────────────────────────────────┘
                    ↓
    ┌───────────────────────────────────────┐
    │ Routes:                               │
    │ • POST /api/auth/** → No rate limit   │
    │ • GET /api/accounts/** → Rate: 100/m  │
    │ • POST /api/transactions/** → 10/m    │
    └───────────────────────────────────────┘
                    ↓
           Banking Backend (Port 8000)
                    ↓
           PostgreSQL + Redis
```

Spring Cloud Gateway Summary

Key Concepts

✅ **Routes** - Define how requests are forwarded
✅ **Predicates** - Match requests (Path, Method, Header, etc.)
✅ **Filters** - Modify requests/responses (Add headers, Rate limit, etc.)
✅ **Global Filters** - Applied to all routes
✅ **Custom Filters** - Reusable filter logic
✅ **Rate Limiting** - Protect backend from overload
✅ **Reactive** - Non-blocking, high performance

Architecture

```
Client → Gateway (Predicates → Filters) → Backend Service
```

Perfect for: Microservices, multi-client apps, API management

Best Practices Summary

Dependency Injection - Best Practices

✅ DO

- Use **constructor injection** (immutable, testable)
- Use `@RequiredArgsConstructor` from Lombok
- Inject **interfaces**, not implementations
- Keep dependencies **minimal** (< 5)

❌ DON'T

- Avoid **field injection** (hard to test)
- Don't create **circular dependencies**
- Don't use `new` for beans (let Spring manage)
- Don't inject **too many** dependencies

Configuration - Best Practices

✅ DO

- Use **profiles** for environments
- Group with `@ConfigurationProperties`
- Externalize **secrets** to environment variables
- Provide **default values**
- **Validate** configuration at startup

❌ DON'T

- Hard-code values
- Commit secrets to Git
- Mix unrelated properties
- Forget to document custom properties

Environment Variables - Best Practices

✅ DO

- Use **UPPER_SNAKE_CASE** naming
- **Validate** required variables at startup
- **Mask** sensitive values in logs
- Use **different values** per environment
- **Rotate secrets** regularly
- Use **.gitignore** for .env files

❌ DON'T

- Commit .env to version control
- Use same credentials everywhere
- Expose secrets in logs
- Use generic variable names

General Spring Boot - Best Practices

✅ DO

1. Follow **convention over configuration**
2. Use **Spring Boot starters** for dependencies
3. Leverage **auto-configuration**
4. Write **unit tests** with mocked dependencies
5. Use **Lombok** to reduce boilerplate
6. Document your **custom properties**
7. Use **profiles** for environment-specific configs

Testing Your Knowledge

Quick Quiz!

Quiz Question 1

What's the default bean scope in Spring?

A) prototype
B) singleton ✅
C) request
D) session

Answer: B) singleton

Every bean is singleton by default - one instance per container.

Quiz Question 2

Which DI method is RECOMMENDED?

A) Field injection
B) Setter injection
C) Constructor injection ✅

Answer: C) Constructor injection

Why? Immutable (final), explicit dependencies, easy to test, thread-safe.

Quiz Question 3

What's the difference between @Component and @Bean?

Answer:

- **@Component**: Class-level, auto-detected by component scanning
- **@Bean**: Method-level in @Configuration, manual bean definition

Use @Component for your classes, @Bean for third-party classes.

Quiz Question 4

Where should you store database passwords?

A) Hard-coded in Java ❌
B) application.properties ⚠️
C) Environment variables ✅
D) In a comment ❌

Answer: C) Environment variables

Most secure, cloud-ready, different per environment.

Resources & Next Steps

Learning Resources

📚 Official Documentation

- Spring Framework Docs
- Spring Boot Reference
- Spring Guides

🎓 Tutorials

- Baeldung Spring Tutorials
- Spring Boot Tutorial by Eugen

💻 Practice

- GitHub repository with all examples
- 50+ REST endpoints to test
- Comprehensive README.md

What's Next?

After mastering basics:

1. **Spring Data JPA** - Database access made easy
2. **Spring Security** - Authentication & authorization
3. **Spring Boot Testing** - Unit & integration tests
4. **Spring AOP** - Aspect-oriented programming
5. **Spring Cloud** - Microservices architecture
6. **Spring WebFlux** - Reactive programming

You now have a solid foundation! 🎉

Project Structure Recap

```
spring-basic/
├── src/main/java/com/springbasic/
│   ├── singleton/          # Singleton examples
│   ├── beans/              # Spring beans
│   ├── di/                 # Dependency injection
│   ├── annotations/        # Annotations
│   ├── config/             # Configuration
│   └── env/                # Environment variables
├── application.properties  # Config file
├── .env.example           # Environment template
└── README.md              # Full documentation
```

Run: `./mvnw spring-boot:run`
Test: `http://localhost:9000/api/...`

Q&A

Questions?

Thank You!

Happy Learning! 🚀

Remember:

- Start with **Singleton & Beans**
- Master **Dependency Injection**
- Use **Constructor Injection**
- Externalize **Configuration**
- Secure with **Environment Variables**

Contact & Resources

📧 Questions?

Check the comprehensive README.md in the repository

💻 Code Examples

All examples are in: `/src/main/java/com/springbasic/`

🧪 Practice

52+ REST endpoints at `http://localhost:9000/api/*`

📖 Documentation

Every class has detailed JavaDoc comments

Now go build something amazing with Spring Boot! 💪