Complete Spring Framework & Spring Boot Guide

Table of Contents

Introduction

Spring is a comprehensive framework ecosystem for Java development that provides infrastructure support for developing Java applications. It handles the plumbing so you can focus on your business logic.

Key Benefits:

- Dependency Injection and Inversion of Control
- Aspect-Oriented Programming
- Simplified database access
- Easy testing
- Modular architecture
- Large ecosystem

Spring Framework vs Spring Boot

Spring Framework

The foundational framework providing:

- **IoC Container**: Manages object creation and dependencies
- **AOP**: Cross-cutting concerns like logging, security
- **Data Access**: JDBC abstraction, ORM integration
- **Web MVC**: Model-View-Controller framework
- **Security**: Authentication and authorization
- **Transaction Management**: Declarative transactions

Spring Boot

Built on top of Spring Framework, adding:

- **Auto-Configuration**: Automatic setup based on dependencies
- **Embedded Servers**: No need for external application servers
- **Production Features**: Metrics, health checks, monitoring
- **Starter Dependencies**: Pre-configured dependency bundles
- **Opinionated Defaults**: Convention over configuration

Key Differences:

Aspect	Spring Framework	Spring Boot
Configuration	Manual XML/Java config	Auto-configuration
Server	External server required	Embedded server
Setup Time	Hours/Days	Minutes
Boilerplate	High	Minimal
Production Ready	Manual setup	Built-in features

Core Spring Concepts

IoC Container and Dependency Injection

Inversion of Control (IoC): The container manages object creation and wiring instead of objects managing their own dependencies.

Dependency Injection Types:

1. **Constructor Injection** (Recommended)

```
@Service
public class UserService {
    private final UserRepository repository;

    public UserService(UserRepository repository) {
        this.repository = repository;
    }
}
```

2. **Setter Injection**

```
@Service
public class UserService {
    private UserRepository repository;

    @Autowired
    public void setRepository(UserRepository repository) {
        this.repository = repository;
    }
}
```

3. **Field Injection** (Not recommended)

```
@Service
public class UserService {
    @Autowired
    private UserRepository repository;
}
```

Bean Scopes

- **Singleton** (Default): One instance per Spring container
- **Prototype**: New instance every time requested
- **Request**: One instance per HTTP request (web apps)
- **Session**: One instance per HTTP session (web apps)
- **Application**: One instance per ServletContext (web apps)

```
@Component
@Scope("prototype")
public class PrototypeBean {
    // New instance created each time
}
```

Bean Lifecycle

1. **Instantiation**: Container creates bean instance
2. **Dependency Injection**: Dependencies are injected
3. **Post-Processing**: BeanPostProcessor methods called
4. **Initialization**: @PostConstruct or custom init methods
5. **Ready for Use**: Bean is fully initialized
6. **Destruction**: @PreDestroy or custom destroy methods

```
@Component
public class LifecycleBean {
    @PostConstruct
    public void init() {
        // Initialization logic
    }

    @PreDestroy
    public void cleanup() {
        // Cleanup logic
    }
}
```

Complete Annotations Reference

Core Stereotype Annotations

@Component

```
@Component
public class GenericComponent {
    // Generic Spring-managed component
}
```

- Base stereotype annotation
- Marks class for auto-detection during component scanning
- Generic when other stereotypes don't fit

@Service

```
@Service
public class UserService {
    // Business logic layer
    public User processUser(User user) {
        // Business operations
        return user;
    }
}
```

- Specialization of @Component
- Indicates business logic layer
- Better readability and potential for additional processing

@Repository

```
@Repository
public class UserRepository {
    // Data access logic
    public User findById(Long id) {
        // Database operations
        return user;
    }
}
```

- Specialization of @Component
- Data access layer annotation
- Provides automatic exception translation
- Converts database exceptions to Spring's DataAccessException

@Controller

```
@Controller
public class UserController {
    @RequestMapping("/users")
    public String listUsers(Model model) {
        // Returns view name
        return "users";
    }
}
```

- Handles web requests
- Returns view names for rendering
- Used with traditional MVC applications

@RestController

```
@RestController
@RequestMapping("/api/users")
public class UserRestController {
    @GetMapping
    public List<User> getUsers() {
        // Returns data directly as JSON
        return userService.getAllUsers();
    }
}
```

- Combines @Controller + @ResponseBody
- Returns data directly to HTTP response
- Perfect for REST APIs

Configuration Annotations

@Configuration

```
@Configuration
public class AppConfig {
    @Bean
    public DataSource dataSource() {
        return new HikariDataSource();
    }
}
```

- Marks class as configuration source
- Replaces XML configuration
- Contains @Bean methods

@Bean

```
@Configuration
public class DatabaseConfig {
    @Bean
    @Primary
    public DataSource primaryDataSource() {
        return new HikariDataSource();
    }

    @Bean("secondaryDS")
    public DataSource secondaryDataSource() {
        return new HikariDataSource();
    }
}
```

- Produces beans managed by Spring
- Method name becomes bean name (unless specified)
- Can specify custom names and qualifiers

@ComponentScan

```
@Configuration
@ComponentScan(basePackages = {"com.example.service", "com.example.reposi
public class AppConfig {
    // Scans specified packages for components
}
```

- Configures component scanning
- Can specify base packages, filters, etc.

@Import

```
@Configuration
@Import({DatabaseConfig.class, SecurityConfig.class})
public class MainConfig {
    // Imports other configuration classes
}
```

@PropertySource

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {
    @Value("${app.name}")
    private String appName;
}
```

Dependency Injection Annotations

@Autowired

```
@Service
public class UserService {
    // Constructor injection (recommended)
    private final UserRepository repository;

    @Autowired
    public UserService(UserRepository repository) {
        this.repository = repository;
    }

    // Field injection (not recommended)
    @Autowired
    private EmailService emailService;

    // Setter injection
    @Autowired
    public void setNotificationService(NotificationService service) {
        this.notificationService = service;
    }
}
```

- Automatic dependency injection
- By type matching
- Can be optional: @Autowired(required = false)

@Qualifier

```
@Service
public class PaymentService {
    @Autowired
    @Qualifier("creditCardProcessor")
    private PaymentProcessor processor;
}
```

```
@Component
@Qualifier("creditCardProcessor")
public class CreditCardProcessor implements PaymentProcessor {
    // Implementation
}
```

- Disambiguates when multiple beans of same type exist
- Works with @Autowired

@Primary

```
@Component
@Primary
public class PrimaryEmailService implements EmailService {
    // This will be injected when multiple EmailService beans exist
}
```

@Value

```
@Component
public class AppSettings {
    @Value("${app.name}")
    private String applicationName;

    @Value("${app.version:1.0}")  // Default value
    private String version;

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

- Injects values from properties
- Supports default values
- Supports Spring Expression Language (SpEL)

Web MVC Annotations

@RequestMapping

```
@Controller
@RequestMapping("/api/v1")
```

```
public class UserController {

    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public ResponseEntity<List<User>> getUsers() {
        return ResponseEntity.ok(users);
    }

    @RequestMapping(
        value = "/users",
        method = RequestMethod.POST,
        consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE
    )
    public ResponseEntity<User> createUser(@RequestBody User user) {
        return ResponseEntity.ok(createdUser);
    }
}
```

HTTP Method Specific Annotations

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

    @GetMapping
    public List<User> getAllUsers() { }

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

    @PostMapping
    public User createUser(@RequestBody User user) { }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User user)

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) { }

    @PatchMapping("/{id}")
    public User partialUpdate(@PathVariable Long id, @RequestBody Map<Str
}
```

Parameter Binding Annotations

```
@RestController
public class UserController {

    // Path variables
    @GetMapping("/users/{id}/orders/{orderId}")
    public Order getOrder(
        @PathVariable Long id,
        @PathVariable("orderId") Long orderIdentifier) { }

    // Query parameters
    @GetMapping("/users")
    public List<User> getUsers(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "10") int size,
        @RequestParam(required = false) String search) { }

    // Request body
    @PostMapping("/users")
    public User createUser(@RequestBody @Valid User user) { }

    // Headers
    @GetMapping("/users")
    public List<User> getUsers(@RequestHeader("Authorization") String aut

    // Cookies
    @GetMapping("/profile")
    public User getProfile(@CookieValue("sessionId") String sessionId) {
}
```

@ResponseBody and @ResponseStatus

```
@Controller
public class UserController {

    @RequestMapping("/users/{id}")
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public User getUser(@PathVariable Long id) {
        return userService.findById(id);
    }
}
```

Validation Annotations

@Valid and @Validated

```
@RestController
public class UserController {

    @PostMapping("/users")
    public User createUser(@Valid @RequestBody User user) {
        return userService.create(user);
    }
}

@Service
@Validated
public class UserService {

    public User create(@Valid User user) {
        return repository.save(user);
    }
}
```

Bean Validation Annotations

```
@Entity
public class User {
    @NotNull(message = "Name cannot be null")
    @Size(min = 2, max = 50, message = "Name must be between 2 and 50 cha
    private String name;

    @Email(message = "Email should be valid")
    @NotBlank(message = "Email cannot be blank")
    private String email;

    @Min(value = 18, message = "Age must be at least 18")
    @Max(value = 120, message = "Age must be less than 120")
    private Integer age;
    @Pattern(regexp = "^[0-9]{10}$", message = "Phone number must be 10 c
    private String phoneNumber;

    @Past(message = "Birth date must be in the past")
    private LocalDate birthDate;

    @DecimalMin(value = "0.0", inclusive = false, message = "Salary must
    private BigDecimal salary;
}
```

Data Access Annotations

JPA Annotations

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "username", unique = true, nullable = false)
    private String username;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, fetch = Fetc
    private List<Order> orders;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "department_id")
    private Department department;

    @CreationTimestamp
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;
}
```

Spring Data Annotations

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u WHERE u.email = ?1")
    User findByEmail(String email);

    @Query(value = "SELECT * FROM users WHERE age > :age", nativeQuery =
    List<User> findUsersOlderThan(@Param("age") int age);

    @Modifying
    @Query("UPDATE User u SET u.active = false WHERE u.lastLogin < :date"
    int deactivateInactiveUsers(@Param("date") LocalDateTime date);
}
```

Transaction Management Annotations

@Transactional

```
@Service
public class UserService {

    @Transactional
    public User createUser(User user) {
        // All operations in single transaction
        User savedUser = userRepository.save(user);
        auditService.logUserCreation(savedUser);
        return savedUser;
    }

    @Transactional(readOnly = true)
    public User getUser(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    @Transactional(
        propagation = Propagation.REQUIRES_NEW,
        isolation = Isolation.READ_COMMITTED,
        timeout = 30,
        rollbackFor = {CustomException.class},
        noRollbackFor = {MinorException.class}
    )
    public void complexOperation() {
        // Complex transactional operation
    }
}
```

Caching Annotations

```
@Service
public class UserService {

    @Cacheable(value = "users", key = "#id")
    public User getUser(Long id) {
        return userRepository.findById(id);
    }

    @CachePut(value = "users", key = "#user.id")
    public User updateUser(User user) {
        return userRepository.save(user);
    }

    @CacheEvict(value = "users", key = "#id")
    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }

    @CacheEvict(value = "users", allEntries = true)
    public void clearAllUsers() {
        // Clear all cached users
    }
}
```

Asynchronous Processing Annotations

```
@Service
public class EmailService {

    @Async
    public CompletableFuture<String> sendEmail(String to, String subject,
        // Async email sending
        emailProvider.send(to, subject, body);
        return CompletableFuture.completedFuture("Email sent");
    }
    @Async("taskExecutor")
    public void processLargeFile(String filePath) {
        // Use specific executor
    }
}

@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean(name = "taskExecutor")
    public Executor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(2);
        executor.setMaxPoolSize(5);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("Async-");
        executor.initialize();
        return executor;
    }
}
```

Scheduling Annotations

```
@Component
public class ScheduledTasks {

    @Scheduled(fixedRate = 5000)  // Every 5 seconds
    public void reportCurrentTime() {
        System.out.println("Current time: " + new Date());
    }

    @Scheduled(fixedDelay = 5000)  // 5 seconds after previous execution
    public void doSomething() {
        // Task execution
    }

    @Scheduled(cron = "0 0 12 * * ?")  // Daily at noon
    public void dailyReport() {
        // Generate daily report
    }
    @Scheduled(cron = "0 0 9 * * MON-FRI", zone = "America/New_York")
    public void weekdayMorningTask() {
        // Weekday morning task
    }
}

@Configuration
@EnableScheduling
public class SchedulingConfig {
    // Enable scheduling
}
```

Event Handling Annotations

```
// Event class
public class UserRegistrationEvent extends ApplicationEvent {
    private final User user;

    public UserRegistrationEvent(Object source, User user) {
        super(source);
        this.user = user;
    }

    public User getUser() { return user; }
}

// Event publisher
@Service
public class UserService {

    @Autowired
    private ApplicationEventPublisher eventPublisher;

    public User registerUser(User user) {
        User savedUser = userRepository.save(user);
        eventPublisher.publishEvent(new UserRegistrationEvent(this, savec
        return savedUser;
    }
}

// Event listeners
@Component
public class UserEventListener {

    @EventListener
    public void handleUserRegistration(UserRegistrationEvent event) {
        // Send welcome email
        emailService.sendWelcomeEmail(event.getUser());
    }

    @EventListener
    @Async
    public void handleUserRegistrationAsync(UserRegistrationEvent event)
        // Async processing
        analyticsService.trackUserRegistration(event.getUser());
    }

    @EventListener(condition = "#event.user.premium")
    public void handlePremiumUserRegistration(UserRegistrationEvent event
        // Handle premium users only
    }
}
```

Testing Annotations

```
@SpringBootTest
@TestPropertySource(locations = "classpath:application-test.properties")
class UserServiceTest {

    @Autowired
    private UserService userService;

    @MockBean
    private UserRepository userRepository;

    @Test
    void testCreateUser() {
        // Test implementation
    }
}

@WebMvcTest(UserController.class)
class UserControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    void testGetUser() throws Exception {
        mockMvc.perform(get("/api/users/1"))
               .andExpect(status().isOk());
    }
}

@DataJpaTest
class UserRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository userRepository;

    @Test
    void testFindByEmail() {
        // Repository test
    }
}
```

Spring Boot Specifics

Auto-Configuration

Spring Boot automatically configures beans based on:

- Dependencies in classpath
- Existing beans
- Property values

```
@Configuration
@ConditionalOnClass(DataSource.class)
@ConditionalOnProperty(name = "spring.datasource.url")
```

```
@EnableConfigurationProperties(DataSourceProperties.class)
public class DataSourceAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public DataSource dataSource(DataSourceProperties properties) {
        return properties.initializeDataSourceBuilder().build();
    }
}
```

Conditional Annotations

```
@Configuration
public class ConditionalConfig {

    @Bean
    @ConditionalOnProperty(name = "feature.email.enabled", havingValue =
    public EmailService emailService() {
        return new EmailServiceImpl();
    }

    @Bean
    @ConditionalOnClass(RedisTemplate.class)
    public CacheManager redisCacheManager() {
        return new RedisCacheManager.Builder().build();
    }

    @Bean
    @ConditionalOnMissingBean(CacheManager.class)
    public CacheManager simpleCacheManager() {
        return new ConcurrentMapCacheManager();
    }

    @Bean
    @ConditionalOnProfile("production")
    public DataSource productionDataSource() {
        return new HikariDataSource();
    }
}
```

Configuration Properties

```
@ConfigurationProperties(prefix = "app.database")
@Component
public class DatabaseProperties {
    private String url;
    private String username;
    private String password;
    private int maxConnections = 10;
    private Duration connectionTimeout = Duration.ofSeconds(30);

    // Getters and setters
}

// application.yml
/*
app:
  database:
    url: jdbc:mysql://localhost:3306/mydb
    username: user
    password: pass
    max-connections: 20
    connection-timeout: 45s
*/
```

Profiles

```
@Configuration
@Profile("development")
public class DevConfig {

    @Bean
    public DataSource devDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .build();
    }
}

@Configuration
@Profile("production")
public class ProdConfig {

    @Bean
    public DataSource prodDataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setJdbcUrl("jdbc:mysql://prod-server:3306/app");
        return dataSource;
    }
}

@Service
@Profile("!test")  // Active in all profiles except test
public class ProductionService {
    // Production-specific logic
}
```

Starter Dependencies

Common starters:

- `spring-boot-starter-web` : Web applications
- `spring-boot-starter-data-jpa` : JPA data access
- `spring-boot-starter-data-mongodb` : MongoDB data access
- `spring-boot-starter-security` : Security features
- `spring-boot-starter-test` : Testing framework
- `spring-boot-starter-actuator` : Production features

Actuator

```
@Component
public class CustomHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        if (checkExternalService()) {
            return Health.up()
                .withDetail("service", "Available")
                .build();
        }
        return Health.down()
            .withDetail("service", "Unavailable")
            .build();
    }
}
```

```
@RestController
@RequestMapping("/actuator")
public class CustomActuatorEndpoint {

    @GetMapping("/custom-info")
    public Map<String, Object> customInfo() {
        Map<String, Object> info = new HashMap<>();
        info.put("app-version", "1.0.0");
        info.put("build-time", LocalDateTime.now());
        return info;
    }
}
```

Web Development

Exception Handling

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ResponseEntity<ErrorResponse> handleUserNotFound(UserNotFoundE
        ErrorResponse error = new ErrorResponse("USER_NOT_FOUND", ex.getM
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
    }

    @ExceptionHandler(ValidationException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ResponseEntity<ValidationErrorResponse> handleValidation(Valic
        ValidationErrorResponse error = new ValidationErrorResponse();
        error.setMessage("Validation failed");
        error.setErrors(ex.getErrors());
        return ResponseEntity.badRequest().body(error);
    }

    @ExceptionHandler(Exception.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public ResponseEntity<ErrorResponse> handleGeneral(Exception ex) {
        ErrorResponse error = new ErrorResponse("INTERNAL_ERROR", "An une
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).bo
    }
}
```

Interceptors

```
@Component
public class LoggingInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletRespo
        System.out.println("Pre Handle method is Calling");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletRespons
        System.out.println("Post Handle method is Calling");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletRe
        System.out.println("Request and Response is completed");
    }
}

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Autowired
    private LoggingInterceptor loggingInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(loggingInterceptor);
    }
}
```

CORS Configuration

```
@Configuration
public class CorsConfig {

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOriginPatterns(Arrays.asList("*"));
        configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PU
        configuration.setAllowedHeaders(Arrays.asList("*"));
        configuration.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigu
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}

// Or using annotations
@RestController
@CrossOrigin(origins = "http://localhost:3000")
public class UserController {
    // Controller methods
}
```

Data Access

Spring Data JPA

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    // Query methods
    List<User> findByName(String name);
    List<User> findByAgeGreaterThan(int age);
    List<User> findByNameContainingIgnoreCase(String name);

    // Custom queries
    @Query("SELECT u FROM User u WHERE u.email = ?1")
    Optional<User> findByEmail(String email);

    @Query(value = "SELECT * FROM users WHERE created_date > :date", nati
    List<User> findUsersCreatedAfter(@Param("date") LocalDateTime date);

    // Projections
    @Query("SELECT u.name as name, u.email as email FROM User u")
    List<UserProjection> findAllProjections();

    // Modifying queries
    @Modifying
    @Query("UPDATE User u SET u.active = false WHERE u.id = ?1")
    int deactivateUser(Long id);

    // Pagination and sorting
    Page<User> findByAgeGreaterThan(int age, Pageable pageable);
    List<User> findByName(String name, Sort sort);
}

// Projection interface
public interface UserProjection {
    String getName();
    String getEmail();
}
```

Custom Repository Implementation

```
public interface UserRepositoryCustom {
    List<User> findUsersWithComplexCriteria(UserSearchCriteria criteria);
}

@Repository
public class UserRepositoryImpl implements UserRepositoryCustom {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public List<User> findUsersWithComplexCriteria(UserSearchCriteria cri
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<User> query = cb.createQuery(User.class);
        Root<User> root = query.from(User.class);

        List<Predicate> predicates = new ArrayList<>();
        if (criteria.getName() != null) {
            predicates.add(cb.like(root.get("name"), "%" + criteria.getNa
        }

        if (criteria.getMinAge() != null) {
            predicates.add(cb.greaterThanOrEqualTo(root.get("age"), crite
        }

        query.where(predicates.toArray(new Predicate[0]));
        return entityManager.createQuery(query).getResultList();
    }
}

public interface UserRepository extends JpaRepository<User, Long>, UserRe
    // Combines default and custom methods
}
```

Database Migrations with Flyway

```sql
-- V1__Create_users_table.sql
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    email VARCHAR(100) NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- V2__Add_user_profile.sql
ALTER TABLE users ADD COLUMN first_name VARCHAR(50);
ALTER TABLE users ADD COLUMN last_name VARCHAR(50);
ALTER TABLE users ADD COLUMN phone VARCHAR(20);
```

```
# application.properties
spring.flyway.enabled=true
spring.flyway.locations=classpath:db/migration
spring.flyway.baseline-on-migrate=true
```

Security

Basic Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exce
        http
            .authorizeHttpRequests(authz -> authz
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .requestMatchers(HttpMethod.GET, "/api/users/**").hasAnyR
                .requestMatchers(HttpMethod.POST, "/api/users/**").hasRol
                .anyRequest().authenticated()
            )
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthConvert
            )
            .csrf(csrf -> csrf.disable())
            .cors(cors -> cors.configurationSource(corsConfigurationSourc

        return http.build();
    }

    @Bean
    public JwtAuthenticationConverter jwtAuthConverter() {
        JwtGrantedAuthoritiesConverter authoritiesConverter = new JwtGran
        authoritiesConverter.setAuthorityPrefix("ROLE_");
        authoritiesConverter.setAuthoritiesClaimName("roles");

        JwtAuthenticationConverter converter = new JwtAuthenticationConve
        converter.setJwtGrantedAuthoritiesConverter(authoritiesConverter)
        return converter;
    }
}
```

JWT Authentication

```
@Component
public class JwtUtils {

    private static final String SECRET = "mySecretKey";
    private static final int JWT_EXPIRATION = 86400000; // 24 hours

    public String generateJwtToken(Authentication authentication) {
        UserPrincipal userPrincipal = (UserPrincipal) authentication.getP

        return Jwts.builder()
                .setSubject(userPrincipal.getUsername())
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + JWT_
                .signWith(SignatureAlgorithm.HS512, SECRET)
                .compact();
    }

    public String getUsernameFromToken(String token) {
        return Jwts.parser()
                .setSigningKey(SECRET)
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(SECRET).parseClaimsJws(token);
            return true;
        } catch (JwtException | IllegalArgumentException e) {
            return false;
        }
    }
}

@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtils jwtUtils;
    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServl
            FilterChain filterChain) throws ServletException, IOExceptior

        String jwt = parseJwt(request);

        if (jwt != null && jwtUtils.validateToken(jwt)) {
            String username = jwtUtils.getUsernameFromToken(jwt);
            UserDetails userDetails = userDetailsService.loadUserByUserna

            UsernamePasswordAuthenticationToken authentication =
                new UsernamePasswordAuthenticationToken(userDetails, null
            authentication.setDetails(new WebAuthenticationDetailsSource(

            SecurityContextHolder.getContext().setAuthentication(authenti
        }

        filterChain.doFilter(request, response);
    }

    private String parseJwt(HttpServletRequest request) {
        String headerAuth = request.getHeader("Authorization");
        if (StringUtils.hasText(headerAuth) && headerAuth.startsWith("Bea
            return headerAuth.substring(7);
        }
        return null;
    }
}
```

Method Level Security

```
@Configuration
@EnableMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig {
    // Method security configuration
}

@Service
public class UserService {
    @PreAuthorize("hasRole('ADMIN')")
    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }

    @PreAuthorize("hasRole('ADMIN') or #username == authentication.name")
    public User getUserByUsername(String username) {
        return userRepository.findByUsername(username);
    }

    @PostAuthorize("returnObject.username == authentication.name or hasRc
    public User getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    @PreAuthorize("@userService.isOwner(#userId, authentication.name)")
    public void updateUserProfile(Long userId, UserProfile profile) {
        // Update user profile
    }

    public boolean isOwner(Long userId, String username) {
        User user = userRepository.findById(userId).orElse(null);
        return user != null && user.getUsername().equals(username);
    }
}
```

Testing

Unit Testing

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

    @Mock
    private UserRepository userRepository;

    @Mock
    private EmailService emailService;

    @InjectMocks
    private UserService userService;

    @Test
    @DisplayName("Should create user successfully")
    void shouldCreateUserSuccessfully() {
        // Given
        User user = new User("john", "john@example.com");
        User savedUser = new User("john", "john@example.com");
        savedUser.setId(1L);

        when(userRepository.save(any(User.class))).thenReturn(savedUser);

        // When
        User result = userService.createUser(user);

        // Then
        assertThat(result.getId()).isEqualTo(1L);
        assertThat(result.getUsername()).isEqualTo("john");
        verify(userRepository).save(user);
        verify(emailService).sendWelcomeEmail(savedUser);
    }

    @Test
    @DisplayName("Should throw exception when user already exists")
    void shouldThrowExceptionWhenUserExists() {
        // Given
        User user = new User("john", "john@example.com");
        when(userRepository.findByUsername("john")).thenReturn(Optional.c

        // When & Then
        assertThrows(UserAlreadyExistsException.class, () -> {
            userService.createUser(user);
        });

        verify(userRepository, never()).save(any());
    }
}
```

Integration Testing

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_POF
@TestPropertySource(locations = "classpath:application-test.properties")
```

```
@Sql(scripts = "/test-data.sql", executionPhase = Sql.ExecutionPhase.BEFO
@Sql(scripts = "/cleanup.sql", executionPhase = Sql.ExecutionPhase.AFTER_
class UserIntegrationTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private UserRepository userRepository;

    @Test
    void shouldCreateUserEndToEnd() {
        // Given
        CreateUserRequest request = new CreateUserRequest("john", "john@e

        // When
        ResponseEntity<User> response = restTemplate.postForEntity("/api/

        // Then
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.CREATED
        assertThat(response.getBody().getUsername()).isEqualTo("john");

        // Verify in database
        Optional<User> savedUser = userRepository.findByUsername("john");
        assertThat(savedUser).isPresent();
    }
}
```

Web Layer Testing

```
@WebMvcTest(UserController.class)
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Autowired
    private ObjectMapper objectMapper;
    @Test
    void shouldReturnUserWhenValidId() throws Exception {
        // Given
        User user = new User("john", "john@example.com");
        user.setId(1L);
        when(userService.getUserById(1L)).thenReturn(user);

        // When & Then
        mockMvc.perform(get("/api/users/1"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.id").value(1))
                .andExpect(jsonPath("$.username").value("john"))
                .andExpect(jsonPath("$.email").value("john@example.com"))
    }

    @Test
    void shouldReturnBadRequestWhenInvalidInput() throws Exception {
        // Given
        CreateUserRequest request = new CreateUserRequest("", "invalid-em

        // When & Then
        mockMvc.perform(post("/api/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.errors").exists());
    }
}
```

Data Layer Testing

```
@DataJpaTest
class UserRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository userRepository;

    @Test
    void shouldFindUserByEmail() {
        // Given
        User user = new User("john", "john@example.com");
        entityManager.persistAndFlush(user);

        // When
        Optional<User> found = userRepository.findByEmail("john@example.c

        // Then
        assertThat(found).isPresent();
        assertThat(found.get().getUsername()).isEqualTo("john");
    }

    @Test
    void shouldReturnEmptyWhenEmailNotFound() {
        // When
        Optional<User> found = userRepository.findByEmail("nonexistent@ex

        // Then
        assertThat(found).isEmpty();
    }
}
```

Test Containers

```
@Testcontainers
@SpringBootTest
class UserServiceIntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("p
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test");

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername)
        registry.add("spring.datasource.password", postgres::getPassword)
    }

    @Autowired
    private UserService userService;

    @Test
    void shouldCreateUserInRealDatabase() {
        // Test with real PostgreSQL database
        User user = new User("john", "john@example.com");
        User saved = userService.createUser(user);

        assertThat(saved.getId()).isNotNull();
    }
}
```

Messaging & Events

Application Events

```
// Custom event
public class OrderCompletedEvent extends ApplicationEvent {
    private final Order order;
    private final User user;

    public OrderCompletedEvent(Object source, Order order, User user) {
        super(source);
        this.order = order;
        this.user = user;
    }

    // Getters
}

// Event publisher
@Service
public class OrderService {

    @Autowired
    private ApplicationEventPublisher eventPublisher;

    @Transactional
    public Order completeOrder(Long orderId) {
        Order order = orderRepository.findById(orderId).orElseThrow();
        order.setStatus(OrderStatus.COMPLETED);
        Order savedOrder = orderRepository.save(order);

        // Publish event
        eventPublisher.publishEvent(new OrderCompletedEvent(this, savedOr

        return savedOrder;
    }
}

// Event listeners
@Component
public class OrderEventListener {

    @EventListener
    @Async
    public void handleOrderCompleted(OrderCompletedEvent event) {
        // Send confirmation email
        emailService.sendOrderConfirmation(event.getUser(), event.getOrde
    }

    @EventListener
    @Async
    public void updateInventory(OrderCompletedEvent event) {
        // Update inventory
        inventoryService.updateStock(event.getOrder());
    }

    @EventListener(condition = "#event.order.totalAmount > 1000")
    public void handleHighValueOrder(OrderCompletedEvent event) {
        // Special handling for high-value orders
        loyaltyService.addBonusPoints(event.getUser(), 100);
    }
}
```

JMS Integration

```
@Configuration
@EnableJms
public class JmsConfig {

    @Bean
    public JmsListenerContainerFactory<?> myFactory(ConnectionFactory cor
            DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory = new DefaultJmsLister
        configurer.configure(factory, connectionFactory);
        return factory;
    }
}

@Component
public class MessageProducer {

    @Autowired
    private JmsTemplate jmsTemplate;

    public void sendMessage(String destination, Object message) {
        jmsTemplate.convertAndSend(destination, message);
    }

    public void sendWithProperties(String destination, Object message, Ma
        jmsTemplate.convertAndSend(destination, message, messagePostProce
            properties.forEach((key, value) -> {
                messagePostProcessor.setObjectProperty(key, value);
            });
            return messagePostProcessor;
        });
    }
}

@Component
public class MessageConsumer {

    @JmsListener(destination = "order.queue")
    public void handleOrderMessage(Order order) {
        // Process order message
        orderService.processOrder(order);
    }

    @JmsListener(destination = "notification.queue",
                 selector = "type = 'EMAIL'")
    public void handleEmailNotification(NotificationMessage message) {
        // Handle email notifications only
        emailService.sendNotification(message);
    }

    @JmsListener(destination = "user.topic",
```

```java
                    containerFactory = "myFactory")
    public void handleUserUpdates(UserUpdateMessage message) {
        // Handle user updates
        userService.handleUpdate(message);
    }
}
```

RabbitMQ Integration

```java
@Configuration
@EnableRabbit
public class RabbitConfig {

    @Bean
    public TopicExchange orderExchange() {
        return new TopicExchange("order.exchange");
    }

    @Bean
    public Queue orderProcessingQueue() {
        return QueueBuilder.durable("order.processing").build();
    }

    @Bean
    public Queue orderNotificationQueue() {
        return QueueBuilder.durable("order.notification").build();
    }

    @Bean
    public Binding orderProcessingBinding() {
        return BindingBuilder
                .bind(orderProcessingQueue())
                .to(orderExchange())
                .with("order.created");
    }

    @Bean
    public Binding orderNotificationBinding() {
        return BindingBuilder
                .bind(orderNotificationQueue())
                .to(orderExchange())
                .with("order.*");
    }
}

@Component
public class RabbitMessageProducer {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void sendOrderCreated(Order order) {
        rabbitTemplate.convertAndSend("order.exchange", "order.created",
    }

    public void sendOrderCompleted(Order order) {
        rabbitTemplate.convertAndSend("order.exchange", "order.completed"
    }
}

@Component
public class RabbitMessageConsumer {

    @RabbitListener(queues = "order.processing")
    public void processOrder(Order order) {
        // Process the order
        orderService.processOrder(order);
    }

    @RabbitListener(queues = "order.notification")
    public void sendNotification(@Payload Order order, @Header Map<String
        // Send notification based on order status
        notificationService.sendOrderNotification(order, headers);
    }
}
```

Microservices & Cloud

Spring Cloud Configuration

```
// Service Discovery with Eureka
@SpringBootApplication
@EnableEurekaClient
```

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

// application.yml
/*
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
*/
```

Circuit Breaker with Resilience4j

```
@Component
public class ExternalApiClient {

    @CircuitBreaker(name = "payment-service", fallbackMethod = "fallbackP
    @Retry(name = "payment-service")
    @TimeLimiter(name = "payment-service")
    public CompletableFuture<PaymentResponse> processPayment(PaymentReque
        return CompletableFuture.supplyAsync(() -> {
            // Call external payment service
            return paymentServiceClient.processPayment(request);
        });
    }

    public CompletableFuture<PaymentResponse> fallbackPayment(PaymentRequ
        // Fallback logic
        PaymentResponse response = new PaymentResponse();
        response.setStatus("PENDING");
        response.setMessage("Payment service temporarily unavailable");
        return CompletableFuture.completedFuture(response);
    }

    @Bulkhead(name = "inventory-service", type = Bulkhead.Type.THREADPOOL
    public CompletableFuture<InventoryResponse> checkInventory(String pro
        return CompletableFuture.supplyAsync(() -> {
            return inventoryServiceClient.checkInventory(productId);
        });
    }
}

// application.yml
/*
resilience4j:
  circuitbreaker:
    instances:
      payment-service:
        failure-rate-threshold: 50
        wait-duration-in-open-state: 30s
        sliding-window-size: 10
        minimum-number-of-calls: 5
  retry:
    instances:
      payment-service:
        max-attempts: 3
        wait-duration: 2s
  timelimiter:
    instances:
      payment-service:
        timeout-duration: 5s
*/
```

API Gateway

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

    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
                .route("user-service", r -> r.path("/api/users/**")
                        .filters(f -> f.addRequestHeader("X-Gateway", "Sp
                        .uri("lb://user-service"))
                .route("order-service", r -> r.path("/api/orders/**")
                        .filters(f -> f.circuitBreaker(config -> config
                                .setName("order-service")
                                .setFallbackUri("forward:/fallback")))
                        .uri("lb://order-service"))
                .build();
    }
}

@RestController
public class FallbackController {

    @RequestMapping("/fallback")
    public ResponseEntity<String> fallback() {
        return ResponseEntity.ok("Service temporarily unavailable. Please
    }
}
```

Configuration Server

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

// application.yml
/*
server:
  port: 8888
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/your-org/config-repo
          default-label: main
*/

// Client configuration
/*
spring:
  application:
    name: user-service
  cloud:
    config:
      uri: http://localhost:8888
      fail-fast: true
*/
```

DevOps & Production

Monitoring with Micrometer

```
@Component
public class CustomMetrics {

    private final Counter orderCounter;
    private final Timer orderProcessingTimer;
    private final Gauge activeUsersGauge;

    public CustomMetrics(MeterRegistry meterRegistry) {
        this.orderCounter = Counter.builder("orders.created")
                .description("Number of orders created")
                .tag("type", "online")
                .register(meterRegistry);

        this.orderProcessingTimer = Timer.builder("orders.processing.time
                .description("Order processing time")
                .register(meterRegistry);

        this.activeUsersGauge = Gauge.builder("users.active")
                .description("Number of active users")
                .register(meterRegistry, this, CustomMetrics::getActiveUs
    }

    public void incrementOrderCount() {
        orderCounter.increment();
    }

    public void recordOrderProcessingTime(Duration duration) {
        orderProcessingTimer.record(duration);
    }
    private double getActiveUserCount() {
        return userService.getActiveUserCount();
    }
}

@Service
public class OrderService {

    @Autowired
    private CustomMetrics metrics;

    @Timed(value = "orders.processing.time", description = "Time taken tc
    public Order processOrder(Order order) {
        metrics.incrementOrderCount();

        // Process order logic
        Order processedOrder = doProcessOrder(order);

        return processedOrder;
    }
}
```

Health Checks

```
@Component
public class DatabaseHealthIndicator implements HealthIndicator {

    @Autowired
    private DataSource dataSource;

    @Override
    public Health health() {
        try (Connection connection = dataSource.getConnection()) {
            if (connection.isValid(1)) {
                return Health.up()
                        .withDetail("database", "Available")
                        .withDetail("validationTimeout", "1 second")
                        .build();
            }
        } catch (SQLException ex) {
            return Health.down()
```

```
                    .withDetail("database", "Unavailable")
                    .withDetail("error", ex.getMessage())
                    .build();
        }

        return Health.down()
                .withDetail("database", "Connection validation failed")
                .build();
    }
}

@Component
public class ExternalServiceHealthIndicator implements HealthIndicator {

    @Autowired
    private RestTemplate restTemplate;

    @Override
    public Health health() {
        try {
            ResponseEntity<String> response = restTemplate.getForEntity(
                    "http://external-service/health", String.class);

            if (response.getStatusCode().is2xxSuccessful()) {
                return Health.up()
                        .withDetail("external-service", "Available")
                        .build();
            }
        } catch (Exception ex) {
            return Health.down()
                    .withDetail("external-service", "Unavailable")
                    .withDetail("error", ex.getMessage())
                    .build();
        }

        return Health.down()
                .withDetail("external-service", "Unexpected response")
                .build();
    }
}
```

Distributed Tracing

```
@Configuration
public class TracingConfig {

    @Bean
    public Sender sender() {
        return OkHttpSender.create("http://zipkin:9411/api/v2/spans");
    }

    @Bean
    public AsyncReporter<Span> spanReporter() {
        return AsyncReporter.create(sender());
    }

    @Bean
    public Tracing tracing() {
        return Tracing.newBuilder()
                .localServiceName("user-service")
                .spanReporter(spanReporter())
                .sampler(Sampler.create(1.0f)) // Sample all traces
                .build();
    }
}

@Service
public class UserService {

    @NewSpan("user-creation")
    public User createUser(@SpanTag("username") String username, User use
        // This method will be traced
        User savedUser = userRepository.save(user);

        // Add custom span annotations
        Span currentSpan = Span.current();
        currentSpan.setAttribute("user.id", savedUser.getId().toString())
        currentSpan.addEvent("User validation completed");

        return savedUser;
    }
}
```

Docker Configuration

```
# Dockerfile
FROM openjdk:17-jre-slim

VOLUME /tmp

COPY target/user-service-1.0.0.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "/app.jar"]

# docker-compose.yml
version: '3.8'
services:
  user-service:
    build: .
    ports:
      - "8080:8080"
    environment:
      - SPRING_PROFILES_ACTIVE=docker
      - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres:5432/userdb
    depends_on:
      - postgres
      - redis

  postgres:
    image: postgres:13
    environment:
      POSTGRES_DB: userdb
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:6-alpine
    ports:
      - "6379:6379"

volumes:
  postgres_data:
```

Best Practices

1. Dependency Injection

- **Use constructor injection** over field injection
- **Make dependencies final** when using constructor injection
- **Avoid circular dependencies** by proper layering

```
// Good
@Service
public class UserService {
    private final UserRepository userRepository;
    private final EmailService emailService;

    public UserService(UserRepository userRepository, EmailService emailS
        this.userRepository = userRepository;
        this.emailService = emailService;
    }
}

// Avoid
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    @Autowired
    private EmailService emailService;
}
```

2. Exception Handling

- **Use specific exception types**
- **Handle exceptions at appropriate layers**
- **Provide meaningful error messages**

```
// Custom exceptions
public class UserNotFoundException extends RuntimeException {
    public UserNotFoundException(Long id) {
        super("User not found with id: " + id);
    }
}

// Service layer
@Service
public class UserService {
    public User getUserById(Long id) {
        return userRepository.findById(id)
                .orElseThrow(() -> new UserNotFoundException(id));
    }
}

// Global exception handler
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ErrorResponse handleUserNotFound(UserNotFoundException ex) {
        return new ErrorResponse("USER_NOT_FOUND", ex.getMessage());
    }
}
```

3. Configuration Management

- **Externalize configuration**
- **Use profiles for different environments**
- **Validate configuration properties**

```
@ConfigurationProperties(prefix = "app")
@Validated
@Component
public class AppProperties {

    @NotBlank
    private String name;

    @Min(1)
    @Max(100)
    private int maxConnections;

    @Valid
    private Database database = new Database();
    public static class Database {
        @NotBlank
        private String url;

        @NotBlank
        private String username;

        // getters and setters
    }

    // getters and setters
}
```

4. Testing Strategy

- **Write unit tests for business logic**
- **Use integration tests for API endpoints**
- **Mock external dependencies**

```
// Unit test
@ExtendWith(MockitoExtension.class)
class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    void shouldCreateUser() {
        // Given
        User user = new User("john");
        when(userRepository.save(any())).thenReturn(user);

        // When
        User result = userService.createUser(user);

        // Then
        assertThat(result.getUsername()).isEqualTo("john");
    }
}
```

5. Security Best Practices

- **Never store passwords in plain text**
- **Use HTTPS in production**
- **Implement proper authentication and authorization**
- **Validate all inputs**

```
@RestController
@Validated
public class UserController {

    @PostMapping("/users")
    public ResponseEntity<User> createUser(
            @Valid @RequestBody CreateUserRequest request) {
        // Input is automatically validated
        User user = userService.createUser(request);
        return ResponseEntity.status(HttpStatus.CREATED).body(user);
    }
}

// Request DTO with validation
public class CreateUserRequest {
    @NotBlank(message = "Username is required")
    @Size(min = 3, max = 20, message = "Username must be between 3 and 2
    private String username;

    @Email(message = "Email must be valid")
    @NotBlank(message = "Email is required")
    private String email;

    // getters and setters
}
```

6. Performance Optimization

- **Use appropriate JPA fetch strategies**
- **Implement caching for frequently accessed data**
- **Use pagination for large datasets**

```
@Entity
public class User {
    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
    private List<Order> orders; // Lazy loading
}

@Service
public class UserService {

    @Cacheable(value = "users", key = "#id")
    public User getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    public Page<User> getUsers(Pageable pageable) {
        return userRepository.findAll(pageable);
    }
}
```

Common Pitfalls

1. Circular Dependencies

Problem:

```
@Service
public class UserService {
    @Autowired
    private OrderService orderService; // Circular dependency
}

@Service
public class OrderService {
    @Autowired
    private UserService userService; // Circular dependency
}
```

Solution:

```
// Use events or extract common logic
@Service
public class UserService {
    @Autowired
    private ApplicationEventPublisher eventPublisher;
    public void createUser(User user) {
        User savedUser = userRepository.save(user);
        eventPublisher.publishEvent(new UserCreatedEvent(savedUser));
    }
}

@EventListener
public void handleUserCreated(UserCreatedEvent event) {
    orderService.initializeUserOrders(event.getUser());
}
```

2. N+1 Query Problem

Problem:

```
@Entity
public class User {
    @OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
    private List<Order> orders;
}

// This causes N+1 queries
List<User> users = userRepository.findAll();
for (User user : users) {
    System.out.println(user.getOrders().size()); // Each iteration trigge
}
```

Solution:

```
// Use JOIN FETCH or @EntityGraph
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT DISTINCT u FROM User u LEFT JOIN FETCH u.orders")
    List<User> findAllWithOrders();

    @EntityGraph(attributePaths = {"orders"})
    List<User> findAll();
}
```

3. Improper Transaction Boundaries

Problem:

```
@Service
public class UserService {

    @Transactional
    public void processUsers() {
        List<User> users = userRepository.findAll();
        for (User user : users) {
            processUser(user); // If this fails, all previous work is rol
        }
    }
}
```

Solution:

```
@Service
public class UserService {

    public void processUsers() {
        List<User> users = userRepository.findAll();
        for (User user : users) {
            processUserSafely(user); // Process each user in separate tra
        }
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void processUserSafely(User user) {
        try {
            processUser(user);
        } catch (Exception e) {
            log.error("Failed to process user: " + user.getId(), e);
            // Continue with next user
        }
    }
}
```

4. Memory Leaks with Prototype Beans

Problem:

```
@Component
@Scope("prototype")
public class PrototypeBean {
    @PreDestroy
    public void cleanup() {
        // This won't be called automatically for prototype beans
    }
}

@Service
public class ServiceWithMemoryLeak {
    @Autowired
    private ApplicationContext context;

    public void doSomething() {
        // Creating many prototype beans without proper cleanup
        for (int i = 0; i < 1000; i++) {
            PrototypeBean bean = context.getBean(PrototypeBean.class);
            // No cleanup - memory leak!
        }
    }
}
```

Solution:

```
@Service
public class ServiceWithProperCleanup {
    @Autowired
    private ApplicationContext context;

    public void doSomething() {
        List<PrototypeBean> beans = new ArrayList<>();
        try {
            for (int i = 0; i < 1000; i++) {
                PrototypeBean bean = context.getBean(PrototypeBean.class)
                beans.add(bean);
                // Use bean
            }
        } finally {
            // Manual cleanup for prototype beans
            beans.forEach(this::destroyBean);
        }
    }
    private void destroyBean(PrototypeBean bean) {
        if (bean instanceof DisposableBean) {
            try {
                ((DisposableBean) bean).destroy();
            } catch (Exception e) {
                log.error("Error destroying bean", e);
            }
        }
    }
}
```

5. Ignoring Exception Handling in Async Methods

Problem:

```
@Service
public class AsyncService {

    @Async
    public void processAsync() {
        throw new RuntimeException("This exception is silently ignored!")
    }
}
```

Solution:

```
@Service
public class AsyncService {

    @Async
    public CompletableFuture<Void> processAsync() {
        try {
            // Async processing logic
            return CompletableFuture.completedFuture(null);
        } catch (Exception e) {
            log.error("Async processing failed", e);
            return CompletableFuture.failedFuture(e);
        }
    }

    @Async
    public void processAsyncWithTryCatch() {
        try {
            // Processing logic
        } catch (Exception e) {
            log.error("Async processing failed", e);
            // Handle exception appropriately
        }
    }
}
```

6. Improper Use of @Autowired

Problem:

```
@Component
public class BadExample {
    @Autowired
    private SomeService someService; // Field injection - harder to test

    @Autowired(required = false)
    private OptionalService optionalService; // Nullable dependencies are
}
```

Solution:

```
@Component
public class GoodExample {
    private final SomeService someService;
    private final Optional<OptionalService> optionalService;

    public GoodExample(SomeService someService,
                       Optional<OptionalService> optionalService) {
        this.someService = someService;
        this.optionalService = optionalService;
    }

    public void doSomething() {
        someService.performAction();
        optionalService.ifPresent(service -> service.performOptionalActic
    }
}
```

7. Not Handling Database Connection Pooling

Problem:

```
// Default connection pool settings might not be optimal
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=user
spring.datasource.password=pass
```

Solution:

```
// Properly configured connection pool
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=300000
spring.datasource.hikari.max-lifetime=600000
spring.datasource.hikari.connection-timeout=20000
spring.datasource.hikari.leak-detection-threshold=60000

@Configuration
public class DatabaseConfig {

    @Bean
    @ConfigurationProperties("spring.datasource.hikari")
    public HikariDataSource dataSource(DataSourceProperties properties) {
        HikariDataSource dataSource = properties
                .initializeDataSourceBuilder()
                .type(HikariDataSource.class)
                .build();

        // Additional configuration
        dataSource.setPoolName("MyAppPool");
        dataSource.addDataSourceProperty("cachePrepStmts", "true");
        dataSource.addDataSourceProperty("prepStmtCacheSize", "250");
        dataSource.addDataSourceProperty("prepStmtCacheSqlLimit", "2048")

        return dataSource;
    }
}
```

8. Poor Error Handling in REST APIs

Problem:

```
@RestController
public class BadController {

    @GetMapping("/users/{id}")
    public User getUser(@PathVariable Long id) {
        return userService.getUserById(id); // What if user doesn't exist
    }

    @PostMapping("/users")
    public User createUser(@RequestBody User user) {
        return userService.createUser(user); // What if validation fails?
    }
}
```

Solution:

```
@RestController
public class GoodController {

    @GetMapping("/users/{id}")
    public ResponseEntity<User> getUser(@PathVariable Long id) {
        try {
            User user = userService.getUserById(id);
            return ResponseEntity.ok(user);
        } catch (UserNotFoundException e) {
            return ResponseEntity.notFound().build();
        }
    }

    @PostMapping("/users")
    public ResponseEntity<?> createUser(@Valid @RequestBody CreateUserRec
        try {
            User user = userService.createUser(request);
            return ResponseEntity.status(HttpStatus.CREATED).body(user);
        } catch (UserAlreadyExistsException e) {
            return ResponseEntity.status(HttpStatus.CONFLICT)
                    .body(new ErrorResponse("USER_EXISTS", e.getMessage()
        }
    }
}
```

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleUserNotFound(UserNotFoundE
        ErrorResponse error = new ErrorResponse("USER_NOT_FOUND", ex.getM
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ValidationErrorResponse> handleValidation(Metho
        ValidationErrorResponse error = new ValidationErrorResponse();
        error.setMessage("Validation failed");

        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(fieldError -> {
            errors.put(fieldError.getField(), fieldError.getDefaultMessag
        });
        error.setErrors(errors);

        return ResponseEntity.badRequest().body(error);
    }
}
```

Advanced Topics

Custom Auto-Configuration

```
@Configuration
@ConditionalOnClass(MyService.class)
@ConditionalOnProperty(name = "myapp.service.enabled", havingValue = "tru
@EnableConfigurationProperties(MyServiceProperties.class)
public class MyServiceAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public MyService myService(MyServiceProperties properties) {
        return new MyServiceImpl(properties);
    }

    @Configuration
    @ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.
    static class ServletConfiguration {

        @Bean
        public MyServiceController myServiceController(MyService myServic
            return new MyServiceController(myService);
        }
    }
}

// META-INF/spring.factories
/*
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.example.autoconfigure.MyServiceAutoConfiguration
*/
```

Custom Starter

```
<!-- my-spring-boot-starter/pom.xml -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure</artifactId>
</dependency>
<dependency>
    <groupId>com.example</groupId>
    <artifactId>my-service-autoconfigure</artifactId>
</dependency>
```

Reactive Programming with WebFlux

```
@RestController
@RequestMapping("/api/reactive")
public class ReactiveUserController {

    @Autowired
    private ReactiveUserService userService;

    @GetMapping("/users")
    public Flux<User> getAllUsers() {
        return userService.findAllUsers();
    }
    @GetMapping("/users/{id}")
    public Mono<ResponseEntity<User>> getUser(@PathVariable String id) {
        return userService.findById(id)
                .map(user -> ResponseEntity.ok(user))
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    @PostMapping("/users")
    public Mono<User> createUser(@RequestBody Mono<User> userMono) {
        return userMono.flatMap(userService::save);
    }

    @GetMapping(value = "/users/stream", produces = MediaType.TEXT_EVENT_
    public Flux<User> streamUsers() {
        return userService.findAllUsers()
                .delayElements(Duration.ofSeconds(1));
    }
}

@Service
public class ReactiveUserService {

    @Autowired
    private ReactiveUserRepository userRepository;

    public Flux<User> findAllUsers() {
        return userRepository.findAll();
    }

    public Mono<User> findById(String id) {
        return userRepository.findById(id);
    }

    public Mono<User> save(User user) {
        return userRepository.save(user);
    }

    public Flux<User> findByAgeGreaterThan(int age) {
        return userRepository.findByAgeGreaterThan(age)
                .doOnNext(user -> System.out.println("Processing user: "
                .filter(user -> user.isActive())
                .take(10);
        }
    }
```

Conclusion

This comprehensive guide covers the essential aspects of Spring Framework and Spring Boot development. Key takeaways:

1. **Start with Spring Boot** for new projects - it provides sensible defaults and auto-configuration
2. **Use constructor injection** for better testability and immutability
3. **Follow layered architecture** with proper separation of concerns
4. **Implement comprehensive testing** at all layers
5. **Handle exceptions gracefully** with proper error responses
6. **Monitor and secure** your applications for production
7. **Keep learning** - Spring ecosystem is vast and constantly evolving

Next Steps for Learning:

1. **Hands-on Practice**: Build a complete application using the concepts covered
2. **Spring Security Deep Dive**: Learn OAuth2, JWT, and method-level security
3. **Spring Cloud**: Explore microservices patterns and distributed systems
4. **Reactive Programming**: Learn WebFlux for high-concurrency applications
5. **Production Deployment**: Study containerization, monitoring, and CI/CD

Additional Resources:

- **Official Documentation**: spring.io
- **Spring Boot Reference**: docs.spring.io/spring-boot
- **Spring Guides**: spring.io/guides
- **Community**: Stack Overflow, Spring Community Forums

Remember: The Spring ecosystem is extensive, and this guide provides a solid foundation. Focus on understanding the core concepts first, then gradually explore advanced topics based on your project requirements.