**25 Advanced Spring Boot Features Explained with Real-World Relevance**

This document explains 25 advanced Spring Boot features with context, why they matter in real-world systems (scalability, resilience, observability, latency, etc.), and code samples to demonstrate their use.

**1. Custom Spring Boot Starters**

**What:** Create reusable dependency groups.  
**Why:** Promotes consistency, simplifies config, and improves collaboration.  
**Use Case:** Internal team uses a common security + logging setup.

<dependency>

<groupId>com.example</groupId>

<artifactId>my-company-starter</artifactId>

</dependency>

**Inside starter:**

@Configuration

public class MyCompanyAutoConfig {

@Bean

public Logger logger() {

return LoggerFactory.getLogger("MyCompany");

}

}

**2. Auto-Configuration Customization**

**What:** Customize default behavior with conditions.  
**Why:** Allows app-specific logic without losing auto-config benefits.  
**Use Case:** Provide default Redis bean if one isn’t defined.

@Configuration

public class MyAutoConfig {

@Bean

@ConditionalOnMissingBean

public RedisTemplate<?, ?> redisTemplate() {

return new RedisTemplate<>();

}

}

**3. @Conditional Annotations**

**What:** Load beans based on presence of properties, classes, etc.  
**Why:** Enable feature toggles, optional modules, or cloud support.  
**Use Case:** Enable Kafka config only if kafka.enabled=true

@Bean

@ConditionalOnProperty(name = "kafka.enabled", havingValue = "true")

public KafkaTemplate<String, String> kafkaTemplate() {

return new KafkaTemplate<>();

}

**4. Custom Health Indicators**

**What:** Expose app-specific health logic via /actuator/health.  
**Why:** Improves observability and monitoring.  
**Use Case:** Track status of third-party APIs.

@Component

public class MyHealthIndicator implements HealthIndicator {

public Health health() {

boolean reachable = checkApi();

return reachable ? Health.up().build() : Health.down().build();

}

}

**5. Kubernetes ConfigMap + Secret Integration**

**What:** Use externalized config via Spring Cloud Kubernetes.  
**Why:** Enables dynamic config changes, secure secrets.  
**Use Case:** Change logging level without restarting app.

apiVersion: v1

kind: ConfigMap

metadata:

name: my-app-config

data:

application.properties: |

logging.level.root=DEBUG

**6. Spring Cloud Config Server**

**What:** Externalize and centralize properties for all services.  
**Why:** Central control of secrets, tuning, feature flags.  
**Use Case:** Update DB URL for 10 microservices from one place.

spring:

cloud:

config:

uri: http://config-server

**7. Spring Boot Admin**

**What:** UI for managing/monitoring apps via actuator.  
**Why:** Central observability without custom dashboards.  
**Use Case:** View memory, endpoints, health of all microservices.

**8. Micrometer Metrics**

**What:** Vendor-neutral metrics API.  
**Why:** Unified monitoring to Prometheus, Datadog, etc.  
**Use Case:** Track latency of controllers, DB pool usage.

@Timed(value = "user.controller.latency", description = "Latency of user calls")

@GetMapping("/users")

public List<User> getUsers() {...}

**9. Custom Actuator Endpoints**

**What:** Build endpoints for internal ops.  
**Why:** Improve observability without exposing to public.  
**Use Case:** Trigger cache refresh or reload config.

@Component

@Endpoint(id = "cache")

public class CacheEndpoint {

@ReadOperation

public String clear() {

cache.clear();

return "Cleared";

}

}

**10. Graceful Shutdown**

**What:** Let app finish requests before stopping.  
**Why:** Prevent request loss during deployment.  
**Use Case:** Deploy with zero downtime.

server.shutdown=graceful

spring.lifecycle.timeout-per-shutdown-phase=30s

**11. Spring Retry**

**What:** Retry failed operations.  
**Why:** Improves resilience to transient faults.  
**Use Case:** Retry HTTP calls to flaky APIs.

@Retryable(maxAttempts = 3, value = HttpServerErrorException.class)

public String callApi() {...}

**12. Rate Limiting**

**What:** Control API usage frequency.  
**Why:** Prevent abuse and maintain service quality.  
**Use Case:** Throttle login attempts.

Bucket bucket = Bucket4j.builder().addLimit(limit).build();

**13. Circuit Breaker with Resilience4j**

**What:** Fail fast when a service is down.  
**Why:** Avoid cascading failures.  
**Use Case:** Stop calling DB if it’s down.

@CircuitBreaker(name = "db", fallbackMethod = "fallback")

public String fetchFromDb() {...}

**14. Caching**

**What:** Avoid recomputing expensive operations.  
**Why:** Improve performance, reduce DB load.  
**Use Case:** Cache search results.

@Cacheable("users")

public List<User> getUsers() {...}

**15. Async Tasks**

**What:** Run tasks in background threads.  
**Why:** Reduce latency for users.  
**Use Case:** Send email after user registers.

@Async

public void sendWelcomeEmail(User user) {...}

**16. Scheduled Jobs**

**What:** Run periodic tasks.  
**Why:** Automate reporting, cleanup.  
**Use Case:** Nightly DB backup.

@Scheduled(cron = "0 0 2 \* \* \*")

public void backup() {...}

**17. Spring Events**

**What:** Decouple components using events.  
**Why:** Better modularity and testability.  
**Use Case:** Send email when user is created.

eventPublisher.publishEvent(new UserCreatedEvent(user));

**18. Custom Error Handling**

**What:** Global control over exception responses.  
**Why:** Consistent client error messages.  
**Use Case:** Return JSON with error codes.

@RestControllerAdvice

public class ErrorHandler {

@ExceptionHandler(NotFoundException.class)

public ResponseEntity<?> handle(NotFoundException e) {...}

}

**19. Validation**

**What:** Ensure input data correctness.  
**Why:** Prevent bugs, reduce errors.  
**Use Case:** Validate email, password policy.

public class User {

@Email

private String email;

}

**20. Profiles**

**What:** Load env-specific beans/configs.  
**Why:** Separate dev, test, prod setups.  
**Use Case:** Use in-memory DB for tests.

@Profile("test")

@Bean

public DataSource testDb() {...}

**21. Spring Security with JWT**

**What:** Secure REST APIs with tokens.  
**Why:** Stateless, scalable security.  
**Use Case:** Secure mobile/backend APIs.

@Bean

public SecurityFilterChain filter(HttpSecurity http) {...}

**22. Integration Testing**

**What:** Test full flow including DB/services.  
**Why:** Catch real-world issues early.  
**Use Case:** Test user registration end-to-end.

@SpringBootTest

public class UserControllerIT {...}

**23. Embedded DB**

**What:** Run DB in memory.  
**Why:** Simplify testing, no need for setup.  
**Use Case:** Unit/integration tests.

spring.datasource.url=jdbc:h2:mem:testdb

**24. GraalVM Native Compilation**

**What:** Compile to native binary.  
**Why:** Reduce memory and startup time.  
**Use Case:** Deploy in serverless or edge.

mvn -Pnative native:compile

**25. Modular Monolith Architecture**

**What:** Organize code in feature modules.  
**Why:** Maintainable without microservice overhead.  
**Use Case:** Large app not ready for microservices.

module user-management

module inventory-management

Let me know if you'd like:

* A GitHub repo with full code for all examples
* A PDF export of this document
* A quiz or mock interview based on these topics