

# Spring Boot & Microservices Cheat Sheet

Complete reference for Spring Boot annotations, configurations, microservices patterns, Docker deployment, and best practices. Perfect for Java developers and architects.

🔗 Spring Boot 3.x • Java 17+ • Spring Cloud 2023+

🔧 Spring Boot

🔗 Microservices

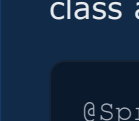
📄 Data & JPA

🔒 Security

🧪 Testing

🐳 Docker & K8s

Search annotations, configurations, patterns... (Press '/' to focus)



## Spring Boot Core

Annotations, Auto-configuration, Properties

### Core Annotations

@SpringBootApplication - Main application class annotation **Essential**

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

📄 Copy

@RestController - Combines @Controller and @ResponseBody

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

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return userService.findById(id);
    }
}
```

📄 Copy

### Configuration

application.properties - Key configurations

```
# Server Configuration
server.port=8080
server.servlet.context-path=/api

# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=secret
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=com.mysql.cj.jdbc.Driver

# Logging
logging.level.org.springframework=INFO
logging.level.com.example=DEBUG
```

📄 Copy

## Microservices

Spring Cloud, Service Discovery, API Gateway

### Spring Cloud

Service Registration (Eureka Client)

```
// pom.xml dependency
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

// Application class
@SpringBootApplication
@EnableEurekaClient
public class ProductServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }
}

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

📄 Copy

### Feign Client

Declarative REST Client

```
// Enable Feign Clients
@SpringBootApplication
@EnableFeignClients
public class OrderServiceApplication {
    // ...
}

// Feign Client Interface
@FeignClient(name = "product-service")
public interface ProductServiceClient {

    @GetMapping("/api/products/{id}")
    Product getProduct(@PathVariable("id") Long id);

    @PostMapping("/api/products")
    Product createProduct(@RequestBody Product product);
}

// Using the client
@Service
public class OrderService {

    @Autowired
    private ProductServiceClient productClient;

    public Order createOrder(OrderRequest request) {
        Product product = productClient.getProduct(request.getId());
        // Process order
    }
}
```

📄 Copy

## Data & JPA

Spring Data JPA, Repositories, Transactions

### Entity & Repository

JPA Entity with Relationships

```
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String name;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List orders = new ArrayList<>();

    @CreationTimestamp
    private LocalDateTime createdAt;

    // Getters and setters
}

// Repository with custom queries
@Repository
public interface UserRepository extends JpaRepository {

    Optional findByEmail(String email);

    @Query("SELECT u FROM User u WHERE u.name LIKE %:name%")
    List findByNameContaining(@Param("name") String name);

    List findByCreatedAtAfter(LocalDateTime date);
}
```

📄 Copy

### Transactions

Transactional Service Method

```
@Service
@Transactional
public class OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private PaymentService paymentService;

    @Transactional(propagation = Propagation.REQUIRED)
    public Order createOrder(OrderRequest request) {
        Order order = new Order();
        // Set order details

        order = orderRepository.save(order);

        // Process payment
        paymentService.processPayment(order);

        return order;
    }

    @Transactional(readOnly = true)
    public Order getOrder(Long id) {
        return orderRepository.findById(id)
            .orElseThrow(() -> new OrderNotFoundException(id));
    }
}
```

📄 Copy



## Spring Security

JWT, OAuth2, Role-based Security

### JWT Configuration

JWT Security Configuration **Spring Security 6.x**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf ->
                .disable()
            )
            .sessionManagement(session ->
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .authorizeHttpRequests(auth ->
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .requestMatchers("/api/user/**").hasAnyRole("ADMIN")
                .anyRequest().authenticated()
            )
            .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    @Bean
    public JwtAuthenticationFilter jwtAuthenticationFilter() {
        return new JwtAuthenticationFilter();
    }

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

📄 Copy

### JWT Utility

JWT Token Utility Class

```
@Component
public class JwtTokenProvider {

    @Value("${app.jwt.secret}")
    private String jwtSecret;

    @Value("${app.jwt.expiration}")
    private long jwtExpiration;

    public String generateToken(UserDetails userDetails) {
        Map claims = new HashMap<>();
        claims.put("roles", userDetails.getAuthorities().stream()
            .map(GrantedAuthority::getAuthority)
            .collect(Collectors.toList()));

        return Jwts.builder()
            .setClaims(claims)
            .setSubject(userDetails.getUsername())
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + jwtExpiration))
            .signWith(SignatureAlgorithm.HS512, jwtSecret)
            .compact();
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(jwtSecret).parseClaims(token);
        } catch (Exception e) {
            return false;
        }
    }
}
```

📄 Copy

## Testing

Unit, Integration, Mocking, Testcontainers

### Unit Tests

Service Unit Test with Mockito

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

    @Mock
    private UserRepository userRepository;

    @Mock
    private PasswordEncoder passwordEncoder;

    @InjectMocks
    private UserService userService;

    @Test
    void createUser_ValidInput_ReturnsUser() {
        // Arrange
        UserRequest request = new UserRequest("john@email.com", "John", "password123");
        User savedUser = new User(1L, "john@email.com", "John");

        when(passwordEncoder.encode("password123"))
            .thenReturn("encodedPassword");

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

        // Act
        User result = userService.createUser(request);

        // Assert
        assertNotNull(result);
        assertEquals("John@email.com", result.getEmail());
        assertEquals(userRepository, times(1), save(any(User.class)));
    }

    @Test
    void getUser_NonExistingId_ThrowsException() {
        // Arrange
        Long userId = 999L;

        when(userRepository.findById(userId))
            .thenReturn(Optional.empty());

        // Act & Assert
        assertThrows(UserNotFoundException.class, () ->
            userService.getUser(userId));
    }
}
```

📄 Copy

### Integration Tests

Controller Integration Test

```
@SpringBootTest
@AutoConfigureMockMvc
class UserControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    void getUser_ExistingId_ReturnsUser() throws Exception {
        // Arrange
        User user = new User(1L, "john@email.com", "John Doe");

        when(userService.getUser(1L)).thenReturn(user);

        // Act & Assert
        mockMvc.perform(get("/api/users/1"))
            .contentType(MediaType.APPLICATION_JSON)
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.id").value(1))
            .andExpect(jsonPath("$.email").value("john@email.com"))
            .andExpect(jsonPath("$.name").value("John Doe"));
    }

    @Test
    void createUser_ValidInput_ReturnsCreated() throws Exception {
        // Arrange
        UserRequest request = new UserRequest("test@email.com", "Test", "pass");
        UserResponse response = new UserResponse(1L, "test@email.com", "Test");

        when(userService.createUser(any(UserRequest)))
            .thenReturn(response);

        // Act & Assert
        mockMvc.perform(post("/api/users"))
            .contentType(MediaType.APPLICATION_JSON)
            .content(asJsonString(request))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.id").exists());
    }

    private static String asJsonString(final Object obj) {
        try {
            return new ObjectMapper().writeValueAsString(obj);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

📄 Copy

## Docker & Kubernetes

Containerization & Orchestration

### Docker Configuration

Dockerfile for Spring Boot

```
# Multi-stage build
# Stage 1: Build
FROM maven:3.8.4-openjdk-17-slim AS build
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline
COPY src ./src
RUN mvn clean package -DskipTests

# Stage 2: Runtime
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar

# Add non-root user
RUN addgroup --system spring && adduser --system --ingroup spring spring
USER spring:spring

# Health check
HEALTHCHECK --interval=30s --timeout=3s \
    CMD curl -f http://localhost:8080/actuator/health || exit 1

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]

# Build commands:
# docker build -t my-spring-app:latest .
# Run command:
# docker run -p 8080:8080 -e SPRING_PROFILES_ACTIVE=prod my-spring-app
```

📄 Copy

### Kubernetes Deployment

K8s Deployment & Service

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
  labels:
    app: user-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
        - name: user-service
          image: myregistry/user-service:1.0.0
          ports:
            - containerPort: 8080
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "prod"
            - name: DB_HOST
              valueFrom:
                configMapKeyRef:
                  name: app-config
                  key: database.host
          resources:
            requests:
              memory: "512Mi"
              cpu: "250m"
            limits:
              memory: "1Gi"
              cpu: "500m"
          livenessProbe:
            httpGet:
              path: /actuator/health/liveness
              port: 8080
            initialDelaySeconds: 60
            periodSeconds: 10
            readinessProbe:
            httpGet:
              path: /actuator/health/readiness
              port: 8080
            initialDelaySeconds: 30
            periodSeconds: 5
```

```
# service.yaml
apiVersion: v1
kind: Service
metadata:
  name: user-service
spec:
  selector:
    app: user-service
  ports:
    - port: 80
      targetPort: 8080
  type: ClusterIP
```

📄 Copy

## Microservices Architecture Pattern



This architecture shows a typical Spring Cloud microservices setup with API Gateway, Service Discovery, and separate databases for each service.

## Spring Boot Quick Reference



### Common Annotations

@SpringBootApplication	Main application annotation
@RestController	REST controller
@Service	Business logic layer
@Repository	Data access layer
@Autowired	Dependency injection
@Value	Inject property values
@Configuration	Configuration class
@Bean	Declare a Spring bean

### Spring Data JPA

findBy(Property)	Auto-generated query
@Query	Custom JPQL/Native query
@Modifying	Update/delete query
@Entity	JPA entity class
@Id	Primary key
@GeneratedValue	Auto-generate ID
@OneToMany	One-to-many relation
@ManyToOne	Many-to-one relation

### Spring Cloud Dependencies

spring-cloud-starter-netflix-eureka-client	Service Discovery
spring-cloud-starter-gateway	API Gateway
spring-cloud-starter-openfeign	Declarative REST Client
spring-cloud-starter-circuitbreaker-resilience4j	Circuit Breaker
spring-cloud-starter-config	Config Server Client
spring-cloud-starter-sleuth	Distributed Tracing
spring-cloud-starter-bus-amqp	Config Refresh (Bus)