Spring Boot Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Implement a custom annotation for method-level caching in Spring Boot with a specified TTL (Time To Live)

Solution:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CustomCache {
   String key() default "";
   long ttlSeconds() default 300;
}
```

Implementation example:

```
@Aspect
@Component
public class CustomCacheAspect {
   @Around("@annotation(customCache)")
   public Object cache(ProceedingJoinPoint joinPoint, CustomCache customCache) {
      // Cache implementation
   }
}
```

2. Create a custom Spring Boot actuator endpoint that monitors application memory usage

```
@Component
@Endpoint(id = "memory")
public class MemoryEndpoint {
   @ReadOperation
   public Map getMemoryStats() {
      Runtime runtime = Runtime.getRuntime();
      return Map.of("total", runtime.totalMemory(),
                  "free", runtime.freeMemory());
   }
}
```

3. Implement a custom RequestBodyAdvice to encrypt/decrypt request data automatically

```
@ControllerAdvice
public class EncryptionRequestBodyAdvice implements RequestBodyAdvice {
   @Override
   public Object afterBodyRead(Object body, HttpInputMessage input,
         MethodParameter param, Type targetType,
         Class> converterType) {
      return decryptBody(body);
   }
}
```

4. Create a custom validation annotation for checking if a String field contains valid JSON

```
@Documented
@Constraint(validatedBy = JsonValidator.class)
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidJson {
   String message() default "Invalid JSON string";
   Class[] groups() default {};
   Class[] payload() default {};
}
```

5. Implement a custom Spring Boot health indicator for external service dependency

```
@Component
public class ExternalServiceHealthIndicator extends AbstractHealthIndicator {
   @Override
   protected void doHealthCheck(Health.Builder builder) {
      try {
         // Check external service health
         builder.up().withDetail("service", "available");
      } catch (Exception e) {
         builder.down().withException(e);
      }
   }
}
```

6. Create a custom ResourceHandler to serve compressed static resources

```
@Configuration
public class CompressedResourceConfig implements WebMvcConfigurer {
   @Override
   public void addResourceHandlers(ResourceHandlerRegistry registry) {
      registry.addResourceHandler("/static/**")
            .addResourceLocations("classpath:/static/")
            .resourceChain(true)
            .addResolver(new GzipResourceResolver());
   }
}
```

7. Implement a custom ThreadPoolTaskExecutor with monitoring capabilities

```
@Configuration
public class MonitoredThreadPoolConfig {
```

```
    @Bean
    public ThreadPoolTaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor() {
            @Override
            protected void beforeExecute(Thread t, Runnable r) {
                // Add monitoring logic
            }
        };
        return executor;
    }
}
```

8. Create a custom error handler for async operations in Spring Boot

```
@Component
public class CustomAsyncExceptionHandler
        implements AsyncUncaughtExceptionHandler {
    @Override
    public void handleUncaughtException(Throwable ex, Method method,
            Object... params) {
        // Custom error handling logic
        log.error("Async error in {}: {}", method.getName(),
                ex.getMessage());
    }
}
```

9. Implement a custom PropertySource that loads configuration from a secure vault

```
@Component
public class VaultPropertySource extends PropertySource {
    public VaultPropertySource() {
        super("vault");
    }

    @Override
    public Object getProperty(String name) {
        return VaultClient.getValue(name);
    }
}
```

10. Create a custom ConversionService to handle complex object transformations

```
@Configuration
public class CustomConversionConfig {
    @Bean
    public ConversionService conversionService() {
        ConversionServiceFactoryBean factory =
            new ConversionServiceFactoryBean();
        factory.setConverters(getCustomConverters());
        factory.afterPropertiesSet();
        return factory.getObject();
    }
}
```

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU Cache using Spring Boot components

Solution:

Here's an efficient LRU Cache implementation using Spring components:

```
@Component
public class LRUCache extends LinkedHashMap {
   private final int capacity;
   public LRUCache(@Value("${cache.capacity:100}") int capacity) {
      super(capacity, 0.75f, true);
      this.capacity = capacity;
   }
   @Override
   protected boolean removeEldestEntry(Map.Entry eldest) {
      return size() > capacity;
   }}
```

Time Complexity: O(1) for both get and put operations

2. Design a Rate Limiter using Spring Boot

Solution:

Implement a sliding window rate limiter:

```
@Service
public class RateLimiter {
   private Map> requests = new ConcurrentHashMap<>();
   public boolean allowRequest(String clientId, int windowMs, int limit) {
      long now = System.currentTimeMillis();
      requests.putIfAbsent(clientId, new ConcurrentLinkedQueue<>());
      cleanOldRequests(clientId, now - windowMs);
      return requests.get(clientId).size() < limit;
   }}
```

3. Implement a Thread-Safe Circular Buffer using Spring Boot

Solution:

```
@Component
public class CircularBuffer {
   private final T[] buffer;
   private int writePos = 0, readPos = 0;
   private final ReentrantLock lock = new ReentrantLock();

   @SuppressWarnings("unchecked")
   public CircularBuffer(@Value("${buffer.size:16}") int size) {
```

```
        buffer = (T[]) new Object[size];
    }
```

Key Features:

- Thread-safe implementation
- Configurable buffer size
- O(1) read/write operations

4. Create a Custom Collection that implements Iterable for Spring Boot pagination

Solution:

```
@Component
public class PaginatedCollection implements Iterable {
    private final List items;
    private final int pageSize;
    public Iterator iterator() {
        return new PaginatedIterator<>(items, pageSize);
    }
```

Benefits:

- Memory efficient pagination
- Lazy loading support
- Compatible with Spring Data

5. Implement a Trie data structure for auto-completion in Spring Boot

Solution:

```
@Component
public class Trie {
    private TrieNode root = new TrieNode();
    public void insert(String word) {
        TrieNode current = root;
        for(char ch : word.toCharArray()) {
            current.children.putIfAbsent(ch, new TrieNode());
            current = current.children.get(ch);
        }
    }
```

Time Complexity:

- Insert: O(m) where m is word length
- Search: O(m) where m is word length

6. Design a Thread-Safe Object Pool in Spring Boot

Solution:

```
@Component
public class ObjectPool {
    private final BlockingQueue pool;
    private final Supplier factory;
    public ObjectPool(Supplier factory, int size) {
        this.factory = factory;
        this.pool = new ArrayBlockingQueue<>(size);
        IntStream.range(0, size).forEach(i -> pool.offer(factory.get()));
    }
```

Features:

- Thread-safe operations
- Configurable pool size
- Automatic resource management

7. Implement a Custom Cache Eviction Strategy in Spring Boot

Solution:

```
@Component
public class TimeBasedCache {
    private final Map> cache = new ConcurrentHashMap<>();
    @Scheduled(fixedRate = 1000)
    public void evictExpiredEntries() {
        long now = System.currentTimeMillis();
        cache.entrySet().removeIf(e -> e.getValue().isExpired(now));
    }
```

Advantages:

- Time-based eviction
- Thread-safe operations
- Customizable expiry policy

8. Create a Custom Priority Queue with Spring Boot

Solution:

```
@Component
public class CustomPriorityQueue {
    private final PriorityQueue queue;
    @Autowired
    public CustomPriorityQueue(Comparator comparator) {
        this.queue = new PriorityQueue<>(comparator);
    }
```

Key Features:

- Configurable comparison logic
- O(log n) insertion
- O(log n) removal
- Spring-managed lifecycle

9. Implement a Sliding Window Rate Limiter with Redis in Spring Boot

Solution:

```
@Service
public class RedisRateLimiter {
    @Autowired
    private RedisTemplate redisTemplate;
    public boolean tryAcquire(String key, int windowSeconds, int limit) {
        long now = Instant.now().getEpochSecond();
        return redisTemplate.execute(new SessionCallback() {
            // Implementation details
        });
    }
```

Benefits:

- Distributed rate limiting
- Sliding window accuracy
- Redis persistence

10. Design a Custom Bloom Filter Implementation in Spring Boot

Solution:

```
@Component
public class BloomFilter {
    private final BitSet bitSet;
    private final int size;
    private final List hashFunctions;
    public boolean mightContain(T item) {
        return hashFunctions.stream()
            .allMatch(h -> bitSet.get(h.hash(item) % size));
    }
```

Characteristics:

- Space-efficient
- Probabilistic data structure
- No false negatives
- Configurable false positive rate

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly using Spring Boot. What would be the key components and considerations?

Key Components:

- **API Layer**: REST endpoints for URL shortening and redirection
- **Encoding Service**: Convert long URLs to short codes
- **Storage Layer**: Distributed database for URL mappings
- **Cache Layer**: Redis for frequently accessed URLs

Implementation Highlights:

```
@Service
public class UrlShortenerService {
    private static final String BASE62 = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    public String shorten(String longUrl) {
        String hash = generateHash(longUrl);
        return encode(hash);
    }
}
```

Scale Considerations:

- Use consistent hashing for database sharding
- Implement rate limiting
- Add CDN for global access

2. How would you design a real-time chat system using Spring WebSocket? Discuss the architecture and scalability aspects.

Architecture Components:

- **WebSocket Server**: Handle bi-directional communication
- **Message Broker**: RabbitMQ/Kafka for message distribution
- **Session Store**: Redis for user sessions

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/app");
    }
}
```

Scalability Features:

- Cluster WebSocket servers with sticky sessions
- Implement message persistence
- Handle reconnection scenarios

3. Design a social media feed service with Spring Boot. How would you handle high read/write throughput?

Core Components:

- **Feed Service**: Generate personalized feeds
- **Post Service**: Handle post creation/updates
- **Fan-out Service**: Distribute posts to followers

```
@Service
public class FeedService {
    @Cacheable(value = "userFeed", key = "#userId")
    public List getUserFeed(String userId, int page) {
        return feedRepository.getFeedItems(userId, PageRequest.of(page, 20));
    }
}
```

Performance Optimizations:

- Implement feed pagination
- Use materialized views for popular users
- Cache hot posts in Redis
- Implement read replicas for scaling

4. Design a distributed rate limiter for a Spring Boot API gateway. How would you implement it?

Implementation Approach:

- **Token Bucket Algorithm**: Control request rate
- **Redis**: Distributed rate counting
- **Sliding Window**: Time-based limiting

```
@Service
public class RateLimiter {
    public boolean allowRequest(String key, int limit) {
        String windowKey = key + ":" + System.currentTimeMillis() / 1000;
        return redisTemplate.opsForValue().increment(windowKey) <= limit;
    }
}
```

Features:

- Configurable rate limits per user/IP
- Circuit breaker integration
- Custom error responses

5. How would you design a distributed caching system with Spring Boot? Discuss invalidation strategies.

Cache Architecture:

- **Multi-level Caching**: Local + Distributed
- **Cache-aside Pattern**: Lazy loading
- **Write-through Cache**: Immediate updates

```
@Cacheable(value = "userCache", key = "#userId", unless = "#result == null")
public User getUser(String userId) {
    return userRepository.findById(userId)
        .orElseThrow(() -> new UserNotFoundException());
}
```

Invalidation Strategies:

- Time-based expiration (TTL)
- Event-driven invalidation
- Version-based invalidation
- Pub/Sub for cross-instance coordination

6. Design a job scheduling system using Spring Boot. How would you ensure reliability and scalability?

System Components:

- **Job Queue**: Distributed message queue
- **Worker Pool**: Scalable execution nodes
- **State Store**: Job status tracking

```
@Scheduled(fixedRate = 60000)
public void processJobs() {
    List jobs = jobQueue.poll(10);
    CompletableFuture.allOf(jobs.stream()
        .map(this::processJobAsync)
        .toArray(CompletableFuture[]::new));
}
```

Reliability Features:

- Dead letter queues
- Job retry mechanisms
- Distributed locking
- Health monitoring

7. Design a notification service that can handle multiple channels (email, SMS, push) using Spring Boot. How would you make it extensible?

Architecture:

- **Strategy Pattern**: Channel-specific implementations
- **Queue System**: Async processing
- **Template Engine**: Message formatting

```
public interface NotificationChannel {
    void send(Notification notification);
}

@Service
class EmailChannel implements NotificationChannel {
    public void send(Notification notification) { ... }
}
```

Features:

- Channel fallback mechanism
- Retry policies
- Template management
- Delivery tracking

8. Design a distributed configuration management system using Spring Boot. How would you handle updates and versioning?

Core Components:

- **Config Store**: Version-controlled repository
- **Change Notification**: Event broadcasting
- **Client Cache**: Local config storage

```
@RefreshScope
@Configuration
public class DynamicConfig {
    @Value("${app.feature.enabled}")
    private boolean featureEnabled;

    @EventListener(RefreshScopeRefreshedEvent.class)
    public void onRefresh() { ... }
}
```

Features:

- Configuration versioning
- Rollback capability
- Environment segregation
- Audit logging

9. Design a distributed session management system for a Spring Boot application. How would you handle session replication?

Architecture Components:

- **Session Store**: Redis/Hazelcast
- **Load Balancer**: Sticky sessions
- **Security Layer**: Session validation

```
@Configuration
public class SessionConfig extends AbstractHttpSessionApplicationInitializer {
    @Bean
    public RedisTemplate redisTemplate() {
        return new RedisTemplate<>();
    }
}
```

Features:

- Session timeout handling
- Cross-DC replication
- Session invalidation
- Failover handling

10. Design a fault-tolerant payment processing system using Spring Boot. How would you ensure transaction consistency?

System Components:

- **Payment Gateway**: External service integration
- **Transaction Manager**: ACID compliance
- **State Machine**: Payment lifecycle

```
@Transactional
public PaymentResult processPayment(Payment payment) {
    try {
        return paymentGateway.process(payment);
    } catch (Exception e) {
        compensateTransaction(payment);
    }
}
```

Reliability Features:

- Idempotency handling
- Distributed transactions
- Circuit breakers
- Transaction logs

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a Spring Boot service method to flatten a nested list of integers. How would you test it?

Solution:

```
@Service
public List flattenList(List nestedList) {
    List flat = new ArrayList<>();
    nestedList.forEach(item -> {
        if (item instanceof List) {
            flat.addAll(flattenList((List) item));
        } else if (item instanceof Integer) {
            flat.add((Integer) item);
        }
    });
    return flat;
}
```

Testing:

```
@Test
void testFlattenList() {
    List nested = Arrays.asList(1, Arrays.asList(2, 3), 4);
    List expected = Arrays.asList(1, 2, 3, 4);
    assertEquals(expected, service.flattenList(nested));
}
```

2. Create a custom annotation to measure method execution time in Spring Boot. How would you implement it?

Implementation:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ExecutionTime {}

@Aspect
@Component
public class ExecutionTimeAspect {
    @Around("@annotation(ExecutionTime)")
    public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.currentTimeMillis();
        Object result = joinPoint.proceed();
        long end = System.currentTimeMillis();
        log.info("{} executed in {}ms", joinPoint.getSignature(), (end - start));
        return result;
    }
}
```

3. Debug a Spring Boot application that's experiencing memory leaks. What tools and approaches would you use?

Memory Leak Debugging Approach:

- **Use JVM Arguments:** -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path/to/dump
- **Tools:**
  - Java Flight Recorder (JFR)
  - JConsole or VisualVM for real-time monitoring
  - Eclipse Memory Analyzer (MAT) for heap dumps

Example Analysis Code:

```
@Component
public class MemoryLeakDetector {
    @Scheduled(fixedRate = 300000)
    public void checkMemoryUsage() {
        Runtime runtime = Runtime.getRuntime();
        long usedMemory = runtime.totalMemory() - runtime.freeMemory();
        log.warn("Memory used: {} MB", usedMemory / (1024 * 1024));
    }
}
```

4. Implement a rate limiter using Spring Boot AOP. How would you handle concurrent requests?

Implementation:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface RateLimit {
    int requests() default 100;
    int timeWindow() default 60;
}

@Aspect
@Component
public class RateLimitAspect {
    private Map buckets = new ConcurrentHashMap<>();

    @Around("@annotation(rateLimit)")
    public Object limitRate(ProceedingJoinPoint joinPoint, RateLimit rateLimit) {
        String key = joinPoint.getSignature().toString();
        Bucket bucket = buckets.computeIfAbsent(key,
            k -> Bucket4j.builder().addLimit(Bandwidth.simple(rateLimit.requests(), Duration.ofSeconds(rateLimit.timeWindow()))).build());

        if (bucket.tryConsume(1)) {
            return joinPoint.proceed();
        }
        throw new RateLimitExceededException();
    }
}
```

5. Create a custom Spring Boot actuator endpoint to monitor application-specific metrics. How would you secure it?

Implementation:

```
@Component
@Endpoint(id = "customMetrics")
public class CustomMetricsEndpoint {
    @ReadOperation
    public Map getMetrics() {
        Map metrics = new HashMap<>();
        metrics.put("activeUsers", getUserCount());
        metrics.put("processingQueue", getQueueSize());
        return metrics;
    }
}
```

Security Configuration:

```
@Configuration
public class ActuatorSecurity {
    @Bean
    public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
        return http.authorizeExchange()
            .pathMatchers("/actuator/customMetrics").hasRole("ADMIN")
            .anyExchange().authenticated()
            .and().build();
    }
}
```

6. Implement a circuit breaker pattern using Spring Cloud Circuit Breaker. How would you handle fallback methods?

Implementation:

```
@Service
public class ResilientService {
```

```
    @CircuitBreaker(name = "externalService", fallbackMethod = "fallbackMethod")
    public String callExternalService() {
        return restTemplate.getForObject("http://external-service/api", String.class);
    }

    public String fallbackMethod(Exception ex) {
        return "Fallback response due to: " + ex.getMessage();
    }
}
```

Configuration:

```
@Configuration
public class CircuitBreakerConfig {
    @Bean
    public Customizer globalCustomConfiguration() {
        return factory -> factory.configureDefault(id -> new Resilience4JConfigBuilder(id)
            .slidingWindowSize(10)
            .failureRateThreshold(50)
            .waitDurationInOpenState(Duration.ofSeconds(10))
            .build());
    }
}
```

7. Write a Spring Boot service that implements the producer-consumer pattern using Spring Integration. How would you handle backpressure?

Implementation:

```
@Configuration
public class IntegrationConfig {
    @Bean
    public MessageChannel inputChannel() {
        return MessageChannels.queue(100).get();
    }

    @Bean
    public IntegrationFlow processFlow() {
        return IntegrationFlows.from(inputChannel())
            .filter(Message.class, m -> m.getPayload() != null)
            .handle(this::processMessage)
            .channel("outputChannel")
            .get();
    }
}
```

8. Implement a custom validation annotation in Spring Boot for checking if a date is in the future. How would you test it?

Implementation:

```
@Documented
@Constraint(validatedBy = FutureDateValidator.class)
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface FutureDate {
    String message() default "Date must be in the future";
    Class[] groups() default {};
    Class[] payload() default {};
}
```

Validator:

```
public class FutureDateValidator implements ConstraintValidator {
    @Override
    public boolean isValid(LocalDate date, ConstraintValidatorContext context) {
        return date != null && date.isAfter(LocalDate.now());
    }
}
```

9. Create a custom Spring Boot starter for handling application events. How would you make it auto-configurable?

Implementation:

```
@Configuration
@ConditionalOnClass(EventHandler.class)
@EnableConfigurationProperties(EventProperties.class)
public class EventAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public EventHandler eventHandler(EventProperties properties) {
        return new EventHandler(properties.getQueueSize());
    }
}
```

Properties:

```
@ConfigurationProperties(prefix = "custom.event")
public class EventProperties {
    private int queueSize = 100;
    // getters and setters
}
```

10. Implement a caching strategy for a Spring Boot REST API using Spring Cache. How would you handle cache invalidation?

Implementation:

```
@Service
public class CacheableService {
    @Cacheable(value = "items", key = "#id", unless = "#result == null")
    public Item getItem(Long id) {
        return repository.findById(id).orElse(null);
    }

    @CacheEvict(value = "items", allEntries = true)
    @Scheduled(fixedRate = 3600000)
    public void clearCache() {
        log.info("Cache cleared");
    }
}
```

Configuration:

```
@EnableCaching
@Configuration
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        return new ConcurrentMapCacheManager("items");
    }
}
```