Java and Spring Boot Interview Comparison Questions

Spring Framework Core

Comparison Question	Key Differen
@Component vs @Bean	@Component: class-level annotation for auto-detection via classpath scanning; @Bean: met
@Controller vs @RestController	@RestController = @Controller + @ResponseBody; @Controller returns views, @RestControlle
@RequestParam vs @PathVariable	@RequestParam: extracts query parameters (?key=value); @PathVariable: extracts values fr
@RequestParam vs @RequestBody	@RequestParam: for form data or query params; @RequestBody: for entire HTTP request body
@Autowired vs @Resource vs @Inject	@Autowired: Spring-specific, by-type injection; @Resource: Java standard, by-name then b
ApplicationContext vs BeanFactory	ApplicationContext: eager loading, more features (events, i18n); BeanFactory: lazy loadi
@Configuration vs @Component	@Configuration: defines beans explicitly, supports @Bean methods; @Component: marks clas
Constructor Injection vs Setter Injection vs Field Injection	Constructor: immutable, testable, recommended; Setter: optional dependencies; Field: sim
@Service vs @Repository vs @Component	Functionally identical, differ semantically: @Service for business logic, @Repository fo
@RequestMapping vs @GetMapping/@PostMapping	@RequestMapping: generic, supports all HTTP methods; @GetMapping/@PostMapping: specializ
@Valid vs @Validated	@Valid: standard JSR-303, triggers validation; @Validated: Spring's extension, supports
@Qualifier vs @Primary	@Qualifier: explicitly names which bean to inject; @Primary: marks default bean when mul
@Scope("prototype") vs @Scope("singleton")	Singleton: one instance per container (default); Prototype: new instance for each reques
@EnableAsync vs @Async	@EnableAsync: enables async processing at application level; @Async: marks specific meth
@EventListener vs @TransactionalEventListener	@EventListener: handles events immediately; @TransactionalEventListener: handles events
@Profile vs @Conditional	@Profile: activates beans for specific profiles; @Conditional: more flexible, custom con
@DependsOn vs @Order	@DependsOn: controls bean initialization order; @Order: controls execution order of comp
@Import vs @ComponentScan	@Import: explicitly imports specific configurations; @ComponentScan: automatically scans
XML Configuration vs Annotation Configuration vs Java Configuration	XML: external, verbose, no compilation needed; Annotations: on classes, concise; Java Co
FactoryBean vs @Bean	FactoryBean: interface for complex bean creation logic; @Bean: annotation for simple met
@PostConstruct vs @PreDestroy vs InitializingBean vs DisposableBean	Annotations (@PostConstruct/@PreDestroy): standard, decoupled; Interfaces: Spring-specif
BeanPostProcessor vs BeanFactoryPostProcessor	BeanPostProcessor: modifies bean instances; BeanFactoryPostProcessor: modifies bean defi

Spring Boot

Comparison Question	Key Differ
Spring vs Spring Boot	Spring: comprehensive framework requiring manual configuration; Spring
@SpringBootApplication vs @EnableAutoConfiguration + @ComponentScan + @Configuration	@SpringBootApplication: convenient combination of three annotations; In
application.properties vs application.yml	Properties: flat key-value pairs; YAML: hierarchical, supports lists, n
@ConfigurationProperties vs @Value	@ConfigurationProperties: type-safe, validates groups of properties; @V
Embedded Server vs External Server	Embedded: packaged in JAR, easier deployment; External: traditional WAR
spring-boot-starter vs Traditional Dependencies	Starters: curated dependency sets with compatible versions; Traditional
@SpringBootTest vs @WebMvcTest vs @DataJpaTest	@SpringBootTest: full context; @WebMvcTest: only web layer; @DataJpaTes
CommandLineRunner vs ApplicationRunner	CommandLineRunner: receives String[] args; ApplicationRunner: receives
@EnableAutoConfiguration vs @Conditional	@EnableAutoConfiguration: triggers Spring Boot's auto-config; @Conditio
Spring Boot Actuator vs Custom Monitoring	Actuator: production-ready endpoints out-of-box; Custom: full control b
DevTools vs Manual Restart	DevTools: automatic restart on file changes; Manual: full control over
Fat JAR vs WAR Deployment	Fat JAR: self-contained with embedded server; WAR: requires external ap
application.properties vs bootstrap.properties	application: standard app config; bootstrap: earlier loading, for cloud
@RefreshScope vs Static Configuration	@RefreshScope: allows runtime config updates; Static: requires restart

Java Core - Language Features

Comparison Question	Key Difference
Abstract Class vs Interface	Abstract class: single inheritance, can have state and constructors; Interface: multipl
Interface (Java 7) vs Interface (Java 8+)	Java 7: only abstract methods; Java 8+: can have default and static methods
Checked Exception vs Unchecked Exception	Checked: must be caught or declared (compile-time); Unchecked: RuntimeException subclas
throw vs throws	throw: actually throws an exception instance; throws: declares method might throw excep
final vs finally vs finalize	final: immutability keyword; finally: always-execute block; finalize: object cleanup be
== vs .equals()	==: reference comparison for objects, value for primitives; .equals(): content comparis
String vs StringBuilder vs StringBuffer	String: immutable; StringBuilder: mutable, not thread-safe, faster; StringBuffer: mutab
pass-by-value vs pass-by-reference in Java	Java is always pass-by-value (for objects, the reference value is passed)
static vs instance methods	Static: belongs to class, no 'this'; Instance: belongs to object, has 'this' reference
static vs non-static nested classes	Static nested: no reference to outer instance; Inner class: implicit reference to outer
Inner Class vs Static Nested Class vs Local Class vs Anonymous Class	Inner: member of outer class; Static nested: no outer reference; Local: defined in meth
Overloading vs Overriding	Overloading: same name, different parameters (compile-time); Overriding: same signature
Compile-time Polymorphism vs Runtime Polymorphism	Compile-time: method overloading; Runtime: method overriding via inheritance
this vs super	this: current object reference; super: parent class reference
package-private vs protected vs public vs private	private: class only; package-private: package only; protected: package + subclasses; pu
transient vs volatile	transient: skip serialization; volatile: thread visibility guarantee
Serializable vs Externalizable	Serializable: automatic serialization; Externalizable: manual control over serializatio
shallow copy vs deep copy	Shallow: copies references; Deep: copies actual objects recursively
Heap vs Stack Memory	Heap: objects storage, shared across threads; Stack: method calls and local variables,
PermGen vs Metaspace	PermGen: fixed size, stores class metadata (Java 7); Metaspace: dynamic size, native me
Young Generation vs Old Generation	Young: new objects, frequent GC; Old: long-lived objects, less frequent GC

Java Collections

Comparison Question	Key Difference
ArrayList vs LinkedList	ArrayList: index-based access O(1), insertion/deletion O(n); LinkedList: sequential access O(n), inse
ArrayList vs Vector	ArrayList: not synchronized, faster; Vector: synchronized, thread-safe but slower
HashSet vs LinkedHashSet vs TreeSet	HashSet: no order; LinkedHashSet: insertion order; TreeSet: sorted order
HashMap vs HashTable vs ConcurrentHashMap	HashMap: not synchronized, allows null; HashTable: synchronized, no nulls; ConcurrentHashMap: segment
HashMap vs TreeMap vs LinkedHashMap	HashMap: no order, O(1) ops; TreeMap: sorted by keys, O(log n); LinkedHashMap: maintains insertion or
HashSet vs HashMap	HashSet: stores values only; HashMap: stores key-value pairs (HashSet internally uses HashMap)
Comparable vs Comparator	Comparable: natural ordering in class (compareTo); Comparator: external custom ordering (compare)
Iterator vs ListIterator vs Spliterator	Iterator: forward only; ListIterator: bidirectional, list-specific; Spliterator: parallel iteration s
Enumeration vs Iterator	Enumeration: legacy, no remove(); Iterator: modern, has remove(), fail-fast
Iterator vs forEach	Iterator: explicit control, can remove; forEach: cleaner syntax, no removal during iteration
Collection vs Collections	Collection: interface for data structures; Collections: utility class with static methods
List vs Set vs Queue	List: ordered, duplicates allowed; Set: no duplicates; Queue: FIFO/priority-based processing
Queue vs Deque	Queue: single-ended (FIFO); Deque: double-ended queue (both ends operations)
PriorityQueue vs TreeSet	PriorityQueue: heap-based, allows duplicates; TreeSet: tree-based, no duplicates, all elements access
WeakHashMap vs HashMap	WeakHashMap: keys can be garbage collected; HashMap: strong references to keys
IdentityHashMap vs HashMap	IdentityHashMap: uses == for key comparison; HashMap: uses .equals() and .hashCode()
Fail-Fast vs Fail-Safe Iterators	Fail-fast: throws ConcurrentModificationException; Fail-safe: works on copy, no exception
Arrays.asList() vs new ArrayList()	Arrays.asList(): fixed-size backed by array; new ArrayList(): fully mutable list
Collections.synchronizedList() vs CopyOnWriteArrayList	synchronizedList: lock on every operation; CopyOnWriteArrayList: copy on write, better for more reads
Collections.unmodifiableList() vs ImmutableList	unmodifiableList: wrapper, original can change; ImmutableList: truly immutable copy

Java Concurrency

Comparison Question	Key Difference
Process vs Thread	Process: independent memory space, heavyweight; Thread: shared memory space, lightweight
User Thread vs Daemon Thread	User: JVM waits for completion; Daemon: JVM can exit while running (e.g., GC thread)
wait() vs sleep()	wait(): releases lock, needs notify(); sleep(): keeps lock, wakes after timeout
wait() vs park()	wait(): Object method, requires monitor; park(): LockSupport method, doesn't need monitor
notify() vs notifyAll()	notify(): wakes one waiting thread; notifyAll(): wakes all waiting threads
Runnable vs Callable	Runnable: no return value, can't throw checked exceptions; Callable: returns value, can throw exc
Future vs CompletableFuture	Future: blocking get(); CompletableFuture: async programming with chaining and callbacks
execute() vs submit() in ExecutorService	execute(): no return, for Runnable; submit(): returns Future, for Runnable or Callable
synchronized method vs synchronized block	Method: locks entire method; Block: locks specific code section, more granular control
volatile vs synchronized	volatile: visibility guarantee only; synchronized: visibility + atomicity (mutual exclusion)
volatile vs Atomic classes	volatile: only memory visibility; Atomic: visibility + atomic operations (CAS-based)
CountDownLatch vs CyclicBarrier	CountDownLatch: one-time use, counts down; CyclicBarrier: reusable, waits for all parties
Semaphore vs Mutex	Semaphore: n permits, multiple thread access; Mutex: binary semaphore, mutual exclusion
ReentrantLock vs synchronized	ReentrantLock: explicit lock, more features (try-lock, interruptible); synchronized: implicit, si
ReadWriteLock vs ReentrantLock	ReadWriteLock: separate read/write locks, concurrent reads; ReentrantLock: exclusive access only
ThreadLocal vs Instance Variables	ThreadLocal: thread-specific copy; Instance: shared across threads
Fork/Join vs ExecutorService	Fork/Join: work-stealing, recursive tasks; ExecutorService: general thread pool
parallelStream() vs stream()	parallelStream(): multi-threaded processing; stream(): single-threaded processing
Thread.start() vs Thread.run()	start(): creates new thread; run(): executes in current thread
interrupted() vs isInterrupted()	interrupted(): checks and clears flag; isInterrupted(): only checks flag
ThreadPoolExecutor vs ForkJoinPool	ThreadPoolExecutor: traditional thread pool; ForkJoinPool: work-stealing algorithm
FixedThreadPool vs CachedThreadPool vs ScheduledThreadPool	Fixed: fixed number of threads; Cached: creates as needed; Scheduled: supports delays/periodic ex

Java 8+ Features

Comparison Question	Key Difference
Stream vs Collection	Stream: lazy evaluation, functional, one-time use; Collection: eager, stores data, reusable
map() vs flatMap()	map(): one-to-one transformation; flatMap(): one-to-many, flattens nested structures
Stream vs Parallel Stream	Stream: sequential processing; Parallel Stream: uses ForkJoinPool for concurrent processing
Optional.of() vs Optional.ofNullable()	of(): throws NPE if null; ofNullable(): returns empty Optional for null
findFirst() vs findAny()	findFirst(): deterministic, first element; findAny(): non-deterministic, better for parallel streams
Predicate vs Function vs Consumer vs Supplier	Predicate: returns boolean; Function: transforms input; Consumer: consumes input; Supplier: produces output
Method Reference vs Lambda Expression	Method reference: cleaner for simple delegation (String::length); Lambda: flexible for complex logic
Default Methods vs Abstract Methods	Default: has implementation, backward compatibility; Abstract: no implementation, must override
Stream.forEach() vs Iterable.forEach()	Stream.forEach(): terminal operation, no guaranteed order in parallel; Iterable.forEach(): always in iteratio
reduce() vs collect()	reduce(): immutable reduction to single value; collect(): mutable reduction to collection
Intermediate Operations vs Terminal Operations	Intermediate: returns Stream, lazy (filter, map); Terminal: triggers processing, returns result (collect, for
anyMatch() vs allMatch() vs noneMatch()	anyMatch(): true if any element matches; allMatch(): true if all match; noneMatch(): true if none match

JPA/Hibernate/Database

Comparison Question	Key Difference
JPA vs Hibernate	JPA: specification/standard; Hibernate: implementation of JPA with extra features
JPA vs JDBC	JDBC: low-level SQL, manual mapping; JPA: high-level ORM, automatic mapping
@OneToMany vs @ManyToOne	@OneToMany: parent side, collection; @ManyToOne: child side, single reference
@ManyToMany vs Two @OneToMany	@ManyToMany: join table, simpler; Two @OneToMany: intermediate entity, more control
FetchType.LAZY vs FetchType.EAGER	LAZY: loads on access, better performance; EAGER: loads immediately, can cause N+1 problems
@JoinColumn vs @JoinTable	@JoinColumn: foreign key in entity table; @JoinTable: separate join table for relationship
First Level Cache vs Second Level Cache	L1: session-scoped, automatic; L2: SessionFactory-scoped, shared across sessions
get() vs load() in Hibernate	get(): returns null if not found, hits DB immediately; load(): returns proxy, throws excepti
save() vs persist() vs saveOrUpdate()	save(): returns ID, Hibernate-specific; persist(): void, JPA standard; saveOrUpdate(): updat
merge() vs update()	merge(): returns managed copy, doesn't attach original; update(): attaches the given instanc
JPQL vs Native SQL vs Criteria API	JPQL: object-oriented queries; Native SQL: database-specific; Criteria: type-safe programmat
@Entity vs @Table	@Entity: marks class as JPA entity; @Table: customizes table name/schema
@Id vs @EmbeddedId vs @IdClass	@Id: single field PK; @EmbeddedId: composite PK as embedded object; @IdClass: composite PK v
@GeneratedValue strategies comparison	AUTO: provider chooses; IDENTITY: DB auto-increment; SEQUENCE: DB sequence; TABLE: separate
@Transactional(readOnly=true) vs @Transactional(readOnly=false)	readOnly=true: optimization hints, no writes; readOnly=false: default, allows modifications
Optimistic Locking vs Pessimistic Locking	Optimistic: version checking, better concurrency; Pessimistic: database locks, prevents conc
@Version vs @Lock	@Version: optimistic locking with version field; @Lock: explicit pessimistic/optimistic lock
CascadeType.ALL vs CascadeType.PERSIST vs CascadeType.MERGE	ALL: all operations cascade; PERSIST: only save cascades; MERGE: only merge cascades
orphanRemoval vs CascadeType.REMOVE	orphanRemoval: removes unreferenced children; CASCADE.REMOVE: deletes children when parent d
@Embeddable vs @Entity	@Embeddable: value object, no identity; @Entity: domain object with identity and lifecycle
@ElementCollection vs @OneToMany	@ElementCollection: for basic/embedded types, no entity; @OneToMany: for entity relationship
@Inheritance strategies comparison	SINGLE_TABLE: one table, discriminator column; JOINED: normalized tables; TABLE_PER_CLASS: c
Named Query vs Dynamic Query	Named: precompiled, cached, validated at startup; Dynamic: flexible, built at runtime
EntityManager vs Session	EntityManager: JPA standard interface; Session: Hibernate-specific with more features
JpaRepository vs CrudRepository vs PagingAndSortingRepository	JpaRepository: JPA-specific methods, batch ops; CrudRepository: basic CRUD; PagingAndSorting
@Modifying vs Regular Query	@Modifying: for UPDATE/DELETE queries, returns void/int; Regular: for SELECT, returns entiti
@Query vs Query Methods vs Specifications	@Query: custom JPQL/SQL; Query Methods: derived from method names; Specifications: programma

Spring Security

Comparison Question	Key Difference
Authentication vs Authorization	Authentication: who you are (identity verification); Authorization: what you can do (access control)
@PreAuthorize vs @PostAuthorize	@PreAuthorize: checks before method execution; @PostAuthorize: checks after execution (can inspect r
@Secured vs @RolesAllowed vs @PreAuthorize	@Secured: Spring-specific, simple roles; @RolesAllowed: Java standard; @PreAuthorize: SpEL expressio
Form-based Authentication vs Token-based Authentication	Form: server sessions, stateful; Token: client stores token, stateless
Session-based vs Stateless Security	Session: server maintains state; Stateless: each request self-contained with credentials
OAuth vs OAuth2	OAuth: complex signatures; OAuth2: simpler with bearer tokens, requires HTTPS
JWT vs Session Tokens	JWT: self-contained claims, stateless; Session: server-side storage, stateful
CSRF vs XSS	CSRF: exploits user's authenticated session; XSS: injects malicious scripts into pages
@EnableWebSecurity vs @EnableGlobalMethodSecurity	@EnableWebSecurity: web-level security config; @EnableGlobalMethodSecurity: method-level annotations
antMatchers() vs mvcMatchers()	antMatchers(): Ant-style patterns; mvcMatchers(): Spring MVC patterns, more secure
hasRole() vs hasAuthority()	hasRole(): automatically prepends "ROLE_"; hasAuthority(): exact match, no prefix
PasswordEncoder vs plain text passwords	PasswordEncoder: hashed/encrypted storage; Plain text: insecure, never use in production
Remember-me vs Persistent Login	Remember-me: cookie-based, limited time; Persistent: long-term token storage
HTTP Basic vs HTTP Digest Authentication	Basic: sends credentials in base64; Digest: sends hashed credentials, more secure

Spring Web/REST

Comparison Question	Key Difference
REST vs SOAP	REST: lightweight, multiple formats, stateless; SOAP: protocol with envelope, XML only, ca
REST vs GraphQL	REST: multiple endpoints, fixed responses; GraphQL: single endpoint, client specifies fiel
@RequestBody vs @ResponseBody	@RequestBody: deserializes request to Java object; @ResponseBody: serializes Java object t
@RequestBody vs @ModelAttribute	@RequestBody: for JSON/XML payload; @ModelAttribute: for form data binding
ResponseEntity vs @ResponseBody	ResponseEntity: full response control (status, headers); @ResponseBody: just body content
PUT vs POST vs PATCH	POST: create resource; PUT: full update/replace; PATCH: partial update
GET vs POST	GET: retrieve data, idempotent, cacheable; POST: submit data, non-idempotent
@RestController vs @Controller + @ResponseBody	@RestController: convenience annotation combining both; separate: more explicit
@ExceptionHandler vs @ControllerAdvice	@ExceptionHandler: handles exceptions in one controller; @ControllerAdvice: global excepti
HandlerInterceptor vs Filter	Interceptor: Spring-specific, access to handler; Filter: Servlet standard, lower level
OncePerRequestFilter vs GenericFilterBean	OncePerRequestFilter: guarantees single execution; GenericFilterBean: basic filter, may ex
Servlet vs Filter	Servlet: handles requests; Filter: preprocesses/postprocesses requests
Session vs Token Authentication	Session: server stores state; Token: client stores token, server stateless
Cookie vs Session	Cookie: client-side storage; Session: server-side storage with session ID in cookie
Stateful vs Stateless	Stateful: server maintains client state; Stateless: each request independent
Content Negotiation vs Fixed Response Format	Content Negotiation: response format based on Accept header; Fixed: always same format
@CrossOrigin vs CORS Filter	@CrossOrigin: per-controller CORS; Filter: global CORS configuration
Forward vs Redirect	Forward: server-side, same request; Redirect: client-side, new request
RequestDispatcher.forward() vs HttpServletResponse.sendRedirect()	forward(): internal, URL unchanged; sendRedirect(): browser redirect, URL changes
WebMvcConfigurer vs WebMvcConfigurerAdapter	WebMvcConfigurer: interface with default methods (Java 8+); Adapter: deprecated abstract c

Spring AOP

Comparison Question	Key Difference
@Before vs @After vs @Around	@Before: executes before method; @After: always after; @Around: wraps method, most powerful
@AfterReturning vs @AfterThrowing	@AfterReturning: only on success; @AfterThrowing: only on exception
JDK Dynamic Proxy vs CGLIB Proxy	JDK: interface-based, Java standard; CGLIB: class-based, creates subclass
Compile-time Weaving vs Load-time Weaving vs Runtime Weaving	Compile: AspectJ compiler; Load: classloader modification; Runtime: proxy-based (Spring default
@Aspect vs @Component	@Aspect: defines aspect with advices; @Component: regular Spring bean
Pointcut vs Join Point vs Advice	Pointcut: where to apply; Join Point: actual execution point; Advice: what to do
AOP Proxy vs Target Object	Proxy: wrapper with aspects applied; Target: original object being proxied
Spring AOP vs AspectJ	Spring AOP: proxy-based, method-only; AspectJ: bytecode weaving, field access, more join points
Method-level AOP vs Class-level AOP	Method: intercepts method calls; Class: can intercept field access, constructor (AspectJ only)

Testing

Comparison Question	Key Difference
@Mock vs @MockBean	@Mock: pure Mockito, unit tests; @MockBean: Spring Boot, replaces beans in context
@Mock vs @Spy	@Mock: complete mock; @Spy: partial mock, real methods unless stubbed
@MockBean vs @SpyBean	@MockBean: replaces with mock; @SpyBean: wraps existing bean with spy
@WebMvcTest vs @SpringBootTest	@WebMvcTest: only web layer; @SpringBootTest: full application context
@DataJpaTest vs @SpringBootTest	@DataJpaTest: only JPA components, in-memory DB; @SpringBootTest: complete context
Unit Testing vs Integration Testing vs End-to-End Testing	Unit: single component; Integration: components interaction; E2E: complete user flow
TDD vs BDD	TDD: test-first development; BDD: behavior-driven with Given-When-Then
JUnit vs TestNG	JUnit: simpler, standard; TestNG: more features, flexible test configuration
MockMvc vs RestTemplate vs TestRestTemplate	MockMvc: no server needed; RestTemplate: real HTTP calls; TestRestTemplate: test-friendly RestTemp
@BeforeEach vs @BeforeAll	@BeforeEach: before each test; @BeforeAll: once before all tests (static)
@Test vs @ParameterizedTest	@Test: single test case; @ParameterizedTest: multiple inputs for same test
@DisplayName vs Method Name	@DisplayName: custom test name in reports; Method name: uses actual method name
Mockito.when() vs Mockito.doReturn()	when(): type-safe, can't use with void; doReturn(): works with void, spies
verify() vs assert()	verify(): checks method invocation; assert(): checks values/state
@DirtiesContext vs Test Isolation	@DirtiesContext: recreates context; Isolation: tests don't affect each other
H2 vs Test Containers	H2: in-memory, fast, may differ from production; TestContainers: real DB in Docker
@Sql vs @DataJpaTest	@Sql: executes SQL scripts; @DataJpaTest: auto-configures test DB

Microservices & Distributed Systems

Comparison Question	Key Difference
Monolithic vs Microservices	Monolithic: single deployable unit; Microservices: multiple independent services
RestTemplate vs WebClient vs FeignClient	RestTemplate: synchronous, deprecated; WebClient: reactive, non-blocking; FeignClient: declarative,
Synchronous vs Asynchronous Communication	Synchronous: waits for response (HTTP); Asynchronous: fire-and-forget (messaging)
HTTP vs Message Queue Communication	HTTP: direct, synchronous; Message Queue: decoupled, asynchronous, reliable
API Gateway vs Load Balancer	API Gateway: routing, auth, rate limiting; Load Balancer: distributes traffic only
Service Discovery vs Load Balancing	Discovery: finds service instances; Load Balancing: distributes requests among instances
Client-side Load Balancing vs Server-side Load Balancing	Client-side: client chooses instance; Server-side: proxy/LB chooses
Eureka vs Consul vs Zookeeper	Eureka: AP in CAP, Spring native; Consul: health checks, KV store; Zookeeper: CP, configuration mar
Circuit Breaker vs Retry	Circuit Breaker: fails fast after threshold; Retry: attempts multiple times
Hystrix vs Resilience4j	Hystrix: Netflix, maintenance mode; Resilience4j: lightweight, functional, actively maintained
Saga Pattern vs Two-Phase Commit	Saga: eventual consistency, compensations; 2PC: strong consistency, blocking
Event Sourcing vs Traditional State Storage	Event Sourcing: stores events; Traditional: stores current state only
CQRS vs Traditional Architecture	CQRS: separate read/write models; Traditional: single model for both
API Gateway vs BFF (Backend for Frontend)	API Gateway: generic for all clients; BFF: tailored per client type
Docker vs Virtual Machine	Docker: shares OS kernel, lightweight; VM: full OS, isolated, heavier
Kubernetes vs Docker Swarm	Kubernetes: complex, more features, industry standard; Swarm: simpler, Docker native
ConfigMap vs Secret in Kubernetes	ConfigMap: non-sensitive config; Secret: sensitive data, base64 encoded
Horizontal Scaling vs Vertical Scaling	Horizontal: add more instances; Vertical: increase instance resources
Stateful vs Stateless Services	Stateless: no session data, easily scalable; Stateful: maintains state, harder to scale
Sidecar vs Library Pattern	Sidecar: separate container/process; Library: embedded in application

Design Patterns

Comparison Question	Key Difference
Factory Pattern vs Abstract Factory Pattern	Factory: creates one product; Abstract Factory: creates families of related products
Factory Method vs Simple Factory	Factory Method: subclasses decide; Simple Factory: single factory class decides
Builder Pattern vs Constructor	Builder: step-by-step construction, fluent; Constructor: all at once, telescoping problem
Singleton vs Static Class	Singleton: instance-based, can implement interfaces; Static: no instance, can't override
Adapter Pattern vs Decorator Pattern	Adapter: changes interface; Decorator: adds functionality, same interface
Adapter vs Facade vs Proxy	Adapter: converts interface; Facade: simplifies interface; Proxy: controls access
Strategy Pattern vs State Pattern	Strategy: interchangeable algorithms; State: behavior changes with internal state
Observer Pattern vs Pub-Sub Pattern	Observer: direct dependency; Pub-Sub: message broker, loosely coupled
Template Method vs Strategy Pattern	Template: inheritance-based, algorithm skeleton; Strategy: composition-based, entire algorithm
Composite Pattern vs Decorator Pattern	Composite: tree structure; Decorator: wraps single object with extras
Chain of Responsibility vs Command Pattern	Chain: passes request along chain; Command: encapsulates request as object
Iterator Pattern vs For-Each Loop	Iterator: explicit traversal control; For-Each: simplified syntax, less control
Flyweight vs Singleton	Flyweight: many shared instances; Singleton: exactly one instance
Bridge Pattern vs Adapter Pattern	Bridge: designed upfront for variation; Adapter: makes incompatible classes work
Mediator vs Observer	Mediator: centralized communication; Observer: direct notification to subscribers
Prototype vs Factory Pattern	Prototype: clones existing instance; Factory: creates new instance from scratch
Command Pattern vs Strategy Pattern	Command: encapsulates request with undo; Strategy: encapsulates algorithm

Spring Reactive

Comparison Question	Key Difference
Spring MVC vs Spring WebFlux	MVC: servlet-based, blocking; WebFlux: reactive, non-blocking
Mono vs Flux	Mono: 0 or 1 element; Flux: 0 to N elements stream
map() vs flatMap() in Reactor	map(): synchronous transformation; flatMap(): async transformation, returns Publisher
subscribe() vs block()	subscribe(): non-blocking, async; block(): blocks thread, waits for result
Reactive Streams vs Java Streams	Reactive: async, backpressure; Java Streams: synchronous, pull-based
Project Reactor vs RxJava	Reactor: Spring native, newer; RxJava: older, more operators
@Controller vs @RestController in WebFlux	Same as MVC but returns reactive types (Mono/Flux)
RouterFunction vs @RequestMapping	RouterFunction: functional endpoints; @RequestMapping: annotation-based
Backpressure vs Throttling	Backpressure: consumer controls flow; Throttling: producer limits rate
Cold Publisher vs Hot Publisher	Cold: starts on subscription; Hot: emits regardless of subscribers
publishOn() vs subscribeOn()	publishOn(): downstream execution thread; subscribeOn(): upstream subscription thread

Caching

Comparison Question	Key Difference
@Cacheable vs @CachePut	@Cacheable: skips method if cached; @CachePut: always executes and updates cache
@CacheEvict vs @CachePut	@CacheEvict: removes from cache; @CachePut: adds/updates cache
Local Cache vs Distributed Cache	Local: in-process, fast; Distributed: shared across instances, network overhead
Ehcache vs Hazelcast vs Redis	Ehcache: Java-focused, local/distributed; Hazelcast: in-memory grid; Redis: key-value store, pe
Cache-aside vs Read-through vs Write-through vs Write-behind	Cache-aside: app manages; Read-through: cache loads; Write-through: sync write; Write-behind: a
TTL vs TTI	TTL: Time To Live from creation; TTI: Time To Idle from last access
LRU vs LFU vs FIFO Eviction	LRU: Least Recently Used; LFU: Least Frequently Used; FIFO: First In First Out
Spring Cache vs JCache	Spring Cache: abstraction layer; JCache (JSR-107): Java standard API

Message Queuing

Comparison Question	Key Difference
JMS vs AMQP	JMS: Java API standard; AMQP: wire protocol, language-agnostic
RabbitMQ vs Kafka	RabbitMQ: message broker, routing; Kafka: event streaming, log-based, higher throughput
Queue vs Topic	Queue: point-to-point, one consumer; Topic: pub-sub, multiple consumers
Point-to-Point vs Publish-Subscribe	P2P: one consumer gets message; Pub-Sub: all subscribers get message
Push Model vs Pull Model	Push: broker sends to consumer; Pull: consumer requests from broker
At-least-once vs At-most-once vs Exactly-once Delivery	At-least-once: may duplicate; At-most-once: may lose; Exactly-once: guaranteed single delivery
Persistent vs Non-persistent Messages	Persistent: survives broker restart; Non-persistent: in-memory only, faster
Synchronous vs Asynchronous Messaging	Synchronous: sender waits; Asynchronous: sender continues immediately

Build Tools & Dependency Management

Comparison Question	Key Difference
Maven vs Gradle	Maven: XML config, convention over configuration; Gradle: Groovy/Kotlin DSL, flexible, faster
compile vs runtime vs provided scope in Maven	compile: all classpaths; runtime: runtime only; provided: compile only (container provides)
implementation vs api in Gradle	implementation: internal dependency; api: exposed to consumers
Maven Repository vs Local Repository	Maven Repo: remote/central; Local: ~/.m2 cache
SNAPSHOT vs RELEASE versions	SNAPSHOT: development version, can change; RELEASE: stable, immutable
Multi-module vs Single-module Project	Multi-module: related projects together; Single: standalone project
BOM vs Direct Dependency Management	BOM: centralized version management; Direct: each project manages versions

Logging & Monitoring

Comparison Question	Key Difference
SLF4J vs Log4j vs Logback	SLF4J: facade/abstraction; Log4j: implementation; Logback: Log4j successor, SLF4J native
ERROR vs WARN vs INFO vs DEBUG vs TRACE	ERROR: failures; WARN: potential issues; INFO: important events; DEBUG: detailed flow; TRACE: most detailed
Console Appender vs File Appender	Console: outputs to stdout; File: writes to disk files
Synchronous Logging vs Asynchronous Logging	Synchronous: blocks until written; Asynchronous: queued, better performance
MDC vs ThreadLocal	MDC: logging-specific context; ThreadLocal: general thread-local storage
Metrics vs Logs vs Traces	Metrics: numerical measurements; Logs: events; Traces: request flow across services
Prometheus vs Graphite	Prometheus: pull model, powerful queries; Graphite: push model, simpler
Pull-based vs Push-based Monitoring	Pull: monitor queries services; Push: services send to monitor

Performance & Optimization

Comparison Question	Key Difference
Eager Initialization vs Lazy Initialization	Eager: at startup, predictable; Lazy: on first use, saves resources
Connection Pooling vs Creating Connections	Pooling: reuses connections, faster; Creating: overhead per request
Database Index vs Full Table Scan	Index: fast lookup, maintenance cost; Full scan: reads all rows, slow
Caching vs Recomputing	Caching: space for time trade-off; Recomputing: always fresh, no memory overhead
Vertical Scaling vs Horizontal Scaling	Vertical: bigger machine; Horizontal: more machines
Synchronous vs Asynchronous Processing	Synchronous: simpler, blocking; Asynchronous: complex, non-blocking, better throughput
Batch Processing vs Real-time Processing	Batch: periodic bulk processing; Real-time: immediate processing
N+1 Query Problem vs Join Fetching	N+1: multiple queries; Join: single query with joins

Cloud & Deployment

Comparison Question	Key Difference
IaaS vs PaaS vs SaaS	IaaS: infrastructure only; PaaS: platform + infrastructure; SaaS: complete application
Public Cloud vs Private Cloud vs Hybrid Cloud	Public: shared infrastructure; Private: dedicated; Hybrid: combination
AWS vs Azure vs GCP	AWS: market leader, most services; Azure: Microsoft integration; GCP: strong in ML/data
EC2 vs Lambda	EC2: virtual servers, always running; Lambda: serverless functions, pay per execution
Container vs Serverless	Container: packaged app environment; Serverless: no infrastructure management
Blue-Green Deployment vs Canary Deployment	Blue-Green: instant switch; Canary: gradual rollout
Rolling Deployment vs Recreate Deployment	Rolling: gradual update; Recreate: all at once with downtime
Terraform vs CloudFormation	Terraform: multi-cloud, HCL; CloudFormation: AWS-only, JSON/YAML

Note: This document contains comparison questions with key differences for Java and Spring Boot interviews. For detailed explanations, candidates should understand the underlying concepts, use cases, trade-offs, and best practices for each comparison.