**Java & Spring boot Interview Questions and Answers:**

1. **Java Fundamentals**
2. **Spring Core & Boot**
3. **Data Access (JPA, transactions)**
4. **REST APIs & Security**
5. **Testing & DevOps**
6. **Cloud Deployment**
7. **Advanced Topics (Microservices, Kafka)**

**Perfect. I’ll now begin with the first set of Spring Boot interview questions – grouped by category. Here are the first 20 questions from the Java Fundamentals and Spring Core/Boot sections.**

**Java Fundamentals**

1. What is the difference between == and. equals() in Java?  
   == compares object references, not content.  
   .equals() compares actual data or content of objects.  
   Used correctly, == is for primitives or reference checks.  
   .equals() is overridden in classes like String.
2. What is the purpose of the final keyword?  
   It marks variables as constants, methods as non-overridable,  
   and classes as non-extendable.  
   Helps enforce immutability and secure inheritance.  
   Frequently used in design patterns and constants.
3. What is multithreading in Java?  
   Multithreading allows concurrent execution of two or more threads.  
   It enhances performance by utilizing CPU efficiently.  
   Java provides Thread and Runnable interfaces for implementation.  
   It is essential for building scalable apps.
4. What is a lambda expression?  
   A lambda is a shorthand for implementing functional interfaces.  
   It reduces boilerplate for anonymous inner classes.  
   Syntax: (param) -> expression.  
   Commonly used in streams and collections.
5. What is the difference between ArrayList and LinkedList?  
   ArrayList is backed by a dynamic array, fast for access.  
   LinkedList uses doubly linked nodes, fast for insert/delete.  
   ArrayList has better performance for index-based operations.  
   LinkedList is better for queue and stack implementations.
6. Explain the use of Optional in Java.  
   Optional helps avoid NullPointerException.  
   It wraps a value that might be null and provides safe methods.  
   Use isPresent(), ifPresent() or orElse() to access data.  
   Helps in writing cleaner and safer code.
7. What is method overloading vs overriding?  
   Overloading: same method name, different parameters.  
   Overriding: child class redefines parent’s method.  
   Overloading happens at compile-time, overriding at runtime.  
   Used for flexibility and polymorphism.
8. What is the difference between HashMap and TreeMap?  
   HashMap stores entries in an unordered way.  
   TreeMap stores keys in sorted order (natural/comparator).  
   HashMap is faster, TreeMap is useful when order matters.  
   Both implement Map but differ in use cases.
9. What are records in Java 16+?  
   Records are immutable data carriers.  
   They reduce boilerplate for POJOs with auto-generated code.  
   Declared with record keyword.  
   Used often in Spring for DTOs and configuration classes.
10. What is the purpose of the volatile keyword?  
    It ensures visibility of changes to variables across threads.  
    Prevents caching of variable value by threads.  
    Useful in double-checked locking and singleton patterns.  
    Not a replacement for synchronization but complements it.

Spring Core / Spring Boot (11–20)

1. What is the Spring Framework?  
   Spring is a powerful Java framework for enterprise applications.  
   It promotes loose coupling via Dependency Injection (DI).  
   It simplifies Java development using various modules.  
   Spring Boot builds on top of it for rapid app development.
2. What is Dependency Injection (DI)?  
   DI is a design pattern where dependencies are provided externally.  
   Spring handles DI via constructors, setters, or field injection.  
   Promotes testability, modularity, and easier maintenance.  
   Core concept of Spring’s IoC container.
3. What is an ApplicationContext?  
   It's the central interface to Spring’s IoC container.  
   It holds bean definitions and manages their lifecycle.  
   Provides advanced features like event propagation and AOP.  
   Used to retrieve and wire beans automatically.
4. What is a Spring Bean?  
   A Spring Bean is an object managed by the Spring container.  
   It is instantiated, assembled, and managed by Spring.  
   Declared using @Component, @Service, @Repository, etc.  
   Beans promote loose coupling and are singleton by default.
5. What is Spring Boot?  
   Spring Boot simplifies building Spring apps quickly.  
   Provides auto-configuration, starter dependencies, and an embedded server.  
   Reduces boilerplate code and promotes convention over configuration.  
   Perfect for REST APIs and microservices.
6. How does Spring Boot auto-configuration work?  
   Spring Boot uses @EnableAutoConfiguration internally.  
   It scans classpath and applies configurations conditionally.  
   It reduces manual bean declaration and setup.  
   Customization is possible via application.properties or YAML.
7. What is the use of application.properties?  
   It allows external configuration of Spring Boot apps.  
   Commonly used for DB URLs, ports, logging, etc.  
   Can be environment-specific (e.g., application-dev.properties).  
   Helps in managing config without touching code.
8. What are starter dependencies in Spring Boot?  
   Starters are convenient dependency descriptors.  
   They bundle commonly used dependencies together.  
   For example, spring-boot-starter-web, data-jpa, etc.  
   Reduces effort in managing dependencies manually.
9. What is the @SpringBootApplication annotation?  
   It is a combination of @Configuration, @EnableAutoConfiguration, @ComponentScan.  
   Acts as the entry point for Spring Boot applications.  
   Placed on the main class to bootstrap the app.  
   Simplifies configuration using a single annotation.
10. How to run a Spring Boot application?  
    Run via the main method using SpringApplication.run().  
    Use Maven/Gradle or IDE to execute.  
    Can also package and run as a jar using java -jar.  
    Supports embedded Tomcat by default.

Great. Here's the next **40 Spring Boot interview questions** covering:

* **Data Access (JPA, Transactions, Queries)**
* **REST APIs**
* **API Security**
* **Persistence & Databases**

**Data Access (21–35)**

1. **What is Spring Data JPA?**  
   Spring Data JPA simplifies data access using JPA.  
   It eliminates boilerplate code with built-in repository support.  
   Interfaces like JpaRepository provide CRUD out-of-the-box.  
   Custom queries can be added easily.
2. **What is the difference between CrudRepository and JpaRepository?**  
   CrudRepository offers basic CRUD operations.  
   JpaRepository extends it with JPA-specific features like flush(), paging.  
   Choose JpaRepository for richer functionality.  
   Both reduce the need for boilerplate DAO code.
3. **What is the purpose of @Entity annotation?**  
   It marks a class as a JPA entity mapped to a table.  
   Each field is a column unless annotated otherwise.  
   Spring Boot automatically scans for it in @EntityScan.  
   Must include an @Id annotated primary key field.
4. **What is the role of @Id and @GeneratedValue?**  
   @Id marks the primary key field.  
   @GeneratedValue auto-generates the value using strategies like AUTO, IDENTITY.  
   It's used in entity classes for key generation.  
   Ensures uniqueness and simplifies DB interaction.
5. **What is the use of JPQL?**  
   JPQL is Java Persistence Query Language.  
   It's object-oriented and operates on entity objects.  
   Used in @Query to fetch data using entity names.  
   Supports joins, projections, and custom queries.
6. **How to create custom queries in Spring Data JPA?**  
   Use @Query with JPQL or native SQL.  
   Supports named parameters and projections.  
   Place annotation above method in Repository interface.  
   Example: @Query("SELECT u FROM User u WHERE u.name = :name").
7. **What is Lazy vs Eager fetching?**  
   Lazy loads data only when accessed.  
   Eager fetches associated entities immediately.  
   Use Lazy for performance and memory efficiency.  
   Defined using @OneToMany(fetch = FetchType.LAZY).
8. **What is a DTO in Spring?**  
   DTO (Data Transfer Object) carries data between layers.  
   Avoids exposing entity structure to API clients.  
   Improves security, flexibility, and abstraction.  
   Often mapped manually or with libraries like MapStruct.
9. **What is the role of @Transactional?**  
   Ensures atomicity of database operations.  
   Used at method or class level for transaction management.  
   If any exception occurs, rollback happens.  
   It integrates with Spring AOP and JPA.
10. **What is the difference between save() and saveAndFlush()?**  
    save() stores entity and commits at transaction end.  
    saveAndFlush() immediately writes changes to DB.  
    Useful in batch updates or when needed mid-transaction.  
    flush() clears persistence context to sync with DB.
11. **How do you handle database schema changes?**  
    Use tools like Flyway or Liquibase for versioned migrations.  
    They manage incremental changes with tracking.  
    Scripts are run automatically at startup.  
    Promotes consistency across environments.
12. **How do you implement pagination in Spring Data?**  
    Use Pageable and return Page<T>.  
    Spring Data handles SQL LIMIT/OFFSET internally.  
    Supports sorting via Sort object.  
    Enables efficient data retrieval for large datasets.
13. **What is the N+1 select problem in JPA?**  
    Occurs when one query fetches N related entities,  
    and each triggers an additional query (N more).  
    Resolved by using fetch join or @EntityGraph.  
    Causes performance issues if unchecked.
14. **What are entity relationships in JPA?**  
    JPA supports @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.  
    Used to define foreign key relations.  
    Each type uses mappedBy or join columns.  
    Important for ORM and query generation.
15. **What is the difference between native and JPQL queries?**  
    Native queries use SQL and DB-specific syntax.  
    JPQL is portable and uses entity names.  
    Use native for performance-heavy or complex joins.  
    JPQL is recommended for cross-DB compatibility.

**REST APIs (36–45)**

1. **What is a RESTful API?**  
   REST is an architecture using HTTP methods for CRUD.  
   It’s stateless, uses JSON/XML, and standard verbs.  
   Spring Boot simplifies REST creation using @RestController.  
   Resources are identified via URLs.
2. **How to create a REST controller in Spring Boot?**  
   Use @RestController and map with @RequestMapping.  
   Define methods using @GetMapping, @PostMapping, etc.  
   Return objects directly – Spring converts to JSON.  
   Handles RESTful requests efficiently.
3. **How to validate request bodies in Spring Boot?**  
   Use @Valid with DTOs and annotate fields.  
   Use annotations like @NotNull, @Size, etc.  
   Bind errors via BindingResult.  
   Validation triggers before controller logic runs.
4. **What is @RequestBody and @ResponseBody?**  
   @RequestBody maps JSON to Java object.  
   @ResponseBody sends Java object as JSON response.  
   @RestController includes @ResponseBody by default.  
   Essential for API data handling.
5. **How do you handle exceptions in REST APIs?**  
   Use @ControllerAdvice with @ExceptionHandler.  
   Handle specific or generic exceptions globally.  
   Return custom error response using ResponseEntity.  
   Improves error consistency and debugging.
6. **What is rate limiting in REST APIs?**  
   Restricts number of API calls per time unit.  
   Protects against abuse and overuse.  
   Use tools like Bucket4j, Redis, or API gateways.  
   Often implemented using filters or interceptors.
7. **How to upload and download files in Spring Boot?**  
   Use MultipartFile in controller to handle uploads.  
   Store files on disk or DB as BLOB.  
   Use ResponseEntity<Resource> for downloads.  
   Add headers for content type and disposition.
8. **What is content negotiation in Spring Boot?**  
   Determines the format of API response (JSON/XML).  
   Spring uses Accept headers or URL suffix.  
   Configured via ContentNegotiationConfigurer.  
   Supports multiple response types.
9. **What is HATEOAS in REST APIs?**  
   HATEOAS adds links to REST responses.  
   Helps clients navigate through API dynamically.  
   Spring HATEOAS provides link-building support.  
   Improves discoverability of APIs.
10. **What is a ResponseEntity?**  
    Wrapper for HTTP response with body, headers, and status.  
    Use it to customize REST responses.  
    Supports generic typing and flexible return types.  
    Improves control over API behavior.

**API Security (46–50)**

1. **How does Spring Security integrate with Spring Boot?**  
   Auto-configures basic authentication on classpath detection.  
   Security filter chains intercept and authorize requests.  
   Customizable with beans like SecurityFilterChain.  
   Supports form login, JWT, OAuth, etc.
2. **What is JWT authentication?**  
   JWT (JSON Web Token) is a stateless token-based auth.  
   Contains claims, signed using secret or key.  
   Used in REST APIs for authentication and authorization.  
   Decoded and verified in filters or interceptors.
3. **How to secure REST APIs with Spring Security?**  
   Use HttpSecurity to configure endpoints.  
   Permit or restrict roles with .antMatchers().  
   Use JWT or session-based authentication.  
   Define UserDetailsService and password encoders.
4. **What is OAuth2 and how is it used in Spring?**  
   OAuth2 is a token-based authorization framework.  
   Spring Security OAuth2 supports login via Google, GitHub, etc.  
   Manages access tokens and user profiles.  
   Configured using spring-security-oauth2-client.
5. **How to define roles and authorities in Spring Security?**  
   Use @PreAuthorize("hasRole('ADMIN')") or .hasAuthority().  
   Roles are prefixed with ROLE\_ internally.  
   Authorities provide fine-grained permission control.  
   Defined in user details or token claims.

Here are the **last 50 Spring Boot interview questions and answers**, covering:

* **Testing**
* **DevOps & Monitoring**
* **Spring Boot with Cloud**
* **Advanced Topics**
* **Real-time Scenarios**

Each answer is limited to around 4 lines as requested.

**Testing in Spring Boot (51–60)**

1. **What is @SpringBootTest used for?**  
   It loads the full application context for integration testing.  
   Simulates real application behavior with full environment.  
   Useful for end-to-end testing.  
   Slower than unit tests due to full context load.
2. **How do you test REST controllers?**  
   Use @WebMvcTest to test only the web layer.  
   Combine with MockMvc for simulating HTTP requests.  
   Keeps tests fast and isolated.  
   Does not load service or repository beans.
3. **How do you write unit tests for services?**  
   Use @Mock for dependencies and @InjectMocks for service.  
   Use Mockito or JUnit for assertions.  
   Focus on logic without involving Spring context.  
   Fast and ideal for CI pipelines.
4. **What is TestRestTemplate?**  
   Used for integration testing of REST endpoints.  
   Sends HTTP requests to running app context.  
   Supports authentication, headers, and JSON handling.  
   Used inside @SpringBootTest.
5. **How to test a JPA repository?**  
   Use @DataJpaTest for in-memory DB testing.  
   Only repository layer is initialized.  
   Uses H2 by default.  
   Tests queries and entity mappings.
6. **What is the difference between @MockBean and @Mock?**  
   @MockBean is Spring-specific; replaces bean in context.  
   @Mock is a pure Mockito mock.  
   Use @MockBean when Spring needs to inject the mock.  
   Use @Mock in pure unit tests.
7. **How do you test exceptions in Spring Boot?**  
   Use MockMvc and assert status codes and error messages.  
   Use assertThrows in unit tests.  
   Test @ControllerAdvice handlers with mock inputs.  
   Ensure proper handling of bad inputs or logic failures.
8. **What is @WebMvcTest?**  
   It loads only controller-related beans.  
   Excludes services, repositories, etc.  
   Used with MockMvc for fast web layer tests.  
   Ideal for REST API unit testing.
9. **How to mock a service in controller tests?**  
   Use @MockBean in @WebMvcTest setup.  
   Spring injects mock into controller automatically.  
   Use Mockito to define behavior.  
   Keeps controller tests isolated.
10. **How do you profile test vs dev environments?**  
    Use @ActiveProfiles("test") in test classes.  
    Keeps configs separate for DB, services, etc.  
    Define application-test.yml for overrides.  
    Prevents polluting production logic.

**DevOps & Monitoring (61–70)**

1. **How to package a Spring Boot app?**  
   Use mvn package or gradle build.  
   Creates a fat JAR with dependencies.  
   Executable via java -jar.  
   Ideal for deployment without external containers.
2. **What is Spring Boot Actuator?**  
   Exposes endpoints for health, metrics, and info.  
   Useful for monitoring and diagnostics.  
   Enable endpoints in application.yml.  
   Secured by default in production.
3. **Which Actuator endpoints are commonly used?**  
   /health, /info, /metrics, /env, /beans.  
   Show system state, properties, and performance.  
   Custom endpoints can also be added.  
   Supports integration with Prometheus, Grafana.
4. **How to monitor Spring Boot applications?**  
   Use Actuator for local metrics.  
   Export to Prometheus or use New Relic, ELK, etc.  
   Enable micrometer integration for dashboards.  
   Monitor logs, traces, and response times.
5. **What is Micrometer in Spring Boot?**  
   A metrics facade for Spring Boot.  
   Integrates with Prometheus, Datadog, etc.  
   Exposes JVM, GC, DB, and HTTP metrics.  
   Enabled automatically with Actuator.
6. **How to externalize configuration in Spring Boot?**  
   Use application.yml, application.properties, or env variables.  
   Supports profiles like application-dev.yml.  
   Priority: command-line > env > file > default.  
   Enables flexibility across environments.
7. **How to run Spring Boot in Docker?**  
   Create a Dockerfile with FROM openjdk.  
   Copy JAR and run with CMD.  
   Expose necessary ports.  
   Use Docker Compose for multi-container setup.
8. **What is a health check endpoint?**  
   /actuator/health shows app readiness.  
   Returns UP or DOWN status.  
   Can include custom indicators (DB, MQ, etc.).  
   Essential for cloud orchestration tools.
9. **How do you manage logs in Spring Boot?**  
   Uses Logback by default.  
   Configurable via logback.xml or application.yml.  
   Supports log levels, patterns, file output.  
   Integrate with ELK or Splunk for central logging.
10. **How to enable graceful shutdown?**  
    Spring Boot supports shutdown hooks.  
    Set server.shutdown=graceful in config.  
    Allows cleanup of resources and threads.  
    Useful for containerized deployments.

**Spring Boot with Cloud (71–80)**

1. **What is Spring Cloud?**  
   Spring Cloud provides tools for microservices.  
   Includes service discovery, config server, API gateway.  
   Integrates with Netflix OSS tools.  
   Optimized for distributed systems.
2. **What is Spring Cloud Config Server?**  
   Centralized external configurations for all apps.  
   Stores configs in Git or filesystem.  
   Clients fetch settings using REST.  
   Reduces config duplication.
3. **What is service discovery?**  
   Allows services to find each other dynamically.  
   Spring uses Eureka (Netflix) for discovery.  
   Clients register and lookup via registry.  
   Avoids hard-coded service URLs.
4. **What is Spring Cloud Gateway?**  
   A modern API gateway for Spring Boot apps.  
   Handles routing, filtering, and rate-limiting.  
   Replaces Zuul 1.x.  
   Supports load balancing with Eureka.
5. **What is a circuit breaker?**  
   Protects services from cascading failure.  
   Stops calls when service is down.  
   Spring Cloud supports Resilience4j for this.  
   Enables fallback logic.
6. **What is a fallback method?**  
   Executed when primary method fails or is short-circuited.  
   Defined using @CircuitBreaker with Resilience4j.  
   Improves fault tolerance.  
   Useful in unstable network environments.
7. **What is Feign Client in Spring Cloud?**  
   Declarative REST client built on top of HTTP.  
   Auto-generates code based on interfaces.  
   Simplifies service-to-service calls.  
   Integrated with Ribbon and Eureka.
8. **What is load balancing in Spring Boot?**  
   Ribbon handles client-side load balancing.  
   Distributes traffic among service instances.  
   Works with Eureka for instance registry.  
   Spring Cloud LoadBalancer is newer alternative.
9. **How to refresh config at runtime?**  
   Use @RefreshScope and /actuator/refresh endpoint.  
   Reloads beans with updated properties.  
   Config Server must support Spring Cloud Bus.  
   Used in distributed config changes.
10. **What is Spring Cloud Bus?**  
    Propagates events across microservices using a message broker.  
    Used for config refresh, alerts, or sync.  
    Works with RabbitMQ, Kafka, etc.  
    Reduces need for individual manual refresh.

**Advanced & Scenario-Based (81–100)**

1. **How to secure endpoints with roles in Spring Boot?**  
   Use @PreAuthorize("hasRole('ADMIN')") or antMatchers() in security config.  
   Secure APIs based on user roles.  
   Supports role hierarchy and custom voters.  
   Spring evaluates before method execution.
2. **How to version a REST API?**  
   Use URI versioning /api/v1/resource.  
   Or use header/version param based approaches.  
   Maintain backward compatibility.  
   Spring supports all methods.
3. **How to handle circular dependencies?**  
   Refactor design to decouple components.  
   Use @Lazy to delay bean injection.  
   Avoid field injection; prefer constructor injection.  
   Use interfaces to break tight coupling.
4. **What causes memory leaks in Spring Boot apps?**  
   Unclosed streams, thread leaks, static maps, large caches.  
   Heavy use of reflection, retained objects.  
   Monitor with tools like VisualVM or JProfiler.  
   Use @PreDestroy for cleanup.
5. **What is an ApplicationContext?**  
   Central interface for accessing beans and environment.  
   Spring Boot auto-configures it at startup.  
   Used for dependency injection and lifecycle management.  
   Sub-interface of BeanFactory.
6. **How to stop bean initialization conditionally?**  
   Use @ConditionalOnProperty, @Conditional, or profiles.  
   Prevents loading beans under certain config.  
   Improves app flexibility.  
   Used in starter modules and features.
7. **How does Spring Boot handle concurrency?**  
   Use @Async for async methods.  
   Use @Scheduled for periodic tasks.  
   Spring creates separate thread pools.  
   Customize with TaskExecutor beans.
8. **How do you handle environment-specific properties?**  
   Use application-{profile}.yml.  
   Activate with --spring.profiles.active=dev.  
   Keeps dev, test, prod isolated.  
   Supports overriding with env vars.
9. **What is a bean lifecycle in Spring?**  
   Beans go through instantiation → dependency injection → initialization → destruction.  
   Hooks: @PostConstruct, @PreDestroy, InitializingBean.  
   Used for resource setup and cleanup.  
   Managed by Spring container.
10. **How to schedule tasks in Spring Boot?**  
    Use @EnableScheduling and @Scheduled.  
    Supports cron, fixedDelay, and fixedRate.  
    Used for background jobs.  
    Spring creates a scheduler behind the scenes.
11. **How do you load data at app startup?**  
    Use CommandLineRunner or ApplicationRunner.  
    Run logic after Spring context initializes.  
    Used for seeding DB or setup.  
    Inject dependencies as needed.
12. **What is ObjectMapper?**  
    Jackson's core JSON mapper used in Spring Boot.  
    Converts Java objects to/from JSON.  
    Can be customized with modules.  
    Auto-configured by Spring Boot.
13. **What are common Spring Boot performance tips?**  
    Use lazy loading, connection pooling, batch inserts.  
    Avoid N+1, minimize bean scanning.  
    Profile JVM and SQL queries.  
    Use Actuator and metrics to monitor.
14. **What is @ConfigurationProperties?**  
    Binds external config to POJOs.  
    Supports nested, validated properties.  
    Use @EnableConfigurationProperties.  
    Cleaner than using @Value repeatedly.
15. **How do you integrate Redis in Spring Boot?**  
    Add Redis starter dependency.  
    Configure connection in YAML.  
    Use RedisTemplate or @Cacheable.  
    Used for caching and pub-sub.
16. **What is the difference between @Component, @Service, @Repository?**  
    All register beans in Spring context.  
    @Service for business logic.  
    @Repository for persistence layer and exception translation.  
    @Component is generic.
17. **What is dependency injection in Spring Boot?**  
    Provides objects their dependencies via constructors or fields.  
    Promotes loose coupling.  
    Managed by Spring IoC container.  
    Supports constructor, setter, and field injection.
18. **How to inject a list of beans?**  
    Use @Autowired List<MyInterface>.  
    Spring injects all beans of that type.  
    Used in strategy pattern.  
    Order can be defined using @Order.
19. **What is an embedded server in Spring Boot?**  
    Spring Boot uses embedded Tomcat, Jetty, or Undertow.  
    No need to deploy WAR to external server.  
    Configured via properties or builder.  
    Runs via main() method.
20. **How to optimize startup time in Spring Boot?**  
    Use lazy init, reduce auto-config, limit classpath scanning.  
    Profile your app with spring.devtools.restart.enabled=false.  
    Exclude unused starters.  
    Split monoliths into smaller apps if needed.

You've covered nearly **all core Spring Boot interview topics** with those 100 questions. However, a few **important or niche topics** often asked in senior interviews are still **missing or underexplored**. Here's a list of **missing or lightly covered areas**, along with **30 bonus questions and answers**, grouped by topic.

1. **Spring Boot and GraphQL**
2. **Spring Boot and WebSockets**
3. **Reactive Programming with Spring WebFlux**
4. **Security Advanced Topics (JWT, OAuth2)**
5. **Multi-module project structure**
6. **Feature toggles and conditional loading**
7. **Spring Boot with gRPC**
8. **Startup performance tuning**
9. **Profile-based bean registration**
10. **Event-driven architecture**

**Bonus 30 Questions and Answers**

**Spring Boot + GraphQL (1–5)**

1. **What is Spring Boot GraphQL?**  
   Spring for GraphQL integrates GraphQL APIs into Spring Boot.  
   It uses annotations like @QueryMapping, @SchemaMapping.  
   It’s strongly typed and supports schema-first design.  
   Optimized for complex nested queries.
2. **How is GraphQL different from REST?**  
   REST uses fixed endpoints; GraphQL uses a single endpoint.  
   GraphQL lets clients query exactly what they need.  
   Reduces over-fetching or under-fetching of data.  
   GraphQL supports types, introspection, and nested objects.
3. **What is a DataFetcher in GraphQL?**  
   DataFetcher defines how to fetch data for a field.  
   Spring hides this using annotations.  
   Can be customized for non-trivial mappings.  
   Helps with fine-grained data control.
4. **What are common GraphQL annotations in Spring Boot?**

* @SchemaMapping – binds method to GraphQL field
* @QueryMapping – defines GraphQL query handler
* @MutationMapping – for mutation operations
* @SubscriptionMapping – for real-time updates

1. **How do you handle validation in GraphQL?**  
   Use Java Bean Validation on input DTOs.  
   Leverage @Valid and @Validated.  
   GraphQL catches and returns validation errors.  
   Custom formatters can be added for error structure.

**WebSockets in Spring Boot (6–8)**

1. **What is WebSocket support in Spring Boot?**  
   Enables full-duplex communication.  
   Use @EnableWebSocket or STOMP with SockJS.  
   Ideal for chat apps, live feeds, etc.  
   Spring handles connection lifecycle and messaging.
2. **How to implement WebSocket endpoint?**  
   Create a config implementing WebSocketConfigurer.  
   Use @ServerEndpoint or message brokers.  
   Handle sessions, messages, and broadcast logic.  
   Can scale with Redis pub-sub or Kafka.
3. **Difference between WebSocket and REST?**  
   REST is stateless and request-response.  
   WebSocket is stateful and bi-directional.  
   WebSocket reduces latency for real-time updates.  
   Needs client-side socket handling.

**Spring WebFlux (Reactive) (9–12)**

1. **What is Spring WebFlux?**  
   A reactive web framework built on Project Reactor.  
   Supports async, non-blocking applications.  
   Scales better under high I/O load.  
   Uses Mono and Flux types.
2. **What is Mono and Flux?**  
   Mono – 0 or 1 item; Flux – 0 to N items.  
   Both are reactive stream types.  
   Emit data asynchronously.  
   Used in reactive APIs for pipelines.
3. **How is WebFlux different from MVC?**  
   WebFlux is non-blocking and reactive.  
   MVC is blocking and thread-per-request.  
   WebFlux uses Netty instead of Tomcat.  
   Better for microservices with high concurrency.
4. **When to choose WebFlux over Spring MVC?**  
   When app needs high concurrency or streaming.  
   Useful for chat apps, real-time feeds.  
   Avoid if blocking DB or legacy libraries are used.  
   More complex debugging.

**Spring Security Advanced (13–17)**

1. **How do you secure REST APIs with JWT?**  
   Use filters to extract JWT from headers.  
   Validate token using a secret or public key.  
   Store user roles in token claims.  
   Spring Security context is set from token.
2. **What is OAuth2 in Spring Boot?**  
   OAuth2 enables third-party login (Google, GitHub).  
   Spring Security provides client and resource server support.  
   Tokens (Access + Refresh) manage sessions.  
   Scopes and consent handled via providers.
3. **What is the difference between JWT and OAuth2?**  
   JWT is a token format.  
   OAuth2 is an auth framework using tokens.  
   JWT can be used in OAuth2.  
   OAuth2 flows define how tokens are issued.
4. **How do you handle role-based access in JWT apps?**  
   Extract roles from JWT claims.  
   Use @PreAuthorize, @Secured, or config rules.  
   Spring Security filters enforce access.  
   Supports fine-grained control per endpoint.
5. **What is CSRF and how to disable it in stateless APIs?**  
   CSRF protects forms in browser-based apps.  
   For stateless JWT APIs, disable using .csrf().disable().  
   No session = no CSRF risk.  
   Always secure endpoints with tokens or headers.

**Multi-module & Large Projects (18–21)**

1. **How to structure a multi-module Spring Boot app?**  
   Split into: common, api, service, web, config.  
   Use Maven/Gradle parent-child setup.  
   Promotes reusability and clean architecture.  
   Each module builds independently.
2. **How to share DTOs across modules?**  
   Place DTOs in a common/shared module.  
   Import as dependency in other modules.  
   Avoid circular dependencies.  
   Apply versioning if used across services.
3. **How do you handle conditional bean loading in modules?**  
   Use @Conditional, @ConditionalOnProperty, @Profile.  
   Control bean loading via env or flags.  
   Improves modularity and feature toggling.  
   Reduces startup cost for unused features.
4. **How to debug bean loading order in multi-module apps?**  
   Enable DEBUG logs for Spring context.  
   Use Actuator /beans endpoint.  
   Add @DependsOn to control order.  
   Check @ComponentScan paths.

**gRPC and Spring Boot (22–23)**

1. **How to use gRPC in Spring Boot?**  
   Add gRPC starter and .proto files.  
   Generate Java stubs using plugin.  
   Implement services and wire with Spring.  
   Faster and type-safe than REST.
2. **gRPC vs REST in Spring Boot?**  
   gRPC uses Protobuf; REST uses JSON.  
   gRPC is binary and faster over network.  
   Ideal for internal microservices.  
   REST is better for external/public APIs.

**Performance & Profiles (24–27)**

1. **How to reduce Spring Boot startup time?**  
   Enable lazy init, avoid unused auto-configs.  
   Use classpath exclusions and slimmer JARs.  
   Profile bean creation with Actuator.  
   Use native image if needed.
2. **How to register beans only for dev or prod?**  
   Use @Profile("dev") or @ConditionalOnExpression.  
   Activate with spring.profiles.active.  
   Keeps environments isolated.  
   Avoids dev-only beans leaking into prod.
3. **What is spring.main.lazy-initialization=true?**  
   Enables lazy init for all beans.  
   Speeds up app startup.  
   Beans created only when needed.  
   Useful for testing or microservices.
4. **How to log bean initialization time?**  
   Enable org.springframework.boot.diagnostics logs.  
   Use ApplicationListener<ApplicationReadyEvent>.  
   Combine with Actuator for profiling.  
   Detect slow or misconfigured beans.

**Event-Driven Architecture (28–30)**

1. **How does Spring Boot support event-driven design?**  
   Use ApplicationEventPublisher and @EventListener.  
   Decouples components.  
   Supports async events with @Async.  
   Helps modularize logic.
2. **What is a domain event in Spring Boot?**  
   Represents business event like OrderPlaced.  
   Emitted by domain logic.  
   Handled by async or local listeners.  
   Encourages DDD practices.
3. **How do you implement async event handling?**  
   Use @EventListener with @Async.  
   Enable @EnableAsync.  
   Events processed in background threads.  
   Does not block main request flow.

**Microservices – 15 Important Questions**

1. **What is a Microservice?**  
   A microservice is a small, independent service that focuses on a specific business function.  
   It runs in its own process and communicates via lightweight protocols (like HTTP/REST).  
   Each service is independently deployable and scalable.  
   Promotes loose coupling and high cohesion.
2. **What are the advantages of Microservices?**  
   Independent deployment and scaling of components.  
   Faster development cycles and easier debugging.  
   Resilient architecture — failure in one doesn’t crash the entire app.  
   Tech stack flexibility per service.
3. **How does Microservices communicate?**  
   Primarily via REST APIs, gRPC, or messaging (Kafka, RabbitMQ).  
   Communication is stateless and can be synchronous or asynchronous.  
   JSON and Protobuf are common data formats.  
   Message queues ensure decoupled interactions.
4. **What is service discovery?**  
   Service discovery helps locate microservices dynamically.  
   Tools like **Eureka, Consul, Zookeeper** manage this.  
   Clients use registries to find service URLs.  
   Vital in dynamic environments like Docker or Kubernetes.
5. **What is the role of an API Gateway?**  
   API Gateway handles routing, authentication, and rate-limiting.  
   It acts as a single entry point to microservices.  
   Popular options: **Zuul, Spring Cloud Gateway, Kong**.  
   Improves security and centralizes cross-cutting concerns.
6. **How do you handle transactions across services?**  
   Use **Saga Pattern** or **Event Choreography**.  
   Avoid distributed transactions when possible.  
   Ensure idempotency for retry logic.  
   Use message brokers for reliable event flow.
7. **What is the Circuit Breaker pattern?**  
   It prevents cascading failures when a service is down.  
   Used with tools like **Resilience4j, Hystrix**.  
   It stops repeated calls to a failing service temporarily.  
   Restores once the service is healthy.
8. **What is fault tolerance in microservices?**  
   Ability to continue working despite failure.  
   Achieved using retries, circuit breakers, fallbacks.  
   Redundant services and timeout configs help.  
   Monitoring is key for reactive behavior.
9. **How do you monitor microservices?**  
   Use centralized logging (ELK stack), metrics (Prometheus + Grafana).  
   Spring Boot Actuator exposes health and metrics.  
   Distributed tracing with **Zipkin** or **Jaeger**.  
   Helps in debugging across services.
10. **How to implement versioning in microservices?**  
    Use URL versioning like /api/v1/users.  
    Can also use headers for backward compatibility.  
    Avoid breaking old consumers when updating APIs.  
    Always document with Swagger/OpenAPI.
11. **What is sidecar pattern in microservices?**  
    Sidecar runs alongside a microservice in the same pod/container.  
    Used for adding features like logging, monitoring, proxying.  
    Common in service meshes (e.g., Istio).  
    Enhances modularity and reuse.
12. **What are challenges in Microservices?**  
    Complex deployment and orchestration.  
    Data consistency and distributed debugging.  
    Network latency and security management.  
    Requires DevOps maturity and tooling.
13. **What is the difference between Monolith and Microservices?**  
    Monolith: single unit, tightly coupled.  
    Microservices: multiple units, loosely coupled.  
    Monolith is easier to develop, harder to scale.  
    Microservices are scalable, but complex to manage.
14. **How is security handled in Microservices?**  
    Use OAuth2 and JWT for authentication.  
    Secure APIs using API Gateway and SSL.  
    Service-to-service auth with mutual TLS.  
    Centralized token validation improves consistency.
15. **What is a bounded context in Microservices?**  
    Defines a service boundary for a specific domain.  
    Each service has its own models and logic.  
    Minimizes data sharing and coupling.  
    Inspired by Domain-Driven Design (DDD).

**Multithreading – 10 Frequently Asked Questions**

1. **What is a thread in Java?**  
   A thread is the smallest unit of execution.  
   Java supports multithreading via Thread and Runnable.  
   Multiple threads run concurrently in a program.  
   Used for improving performance and responsiveness.
2. **Difference between process and thread?**  
   Process has its own memory space.  
   Thread shares memory within a process.  
   Processes are heavier than threads.  
   Thread switching is faster than process switching.
3. **What is thread safety?**  
   Ensuring shared resources are accessed by one thread at a time.  
   Use synchronized blocks or locks to avoid race conditions.  
   Stateless objects are naturally thread-safe.  
   Use Atomic, Concurrent classes for safe access.
4. **What is a race condition?**  
   Occurs when threads access shared data simultaneously.  
   Result depends on the execution order.  
   Leads to inconsistent or corrupt state.  
   Solved by using synchronization or locks.
5. **How to create threads in Java?**  
   Extend Thread class or implement Runnable interface.  
   Start using .start() method.  
   Executor framework is preferred for production use.  
   Avoid manual thread management.
6. **What is the Executor framework?**  
   Manages thread pools and task submission.  
   Use ExecutorService, ThreadPoolExecutor.  
   Better than creating threads manually.  
   Helps in scalability and resource management.
7. **What is a deadlock?**  
   Two threads wait indefinitely for each other’s resources.  
   Occurs due to incorrect locking order.  
   Use timeout or lock ordering to avoid.  
   Thread dump analysis helps detect it.
8. **What is a thread pool?**  
   A pool of reusable threads.  
   Reduces overhead of creating new threads.  
   Use Executors.newFixedThreadPool() for fixed size.  
   Helps in parallel processing.
9. **What is synchronized keyword?**  
   Used to lock critical sections of code.  
   Ensures only one thread executes it at a time.  
   Prevents race conditions.  
   Adds performance overhead.
10. **Difference between synchronized and Lock?**  
    Lock offers more flexibility than synchronized.  
    Lock allows try-lock, timed lock, and interruptible locks.  
    ReentrantLock is a popular Lock implementation.  
    Lock must be manually released.