1️⃣ Spring Boot Core Concepts

1. What is Spring Boot?

Answer:
Spring Boot is an extension of the Spring framework that simplifies building production-ready applications. It eliminates boilerplate configuration by providing auto-configuration, embedded servers, and opinionated defaults. Developers can focus on writing business logic instead of spending time setting up infrastructure. For example, a REST API can be started with just a single main class annotated with @SpringBootApplication.

2. What is auto-configuration in Spring Boot?

Answer:
Auto-configuration is a mechanism where Spring Boot automatically configures beans and components based on classpath dependencies, existing beans, and defined properties. For instance, if spring-boot-starter-web is on the classpath, Spring Boot will automatically configure an embedded Tomcat server, Jackson for JSON processing, and MVC components without writing any XML or Java config.

3. What is the purpose of @SpringBootApplication?

Answer:
@SpringBootApplication is a convenience annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. It marks the main class of a Spring Boot application, triggers auto-configuration, and scans the base package for beans and components.

4. What is a Starter Dependency?

Answer:
A starter dependency is a pre-packaged set of libraries grouped for specific functionality. For example, spring-boot-starter-data-jpa includes Spring Data JPA, Hibernate, and H2 database dependencies. Starters help avoid managing individual dependencies manually.

5. How does Spring Boot run without an external server?

Answer:
Spring Boot embeds servers like Tomcat, Jetty, or Netty directly into the application. This allows the application to run as a self-contained JAR file without requiring an external servlet container. Developers just execute the main() method, and the embedded server starts automatically.

6. How do you externalize configuration in Spring Boot?

Answer:
External configuration allows separating environment-specific settings from code. Spring Boot supports application.properties or application.yml files, system environment variables, and

command-line arguments. For example, database URLs, usernames, and passwords can be defined in application.yml for development, staging, and production environments.

7. What is Actuator in Spring Boot?

Answer:
Spring Boot Actuator provides production-ready endpoints to monitor and manage applications. It exposes endpoints such as /health (system health), /metrics (application metrics), /env (environment properties), and /beans (list of beans). These endpoints help DevOps teams monitor services and troubleshoot issues in real time.

8. Difference between application.properties and application.yml?

Answer:
Both files store configuration, but YAML (.yml) supports hierarchical structure and is more readable for nested properties. For example:

```
server:
  port: 8081
spring:
  datasource:
    url: jdbc:h2:mem:testdb
```

Whereas .properties would flatten these settings using dot notation.

9. How does Profile Management work?

Answer:
Profiles allow defining environment-specific configurations. Using spring.profiles.active, you can switch between dev, test, or prod. Spring loads the corresponding application-dev.yml or application-prod.yml file to override defaults, ensuring environment-specific behavior.

10. How do you enable asynchronous execution?

Answer:
Spring Boot supports asynchronous methods using @EnableAsync at the configuration class and @Async on methods. When a method is annotated with @Async, Spring executes it in a separate thread from a configured thread pool, allowing non-blocking operations like sending emails or processing files.

11. How to create a custom starter?

Answer:
A custom starter packages reusable code and auto-configuration into a library. It contains a dependency module and an @Configuration class registered in META-INF/spring.factories or AutoConfiguration.imports. Other projects can include it as a dependency to get pre-configured beans automatically.

12. What is Spring Boot CLI?

Answer:
The Spring Boot CLI is a command-line tool for rapid application development. It allows you to write Groovy scripts with Spring Boot annotations, automatically resolving dependencies and running applications without explicit project setup. Ideal for prototyping.

13. What is DevTools?

Answer:
DevTools provides features like automatic restart, live reload, and configuration for development. When code changes, Spring reloads only application classes (not the whole framework) for faster feedback during development.

14. What is the default logging framework?

Answer:
Spring Boot uses **Logback** as the default logging framework. You can configure logging levels in application.properties or use external log frameworks like Log4j2 by excluding the default Logback dependency.

15. How to secure Actuator endpoints?

Answer:
By default, Spring Boot exposes Actuator endpoints publicly. You can secure them with Spring Security by restricting access to specific roles, using HTTP basic authentication, or enabling user-defined credentials in application.properties.

16. What is the default port in Spring Boot?

Answer:
The default port is 8080. You can change it using server.port=9090 in application.properties or application.yml.

17. How to change the default port?

Answer:
Set the property server.port to the desired value. For example:

server.port=9090
This will start the application on port 9090 instead of 8080.

18. What are Banner customizations?

Answer:
You can customize the Spring Boot startup banner by creating a banner.txt file in src/main/

resources. This can display ASCII art, application info, or version details when the application starts.

19. How to disable auto-configuration?

Answer:
Use @SpringBootApplication(exclude = {...}) to exclude specific auto-configuration classes. For example, to disable DataSource auto-configuration:

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
```

20. What is CommandLineRunner?

Answer:
CommandLineRunner is an interface that allows executing code after the Spring Boot application context is initialized. It's commonly used for startup tasks like seeding databases or initializing caches. Example:

```
@Bean
public CommandLineRunner init(UserRepository repo) {
   return args -> {
      repo.save(new User("John"));
   };
}
```

2️⃣ Spring Bean Lifecycle

1. What is a Spring Bean?

Answer:
A Spring Bean is an object instantiated, assembled, and managed by the Spring IoC container. The container handles the bean's lifecycle, including creation, dependency injection, initialization, and destruction.

2. Explain Spring Bean lifecycle.

Answer:
Bean lifecycle starts with instantiation, followed by property injection. Spring calls Aware interfaces (e.g., BeanNameAware) if implemented. Then @PostConstruct or afterPropertiesSet() executes for initialization. Finally, the bean is ready for use. On container shutdown, destroy methods or @PreDestroy are called.

3. What is @PostConstruct?

Answer:
@PostConstruct is a JSR-250 annotation called after all bean properties are set and dependencies injected. It is ideal for initialization logic like opening a database connection.

4. What is InitializingBean?

Answer:
InitializingBean is a Spring interface with afterPropertiesSet() method. It allows beans to perform initialization after dependency injection. It's Spring-specific and an alternative to @PostConstruct.

5. What is DisposableBean?

Answer:
DisposableBean is a Spring interface with a destroy() method, executed before bean destruction. Useful for releasing resources, closing connections, or cleanup tasks.

6. What are Bean Scopes?

Answer:
Bean scope defines how many instances of a bean exist and its lifecycle. Common scopes: Singleton, Prototype, Request, Session, and Application. Singleton is default; one instance per container.

7. What is Singleton scope?

Answer:
Only one bean instance is created per Spring container and reused wherever it is injected.

8. What is Prototype scope?

Answer:
A new bean instance is created each time it is requested. The container doesn't manage the destruction of prototype beans.

9. How to define custom init/destroy methods?

Answer:
You can define methods and specify them in @Bean(initMethod="init", destroyMethod="cleanup"). These methods will be called after initialization and before destruction.

10. Difference between @PostConstruct and init method?

Answer:
@PostConstruct is part of JSR-250 (standard), while init-method is Spring-specific. Both run after bean creation, but @PostConstruct works even outside Spring XML config.

11. Lifecycle callback sequence?

Answer:
Constructor → Setter injection → BeanNameAware → BeanFactoryAware → PostConstruct →

InitializingBean.afterPropertiesSet() → custom init method → Ready → container shutdown → custom destroy → DisposableBean.destroy() → @PreDestroy.

12. What is BeanFactoryPostProcessor?

Answer:
It allows modifying bean definitions before bean instantiation. Useful for changing properties programmatically before beans are created.

13. What is BeanPostProcessor?

Answer:
It allows intercepting bean initialization, executing logic before and after a bean is initialized. Commonly used for logging, proxy creation, or validation.

14. What is Lazy Initialization?

Answer:
Bean is created only when requested, not at container startup. Improves startup time for large applications.

15. What is Eager Initialization?

Answer:
Bean is created at container startup. Default behavior for singleton beans, ensuring readiness immediately.

16. What is @DependsOn?

Answer:
@DependsOn("beanName") ensures that a specific bean is created before the current bean.

17. How to disable bean overriding?

Answer:
Set spring.main.allow-bean-definition-overriding=false in application.properties. Spring will throw an exception if two beans have the same name.

18. How are Prototype beans destroyed?

Answer:
Spring does not manage destruction of prototype beans; you must destroy them manually if resources need cleanup.

19. How do you debug bean lifecycle?

Answer:
Enable Spring debug logs or implement a custom BeanPostProcessor to track initialization and destruction of each bean.

20. Difference between Singleton and Prototype in lifecycle management?

Answer:
Singleton: created at container startup, container manages full lifecycle.
Prototype: created on demand, container manages creation but not destruction.

I can now **continue with the next topics**:

- Spring AOP
- Spring Security

3️⃣ SPRING AOP

1. What is Spring AOP and why do we use it?

Spring AOP (Aspect-Oriented Programming) is a technique for separating cross-cutting concerns—functionalities that appear across multiple modules but do not belong to any one component. Common examples include logging, security checks, exception tracking, auditing, and transaction handling.

Instead of repeating the same code in multiple services or controllers, AOP allows you to write that behavior once and apply it wherever needed, keeping the code cleaner, modular, and easier to maintain.

2. What is a cross-cutting concern with a real example?

A cross-cutting concern is logic that affects multiple parts of the application.
Example: Logging every request to a microservice.

Without AOP:
You'd add logger.info() everywhere → messy and repetitive.

With AOP:
One logging aspect → automatically logs all controller methods.
This separates business logic from system-level concerns.

3. What is an Aspect?

An Aspect is a class that contains cross-cutting logic.
For example, a LoggingAspect class can contain methods that run before or after your business methods.

```
@Aspect
@Component
public class LoggingAspect {
   @Before("execution(* com.demo.service.*.*(..))")
   public void logBefore() {
      System.out.println("Executing method...");
   }
}
```

4. What is a Join Point?

A Join Point is any point in application execution where you can apply AOP logic — typically method execution.

For example, every method inside UserService is a join point.

5. What is a Pointcut?

A Pointcut defines where an aspect should run.
It uses expressions to select specific methods.

Example:

execution(* com.demo.service.*.*(..))
This targets all methods inside service package.

6. What is Advice in AOP?

Advice defines what should run at the join point.
Types of advice include:

- Before
- After
- AfterReturning
- AfterThrowing
- Around

Each type serves a different purpose (logging, measuring time, error handling, etc.).

7. Explain each type of Advice with real-world use case.

Before Advice

Runs before a method executes.
Use case: Validate API key before processing request.

After Advice

Runs after method execution, regardless of success/fail.
Use case: Logging that a method finished.

AfterReturning

Runs only when method returns successfully.
Use case: Capture a response payload for auditing.

AfterThrowing

Runs when method throws exception.
Use case: Send alert to monitoring system (e.g., Sentry).

Around

Wraps the method execution.
Use case: Time taken by a method (performance profiling).

8. What is weaving?

Weaving is the process of linking aspects to target objects.

Spring uses runtime weaving through proxies, meaning aspects are applied dynamically at runtime (not at compile-time).

9. How does Spring AOP work internally?

Spring AOP uses dynamic proxies:

- If the target implements an interface → JDK dynamic proxy
- If not → CGLIB proxy (creates subclass)

When a method is called:

1. The call goes to proxy.
2. Proxy checks if any pointcut matches.
3. If yes, relevant advice runs.
4. Then actual business method runs.

10. Difference between Spring AOP and AspectJ?

Feature	Spring AOP	AspectJ

Weaving	Runtime	Compile-time / load-time
Performance	Moderate	Faster
Features	Limited	Full AOP support
Complexity	Simple	Complex

Spring AOP is widely used because it's simpler and fits 95% use cases.

11. Why AOP is useful in Microservices?

In microservices, AOP helps with:
✔ Logging incoming/outgoing API calls
✔ Distributed tracing (Zipkin/Jaeger)
✔ Security validations
✔ Rate-limiting
✔ Transaction boundaries

Without AOP, these concerns would clutter business logic.

12. Can we apply AOP on private methods?

No.
Spring AOP works using proxies, and proxies cannot intercept private or final methods.

13. What problems occur when using AOP incorrectly?

- Pointcuts becoming too broad → accidental behavior
- Hard-to-debug issues due to hidden method interception
- Performance overhead if overused
- Forgetting to annotate @EnableAspectJAutoProxy

14. How to enable AOP in Spring Boot?

Simply annotate a configuration class with:

@EnableAspectJAutoProxy
or rely on auto-configuration (Spring Boot does it automatically in most cases).

15. How do you log execution time using AOP?

Use Around Advice:

@Around("execution(* com.demo..*(..))")

```
public Object trackTime(ProceedingJoinPoint pjp) throws Throwable {
   long start = System.currentTimeMillis();
   Object obj = pjp.proceed();
   System.out.println("Time: " + (System.currentTimeMillis() - start));
   return obj;
}
```

16. How do you handle exceptions using AOP?

Use AfterThrowing advice:

```
@AfterThrowing(pointcut="execution(* com.demo.*.*(..))", throwing="ex")
public void logError(Exception ex) {
   System.out.println("Error: " + ex.getMessage());
}
```

17. Can AOP modify method return values?

Yes — using Around advice.
You can intercept the response, modify it, and return the updated value.

18. How does proxy creation impact performance?

Proxies add a small overhead because method calls go through an interceptor chain.
This is usually negligible but can be noticeable if AOP is used for highly frequent operations (e.g., millions of calls).

19. Can AOP work on final classes?

CGLIB proxy cannot subclass a final class — so NO.

20. Real-world scenario where you used AOP?

Example answer:
"In one of my microservices, we needed unified logging for every incoming request, along with request ID tracking. Instead of adding logging in every controller, I created a LoggingAspect that intercepted all controller methods, extracted request metadata, logged execution time, and passed control to the actual method. This reduced repetitive code and improved maintainability."

4️⃣ SPRING SECURITY

1. What is Spring Security?

Spring Security is a framework that provides authentication, authorization, CSRF protection, password hashing, session management, and other security features.

Its biggest strength is that it integrates deeply into the Spring ecosystem with very minimal configuration required.

2. What are Authentication and Authorization?

Authentication → "Who are you?"

Example: Logging in with username/password.

Authorization → "What are you allowed to do?"

Example: Admins can delete users, normal users cannot.

Spring Security handles both using filters, providers, and configuration classes.

3. How does Spring Security work internally?

When a request comes in:

1. The request enters the Security Filter Chain.
2. Filters such as UsernamePasswordAuthenticationFilter intercept the request.
3. The user details are passed to AuthenticationManager.
4. Manager uses configured AuthenticationProvider (e.g., DaoAuthenticationProvider).
5. Provider loads user from database using UserDetailsService.
6. Password comparison happens using PasswordEncoder.
7. If correct → Spring stores authentication in SecurityContext.
8. The request proceeds to the controller.

This all happens transparently.

4. What is the Security Filter Chain?

A set of around 15 built-in filters that intercept requests for security-related checks.

Examples:

- CSRF filter
- Authentication filter
- Authorization filter
- Session management filter

- Exception translation filter

Each filter plays a specific role.

5. What is UserDetailsService?

It's the interface used by Spring Security to load user information from DB or any other source.
You simply implement loadUserByUsername() and return user details.

```
@Override
public UserDetails loadUserByUsername(String username) {
   User user = userRepo.findByUsername(username);
   return new org.springframework.security.core.userdetails.User(
      user.getUsername(), user.getPassword(), authorities
   );
}
```

6. Why do we need PasswordEncoder?

To ensure passwords aren't stored in plain text.
Spring Security never compares passwords directly — it always hashes and verifies.

Common encoders:

- BCryptPasswordEncoder
- PBKDF2
- Argon2

BCrypt is the industry standard.

7. What is CSRF Protection?

CSRF is an attack where someone tricks a user into submitting a malicious request.
Spring Security prevents this by adding a secret token to every request.

Important:
For REST APIs, CSRF is often disabled as APIs are usually stateless and rely on JWT or OAuth.

8. What is the difference between Stateless and Stateful Authentication?

Stateful

Server stores session. (Session cookies)
Example: Traditional web apps.

Stateless

Server stores nothing. (JWT tokens)
Example: Microservices & REST APIs.

Spring Security supports both.

9. How to secure APIs using JWT?

Typical steps:

1. User sends username/password.
2. Server validates and returns JWT token.
3. Client sends token in Authorization header.
4. A JWT filter validates the token on every request.
5. If valid, the SecurityContext is populated.
6. Request proceeds.

JWT is preferred in distributed systems.

10. What is OAuth2 and OIDC?

OAuth2 provides delegated authorization (e.g., login with Google, Facebook).
OIDC adds identity layer on top of OAuth2 (user info, profile, email).

Spring Security supports both with minimal configuration.

11. How to create a custom login page?

You override the configuration:

```
.formLogin()
  .loginPage("/login")
  .permitAll();
```

Spring Security then uses your custom view.

12. How do you handle Role-Based Authorization?

Use hasRole('ADMIN') or hasAuthority('ROLE_ADMIN').

Example:

```
@PreAuthorize("hasRole('ADMIN')")
public void deleteUser() {}
```

13. What is Method-Level Security?

Instead of securing endpoints at HTTP level, you secure methods using:

- @PreAuthorize
- @PostAuthorize
- @Secured

Example:

```
@PreAuthorize("hasAuthority('UPDATE_USER')")
public void updateUser() {}
```

14. What is SecurityContext?

It holds authentication details of the current user.
Stored in a ThreadLocal so controllers and services can access it.

15. What is the difference between WebSecurityConfigurerAdapter and new Spring Security approach?

Spring 5.7 deprecated the old WebSecurityConfigurerAdapter approach.
Now we use bean-based lambda configuration:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   return http
       .authorizeHttpRequests(auth -> auth.anyRequest().authenticated())
       .build();
}
```

This is cleaner and more readable.

16. What is ExceptionTranslationFilter?

If authentication or authorization fails:

- Converts exceptions into proper HTTP responses (403 / 401)
- Redirects users to login page if needed

It ensures user-friendly error handling.

17. How do you disable security for specific URLs?

By configuring permitAll for specific endpoints:

```
.authorizeHttpRequests(req ->
    req.requestMatchers("/public/**").permitAll()
)
```

18. How does Spring Security handle sessions?

Supports:

- ALWAYS (create a session every time)
- IF_REQUIRED
- NEVER
- STATELESS

You can configure it easily:

```
.sessionManagement()
  .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
```

19. What are Granted Authorities?

They represent permissions assigned to a user.
Example:

- READ_USER
- WRITE_USER
- DELETE_USER

Roles (like ADMIN) are higher-level wrappers around authorities.

20. Mention a real-world scenario where you used Spring Security.

Example answer:
“In our payment microservice, we implemented JWT-based authentication using Spring Security. Each request went through a JWT filter that validated the token, extracted user roles, and populated the SecurityContext. Additionally, we used method-level security to restrict high-risk operations like initiating refunds. This setup ensured our APIs were stateless, scalable, and secure.”

5️⃣ SPRING DATA JPA

1. What is Spring Data JPA?

Answer:
Spring Data JPA is a high-level abstraction over JPA/Hibernate that allows developers to interact with the database without writing boilerplate code. Instead of building DAO classes manually, Spring Data JPA provides repository interfaces where CRUD methods, pagination, sorting, and even dynamic query creation are available out of the box.

It dramatically reduces development time and ensures clean, readable, and maintainable data-access code.

Internal Working:

1. Spring scans your JpaRepository/CrudRepository interfaces at application startup.
2. It creates a proxy object for each repository.
3. When you call repository methods (like findAll()), the proxy delegates the call to the underlying EntityManager.
4. EntityManager uses Hibernate to generate SQL queries.
5. Hibernate maps results to Java objects (entities).

Real-World Use Case:

Any enterprise application like e-commerce, banking, payments, or healthcare systems use JPA for storing users, orders, transactions, logs, etc.

2. What is an Entity in JPA?

Answer:
An entity is a Java class that maps to a table in the relational database. Every field inside the class becomes a column, and every object instance becomes a row.

Entities are the cornerstone of ORM (Object Relational Mapping).
They help developers work with objects instead of writing SQL manually.

Internal Working:

- Hibernate scans classes annotated with @Entity.
- It creates metadata: table name, columns, primary keys.
- Hibernate uses reflection to read/write fields on these entity objects.

Real-World Use Case:

Class User → Table users
Class Order → Table orders

3. What is the importance of @Id and @GeneratedValue?

Answer:
@Id defines the primary key of an entity, ensuring every row is uniquely identifiable.
@GeneratedValue automatically generates the ID using strategies like:

- IDENTITY → Auto-increment
- SEQUENCE → Database sequence
- AUTO → Hibernate chooses the best strategy

This allows you to avoid manually assigning IDs.

Internal Working:

- Before saving a record, Hibernate generates the ID.
- It inserts it into the SQL INSERT statement.
- Database returns the generated key if needed.

Real Use Case:

Every database table requires a unique key; JPA simplifies its generation.

4. CRUDRepository vs JpaRepository vs PagingAndSortingRepository

Answer:
All three are part of Spring Data but differ in features:

CrudRepository

Basic CRUD operations only.
Good for simple data access.

PagingAndSortingRepository

Adds pagination and sorting functionality.
Useful when displaying lists (products, users).

JpaRepository

Extends both above, plus:

- Batch operations
- Flush
- Advanced JPA features
- Better default implementations

Internal Working:

Each repository builds on the other using interface inheritance.
Spring Data generates implementations using proxies.

5. How does Spring Data JPA derive queries from method names?

Answer (Elaborated):
Spring Data JPA can automatically generate SQL queries based on your repository method names,

without writing any query manually. It does this by parsing the method name and identifying keywords like:

- find
- count
- exists
- delete
- By
- And / Or
- LessThan / GreaterThan
- Like / Containing
- Between
- OrderBy

For example:

findByEmail(String email)
Spring understands this as:

“Fetch the record where email = given value”

Internal Working:

- Method name is analyzed using a naming parser.
- The parser converts it into JPQL.
- JPQL is converted to SQL by Hibernate.
- SQL is executed against DB.
- Hibernate maps results to entities.

Real Use Case:

Search users by: email, phone, status, age range, name containing specific text.

6. What is the @Query annotation and why do we need it?

Answer:
While derived queries work for simple cases, complex queries often require custom logic.
@Query allows writing:

- JPQL queries

- Native SQL queries
- Named parameters
- Complex joins

Example:

```
@Query("SELECT u FROM User u WHERE u.status = :status")
List<User> findActiveUsers(String status);
```

Internal Working:

- Spring reads the annotation at runtime.
- Converts JPQL to SQL (unless native query).
- Hibernate executes the SQL.

Real Use Case:

Reports, analytics, multi-table joins, aggregation functions.

7. Difference between JPQL and SQL

Answer:
JPQL works with Java entity names and fields.
SQL works with table and column names.

Example JPQL:
SELECT u FROM User u

Example SQL:
SELECT * FROM users

Use Case:

JPQL is used for ORM-based logic.
SQL is used when database-specific features are required.

8. What is EntityManager?

Answer:
The EntityManager is the core API of JPA.
It manages:

- Persistence
- Updates
- Deletes
- Query execution

- Transactions

Spring Data hides EntityManager behind repositories, making work easier.

Internal Working:

EntityManager interacts with Hibernate's Session API internally.

9. Explain the N+1 query problem.

Answer:
N+1 happens when:

- 1 query loads a list of parent records
- N additional queries load each child record

This leads to bad performance.

Internal Working:

Caused by default LAZY loading in child entities.

Solution:

Use JOIN FETCH or DTO projections.

10. What is Lazy Loading vs Eager Loading?

Lazy: Loads child objects only when accessed.
Eager: Loads child objects immediately.

Real Use Case:

Lazy is recommended for performance, except when relationship is always needed.

11. What is @Transactional? Why is it important?

Answer:
@Transactional ensures that a block of code executes in a single transaction.
If any operation fails → entire transaction rolls back.

Internal Working:

Spring wraps the method in a proxy and manages commit/rollback.

12. Difference between save() and saveAndFlush()

save() → does not push immediately to DB; stored in persistence context
saveAndFlush() → forces immediate SQL execution

Useful when you need ID instantly.

13. What is cascading in JPA?

Defines how operations on parent entities affect children:

- CascadeType.ALL
- PERSIST
- REMOVE
- MERGE
- REFRESH

Real Use Case:

Saving an order should automatically save order items.

14. Purpose of @OneToOne, @ManyToOne, @OneToMany, @ManyToMany

Defines entity relationships.
Used to map table relationships directly in Java.

15. What is a JPA proxy?

When fetching entities lazily, Hibernate creates a proxy object instead of the real object.
Actual data is fetched only when a getter is called.

Understood. I will **rewrite from Q14 onward** with the **same detailed, human-explainable, interview-ready style** I used for the earlier questions (1–13).
Topic: **Spring Data JPA**
Starting from **Q14** with fully elaborated answers.

16. What is the difference between findById() and getById() in Spring Data JPA?

findById() immediately hits the database and returns an **Optional**, meaning it safely handles cases where the entity may not exist.

getById() (earlier getOne()) **does not hit the DB immediately**.
Instead, it returns a **lazy-loading proxy object**, and the actual DB call is triggered **only when a property of that object is accessed**.

This means:

- If you call getById() outside a transaction → accessing data triggers **LazyInitializationException**.
- findById() is safe, eager, and preferred in REST layers.

Internal Working:

- findById() → calls EntityManager's **find()** → immediate SQL SELECT.
- getById() → calls EntityManager's **getReference()** → returns a proxy (Hibernate uses ByteBuddy/CGLIB).

Real-world Use Case:

Use getById() when:

- You only need a reference to set in another entity (e.g., setting foreign keys).
  Use findById() when:
- You need *actual data* (like sending DTO in API).

17. What is the N+1 query problem? How do you solve it in JPA?

N+1 happens when a parent entity is fetched in **1 query**, but its child entities are fetched lazily in **N additional queries**.

Example:
Fetching 100 users, and for each user fetching its orders → results in **101 queries**.
This slows down performance drastically.

Why It Happens?

Collections (like @OneToMany) default to **LAZY**.
When you access the child collection, Hibernate fires another query.

Solutions:

1. **Fetch Join**

```
@Query("SELECT u FROM User u JOIN FETCH u.orders")
List<User> findUsersWithOrders();
```

2. **Entity Graphs**

```
@EntityGraph(attributePaths = {"orders"})
List<User> findAll();
```

3. **Batch Size**

```
@BatchSize(size = 20)
```

Real-world Use Case:

When building dashboards that load parent-child data (Users + Orders), N+1 will kill performance unless fetch joins/entity graphs are used.

18. What are the advantages of using Spring Data JPA over plain Hibernate?

Spring Data JPA **reduces 70–80% of boilerplate** needed in Hibernate.

Key benefits:

- No need to write DAO classes manually.
- Repository interfaces generate queries using **method names**, reducing effort.
- Built-in paging, sorting, auditing, specification support.
- Transaction management + exception translation automatically handled.

It feels like JPA on steroids—faster development, cleaner code, fewer bugs.

Internal Working:

Spring creates **proxy implementations** of your repository interface at runtime using JDK dynamic proxies and delegates calls to JPA's EntityManager.

Real-World Use Case:

Enterprise apps with 100+ entities benefit massively—repository layer becomes extremely compact.

19. How does Spring Data JPA handle transactions?

Spring Data JPA uses **@Transactional** annotation under the hood for repository methods.

- **Read operations** → default **read-only transactions**, optimized for performance.
- **Save/Delete operations** → wrapped in **write transactions** that commit changes automatically.

If something fails, Spring triggers **rollback()** so your DB remains consistent.

Internal Working:

Spring AOP wraps the method call with a transactional proxy using platform TransactionManager (Hibernate/JPA).

Real-world Use Case:

When updating multiple tables in one service method → a single transactional boundary ensures **all-or-nothing** behavior.

20. How does Hibernate's first-level cache work in JPA?

First-level cache is **Session-level**, meaning:
During one Hibernate session (usually 1 request), Hibernate stores all fetched entities in memory.

So if you call:

userRepo.findById(1);
userRepo.findById(1);
Only **one SQL query** is executed. The second call returns the cached object.

Internal Working:

Hibernate keeps an in-memory map of:

<EntityType + PrimaryKey, EntityInstance>
Real-world Use Case:

Saving repeated DB calls for frequently accessed entities inside the same transaction.

21. What is Dirty Checking in JPA?

Dirty checking means Hibernate automatically tracks changes to managed entities and updates the DB **only when needed**.

Example:

User u = repo.findById(1).get();
u.setName("John");
You never call **update** manually.
Hibernate detects the change at **transaction commit** and performs:

UPDATE user SET name='John' WHERE id=1
Internal Working:

Hibernate maintains a “snapshot” of the entity and compares it with the current state during flush.

Real-world Use Case:

Makes service layer cleaner—no need for explicit update statements.

22. What is Cascading in JPA? Why is it important?

Cascade means performing operations on a parent entity will automatically apply to its child entities.

Example:

CascadeType.ALL
If you save/delete a User → Orders linked with that user are also saved/deleted.

Internal Working:

Hibernate propagates operations down the entity graph before flush.

Real-world Use Case:

Useful when inserting an order with its items in a single call, avoiding multiple manual saves.

23.Difference between @OneToOne, @OneToMany, and ManyToMany

These define **relationship cardinality**:

- **OneToOne** → One entity mapped to exactly one other entity.
- **OneToMany** → One parent can have multiple child entities.
- **ManyToMany** → Both sides have multiple references (requires join table).

Internal Working:

Hibernate decides how to place FK columns or create join tables depending on annotation.

Real-world Use Case:

OneToMany for **User → Orders**,
ManyToMany for **Student ↔ Course** relationship.

24. What is the purpose of @Modifying and @Query together?

@Query allows writing custom JPQL/SQL.

When the query changes data (DELETE/UPDATE), we add **@Modifying** so Spring knows it must execute an update instead of a select.

@Modifying
@Query("UPDATE User u SET u.active=false WHERE u.id=:id")
void deactivate(int id);
Without @Modifying, Spring will throw an error because it expects the query to return a result.

Real-world Use Case:

Bulk updates that are faster than fetching entities and modifying them.

25. Why do we use DTOs instead of returning Entities?

Returning entities can cause:

- Lazy loading exceptions
- Exposing internal DB structure to clients
- Infinite recursion in JSON (bi-directional mappings)
- Security issues (fields you don't want to expose)

DTOs solve these problems by returning only the required fields.

Real-world Use Case:

REST APIs should **never** expose full entity objects.

26. What is @EntityGraph and why is it used?

@EntityGraph defines which attributes should be eagerly fetched to avoid **N+1 problem**.

Example:

```
@EntityGraph(attributePaths = {"orders"})
List<User> findAll();
```

This tells Hibernate:

Load users AND their orders in one SQL query.

Real-world Use Case:

Fetching parent-child relationships in dashboards.

27. What is optimistic locking in JPA?

Optimistic locking uses a @Version field to ensure **no two users overwrite each other's changes**.

If two users try to update the same record, Hibernate throws:

OptimisticLockException
Real-world Use Case:

Banking transactions, ticket booking, inventory updates.

28.How does save() behave for new vs existing entities?

Spring Data JPA uses save() for both **INSERT and UPDATE**:

- If id == null → performs INSERT
- If id != null → performs UPDATE

Internal Working:

Delegates to JPA EntityManager's merge().

29. What is flush() in JPA?

flush() forces Hibernate to synchronize entity state with the DB immediately.

Hibernate normally flushes automatically:

- Before commit
- Before executing a query

Real-world Use Case:

Used when you want DB constraints to validate data immediately.

30. What is the purpose of @Transactional(readOnly=true)?

Optimizes read operations:

- Hibernate skips dirty checking
- Database uses optimized read-only transactions
- Prevents accidental writes

31. Why is LAZY loading recommended over EAGER?

EAGER looks simple but is dangerous:

- Loads too much data
- Causes huge SQL queries
- Slows performance
- Often leads to N+1 problems

LAZY gives full control—you load only what you use.

32. What are specifications in Spring Data JPA?

Specifications allow building **dynamic queries** using a fluent API.

Good for:

- Advanced search filters
- Multiple optional parameters

Example:

```
Specification<User> spec = UserSpecs.hasName(name)
        .and(UserSpecs.hasStatus(status));
```

6️⃣ Spring Cloud

1. What is Spring Cloud and why do we use it?

Answer:
Spring Cloud is a framework built on top of Spring Boot that simplifies building distributed systems. In microservices environments, we face common challenges such as configuration management, service discovery, load balancing, API routing, and fault tolerance. Spring Cloud provides ready-made components for these problems so developers don't have to build them manually.

Internal Working:
Spring Cloud integrates with tools like Eureka, Config Server, Gateway, Feign, and Resilience4j. It uses Spring Boot's auto-configuration to register these components at runtime, making distributed system features available with minimal code.

Real-world Use Case:
A company running 20 microservices uses Spring Cloud to centralize configs, handle routing, perform circuit breaking, gather logs, and enable inter-service communication efficiently.

2. What is the role of Spring Cloud Config Server?

Answer:
Spring Cloud Config Server externalizes configuration for all microservices. Instead of storing properties within each service, we keep them in a central Git repository. This ensures consistency across environments and allows configuration changes without redeploying services.

Internal Working:

- Config Server reads values from Git or local files.
- Config Client services request configuration during startup.
- Changes propagate using Spring Cloud Bus with RabbitMQ/Kafka.
- /actuator/refresh reloads configuration dynamically.

Real-world Use Case:
Updating database URLs, enabling a new feature flag, or modifying logging levels across multiple microservices without restarting applications.

3. What is Service Discovery, and how does Eureka work?

Answer:
In microservices, service instances scale up and down dynamically. Hardcoding URLs is not feasible. Eureka provides service discovery where each service registers itself, and other services find it via Eureka instead of knowing its IP or port.

Internal Working:

- Each service registers itself with Eureka using heartbeats.

- Eureka maintains a registry of all active services.
- Clients query Eureka before making service-to-service calls.
- Ribbon or Spring LoadBalancer uses this registry for load balancing.

Real-world Use Case:
An Order service needs to call Payment service. It asks Eureka for available Payment instances and selects one dynamically.

4. What is the difference between Client-Side and Server-Side Load Balancing?

Answer:
Client-side load balancing: the client calls the service directly and chooses the instance using a load balancing algorithm.
Server-side load balancing: a central load balancer like Nginx or AWS ELB sits in front of services and distributes requests.

Internal Working:
Spring Cloud LoadBalancer fetches instance details from Eureka and applies strategies like round-robin or random selection.

Real-world Use Case:
Internal microservice-to-microservice communication typically uses client-side load balancing.

5. What is Spring Cloud Gateway?

Answer:
Spring Cloud Gateway is an API Gateway responsible for routing external requests to the appropriate microservices. It also supports authentication, rate limiting, logging, and filter-based processing.

Internal Working:

- Built on Spring WebFlux and Netty for non-blocking routing.
- Routes are defined using predicates and filters.
- Gateway communicates with Eureka to discover service locations.

Real-world Use Case:
All client requests (mobile/web) first reach Gateway, which routes them to User, Order, Payment, or Inventory services.

6. What is Feign Client and how does it work?

Answer:
Feign is a declarative REST client. Instead of writing RestTemplate code, you create a simple interface and Feign generates the HTTP communication code automatically.

Internal Working:

- Spring creates a dynamic proxy for the interface.
- Proxy performs HTTP calls behind the scenes.
- Integrates with Eureka for service discovery.
- Works with Resilience4j for retries and circuit breaking.

Real-world Use Case:
The Order service calls Payment service using a simple Java interface annotated with @FeignClient.

7. What is a Circuit Breaker and why do we need it?

Answer:
A circuit breaker prevents one failing service from crashing or slowing down other services. If a service repeatedly fails, the circuit opens and stops further calls, returning a fallback response.

Internal Working:
Resilience4j provides three states:

- Closed (normal)
- Open (service failing, block requests)
- Half-open (test if service is back)

Real-world Use Case:
If Payment service is down, Order service immediately returns a fallback message instead of waiting for timeouts.

8. What is Resilience4j and how is it different from Hystrix?

Answer:
Hystrix is deprecated. Resilience4j is modern, lightweight, modular, built for Java 8+, and integrates well with Spring Boot.

It supports:

- Circuit Breaker
- Retry
- Rate Limiter
- Bulkhead
- Timeouts

Real-world Use Case:
Retry failed external API calls, prevent service outages, and add fault tolerance automatically.

9. What is the difference between API Gateway and Service Discovery?

Answer:
Service discovery is for internal service-to-service communication.
API Gateway is for external client-to-service communication.

Gateway handles routing, filtering, and security.
Discovery handles service instance registration and lookup.

10. What is Spring Cloud Bus and what problem does it solve?

Answer:
Spring Cloud Bus links distributed services with a lightweight message broker (Kafka/RabbitMQ) and helps broadcast configuration changes to all microservices.

Internal Working:
When Config Server configuration changes, it publishes a refresh event to the message bus. All services receive the event and refresh automatically.

Real-world Use Case:
Dynamic refresh of 50 microservices after updating Git configuration.

11. What is Distributed Tracing? How do Sleuth and Zipkin work together?

Answer:
Distributed tracing tracks a request across multiple microservices. Sleuth adds trace IDs in logs, and Zipkin stores and visualizes traces.

Internal Working:

- Sleuth intercepts HTTP calls and adds traceId and spanId.
- Zipkin receives spans and builds a trace timeline.

Real-world Use Case:
Tracing performance issues or debugging failures across a chain of microservices.

12. What is Centralized Logging and why is it needed?

Answer:
In a microservice architecture, logs are spread across multiple services and servers. Centralized logging aggregates them in one place (ELK Stack or Loki/Grafana), making debugging easier.

Real-world Use Case:
Investigating a production error by searching logs by traceId across services.

13. What is the difference between Zipkin and ELK?

Answer:
Zipkin → Used for distributed tracing (tracks request flow).
ELK (Elasticsearch, Logstash, Kibana) → Used for centralized logging, searching, and analytics.

They solve different problems but complement each other.

14. What is Spring Cloud LoadBalancer?

Answer:
Spring Cloud LoadBalancer is the modern client-side load balancer that replaced Ribbon. It selects an appropriate service instance from the service registry.

Internal Working:
Uses ServiceInstanceListSupplier to fetch instances and apply load balancing algorithms.

Real-world Use Case:
Distribute load across multiple instances of Payment service.

15. What is Spring Cloud Stream?

Answer:
Spring Cloud Stream makes it easy to build message-driven microservices using Kafka or RabbitMQ. It uses binders to abstract the messaging platform.

Internal Working:

- Defines input and output channels.
- Binds channels to topics using a binder (Kafka/RabbitMQ).
- Handles serialization, partitioning, retries.

Real-world Use Case:
Building event-driven systems like order-placed → payment-initiated → notification-sent.

16. What is Spring Cloud Sleuth?

Answer:
Sleuth provides unique trace IDs for each request and attaches them to logs. It helps correlate logs across multiple services for debugging.

Real-world Use Case:
Tracing a user's order journey across User, Order, Inventory, and Payment services.

17. What is Rate Limiting and how is it applied in Spring Cloud?

Answer:
Rate limiting restricts the number of requests allowed in a time window. This prevents abuse, ensures fair usage, and protects downstream services.

Internal Working:
Implemented using Gateway filters, Bucket4j, or Resilience4j RateLimiter module.

Real-world Use Case:
Preventing a client from hitting an API 1000 times per second.

18. What is the Bulkhead Pattern?

Answer:
The bulkhead pattern isolates resources between components so that a failure in one component does not consume all threads or memory.

Internal Working:
Resilience4j provides thread pool and semaphore-based bulkheads.

Real-world Use Case:
If Inventory service is slow, Order service should not freeze; only the thread pool dedicated to Inventory calls is affected.

19. What is the difference between Local Configuration and Distributed Configuration?

Answer:
Local configuration: stored inside each application. Requires redeploying the service after changes.

Distributed configuration: stored in Config Server. Allows dynamic updates without restarts and ensures consistency.

Real-world Use Case:
Updating payment gateway credentials across environments without restarting services.

20. What is the difference between API Gateway and Load Balancer?

Answer:
Load Balancer distributes incoming traffic among service instances.
API Gateway performs request routing, authentication, rate limiting, header manipulation, and more.

Gateway = Advanced routing + security

Microservices

1. What are Microservices?

Microservices is an architectural style where an application is broken into **small, independent services**, each responsible for a single business capability.
Instead of one large monolithic block, you build multiple loosely-coupled services that can be **developed, deployed, and scaled independently**.

Why companies use it: Faster deployments, better scalability, independent tech stack choices, and improved fault isolation.

2. How do Microservices communicate with each other?

Microservices typically communicate through **lightweight protocols**.
The most common approaches are:

- **Synchronous** → REST APIs, gRPC
- **Asynchronous** → Kafka, RabbitMQ, event streams

Interviews expect clarity:
Synchronous = immediate response; tightly dependent
Asynchronous = event-driven; loosely coupled; resilient

Companies prefer **async messaging** for high scalability.

3. What are the main characteristics of Microservices?

Key characteristics include:

1. **Single Responsibility –** each service handles one business domain.
2. **Decentralized Governance –** teams choose their own tech stack.
3. **Independent Deployment –** each service deploys without affecting others.
4. **Resilience –** one service failing must not take down the system.
5. **Automated CI/CD –** microservices demand automation.
6. **Observability –** logs, traces, metrics are mandatory.

These characteristics make microservices scalable and maintainable.

4. What is the difference between Monolithic Architecture and Microservices?

Monolith:

- Single codebase
- Shared database
- Single deployment
- Tight coupling

Microservices:

- Multiple small services

- Decentralized data
- Individual deployments
- Loose coupling

Example:
E-commerce monolith = single app
E-commerce microservices = Order, Payment, Cart, Delivery, Notification services separately.

5. Why do we need API Gateway in Microservices?

An **API Gateway** is the single entry point for external clients.
It solves major microservices issues:

- Authentication & Authorization
- Routing requests to appropriate services
- Rate limiting
- Load balancing
- Aggregating responses from multiple services
- Hiding internal service details

Popular gateways: Spring Cloud Gateway, Kong, NGINX, Zuul.

6. What is Service Discovery and why do we need it?

In microservices, services scale dynamically and their IP addresses change.
Service Discovery helps clients find services without hardcoding URLs.

Two types:

- **Client-side discovery** → Eureka
- **Server-side discovery** → AWS ELB, Kubernetes

Without discovery, microservices cannot scale or communicate reliably.

7. What is the role of Eureka Server?

Eureka is a **Service Registry**.
Services register themselves on startup.
Clients ask Eureka for the location of services, enabling:

- Load balancing
- Fault tolerance

- Zero hardcoded URLs

Eureka = “yellow pages” of microservices.

8. What is load balancing in Microservices?

Load balancing distributes traffic across multiple service instances to:

- Avoid overload on a single instance
- Increase availability
- Improve performance

Types:

- **Client-side (Ribbon)** → client chooses instance
- **Server-side (gateway/K8s)** → gateway decides

9. What is Circuit Breaker Pattern?

Circuit Breaker prevents a cascading failure.
If a service repeatedly fails, the breaker “opens” and stops calls temporarily.

States:

- **Closed** → normal
- **Open** → block calls
- **Half-open** → test a limited number of calls

Tools: Resilience4J, Hystrix
It ensures stability when dependent services fail.

10. What is the Saga Pattern?

Saga handles **distributed transactions** across microservices without using 2-phase commit.

Two ways:

- **Choreography** → events
- **Orchestration** → central controller

Example:
Order → Payment → Inventory → Shipping
If Inventory fails, saga triggers compensating actions like refund.

11. How do Microservices handle data consistency?

Microservices avoid shared databases.
To maintain consistency across multiple services:

- Saga pattern
- Event sourcing
- CQRS
- Asynchronous events
- Compensating transactions

Consistency is **eventual**, not immediate.

12. What is the difference between Orchestration and Choreography?

Orchestration:
A central "orchestrator" controls the workflow.
Easier to track; more structured.

Choreography:
Services interact using events with no central controller.
More scalable; more distributed.

13. How do you secure Microservices?

Key approaches:

- JWT for authentication
- OAuth2 / Keycloak / Okta
- API Gateway for centralized security
- mTLS (service-to-service encryption)
- Role-based access control

Microservices must never trust internal traffic blindly.

14. What are common challenges in Microservices?

Major challenges:

- Distributed debugging
- Network latency

- Data integrity issues
- Managing deployments
- Inter-service communication failures
- Complex monitoring

Tools like Prometheus, Grafana, ELK, Jaeger solve these challenges.

15. What is Distributed Tracing?

Distributed tracing tracks a request across multiple services.
It uses:

- Trace ID
- Span ID

Tools: Jaeger, Zipkin
It helps debug latency, failures, and bottlenecks in complex microservices.

16. What is Event-Driven Architecture in Microservices?

In event-driven architecture, services communicate through **events**, not direct calls.

Example:
Order Placed → Order Event → Payment listens → Inventory listens.

Advantages: Loose coupling, scalability, resilience.
Implemented using Kafka, RabbitMQ, SNS/SQS.

17. How do you test Microservices?

Testing layers:

- **Unit tests**
- **Component tests**
- **Contract tests (Consumer-Driven Contracts)**
- **Integration tests**
- **End-to-end tests**

For contract testing: Pact.io is commonly used.

18. What is the Strangler Fig Pattern?

Used for **migrating monoliths to microservices**.
You gradually route traffic away from old system into new microservices.
Eventually the monolith is “strangled”.

Companies heavily use it during modernization.

19. What is Sidecar Pattern?

Sidecar is an additional container that runs beside your service and provides cross-cutting functionalities like:

- Logging
- Monitoring
- Proxying
- Service mesh (Istio/Envoy)

It simplifies service code by offloading infrastructure tasks.

20. What is a Service Mesh?

A service mesh manages **service-to-service communication** with features like:

- mTLS
- Traffic routing
- Retries
- Rate-limiting
- Circuit breaking

Popular meshes: **Istio**, **Linkerd**
It offloads network concerns from microservices.