1) You might have created REST API from scratch or you might know how to create it, right? So, tell me 5 REST API annotations.

When creating REST APIs in Spring Boot, some of the essential annotations used are:

1. @RestController - Designates a class as a controller where every method returns a domain object instead of a view.
2. @RequestMapping - Maps HTTP requests to handler methods of MVC and REST controllers.
3. @GetMapping - A shortcut for @RequestMapping(method = RequestMethod.GET) used to handle GET requests.
4. @PostMapping - A shortcut for @RequestMapping(method = RequestMethod.POST), used for handling POST requests.
5. @PathVariable - Indicates that a method parameter should be bound to a URI template variable.

These annotations help efficiently handle different types of HTTP requests, making it easier to build robust and scalable RESTful services.

2) Have you encountered any real-world bugs due to the incorrect choice between PUT and POST? How did you resolve it?

Using POST instead of PUT can lead to duplicate data creation if the endpoint is called multiple times. To resolve such issues, one would switch to using PUT where appropriate to ensure idempotence, meaning subsequent identical requests won't create extra data but will update the existing record. Additionally, implementing checks for existing data before creation can prevent duplicates when using POST.

3) If you have multiple beans of the same type in your Spring application context, how would you handle conflicts using @Autowired and @Qualifier? What potential issues could arise if you mix these annotations incorrectly?

In Spring, if we have multiple beans of the same type and use @Autowired without specifying which bean to inject, Spring will throw a NoUniqueBeanDefinitionException due to ambiguity. To resolve this, use @Qualifier alongside @Autowired to specify the exact bean by its name. Incorrectly mixing these annotations, like using a wrong qualifier name or omitting @Qualifier when needed, can lead to the wrong bean being injected or runtime errors, disrupting the application's behavior and dependency management.

4) If your application needs to handle file uploads, which controller would you prefer and why? Can you describe a scenario where using a @RestController could lead to complications?

For handling file uploads, a @RestController is typically preferred because it simplifies handling multipart file data directly via HTTP and is well-suited for building RESTful services where responses are data rather than views. However, using a @RestController could lead to complications if the application needs to directly manipulate response headers or directly manage streamed file data, as the abstraction level of @RestController might not provide fine-

grained control over the response compared to using @Controller with added @ResponseBody on methods, which allows more detailed handling of the HTTP response and headers.

5) Are you using cache, if yes, which scenario, when you are removing/refreshing data from cache?

In web applications, caching is often used to store frequently accessed data like user session information or product details. The cache is refreshed or cleared when data changes, ensuring consistency. For example, if product prices are updated in a database, the related cache would be invalidated or refreshed to reflect the new prices, preventing outdated information from being served to users. This approach enhances performance while maintaining data accuracy.

6) Assume your cache is stale due to frequent updates in the underlying data source. How would you determine when to refresh the cache, and what strategies would you use to ensure minimal disruption to users?

To handle frequent updates in the underlying data source and minimize cache staleness, employ a cache invalidation strategy. One effective method is to use event-driven cache refreshes: whenever data is updated in the database, an event triggers the cache to refresh. This ensures that the cache always reflects the most current data. Additionally, consider implementing a time-to-live (TTL) policy for cache entries to automatically refresh data at regular intervals, balancing between freshness and performance.

7) How are you verifying your changes once your changes are deployed to production?

To verify changes after deployment to production, a combination of automated and manual checks is used. Immediately after deployment, automated health checks and integration tests run to ensure that the application is functioning as expected. Monitoring tools track performance and error rates against predefined thresholds to detect anomalies. Additionally, manual testing or user acceptance testing (UAT) might be conducted for critical features. This comprehensive approach helps ensure that the deployment is stable and operates as intended in the production environment.

8) If you discover a critical bug shortly after deployment, what processes do you have in place to roll back the changes quickly and safely?

If a critical bug is discovered after deployment, a rollback process is initiated to revert to the previous stable version of the application. This is typically done using version control systems and deployment tools that support rollback capabilities. The process involves stopping the current deployment, activating the last known good configuration, and restarting the services. Continuous monitoring and automated alerts ensure that any degradation in service is quickly detected and addressed, minimizing downtime and impact on users.

9) Where are you storing your artifacts?

Artifacts are typically stored in a repository management system, such as Artifactory or Nexus, which acts as a centralized storage and management solution for binary artifacts. These systems help in versioning, tracking, and organizing the artifacts generated from builds. They also ensure that artifacts are available for deployment to various environments and facilitate sharing across different development teams, improving efficiency and consistency in the build and release process.

10) If your artifact storage becomes inaccessible, what contingency plans do you have to ensure your deployment process can continue?

If the primary artifact storage becomes inaccessible, a contingency plan involves having a secondary, redundant artifact repository configured as a fallback. This backup should be regularly synchronized with the primary to ensure it has the latest versions of all artifacts. Additionally, ensuring that critical artifacts are cached locally or in a distributed cache can provide temporary relief during outages. These strategies ensure deployments can continue without significant delays, maintaining operational continuity.

11) What is the CAP theorem?

The CAP theorem is a principle that states a distributed database system can only simultaneously provide two out of the following three guarantees: Consistency (all nodes see the same data at the same time), Availability (every request receives a response about whether it was successful or failed), and Partition Tolerance (the system continues to operate despite arbitrary partition failures). This theorem helps guide the design choices for distributed systems, balancing these critical aspects based on the application's specific requirements.

12) Can you provide a real-world scenario where you had to prioritize between consistency, availability, and partition tolerance? What decision did you make?

In a real-world scenario, such as building an e-commerce platform, I had to prioritize availability and partition tolerance over strict consistency. During high-traffic events like sales or festive seasons in India, ensuring the site remains responsive (availability) even if some servers are unreachable (partition tolerance) was crucial. While minor data inconsistencies (like showing stock that was just sold out) are acceptable temporarily, the priority was to keep the system running for all users, preventing downtime and lost sales.

13) Can you talk about a time when a misconfiguration in the properties file caused an issue in production?

Once, a misconfiguration in the properties file caused an issue in a production environment by incorrectly setting the database connection timeout too low. This led to frequent database connection drops under high load, affecting application performance and user experience. The issue was identified through error logs and monitoring tools. The resolution involved correcting the timeout setting in the properties file and redeploying the application, which restored stability and performance to the system.

14) Consider a scenario where a transaction spans multiple service calls. How would you decide the appropriate propagation level for each service call, and what potential pitfalls could arise with your chosen approach?

In a scenario where a transaction spans multiple service calls, choosing the right transaction propagation level is crucial. For instance, if service calls are tightly coupled and need to be atomic, REQUIRED propagation ensures that all services join the existing transaction. However, using REQUIRES_NEW could be appropriate when services need isolation from each other. A pitfall of REQUIRED is potential deadlocks or longer waits due to lock contention. With REQUIRES_NEW, resource consumption increases, and managing multiple transactions can complicate rollback scenarios if one service fails.

15) What are the best practices for designing a custom exception handling framework in Spring Boot?

For designing a custom exception handling framework in Spring Boot, best practices include using @ControllerAdvice to handle exceptions globally across all controllers. Define a base custom exception class for specific errors and extend it for various error types. Use @ExceptionHandler methods within the @ControllerAdvice class to catch and handle different exceptions, returning appropriate HTTP statuses and error messages. This centralized approach ensures consistent error handling and improves the maintainability of the code.

16) How do you manage configuration changes for your Dockerized Spring Boot application across different environments (development, testing, production)?

To manage configuration changes for a Dockerized Spring Boot application across different environments, use environment variables. In Docker, we can set these variables in the docker-compose.yml or pass them directly in the Docker run command. This method allows us to keep sensitive and environment-specific settings (like database URLs and credentials) out of our application's codebase. Configurations can be adjusted dynamically when starting containers, ensuring flexibility and security across environments.

17) As part of a move to a microservices architecture, you are tasked with containerizing your Spring Boot applications using Docker. What are the steps to prepare your Spring Boot application for Docker, and what best practices would you follow?

To containerize a Spring Boot application using Docker, start by creating a Dockerfile in our project root. This file should define the base image (e.g., a lightweight Java image), add our application's jar file, and set the command to run our application. Best practices include keeping our Docker images as small as possible by using multi-stage builds and Alpine-based images. Additionally, externalize configuration using environment variables to manage different environments effectively. This approach ensures a scalable, maintainable, and efficient deployment process.

18) Your application is going live, and you are responsible for setting up monitoring tools. How can Spring Boot Actuator be customized to provide more detailed health metrics specific to your application's needs?

To customize Spring Boot Actuator for detailed health metrics, we can extend the existing health indicators or create new ones specific to our application's needs. Implement the HealthIndicator interface to define custom checks, such as database connections or external API availability. Add these custom indicators to your application's configuration. Additionally, configure the management.endpoint.health.show-details property to 'always' to expose detailed health information. This tailored setup helps in actively monitoring critical components, ensuring timely detection and resolution of issues.

19) You need to integrate Kafka to handle real-time notifications in a social media application built with Spring Boot. How would you set up and configure this integration?

To integrate Kafka for real-time notifications in a Spring Boot application, we can start by adding the Spring Kafka dependency in our project's build file (pom.xml or build.gradle). Configure Kafka properties in the application.properties or application.yml file, specifying the broker address, producer, and consumer settings. Implement Kafka producer and consumer services using @EnableKafka. The producer service sends messages to a Kafka topic, while the consumer service listens to the topic and processes incoming messages. This setup ensures efficient handling of real-time events in your application.

20) How would you ensure that your custom starter doesn't interfere with Spring Boot's own auto-configuration?

To ensure that our custom starter doesn't interfere with Spring Boot's auto-configuration, use conditional annotations such as @ConditionalOnMissingBean, @ConditionalOnClass, and @ConditionalOnProperty. These annotations prevent our configurations from being applied if certain classes, beans, or properties are already defined by Spring Boot's default configuration. Place these conditions in your auto-configuration classes. This approach ensures that our starter adds functionality only when it doesn't conflict with what's already configured, maintaining compatibility and preventing configuration clashes.

21) You're tasked with creating a custom starter for handling cloud-based message queuing in multiple microservices across your organization. What key components will you include in your starter?

For a custom starter handling cloud-based message queuing, key components would include auto-configuration classes to set up and configure the message queue client, service abstractions for publishing and consuming messages, and common utilities like message serialization and deserialization tools. Also, provide default configurations that can be overridden as needed. Include conditional annotations to ensure compatibility with existing configurations and documentation to guide users on integrating and using the starter across different microservices effectively.

22) Suppose you need to ensure zero downtime deployments for a Spring Boot e-commerce application. What approach would you take?

For zero downtime deployments in a Spring Boot e-commerce application, I recommend using a blue-green deployment strategy. This approach involves having two identical production environments (blue and green). Deploy the new version to the green environment while the blue environment remains active. After testing the green environment to ensure it's functioning properly, switch traffic from blue to green. This method minimizes downtime and risk by ensuring there is always a live environment available to users.

23) What are the basic commands to create a docker image and where are you storing your docker images?

To create a Docker image, we can start by writing a Dockerfile which specifies the base image and the steps needed to configure our application. Then, use the command docker build -t your-image-name . in the terminal where our Dockerfile is located to build the image. Docker images are usually stored in a Docker registry, such as Docker Hub, or we can use private registries like AWS ECR or Azure Container Registry for more control and privacy.

24) What is the use of sleuth in spring boot microservice application? As Spring boot 3.x no more supporting sleuth, please name the alternative for the same.

In Spring Boot microservice applications, Sleuth is used for distributed tracing, which helps in tracking requests as they flow through various microservices, making it easier to understand and debug complex transactions. With the release of Spring Boot 3.x, Sleuth is no longer supported. The alternative to Sleuth is OpenTelemetry, an open-source project that provides APIs, libraries, and agents to enable observability for distributed systems, including tracing, metrics, and logging.

25) What is Micrometer Tracing in spring boot 3?

Micrometer Tracing in Spring Boot 3 is a tool for capturing distributed tracing data across microservices. It helps track how requests flow through different services, making it easier to monitor, debug, and improve performance. It integrates with other monitoring tools and replaces Spring Sleuth, providing a unified way to capture metrics and traces. This helps developers understand system behavior and troubleshoot issues efficiently.

26) How will enable and disable auto configuration in spring boot?

In Spring Boot, auto-configuration simplifies the initial setup of applications by automatically configuring Spring components based on the libraries present in your classpath. To disable specific auto-configurations, you can use the **@EnableAutoConfiguration** annotation with the exclude attribute. For example, **@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})** disables the auto-configuration for the data source. To disable all auto-configurations, set **spring.autoconfigure.exclude** property in your **application.properties** or **application.yml** file.

27) What is traceId and span Id in spring boot microservice application and what is the use of these ids?

In a Spring Boot microservice application, **traceId** and **spanId** are unique identifiers used for tracking requests across multiple services. A **traceId** tracks the entire request journey across all microservices, while a **spanId** tracks a single unit of work within a service. These IDs help in monitoring, debugging, and understanding the flow of requests through distributed systems, making it easier to identify issues.

28) What is webflux and mono in spring boot?

WebFlux is a reactive, non-blocking framework in Spring Boot used for building efficient web applications that handle many requests with fewer resources. It allows handling asynchronous data streams. **Mono** is a type in WebFlux that represents a single item or an empty result. It's used to handle asynchronous operations that return one value, like fetching a single record from a database or API response.

29) How will you create custom annotation?

To create a custom annotation in Java, use the @interface keyword. For example, @interface MyAnnotation defines a basic annotation. You can add elements inside the annotation, similar to method declarations. After defining it, you can use the custom annotation on classes, methods, or fields by placing @MyAnnotation above them. Additionally, you can specify the target and retention policy with @Target and @Retention annotations to control its usage and scope.

30) How will you make two ambiguity URL working in spring boot without changing the HTTP method type and no change will be accepted in URL as well?

In Spring Boot, we can resolve ambiguous URLs without changing the HTTP method or URL by using request parameters or path variables. For example, define two methods with the same URL but add different query parameters like @RequestParam. This allows both URLs to exist, but they are distinguished by the presence of certain parameters. This way, we don't need to modify the URL or HTTP method.