# **Câu hỏi về Java và các Framework của Java**

## **Spring Core**

### **Spring là gì? Nêu các tính năng chính của Spring Framework.**

**Trả lời:** Là một Framework mã nguồn mở của Java

**Tính năng chính:**  **Inversion of Control (IoC):** Quản lý các đối tượng (beans) và sự phụ thuộc giữa chúng.

**Dependency Injection (DI):** Cung cấp các phụ thuộc cho đối tượng thay vì để đối tượng tự tạo ra chúng.

**Aspect-Oriented Programming (AOP):** Hỗ trợ tách biệt các mối quan tâm (cross-cutting concerns) như logging, security.

**Data Access:** Hỗ trợ tích hợp với các cơ sở dữ liệu thông qua JDBC, ORM (Hibernate, JPA).

**Transaction Management:** Quản lý giao dịch một cách nhất quán.

**MVC Framework:** Hỗ trợ phát triển ứng dụng web.

### **IoC (Inversion of Control) và DI (Dependency Injection) là gì? Cho ví dụ.**

**IoC:** Là một nguyên lý trong đó việc kiểm soát luồng điều khiển được đảo ngược so với lập trình truyền thống. Thay vì code gọi các thư viện, framework sẽ gọi code của bạn.

**DI:** Là một cách triển khai **IoC**, trong đó các phụ thuộc (dependencies) được "tiêm" vào đối tượng thay vì đối tượng tự tạo ra chúng.

### **Bean trong Spring là gì? Làm thế nào để định nghĩa một bean?**

**Bean**: Là một đối tượng được quản lý bởi Spring IoC container. Nó được tạo, cấu hình, và quản lý bởi Spring.

**Cách định nghĩa:**

Sử dụng annotation: @Component, @Service, @Repository, @Controller.

Cấu hình XML:

*<bean id="userService" class="com.example.UserService"/>*

1. **Sự khác biệt giữa @Component, @Service, @Repository, và @Controller?**

**@Component**: Dùng để đánh dấu một class là bean, có thể được quản lý bởi Spring.

**@Service**: Dùng cho các class thuộc tầng service (business logic).

**@Repository**: Dùng cho các class thuộc tầng DAO (Data Access Object), thường tích hợp với database.

**@Controller**: Dùng cho các class thuộc tầng controller trong ứng dụng web (Spring MVC).

**@Configuration**:

Được sử dụng để đánh dấu một lớp là nguồn cấu hình bean. Các phương thức trong lớp được đánh dấu bằng **@Bean** sẽ trả về một instance của bean được quản lý bởi Spring container

**@Bean:**

Được sử dụng trong các lớp **@Configuration** để khai báo một bean. Phương thức được đánh dấu bằng **@Bean** sẽ trả về một instance của bean đó.

**@RestController:**

Là một phiên bản đặc biệt của **@Controller**, được sử dụng để xây dựng các RESTful web services. Nó kết hợp **@Controller** và **@ResponseBody**.

**@ControllerAdvice:**

Được sử dụng để xử lý các ngoại lệ toàn cục trong ứng dụng Spring MVC. Nó cho phép bạn xử lý các ngoại lệ một cách tập trung.

**@RestControllerAdvice:**

Tương tự như **@ControllerAdvice**, nhưng được sử dụng cho các RESTful web services. Nó kết hợp **@ControllerAdvice** và **@ResponseBody**.

**@Scope:**

Được sử dụng để xác định phạm vi (scope) của một bean, ví dụ như singleton, prototype, request, session, v.v.

**@Lazy:**

Được sử dụng để chỉ định rằng một bean chỉ nên được khởi tạo khi nó được yêu cầu lần đầu tiên, thay vì khi ứng dụng khởi động.

**@Primary:**

Được sử dụng để chỉ định rằng một bean nên được ưu tiên khi có nhiều bean cùng loại được khai báo.

**@Qualifier:**

Được sử dụng để chỉ định rõ ràng bean nào nên được sử dụng khi có nhiều bean cùng loại.

**@Profile:**

Được sử dụng để chỉ định rằng một bean chỉ nên được tạo ra khi một profile cụ thể được kích hoạt.

**@Conditional:**

Được sử dụng để tạo ra các bean dựa trên các điều kiện cụ thể. Bạn có thể tạo các điều kiện tùy chỉnh bằng cách triển khai giao diện Condition.

**@Import**:

Được sử dụng để import các lớp cấu hình khác vào trong một lớp cấu hình.

**@ImportResource:**

Được sử dụng để import các tài nguyên cấu hình XML vào trong một lớp cấu hình.

**@PropertySource:**

Được sử dụng để load các file cấu hình properties vào trong môi trường Spring.

**@Value:**

Được sử dụng để inject giá trị từ các file cấu hình properties vào các biến trong bean.

**@Autowired:**

Được sử dụng để tự động inject các dependency vào các biến, phương thức, hoặc constructor.

**@Required**:

Được sử dụng để chỉ định rằng một thuộc tính bắt buộc phải được thiết lập thông qua dependency injection.

**@EventListener**:

Được sử dụng để đăng ký một phương thức như một listener cho các sự kiện trong Spring.

**@Scheduled:**

Được sử dụng để lên lịch thực hiện các tác vụ định kỳ.

**@Async**:

Được sử dụng để chỉ định rằng một phương thức nên được thực hiện bất đồng bộ.

### **Spring ApplicationContext là gì? Nó khác gì với BeanFactory?**

**ApplicationContext**: Là một interface kế thừa từ BeanFactory, cung cấp thêm các tính năng như AOP, event handling, i18n

**BeanFactory**: Chỉ cung cấp các tính năng cơ bản của IoC container.

### **Các scope của bean trong Spring? Giải thích từng scope.**

**Singleton**: Một bean duy nhất được tạo cho mỗi Spring IoC container (mặc định).

**Prototype**: Một bean mới được tạo mỗi khi được yêu cầu.

**Request**: Một bean được tạo cho mỗi HTTP request (chỉ dùng trong ứng dụng web).

**Session**: Một bean được tạo cho mỗi HTTP session (chỉ dùng trong ứng dụng web).

**Global** **Session**: Dùng cho các ứng dụng portlet.

### **Làm thế nào để cấu hình Spring mà không sử dụng XML?**

Sử dụng annotation và Java-based configuration

### **Spring Profiles là gì? Làm thế nào để sử dụng chúng?**

Profiles: Cho phép cấu hình ứng dụng khác nhau cho các môi trường khác nhau (ví dụ: dev, test, uat, prod).

Kích hoạt profile bằng cách đặt ***spring.profiles.active=dev*** trong ***application.properties***.

### **AOP (Aspect-Oriented Programming) trong Spring là gì? Cho ví dụ về cách sử dụng @Aspect.**

**AOP**: Là một kỹ thuật để tách biệt các mối quan tâm (cross-cutting concerns) như logging, security.

### **Sự khác biệt giữa @Autowired và @Resource?**

**@Autowired:** Thuộc về Spring, tự động tiêm bean theo kiểu dữ liệu.

**@Resource:** Thuộc về JSR-250, tiêm bean theo tên.

## **Spring Boot**

1. Spring Boot là gì? Tại sao nó được sử dụng phổ biến?

**Spring Boot** là một framework dựa trên nền tảng Spring, được thiết kế để đơn giản hóa quá trình phát triển ứng dụng Java. Nó cung cấp các cấu hình mặc định và các công cụ tích hợp sẵn, giúp nhà phát triển nhanh chóng tạo ra các ứng dụng độc lập, sẵn sàng để chạy mà không cần phải cấu hình phức tạp.

**Lý do phổ biến:**

**Đơn giản hóa cấu hình:** Spring Boot tự động cấu hình nhiều thành phần, giảm thiểu công sức cấu hình thủ công.

**Tích hợp sẵn các công cụ:** Nó tích hợp sẵn các công cụ như Tomcat, Jetty, và các thư viện khác, giúp ứng dụng chạy ngay lập tức.

**Microservices**: Spring Boot rất phù hợp để xây dựng các ứng dụng microservices nhờ vào khả năng tạo các ứng dụng độc lập và nhẹ.

**Hỗ trợ Actuator:** Cung cấp các công cụ giám sát và quản lý ứng dụng thông qua Spring Boot Actuator.

1. **Spring Boot Auto-Configuration là gì? Nó hoạt động như thế nào?**

**Spring Boot Auto-Configuration** là cơ chế tự động cấu hình các thành phần của ứng dụng dựa trên các dependencies có trong classpath. Nó giúp giảm thiểu việc cấu hình thủ công bằng cách tự động thiết lập các bean cần thiết.

**Cách hoạt động:**

Spring Boot quét classpath để xác định các thư viện có sẵn.

Dựa trên các thư viện này, nó tự động cấu hình các bean tương ứng. Ví dụ, nếu bạn có thư viện spring-boot-starter-data-jpa trong classpath, Spring Boot sẽ tự động cấu hình DataSource, EntityManager, v.v.

Các cấu hình tự động này có thể được ghi đè bằng cách cung cấp các cấu hình tùy chỉnh trong file **application.properties** hoặc **application.yml**

1. Spring Boot Actuator là gì? Nó dùng để làm gì?

**Spring Boot Actuator** là một module cung cấp các endpoint để giám sát và quản lý ứng dụng Spring Boot. Nó cung cấp các thông tin về sức khỏe, metrics, log, và nhiều thông tin khác của ứng dụng

Các tính năng chính:

**Health Checks**: Kiểm tra tình trạng sức khỏe của ứng dụng.

**Metrics**: Thu thập và hiển thị các chỉ số về hiệu suất.

**Environment**: Hiển thị các thông tin về môi trường.

**Loggers**: Quản lý và cấu hình log.

1. Cách sử dụng @SpringBootApplication annotation?

**@SpringBootApplication** là một annotation kết hợp của ba annotation khác:

**@Configuration:** Đánh dấu lớp là nguồn cấu hình bean.

**@EnableAutoConfiguration:** Kích hoạt cơ chế auto-configuration của Spring Boot.

**@ComponentScan:** Quét các component, service, repository trong package và các package con.

## **Spring Data JPA**

1. **JPA là gì? Nó khác gì với Hibernate?**

**Bản chất:**

JPA là specification (tiêu chuẩn)

Hibernate là implementation (triển khai)

**Vai trò :**

JPA định nghĩa cách thức làm việc với database

Hibernate triển khai các chức năng thực tế

**Tính chất :**

JPA chỉ là interface, không có code triển khai

Hibernate có code logic đầy đủ để xử lý dữ liệu

**JPA (Java Persistence API):** là một đặc tả (**specification**) của Java EE (nay là Jakarta EE) dùng để quản lý mapping giữa các đối tượng Java (POJO) và cơ sở dữ liệu quan hệ (RDBMS). Nó giúp các ứng dụng Java làm việc với cơ sở dữ liệu mà không cần viết SQL thuần, nhờ vào cơ chế **ORM (Object-Relational Mapping)**..

**JPA** chỉ định các annotation và API để thực hiện các thao tác CRUD (Create, Read, Update, Delete) và các truy vấn phức tạp.

**Hibernate**:

**Hibernate** là một framework ORM (Object-Relational Mapping) mã nguồn mở, triển khai đặc tả JPA. Nó cung cấp các tính năng bổ sung ngoài những gì JPA quy định.

**Hibernate** có thể được sử dụng độc lập mà không cần JPA, nhưng nó cũng hỗ trợ đầy đủ các annotation và API của JPA.

Sự khác biệt:

**JPA** là một đặc tả, trong khi **Hibernate** là một triển khai cụ thể của đặc tả đó.

**Hibernate** cung cấp nhiều tính năng mở rộng và tùy chỉnh hơn so với JPA.

Như vậy, JPA và Hibernate không phải là hai thứ đối lập nhau mà là bổ trợ cho nhau, trong đó JPA cung cấp tiêu chuẩn còn Hibernate là một trong những triển khai phổ biến nhất của tiêu chuẩn đó.

1. Sự khác biệt giữa CrudRepository và JpaRepository?

**CrudRepository:** Cung cấp các phương thức CRUD cơ bản như save, findById, findAll, delete, v.v. Nó không hỗ trợ phân trang và sắp xếp.

**JpaRepository:** Kế thừa từ CrudRepository và PagingAndSortingRepository, cung cấp thêm các phương thức liên quan đến phân trang và sắp xếp, cũng như các phương thức JPA-specific như flush, deleteInBatch, v.v.

**Sự khác biệt chính:**

**CrudRepository** chỉ cung cấp các phương thức CRUD cơ bản.

**JpaRepository** cung cấp thêm các phương thức phân trang, sắp xếp, và các phương thức JPA-specific.

1. Cách xử lý transaction trong Spring Data JPA?

**Spring Data JPA** sử dụng cơ chế quản lý transaction của Spring. Bạn có thể sử dụng annotation **@Transactional** để đánh dấu một phương thức hoặc lớp là transactional.

+ Sử dụng **@Transactional** annotation:

Đánh dấu một phương thức hoặc lớp để Spring quản lý transaction

+ Cấu hình transaction manager:

Spring Boot tự động cấu hình PlatformTransactionManager nếu bạn sử dụng Spring Data JPA.

**@Transactional** có thể được sử dụng ở mức lớp hoặc phương thức.

Mặc định, Spring sẽ rollback transaction nếu có một unchecked exception (RuntimeException) được ném ra.

**Annotation** **@Transactional** là một trong những annotation quan trọng nhất trong Spring để quản lý các giao dịch (**transactions**) trong ứng dụng. Nó giúp đảm bảo tính toàn vẹn của dữ liệu bằng cách quản lý các thao tác trên cơ sở dữ liệu trong một phạm vi giao dịch. Dưới đây là chi tiết về cách sử dụng **@Transactional**, từ cơ bản đến nâng cao.

1.Cơ bản về @Transactional

**1.1. Cách sử dụng cơ bản**

Annotation @Transactional được sử dụng để đánh dấu một phương thức hoặc lớp là **transactional**, nghĩa là các thao tác trong phương thức đó sẽ được thực hiện trong một giao dịch.

Khi phương thức được gọi, Spring sẽ tự động bắt đầu một giao dịch. Nếu phương thức thực hiện thành công, giao dịch sẽ được commit. Nếu có ngoại lệ (exception) xảy ra, giao dịch sẽ được rollback.

**1.2. Phạm vi của @Transactional**

**Mức phương thức:** Khi @Transactional được đặt trên một phương thức, chỉ phương thức đó được quản lý bởi giao dịch.

**Mức lớp:** Khi **@Transactional** được đặt trên một lớp, tất cả các phương thức public trong lớp đó sẽ được quản lý bởi giao dịch.

2. Các thuộc tính của @Transactional

Annotation **@Transactional** có nhiều thuộc tính để tùy chỉnh hành vi của giao dịch. Dưới đây là các thuộc tính quan trọng

**2.1. propagation**

Định nghĩa cách thức lan truyền (propagation) của giao dịch. Có các giá trị sau:

**Propagation.REQUIRED (mặc định):** Nếu đã có giao dịch hiện tại, sử dụng giao dịch đó. Nếu không, tạo giao dịch mới.

**Propagation.REQUIRES\_NEW:** Luôn tạo một giao dịch mới, tạm dừng giao dịch hiện tại (nếu có).

**Propagation.SUPPORTS:** Sử dụng giao dịch hiện tại (nếu có), nếu không, thực hiện không có giao dịch.

**Propagation.NOT\_SUPPORTED:** Thực hiện không có giao dịch, tạm dừng giao dịch hiện tại (nếu có).

**Propagation.MANDATORY:** Bắt buộc phải có giao dịch hiện tại, nếu không sẽ ném ra ngoại lệ.

**Propagation.NEVER:** Không được phép có giao dịch hiện tại, nếu có sẽ ném ra ngoại lệ.

**Propagation.NESTED:** Tạo một giao dịch con (nested transaction) trong giao dịch hiện tại (nếu có).

**2.2. isolation**

Định nghĩa mức độ cô lập (isolation) của giao dịch. Có các giá trị sau:

**Isolation.DEFAULT (mặc định):** Sử dụng mức cô lập mặc định của cơ sở dữ liệu.

**Isolation.READ\_UNCOMMITTED:** Cho phép đọc dữ liệu chưa được commit (có thể dẫn đến dirty reads).

**Isolation.READ\_COMMITTED:** Chỉ cho phép đọc dữ liệu đã được commit.

**Isolation.REPEATABLE\_READ:** Đảm bảo rằng nếu một transaction đọc cùng một dữ liệu nhiều lần, kết quả sẽ giống nhau.

**Isolation.SERIALIZABLE:** Mức cô lập cao nhất, đảm bảo không có dirty reads, non-repeatable reads, và phantom reads.

**2.3. readOnly**

Chỉ định giao dịch là chỉ đọc (read-only). Mặc định là false.

Khi **readOnly = true,** Spring có thể tối ưu hóa hiệu suất bằng cách sử dụng các cơ chế như **connection pooling.**

**2.4. timeout**

Chỉ định thời gian tối đa (tính bằng giây) mà giao dịch có thể thực hiện. Nếu vượt quá thời gian này, giao dịch sẽ bị rollback.

Mặc định là -1 (không giới hạn thời gian).

**2.5. rollbackFor và noRollbackFor**

**rollbackFor**: Chỉ định các ngoại lệ cụ thể sẽ kích hoạt rollback.

**noRollbackFor:** Chỉ định các ngoại lệ cụ thể sẽ không kích hoạt rollback.

3. Các vấn đề nâng cao

**3.1. Self-invocation (Gọi nội bộ)**

Khi một phương thức trong cùng một lớp gọi một phương thức khác có **@Transactional**, giao dịch sẽ không được áp dụng do cơ chế proxy của Spring.

Giải pháp: Sử dụng AspectJ hoặc tách phương thức sang một lớp dịch vụ khác.

**3.2. Transactional với nhiều DataSources**

Khi ứng dụng sử dụng nhiều nguồn dữ liệu (DataSources), bạn cần cấu hình nhiều transaction managers.

Sử dụng **@Transactional** với thuộc tính value để chỉ định transaction manager cụ thể.

**3.3. Transactional với Event Listeners**

Khi sử dụng Spring Events, bạn có thể muốn gửi sự kiện sau khi giao dịch được commit.

Sử dụng @TransactionalEventListener để lắng nghe sự kiện sau khi giao dịch hoàn tất.

## **Spring Security**

1. Spring Security là gì? Nó dùng để làm gì?
2. Cách cấu hình Spring Security để bảo vệ ứng dụng?
3. Sự khác biệt giữa Authentication và Authorization?

**Authentication (Xác thực):** Là quá trình xác minh danh tính của người dùng. Ví dụ: Khi người dùng đăng nhập bằng username và password, hệ thống sẽ kiểm tra xem thông tin đó có hợp lệ hay không.

**Authorization (Ủy quyền):** Là quá trình kiểm tra quyền truy cập của người dùng đã được xác thực. Ví dụ: Sau khi đăng nhập, hệ thống sẽ kiểm tra xem người dùng có quyền truy cập vào một tài nguyên cụ thể hay không.

Tóm tắt:

**Authentication**: "Bạn là ai?"

**Authorization**: "Bạn có quyền làm gì?"

1. Cách sử dụng JWT (JSON Web Token) trong Spring Security?

**4.1: JWT (JSON Web Token)** là một chuẩn mở (RFC 7519) định nghĩa cách truyền thông tin an toàn giữa các bên dưới dạng JSON object. JWT thường được sử dụng để xác thực (authentication) và ủy quyền (authorization) trong các ứng dụng web và API. Token này có thể được ký (signed) để đảm bảo tính toàn vẹn và mã hóa (encrypted) để bảo mật thông tin.

Một JWT bao gồm 3 phần, được phân tách bằng dấu chấm (.):

1. **Header**:

Chứa thông tin về loại token (typ) và thuật toán ký (alg), ví dụ: HMAC SHA256 hoặc RSA.

1. **Payload**:

* Chứa các thông tin (claims) về người dùng hoặc dữ liệu khác. Có 3 loại claims:

**Registered Claims**: Các claims được định nghĩa sẵn, ví dụ: iss (issuer), exp (expiration time), sub (subject), aud (audience).

**Public Claims**: Các claims tùy chỉnh, có thể được định nghĩa bởi người dùng.

**Private Claims**: Các claims dùng để chia sẻ thông tin giữa các bên.

1. **Signature**:

Được tạo bằng cách mã hóa (encode) header và payload, sau đó ký (sign) bằng thuật toán được chỉ định trong header.

**4.2: Tại sao sử dụng JWT?**

**Stateless (Không lưu trữ trạng thái):**

JWT chứa tất cả thông tin cần thiết trong chính nó, không cần lưu trữ trên server. Điều này giúp giảm tải cho server và dễ dàng mở rộng hệ thống.

**Bảo mật:**

JWT có thể được ký (signed) để đảm bảo tính toàn vẹn và mã hóa (encrypted) để bảo mật thông tin.

**Linh hoạt:**

JWT có thể chứa bất kỳ thông tin nào, phù hợp với nhiều trường hợp sử dụng khác nhau.

**Dễ dàng tích hợp:**

JWT có thể được sử dụng trong nhiều ngôn ngữ và framework khác nhau.

**4.3: Cách hoạt động của JWT trong Spring Security**

1. Người dùng đăng nhập:

Người dùng gửi thông tin đăng nhập (username/password) đến server.

Server xác thực thông tin và tạo JWT chứa thông tin người dùng (ví dụ: username, roles).

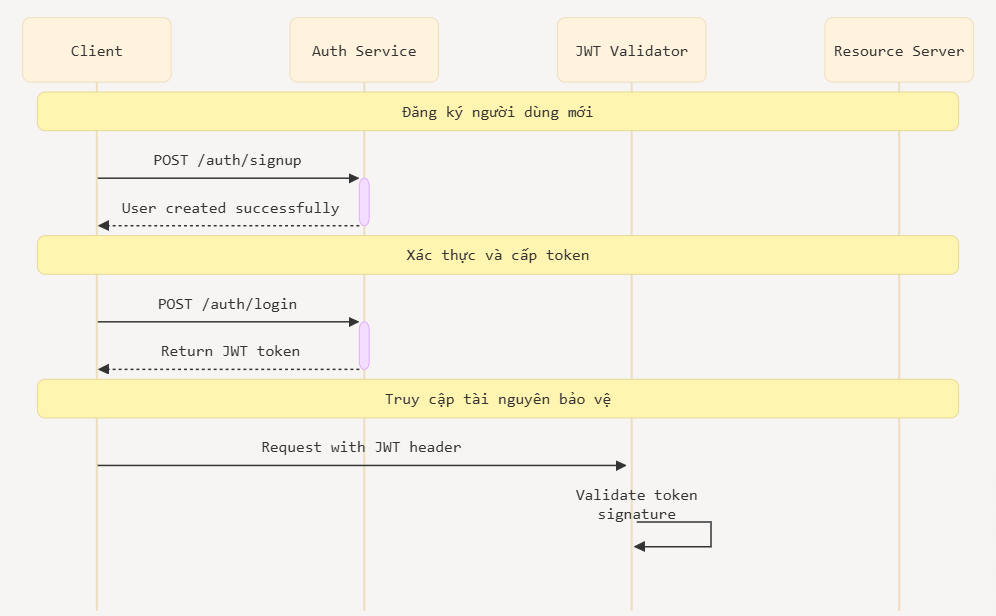
2. Server trả về JWT:

Server trả về JWT cho client (thường qua header hoặc body của response).

3. Client sử dụng JWT:

Client lưu trữ JWT (thường trong localStorage hoặc cookies).

Trong các request tiếp theo, client gửi JWT trong header Authorization::



Trong sơ đồ trên:

**Client**: Ứng dụng hoặc dịch vụ gọi đến API của chúng ta

**Auth Service**: Dịch vụ xử lý đăng ký và xác thực người dùng

**JWT Validator**: Bộ phận kiểm tra tính hợp lệ của token

**Resource Server**: Máy chủ chứa các tài nguyên cần bảo vệ.

1. Làm thế nào để triển khai OAuth2 trong Spring Security?

## **Spring Integration**

1. Spring Integration là gì? Nó dùng để làm gì?

**Spring Integration**: Là một framework mở rộng của Spring, dùng để tích hợp các hệ thống khác nhau thông qua các kênh (channels) và endpoints.

**Mục đích:** Giúp xây dựng các ứng dụng tích hợp dễ dàng và linh hoạt.

1. Các thành phần chính trong Spring Integration (Message, Channel, Endpoint)?

**Message**: Đơn vị dữ liệu được trao đổi giữa các thành phần.

**Channel**: Kênh truyền thông giữa các thành phần.

**Endpoint**: Các thành phần xử lý message (ví dụ: service activator, transformer).

1. Làm thế nào để tích hợp Spring Integration với hệ thống bên ngoài (ví dụ: JMS, FTP)?

Sử dụng các adapter như **JmsMessageDrivenChannelAdapter** hoặc **FtpInboundFileSynchronizer**.

1. Sự khác biệt giữa Point-to-Point Channel và Publish-Subscribe Channel?

**Point-to-Point**: Một message được gửi đến một consumer duy nhất.

**Publish-Subscribe**: Một message được gửi đến nhiều consumer.

1. **Cách sử dụng @Transformer, @Filter, và @Router trong Spring Integration?**

**@Transformer:** Chuyển đổi message từ dạng này sang dạngkhác**.**

**@Filter**: Lọc message dựa trên điều kiện.

**@Router:** Định tuyến message đến các kênh khác nhau.

1. **Làm thế nào để xử lý lỗi trong Spring Integration?**
2. **Cách tích hợp Spring Integration với Kafka hoặc RabbitMQ?**
3. **Spring Integration DSL là gì? Cho ví dụ về cách sử dụng.**
4. **Làm thế nào để monitor và debug các luồng tích hợp trong Spring Integration?**
5. **Cách sử dụng Spring Integration để xử lý bất đồng bộ?**

## **Java Core**

**Java Basics**

1. Sự khác biệt giữa JDK, JRE, và JVM?

**JDK (Java Development Kit):** Cung cấp công cụ để phát triển ứng dụng Java (compiler, debugger).

**JRE (Java Runtime Environment):** Cung cấp môi trường để chạy ứng dụng Java.

**JVM (Java Virtual Machine):** Thực thi bytecode và quản lý bộ nhớ.

### **Các tính năng chính của Java?**

Đơn giản, hướng đối tượng, độc lập nền tảng, bảo mật, đa luồng, hiệu suất cao.

1. **Sự khác biệt giữa == và .equals()?**

**==:** So sánh tham chiếu (địa chỉ bộ nhớ).

**.equals():** So sánh giá trị (có thể được override).

1. String trong Java là immutable. Tại sao?

**Immutable:** Đảm bảo an toàn luồng (thread-safe), tối ưu hóa bộ nhớ (string pool).

1. Sự khác biệt giữa String, StringBuilder, và StringBuffer?

**String:** Immutable**.**

**StringBuilder:** Mutable, không đồng bộ.

**StringBuffer:** Mutable, đồng bộ.

### **Các kiểu dữ liệu nguyên thủy (primitive types) trong Java?**

**Kiểu số nguyên:**

**byte**: 8-bit, giá trị từ -128 đến 127**.**

**short:** 16-bit, giá trị từ -32,768 đến 32,767.

**int:** 32-bit, giá trị từ -2^31 đến 2^31 - 1.

**long:** 64-bit, giá trị từ -2^63 đến 2^63 - 1.

**Kiểu số thực:**

**float:** 32-bit, dùng cho số thực dấu phẩy động.

**double**: 64-bit, dùng cho số thực dấu phẩy động với độ chính xác cao hơn.

**Kiểu ký tự:**

**char:** 16-bit, dùng để lưu trữ một ký tự Unicode.

**Kiểu logic:**

**boolean:** Chỉ có hai giá trị là true hoặc false.

1. Wrapper classes trong Java là gì? Tại sao chúng được sử dụng?

**Wrapper classes** là các lớp đối tượng (object) tương ứng với các kiểu dữ liệu nguyên thủy (primitive types).

**Hỗ trợ Collections Framework:** Các collection (như ArrayList, HashMap) chỉ làm việc với các đối tượng, không làm việc với kiểu nguyên thủy.

**Cung cấp các phương thức tiện ích:** Ví dụ: Integer.parseInt(), Double.toString(), v.v.

**Hỗ trợ Autoboxing và Unboxing:** Tự động chuyển đổi giữa kiểu nguyên thủy và wrapper class.

**Autoboxing:** Chuyển đổi tự động từ kiểu nguyên thủy sang wrapper class.

**Unboxing:** Chuyển đổi tự động từ wrapper class sang kiểu nguyên thủy.

1. Sự khác biệt giữa final, finally, và finalize?

**final:** Là một từ khóa dùng để đánh dấu một thực thể là không thể thay đổi.

**Có thể áp dụng cho:**

**Biến:**Biến không thể thay đổi giá trị sau khi khởi tạo.

**Phương thức:** Phương thức không thể ghi đè (override) trong lớp con.

**Lớp:** Lớp không thể kế thừa.

**finally:** Là một khối trong cấu trúc try-catch-finally, luôn được thực thi dù có ngoại lệ (exception) xảy ra hay không.

**finalize:** Là một phương thức của lớp Object, được gọi bởi Garbage Collector trước khi đối tượng bị thu hồi bộ nhớ.

9. Sự khác biệt giữa ArrayList và LinkedList?

**Cả ArrayList và LinkedList**đều là các lớp triển khai của interface List trong Java, nhưng chúng có cách tổ chức dữ liệu và hiệu suất khác nhau tùy thuộc vào trường hợp sử dụng. Dưới đây là sự khác biệt chính giữa chúng:

**1. Cấu trúc dữ liệu**

**ArrayList:**

Sử dụng một mảng động (dynamic array) để lưu trữ các phần tử.

Khi kích thước vượt quá dung lượng hiện tại, ArrayList tự động tăng kích thước mảng (thường là gấp đôi).

**LinkedList:**

Sử dụng danh sách liên kết đôi (doubly linked list) để lưu trữ các phần tử.

Mỗi phần tử (node) trong LinkedList chứa dữ liệu và hai tham chiếu (con trỏ) đến phần tử trước và sau.

**2. Hiệu suất**

**Thao tác** ArrayList LinkedList

**Truy cập phần tử** O(1) O(n)

**Thêm phần tử ở đầu** O(n) (do phải dịch chuyển các phần tử) O(1)

**Thêm phần tử ở cuối** O(1) (nếu không cần mở rộng mảng) O(1)

**Thêm phần tử ở giữa** O(n) O(1) (nếu đã biết vị trí chèn)

**Xóa phần tử ở đầu** O(n) O(1)

**Xóa phần tử ở cuối** O(1) O(1)

**Xóa phần tử ở giữa** O(n) O(1) (nếu đã biết vị trí xóa)

**Tìm kiếm phần tử** O(n) O(n)

**3. Ưu điểm**

**ArrayList:**

Truy cập ngẫu nhiên (random access) nhanh (O(1)).

Hiệu suất tốt khi thêm/xóa phần tử ở cuối danh sách.

Tiết kiệm bộ nhớ hơn vì chỉ lưu trữ dữ liệu, không cần lưu thêm con trỏ.

**LinkedList:**

Thêm/xóa phần tử ở đầu hoặc giữa danh sách nhanh (O(1)).

Linh hoạt hơn khi cần thay đổi cấu trúc dữ liệu thường xuyên.

1. **Nhược điểm**

**ArrayList:**

Thêm/xóa phần tử ở đầu hoặc giữa danh sách chậm (O(n)) do phải dịch chuyển các phần tử.

Cần mở rộng mảng khi vượt quá dung lượng, gây tốn chi phí**.**

**LinkedList:**

Truy cập ngẫu nhiên chậm (O(n)) do phải duyệt từ đầu hoặc cuối danh sách.

Tốn bộ nhớ hơn do lưu trữ thêm con trỏ.

**Sử dụng ArrayList khi:**

Cần truy cập phần tử thường xuyên (ví dụ: đọc dữ liệu từ danh sách).

Thêm/xóa phần tử chủ yếu ở cuối danh sách.

Cần tiết kiệm bộ nhớ.

**Sử dụng LinkedList khi:**

Cần thêm/xóa phần tử thường xuyên ở đầu hoặc giữa danh sách.

Không cần truy cập ngẫu nhiên thường xuyên.

Cần một cấu trúc dữ liệu linh hoạt.

**Kết luận**

**ArrayList**: Phù hợp khi cần truy cập phần tử nhanh và thêm/xóa phần tử ở cuối danh sách.

**LinkedList**: Phù hợp khi cần thêm/xóa phần tử thường xuyên ở đầu hoặc giữa danh sách.

## **OOP (Object-Oriented Programming)**

1. 4 nguyên lý cơ bản của OOP? Giải thích từng nguyên lý.

**a. Encapsulation (Đóng gói)**

**Định nghĩa**: Đóng gói dữ liệu và phương thức liên quan vào một đơn vị (class), đồng thời kiểm soát quyền truy cập vào dữ liệu thông qua các access modifiers.

**Mục đích:** Bảo vệ dữ liệu khỏi truy cập trái phép và đảm bảo tính toàn vẹn của dữ liệu.

**b. Inheritance (Kế thừa)**

**Định nghĩa:** Cho phép một class (lớp con) kế thừa các thuộc tính và phương thức từ một class khác (lớp cha).

**Mục đích:** Tái sử dụng code và tạo ra mối quan hệ "is-a" giữa các lớp.

**c. Polymorphism (Đa hình)**

**Định nghĩa**: Khả năng một đối tượng có thể thực hiện các hành động khác nhau tùy thuộc vào ngữ cảnh.

**Mục đích:** Làm cho code linh hoạt và dễ mở rộng.

**d. Abstraction (Trừu tượng hóa)**

**Định nghĩa:** Ẩn đi các chi tiết phức tạp và chỉ hiển thị các tính năng cần thiết.

**Mục đích:** Giảm độ phức tạp và tập trung vào các khía cạnh quan trọng.

2. Sự khác biệt giữa lớp trừu tượng (abstract class) và interface?

**Abstract Class Interface**

Có thể chứa phương thức có thân (implemented). Chỉ chứa phương thức không có thân (trước Java 8).

Có thể chứa các biến instance. Chỉ chứa các biến static final (hằng số).

Một lớp chỉ có thể kế thừa một abstract class. Một lớp có thể implement nhiều interface.

Dùng để chia sẻ code giữa các lớp con. Dùng để định nghĩa một hợp đồng (contract).

3. Overloading và Overriding là gì? Cho ví dụ.

**Overloading (Nạp chồng)**

**Định nghĩa:** Cùng một tên phương thức nhưng khác nhau về số lượng hoặc kiểu tham số.

**Overriding (Ghi đè)**

**Định nghĩa**: Lớp con định nghĩa lại phương thức của lớp cha.

### **4. Sự khác biệt giữa public, private, protected, và default access modifiers**?

5. Cách sử dụng super và this trong Java?

**super:** Dùng để truy cập các thành phần (phương thức, biến) của lớp cha.

**this:** Dùng để tham chiếu đến đối tượng hiện tại.

6. Sự khác biệt giữa composition và inheritance?

**Composition Inheritance**

Mối quan hệ "has-a" (có một). Mối quan hệ "is-a" (là một).

Lớp chứa đối tượng của lớp khác. Lớp con kế thừa từ lớp cha.

Linh hoạt hơn, dễ thay đổi. Cứng nhắc hơn, khó thay đổi.

Ví dụ: Car có một Engine. Ví dụ: Dog là một Animal.

7. Cách sử dụng static trong Java?

**Biến static:** Dùng chung cho tất cả các đối tượng của lớp.

**Phương thức static:** Gọi trực tiếp từ tên lớp mà không cần tạo đối tượng.

## **Collections**

1. Các interface chính trong Java Collections Framework?

**Java Collections Framework bao gồm các interface chính sau:**

**Collection**: Interface gốc, đại diện cho một nhóm các đối tượng (elements).

**List**: Một collection có thứ tự, cho phép trùng lặp phần tử.

**Set**: Một collection không có thứ tự, không cho phép trùng lặp phần tử.

**Map**: Lưu trữ dữ liệu dưới dạng cặp key-value, không cho phép trùng lặp key.

**Queue**: Một collection dùng để lưu trữ các phần tử trước khi xử lý (FIFO hoặc LIFO).

**Deque:** Một queue hai đầu (Double-Ended Queue).

2. Sự khác biệt giữa List, Set, và Map?

|  |  |  |  |
| --- | --- | --- | --- |
| **Đặc điểm** | **List** | **Set** | **Map** |
| Thứ tự | Có thứ tự (theo chỉ số). | Không có thứ tự. | Không có thứ tự (trừ LinkedHashMap). |
| Trùng lặp | Cho phép trùng lặp phần tử. | Không cho phép trùng lặp phần tử. | Không cho phép trùng lặp key. |
| Cấu trúc dữ liệu | Dựa trên chỉ số (index). | Dựa trên hash table hoặc cây. | Dựa trên cặp key-value. |
| Ví dụ | ArrayList, LinkedList. | HashSet, TreeSet. | HashMap, TreeMap. |

3. Cách sử dụng HashMap? Làm thế nào nó hoạt động?

**Cách hoạt động của HashMap**

**Hash Table:** HashMap sử dụng một mảng (array) để lưu trữ các bucket.

**Hash Function:** Mỗi key được ánh xạ thành một index trong mảng thông qua hash function.

**Collision Handling:** Nếu nhiều key có cùng hash value, chúng được lưu trữ trong một linked list hoặc cây (từ Java 8).

4. Sự khác biệt giữa HashMap và Hashtable?

|  |  |  |
| --- | --- | --- |
| **Đặc điểm** | **HashMap** | **Hashtable** |
| Đồng bộ (Synchronized) | Không đồng bộ. | Đồng bộ (thread-safe). |
| Cho phép null | Cho phép một key null và nhiều value null. | Không cho phép key hoặc value null. |
| Hiệu suất | Nhanh hơn vì không đồng bộ. | Chậm hơn vì đồng bộ. |
| Kế thừa | Kế thừa từ AbstractMap. | Kế thừa từ Dictionary. |

5. Cách sử dụng Comparator và Comparable?

**Comparable:**  Interface để sắp xếp các đối tượng dựa trên thuộc tính tự nhiên (natural ordering).

**Comparator**: Interface để sắp xếp các đối tượng dựa trên logic tùy chỉnh.

6. Sự khác biệt giữa ArrayList và Vector?

|  |  |  |
| --- | --- | --- |
| Đặc điểm | ArrayList | Vector |
| Đồng bộ (Synchronized) | Không đồng bộ. | Đồng bộ (thread-safe). |
| Hiệu suất | Nhanh hơn vì không đồng bộ. | Chậm hơn vì đồng bộ. |
| Tăng kích thước | Tăng 50% kích thước hiện tại. | Tăng gấp đôi kích thước hiện tại. |
| Kế thừa | Kế thừa từ AbstractList. | Kế thừa từ AbstractList. |

7. Cách sử dụng Iterator và ListIterator?

**Iterator:** Dùng để duyệt qua các phần tử của collection.

**ListIterator:** Dùng để duyệt qua các phần tử của List, hỗ trợ duyệt cả hai chiều (tiến và lùi).

8. Concurrent Collections trong Java là gì? Cho ví dụ.

**Định nghĩa:** Các collection được thiết kế để hỗ trợ đa luồng (thread-safe) mà không cần đồng bộ hóa bên ngoài.

9. So sánh Array và ArratList?

|  |  |  |
| --- | --- | --- |
| Đặc điểm | Array | ArrayList |
| Kích thước | Cố định (không thể thay đổi sau khi khai báo). | Linh hoạt (có thể thay đổi kích thước động). |
| Cú pháp khai báo | int[] arr = new int[10]; | ArrayList<Integer> list = new ArrayList<>(); |
| Kiểu dữ liệu | Chỉ hỗ trợ kiểu dữ liệu nguyên thủy và đối tượng. | Chỉ hỗ trợ kiểu dữ liệu đối tượng (sử dụng wrapper class cho kiểu nguyên thủy). |
| Hiệu suất | Nhanh hơn do truy cập trực tiếp bằng chỉ số. | Chậm hơn do cần thêm các thao tác quản lý bộ nhớ động. |
| Bộ nhớ | Tiết kiệm bộ nhớ hơn. | Tốn nhiều bộ nhớ hơn do lưu trữ thêm thông tin về kích thước và các phương thức hỗ trợ. |
| Thao tác thêm/xóa phần tử | Không hỗ trợ thêm/xóa phần tử. | Hỗ trợ thêm/xóa phần tử linh hoạt. |
| Các phương thức hỗ trợ | Không có phương thức hỗ trợ sẵn. | Có nhiều phương thức hỗ trợ như add(), remove(), get(), set(), v.v. |
| Đồng bộ (Thread-safe) | Không đồng bộ. | Không đồng bộ (cần sử dụng Collections.synchronizedList để đồng bộ). |
| Kích thước mặc định | Kích thước được chỉ định khi khai báo. | Kích thước mặc định là 10, tự động tăng khi cần. |
| Ví dụ | int[] arr = {1, 2, 3}; | ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3)); |

10. Sự khác biệt giữa parallelStream và stream?

|  |  |  |
| --- | --- | --- |
| Đặc điểm | stream | parallelStream |
| Xử lý | Tuần tự (single-threaded). | Song song (multi-threaded). |
| Hiệu suất | Phù hợp với dữ liệu nhỏ. | Phù hợp với dữ liệu lớn và xử lý phức tạp. |
| Ví dụ | list.stream().forEach(...) | list.parallelStream().forEach(...) |

**Multithreading**

1. **Cách tạo một thread trong Java?**
2. **Sự khác biệt giữa Thread và Runnable?**
3. **Cách sử dụng synchronized trong Java?**
4. **Sự khác biệt giữa wait(), notify(), và notifyAll()?**
5. **Cách sử dụng ExecutorService để quản lý thread pool?**
6. **Sự khác biệt giữa Callable và Runnable?**
7. **Cách sử dụng Future và CompletableFuture?**
8. **Deadlock là gì? Làm thế nào để tránh deadlock?**
9. **Cách sử dụng volatile trong Java?**
10. **Sự khác biệt giữa ReentrantLock và synchronized**?

**Exception Handling**

1. **Sự khác biệt giữa checked và unchecked exceptions?**
2. **Cách sử dụng try-catch-finally?**
3. **Sự khác biệt giữa throw và throws?**
4. **Cách tạo một custom exception?**
5. **Cách sử dụng try-with-resources?**

**4. Câu hỏi kỹ năng mềm**

1. Bạn đã từng làm việc trong một dự án lớn với nhiều thành viên chưa? Bạn đã đóng góp như thế nào?
2. Làm thế nào để bạn quản lý thời gian và ưu tiên công việc khi có nhiều task cùng lúc?
3. Bạn đã từng gặp xung đột trong team chưa? Bạn đã giải quyết như thế nào?
4. Bạn đã từng xử lý sự cố trên production chưa? Hãy kể về một tình huống cụ thể.
5. Làm thế nào để bạn học một công nghệ mới?

**1. Câu hỏi nâng cao về Spring Framework**

**Spring Core**

1. Làm thế nào để tạo một custom BeanPostProcessor trong Spring? Cho ví dụ.
2. Cách sử dụng @Conditional annotation trong Spring? Khi nào nên sử dụng nó?
3. Spring FactoryBean là gì? Khi nào nên sử dụng nó?
4. Cách sử dụng @Import và @ImportResource trong Spring?
5. Làm thế nào để tạo một custom scope trong Spring? Cho ví dụ.

**Spring Boot**

1. Cách tạo một custom Spring Boot Starter?
2. Làm thế nào để tối ưu hóa hiệu suất của ứng dụng Spring Boot?
3. Cách sử dụng @ConfigurationProperties để binding dữ liệu từ file cấu hình?
4. Làm thế nào để tích hợp Spring Boot với GraphQL?
5. Cách sử dụng Spring Boot Actuator để monitor và quản lý ứng dụng?

**Spring Security**

1. Cách triển khai OAuth2 Resource Server trong Spring Security?
2. Làm thế nào để tích hợp Spring Security với Keycloak?
3. Cách sử dụng @PreAuthorize và @PostAuthorize để kiểm soát quyền truy cập?
4. Làm thế nào để triển khai Multi-Tenancy Security trong Spring Security?
5. Cách sử dụng Spring Security để bảo vệ WebSocket?

**Spring Data JPA**

1. Cách sử dụng @EntityGraph để tối ưu hóa truy vấn JPA?
2. Làm thế nào để sử dụng @Query với native query trong Spring Data JPA?
3. Cách triển khai Auditing (ghi lại thay đổi) trong Spring Data JPA?
4. Làm thế nào để sử dụng @Modifying và @Transactional trong Spring Data JPA?
5. Cách sử dụng Specification và Criteria API trong Spring Data JPA?

**2. Câu hỏi nâng cao về Spring Integration**

1. Làm thế nào để xử lý lỗi (error handling) trong Spring Integration?
2. Cách sử dụng Aggregator và Splitter trong Spring Integration?
3. Làm thế nào để tích hợp Spring Integration với Apache Kafka?
4. Cách sử dụng MessageChannel và MessageHandler để xử lý message?
5. Làm thế nào để triển khai một hệ thống event-driven sử dụng Spring Integration?

**3. Câu hỏi nâng cao về Java Core**

**Java Memory Model**

1. Java Memory Model (JMM) là gì? Giải thích về happens-before relationship.
2. Cách sử dụng volatile để đảm bảo tính nhất quán trong đa luồng?
3. Sự khác biệt giữa synchronized và ReentrantLock? Khi nào nên sử dụng từng loại?
4. Làm thế nào để tránh deadlock trong Java?
5. Cách sử dụng ThreadLocal trong Java? Khi nào nên sử dụng nó?

**Concurrency**

1. Cách sử dụng CompletableFuture để xử lý bất đồng bộ trong Java?
2. Làm thế nào để sử dụng ForkJoinPool và RecursiveTask?
3. Cách sử dụng StampedLock để tối ưu hóa hiệu suất trong đa luồng?
4. Làm thế nào để triển khai một custom ThreadPoolExecutor?
5. Cách sử dụng Phaser để đồng bộ hóa các luồng trong Java?

**Performance Tuning**

1. Làm thế nào để tối ưu hóa hiệu suất của ứng dụng Java?
2. Cách sử dụng Java Flight Recorder (JFR) để phân tích hiệu suất?
3. Làm thế nào để sử dụng Garbage Collection tuning để cải thiện hiệu suất?
4. Cách sử dụng JMH (Java Microbenchmark Harness) để đo lường hiệu suất?
5. Làm thế nào để sử dụng Profiling Tools như VisualVM hoặc YourKit?

**4. Câu hỏi nâng cao về Java Collection Framework**

1. Cách sử dụng ConcurrentHashMap để đảm bảo thread-safe?
2. Làm thế nào để triển khai một custom Comparator và Comparable?
3. Cách sử dụng Stream API để xử lý dữ liệu trong Java 8?
4. Làm thế nào để sử dụng Collectors để nhóm dữ liệu trong Stream API?
5. Cách sử dụng Spliterator để chia nhỏ và xử lý dữ liệu song song?

**5. Câu hỏi nâng cao về Hibernate**

Hibernate Core

1. Cách sử dụng @BatchSize để tối ưu hóa truy vấn trong Hibernate?
2. Làm thế nào để sử dụng @Fetch và @FetchMode để điều khiển fetching strategy?
3. Cách sử dụng @Filter để lọc dữ liệu trong Hibernate?
4. Làm thế nào để sử dụng @NamedQuery và @NamedNativeQuery trong Hibernate?
5. Cách sử dụng @Version để triển khai optimistic locking trong Hibernate?

**Performance Tuning**

1. Làm thế nào để tối ưu hóa hiệu suất của ứng dụng sử dụng Hibernate?
2. Cách sử dụng Second-Level Cache trong Hibernate?
3. Làm thế nào để sử dụng Query Cache để tối ưu hóa truy vấn?
4. Cách sử dụng Batch Processing để cải thiện hiệu suất khi xử lý dữ liệu lớn?
5. Làm thế nào để sử dụng Stateless Session trong Hibernate?

**Advanced Mapping**

1. Cách sử dụng @Inheritance để triển khai inheritance mapping trong Hibernate?
2. Làm thế nào để sử dụng @ElementCollection để mapping collections trong Hibernate?
3. Cách sử dụng @JoinTable và @JoinColumn để mapping quan hệ nhiều-nhiều?
4. Làm thế nào để sử dụng @Embeddable và @Embedded để mapping composite keys?
5. Cách sử dụng @Formula để tính toán giá trị động trong Hibernate?

**6. Câu hỏi về Design Patterns và Best Practices**

1. Làm thế nào để áp dụng Dependency Injection (DI) trong Spring một cách hiệu quả?
2. Cách sử dụng Factory Pattern để tạo đối tượng trong Spring?
3. Làm thế nào để áp dụng Proxy Pattern trong Spring AOP?
4. Cách sử dụng Builder Pattern để tạo đối tượng phức tạp trong Java?
5. Làm thế nào để áp dụng Observer Pattern trong Spring Integration?

**7. Câu hỏi về Testing**

1. Cách sử dụng Mockito để viết unit test trong Spring?
2. Làm thế nào để sử dụng Spring Boot Test để viết integration test?
3. Cách sử dụng Testcontainers để kiểm thử ứng dụng với database thật?
4. Làm thế nào để sử dụng JUnit 5 để viết unit test trong Java?
5. Cách sử dụng WireMock để mock các dịch vụ bên ngoài trong Spring Boot?
6. **Làm thế nào để đảm bảo tính nhất quán dữ liệu trong hệ thống Microservices?**

Để đảm bảo tính nhất quán dữ liệu trong hệ thống Microservices, tôi thường áp dụng các phương pháp sau:

1. **Saga Pattern**: Sử dụng Saga để quản lý các giao dịch phân tán. Mỗi service thực hiện một phần của giao dịch và gửi event để service tiếp theo xử lý. Nếu có lỗi, các compensating transaction sẽ được thực hiện để rollback.
2. **Eventual Consistency**: Chấp nhận tính nhất quán cuối cùng bằng cách sử dụng message queue (như RabbitMQ hoặc Kafka) để đồng bộ dữ liệu giữa các service.
3. **Distributed Locking**: Sử dụng distributed lock (ví dụ: Redis Lock) để đảm bảo chỉ một service có thể thay đổi dữ liệu tại một thời điểm.
4. **Database per Service**: Mỗi service có database riêng để tránh tranh chấp dữ liệu.
5. **Bạn hiểu gì về kiến trúc event-driven? Làm thế nào để xử lý message loss trong RabbitMQ?**

Kiến trúc event-driven là một mô hình trong đó các service giao tiếp thông qua events (sự kiện). Khi một service thực hiện một hành động, nó sẽ publish một event, và các service khác subscribe để xử lý event đó.

Để xử lý message loss trong RabbitMQ, tôi thường áp dụng các phương pháp sau:

**+ Message Acknowledgment**: Đảm bảo consumer gửi acknowledgment sau khi xử lý message thành công.

**+ Persistent Messages**: Đánh dấu message là persistent để RabbitMQ lưu message vào disk, tránh mất dữ liệu khi server crash.

**+ Dead Letter Exchange (DLX):** Sử dụng DLX để xử lý các message không thể xử lý được sau một số lần retry.

**+ Monitoring và Alerts**: Sử dụng công cụ như Prometheus và Grafana để theo dõi hệ thống và cảnh báo sớm khi có sự cố.

1. **Làm thế nào để thiết kế một API RESTful tốt?**

**RESTful Principles**: Sử dụng đúng HTTP methods (GET, POST, PUT, DELETE) và status codes (200, 201, 400, 404, 500).

**Versioning**: Sử dụng versioning (ví dụ: /api/v1/resource) để đảm bảo tương thích ngược.

**Documentation**: Sử dụng Swagger để tạo tài liệu API chi tiết và dễ hiểu.

**Security**: Áp dụng OAuth2 hoặc JWT để bảo mật API.

**Pagination và Filtering**: Hỗ trợ pagination và filtering để tối ưu hiệu suất.

Thiết kế một hệ thống có thể chịu được **150 triệu request/ngày** (khoảng **1,736 request/giây**) đòi hỏi phải áp dụng các nguyên tắc kiến trúc hiệu suất cao, khả năng mở rộng (scalability), và độ tin cậy (reliability).

**1. Đảm bảo perfomance**

**Cân bằng tải với Load Balancer**: Load Balancer là một thiết bị (phần cứng hoặc phần mềm) cho phép cân bằng tải đến nhiều server.

A diagram of a network

AI-generated content may be incorrect.

**Phân tán dữ liệu với Content Delivery Network (CDN):** CDN là một mạng lưới các server được phân bố trên nhiều khu vực.

**Caching**: Cache là một kĩ thuật để tăng tốc độ đọc dữ liệu, bằng cách lưu dữ liệu sẵn vào cache server để tăng tốc độ đọc ở những lần sau.

**2. Đảm bảo Availability**

**Master/Slave:** Thay vì chỉ chạy 1 server, ta chạy 2 hoặc nhiều hơn. 1 server chính gọi là master, các server còn lại là slave. Khi master có vấn đề (sập nguồn hay crash), một slave sẽ được chỉ định để lên thay thế master.

**Replication**: Thường được kết hợp chung với Load Balancer. Code của ứng dụng sẽ được deploy lên nhiều server. Khi có 1 server die, load balancer sẽ chuyển request sang server khác, đảm bảo request vẫn được thực hiện.

**3. Đảm bảo scalability**

**Vertical Scaling**: Tăng sức mạnh phần cứng cho server

**Horizontal Scaling**: Thay vì tăng sức mạnh cho 1 server, ta thêm nhiều server vào hệ thống và chạy cùng lúc.

1. **Bộ nhớ trong Java**

**4.1 Bộ nhớ Stack (Stack Memory)**

**Mục đích:** Lưu trữ các biến cục bộ (local variables), tham số của phương thức, và các lời gọi phương thức (method calls).

**Đặc điểm:**

Hoạt động theo cơ chế LIFO (Last In First Out).

Mỗi luồng (thread) có một stack riêng.

Kích thước nhỏ hơn so với Heap.

Tốc độ truy cập nhanh hơn Heap.

Bộ nhớ được giải phóng tự động khi phương thức kết thúc.

**4.** **2. Bộ nhớ Heap (Heap Memory)**

**Mục đích**: Lưu trữ các đối tượng (objects) và biến instance (instance variables).

**Đặc điểm:**

Tất cả các đối tượng được tạo bằng từ khóa new đều được lưu trong Heap.

Heap được chia sẻ giữa tất cả các luồng (threads).

Kích thước lớn hơn so với Stack.

Tốc độ truy cập chậm hơn Stack.

Bộ nhớ được quản lý bởi Garbage Collector (GC), tự động giải phóng khi không còn tham chiếu đến đối tượng.

Bộ nhớ Java

├── Stack Memory

├── Heap Memory

│ ├── String Pool

│ └── Các đối tượng

├── Method Area (Metaspace)

│ ├── Runtime Constant Pool

│ └── Thông tin lớp, biến static

├── PC Register

└── Native Method Stack

|  |  |  |
| --- | --- | --- |
| **Loại bộ nhớ** | **Mục đích chính** | **Vị trí lưu trữ** |
| **Stack** | Biến cục bộ, lời gọi phương thức | Bộ nhớ riêng cho mỗi luồng |
| **Heap** | Đối tượng, biến instance | Bộ nhớ chia sẻ |
| **Method Area (Metaspace)** | Thông tin lớp, biến static, hằng số | Bộ nhớ native (từ Java 8) |
| **PC Register** | Địa chỉ lệnh hiện tại | Bộ nhớ riêng cho mỗi luồng |
| **Native Method Stack** | Phương thức native | Bộ nhớ riêng cho mỗi luồng |
| **String Pool** | Lưu trữ chuỗi literal | Nằm trong Heap |
| **Runtime Constant Pool** | Lưu trữ hằng số | Nằm trong Method Area |