**INTERVIER JAVA**

# JAVA CORE

## Java vừa là ngôn ngữ biên dịch, vừa là ngôn ngữ thông dịch

* Biên dịch từ file code (.java) sang file .class
* Dùng máy ảo JVM thông dịch từ file .class sang ngôn ngữ máy

## OOP

* Đóng gói: private in class
* Trừu tượng:
* Interface: chỉ chứa Abstract Class
* Abstract: chứa Abstract Class và non Abstract Class (phương thức trừu tượng & class)
* Kế thừa:
* Extend: kế thừa từ Class
* Implement: kế từ từ Interface
* Đa hình:
  + Overloading (nạp chồng): (add (int a, int b))
  + Overriding (ghi đè): extend class

## Sự khác nhau giữa abtract và interface

Nhắc đến **Interface** và **abstract class** hãy nhớ 2 từ này khá clear rồi, Khi sử dụng **Interface** là bạn **Implement** còn sử dụng **abstract class** là bạn **extend**

* **Interface:**
  + Không phải là class.
  + Chỉ chứa những method/properties trống không có thực thi.
  + Nó giống như một khuôn mẫu, một khung để để các lớp implement và follow.
  + Các lớp có thể implements nhiều interface.
  + Là một contract, các class implement phải triển khai các method theo như interface đã định nghĩa.
* **Abstract class:**
  + Khá giống Interface nhưng nó có thể làm nhiều việc hơn.
  + Có 2 loại method là abstract method và method thường:
    - abstract method là method trống không có thực thi.
    - method thường là method có thực thi.
  + Các lớp chỉ có thể kế thừa một Abstract class
  + Hướng đến tính năng và những tính năng có thực thi được sử dụng làm hàm chung cho các class extend.

**Ưu điểm:**

Interface

Có thể kế thừa nhiều interface(tính đa hình).

Xây dựng được bộ khung mẫu mà các lớp phải follow theo.

Giúp quản lý tốt, nắm bắt được các chức năng phải có cho một đối tượng nào đó.

Abstract class

Có thể linh động các method. giống như một class thông thường.

Các class extend có thể override hoặc không override các method thường.

**Nhược điểm:**

Interface:

Mỗi khi định nghĩa thêm tính năng, các class impelement nó đồng lọat phải thêm tính năng đó, khả năng cao sẽ không có xử lý gì.

Abstract class

Không thể extend nhiều abstract class.

## Những collection hay sử dụng trong java

1. ArrayList: ArrayList là một danh sách có thể mở rộng, được triển khai bằng mảng. Nó cung cấp các phương thức để thêm, xóa, truy xuất và sắp xếp các phần tử. ArrayList cho phép các phần tử trùng lặp và hỗ trợ truy xuất ngẫu nhiên.
2. LinkedList: LinkedList là một danh sách liên kết, mỗi phần tử có tham chiếu đến phần tử tiếp theo trong danh sách. Nó cũng cung cấp các phương thức để thêm, xóa, truy xuất và sắp xếp các phần tử. LinkedList tốt hơn ArrayList trong việc thêm/xóa phần tử ở giữa danh sách, nhưng truy cập ngẫu nhiên chậm hơn.
3. HashSet: HashSet là một tập hợp không có thứ tự và không cho phép các phần tử trùng lặp. Nó lưu trữ các phần tử dựa trên giá trị băm (hash) của chúng, do đó, việc truy xuất các phần tử nhanh chóng và hiệu quả.
4. TreeSet: TreeSet là một tập hợp được sắp xếp theo thứ tự tăng dần hoặc giảm dần. Nó sử dụng cấu trúc dữ liệu cây đỏ-đen để duy trì sự sắp xếp của các phần tử. TreeSet không cho phép các phần tử trùng lặp và cung cấp các phương thức để thêm, xóa và truy xuất phần tử theo thứ tự.
5. HashMap: HashMap lưu trữ dữ liệu theo cặp khóa-giá trị (key-value). Nó không có thứ tự và không cho phép khóa trùng lặp. HashMap sử dụng hàm băm để xác định vị trí lưu trữ và cung cấp các phương thức để thêm, xóa và truy xuất giá trị thông qua khóa.
6. TreeMap: TreeMap là một ánh xạ được sắp xếp theo thứ tự tăng dần hoặc giảm dần của các khóa. Nó sử dụng cấu trúc dữ liệu cây đỏ-đen để duy trì sự sắp xếp của các phần tử. TreeMap không cho phép các khóa trùng lặp và cung cấp các phương thức để thêm, xóa và truy xuất giá trị theo thứ tự.

## Sự khác nhau giữa LinkedList và ArrayList

1. Cấu trúc dữ liệu:

* ArrayList: ArrayList được triển khai bằng mảng, mỗi phần tử được lưu trữ trong một vị trí cố định trong mảng. Khi mảng đầy, ArrayList có thể mở rộng kích thước của nó tự động.
* LinkedList: LinkedList được triển khai bằng danh sách liên kết, trong đó mỗi phần tử lưu trữ dữ liệu và một tham chiếu đến phần tử tiếp theo. Mỗi phần tử trong LinkedList được gọi là "nút".

1. Hiệu suất:

* Truy cập ngẫu nhiên: ArrayList hỗ trợ truy cập ngẫu nhiên nhanh hơn LinkedList. Vì ArrayList được triển khai bằng mảng, việc truy cập vào phần tử bằng chỉ số là hiệu quả O(1). Trong khi đó, LinkedList cần phải truy cập tuần tự từ đầu đến vị trí cần truy xuất, do đó, truy cập ngẫu nhiên chậm hơn với độ phức tạp O(n).
* Thêm và xóa phần tử: LinkedList có hiệu suất tốt hơn trong việc thêm và xóa phần tử ở giữa danh sách. Vì LinkedList chỉ cần cập nhật các tham chiếu giữa các nút, việc thêm và xóa phần tử là hiệu quả O(1). Trong khi đó, ArrayList cần phải dịch chuyển các phần tử sau vị trí thêm hoặc xóa, gây ra độ phức tạp O(n).
* Sử dụng bộ nhớ: ArrayList cần một khối lượng bộ nhớ liền kề để lưu trữ các phần tử, kích thước của nó được mở rộng khi cần thiết. Trong khi đó, LinkedList sử dụng một lượng bộ nhớ lớn hơn do việc lưu trữ tham chiếu đến các nút.

1. Sử dụng:

* ArrayList thường được sử dụng khi cần truy cập ngẫu nhiên vào phần tử, và việc thêm/xóa phần tử không phổ biến hoặc xảy ra ở cuối danh sách.
* LinkedList thường được sử dụng khi cần thêm/xóa phần tử ở giữa danh sách thường xuyên, trong khi truy cập ngẫu nhiên không quan trọng.

Lựa chọn giữa LinkedList và ArrayList phụ thuộc vào các yêu cầu cụ thể của ứng dụng. Nếu bạn cần truy cập ngẫu nhiên nhanh chóng và việc thêm/xóa phần

## Một rest api bao gồm những thông tin gì

1. Địa chỉ URL (Endpoint): REST API sử dụng các đường dẫn URL (Uniform Resource Locator) để xác định tài nguyên và hành động liên quan đến tài nguyên đó. Địa chỉ URL cho biết đường dẫn đến tài nguyên API và xác định hành động mà người dùng muốn thực hiện.
2. Phương thức HTTP: REST API sử dụng các phương thức HTTP như GET, POST, PUT, DELETE để xác định các hành động được thực hiện trên tài nguyên. Ví dụ: GET để truy xuất thông tin, POST để tạo mới tài nguyên, PUT để cập nhật tài nguyên, DELETE để xóa tài nguyên.
3. Định dạng dữ liệu: REST API sử dụng các định dạng dữ liệu phổ biến như JSON (JavaScript Object Notation) hoặc XML (eXtensible Markup Language) để truyền tải dữ liệu giữa máy khách và máy chủ. Định dạng dữ liệu cho biết cách dữ liệu được đóng gói và phân tích.
4. Thông tin xác thực (Authorization): REST API thường yêu cầu các phương thức xác thực để bảo mật và kiểm soát quyền truy cập vào tài nguyên. Thông tin xác thực có thể được truyền qua tiêu đề (header) của yêu cầu HTTP, ví dụ: mã thông báo truy cập (access token) hoặc chứng chỉ xác thực (authentication certificate).
5. Thông tin tham số (Parameters): REST API có thể yêu cầu các tham số bổ sung để thực hiện hành động cụ thể. Các tham số này có thể được truyền dưới dạng tham số trên URL (query parameters) hoặc dưới dạng tham số trong phần thân yêu cầu (request body).
6. Phản hồi (Response): REST API trả về phản hồi từ máy chủ cho máy khách sau khi thực hiện yêu cầu. Phản hồi thường chứa mã trạng thái HTTP để chỉ ra kết quả của yêu cầu (ví dụ: 200 OK, 404 Not Found), cùng với dữ liệu trả về (nếu có) được đóng gói trong định dạng dữ liệu đã chỉ định (JSON, XML, ...).

Tùy thuộc vào yêu cầu cụ thể của REST API, có thể có thêm các thông tin bổ sung như phiên bản API, các tiêu đề tùy chọn (header), giới hạn tốc độ (rate limiting), và các quy định bảo mật khác. T

## Muốn lọc dữ liệu từ mảng thì làm như nào

1. Sử dụng vòng lặp: Bạn có thể sử dụng vòng lặp để duyệt qua từng phần tử trong mảng và kiểm tra điều kiện để lọc dữ liệu. Nếu phần tử thỏa mãn điều kiện, bạn có thể thêm nó vào một danh sách hoặc mảng mới.

2.Sử dụng Stream API: Trong Java 8 và các phiên bản sau đó, bạn có thể sử dụng Stream API để lọc dữ liệu từ mảng. Stream API cung cấp các phương thức như **filter()** để chọn lọc các phần tử theo điều kiện và **collect()** để thu thập kết quả vào một danh sách hoặc mảng mới.

1. Sử dụng các thư viện bên thứ ba: Ngoài các phương pháp trên, bạn cũng có thể sử dụng các thư viện bên thứ ba như Apache Commons Collections hoặc Guava để lọc dữ liệu từ mảng.

## String, String builder & String buffer

* Cơ bản về 3 class này như sau:
* String là không thể thay đổi (immutable), và không cho phép có class con.
* StringBuffer, StringBuilder có thể thay đổi (mutable)
* StringBuilder và StringBuffer là giống nhau, nó chỉ khác biệt tình huống sử dụng có liên quan tới đa luồng (Multi Thread). => về tốc độ xử lý StringBuilder là tốt nhất, sau đó StringBuffer và cuối cùng mới là String.

## Delete và Truncate trong database

* DELETE : Xóa một hay tất cả dòng trong một bảng theo một điều kiện nhất định, dữ liệu có thể phục hồi lại
  + Delete là một lệnh SQL được sử dụng để xóa các bản ghi (rows) cụ thể từ bảng dữ liệu.
  + Delete hoạt động bằng cách xác định một điều kiện (condition) để tìm kiếm và xóa các bản ghi thỏa mãn điều kiện đó.
  + Delete chỉ xóa các bản ghi được xác định và không thay đổi cấu trúc bảng.
  + Delete ghi log việc xóa từng bản ghi, điều này có thể tạo ra lưu lượng log lớn và tốn tài nguyên.
* TRUNCATE : Xóa toàn bộ các dòng của bảng, giải phóng bộ nhớ và không thể phục hồi lại
  + Truncate là một lệnh SQL được sử dụng để xóa toàn bộ nội dung của một bảng.
  + Truncate không xác định điều kiện và xóa tất cả các bản ghi trong bảng.
  + Truncate thực hiện việc xóa nhanh hơn delete vì nó không ghi log việc xóa từng bản ghi và không cần quét qua từng bản ghi để xóa.
  + Truncate có thể thiết lập lại (reset) giá trị các trường tự tăng (auto-increment) về giá trị ban đầu.
  + Truncate là một hoạt động không phục hồi, có nghĩa là dữ liệu đã bị xóa không thể khôi phục lại sau khi thực hiện lệnh truncate.
* DROP : Xóa một bảng khỏi database

## Khác nhau giữa stored proceduce và query thông thường

**Query thông thường:**

* Query thông thường là một câu truy vấn SQL (như SELECT, INSERT, UPDATE, DELETE) được gọi từ một ứng dụng hoặc giao diện người dùng để truy xuất hoặc thay đổi dữ liệu trong cơ sở dữ liệu.
* Query thông thường được thực thi ngay lập tức khi được gọi và trả về kết quả tương ứng (nếu có) cho ứng dụng hoặc giao diện người dùng.
* Query thông thường không được lưu trữ trong cơ sở dữ liệu và thường được viết và thực thi tại thời điểm cần sử dụng.

**Stored Procedure:**

* Stored procedure là một khối mã SQL được lưu trữ trong cơ sở dữ liệu.
* Stored procedure thường là một tập hợp các câu truy vấn và logic xử lý phức tạp hơn, được sử dụng để thực hiện các tác vụ cụ thể hoặc tương tác với dữ liệu trong cơ sở dữ liệu.
* Stored procedure được lưu trữ một lần trong cơ sở dữ liệu và có thể được gọi nhiều lần từ các ứng dụng khác nhau.
* Stored procedure có thể chứa các tham số, điều kiện logic, vòng lặp và các câu truy vấn phức tạp hơn để thực hiện các tác vụ đa dạng.
* Stored procedure có thể trả về kết quả hoặc chỉ thực hiện một công việc nội bộ trong cơ sở dữ liệu.

Lưu ý rằng sự khác nhau chính giữa stored procedure và query thông thường nằm trong tính chất của chúng. Stored procedure được lưu trữ một lần và có thể được gọi nhiều lần từ các ứng dụng khác nhau, trong khi query thông thường được viết và thực thi ngay lập tức khi cần sử dụng. Stored procedure thường được sử dụng trong các tình huống yêu cầu xử lý logic phức tạp hoặc thao tác với dữ liệu trên phía máy chủ cơ sở dữ liệu.

## JDK, JVM, JRE

Dưới đây là sự phân biệt giữa JDK, JRE và JVM:

JDK (Java Development Kit): JDK là viết tắt của Java Development Kit. Nó là một bộ công cụ phát triển Java đầy đủ, cung cấp các công cụ và tài liệu cần thiết để phát triển, biên dịch, thực thi và gỡ lỗi ứng dụng Java. JDK bao gồm Java Runtime Environment (JRE), các công cụ phát triển như trình biên dịch (javac), trình gỡ lỗi (jdb), trình tạo tài liệu (javadoc) và các thư viện hỗ trợ phát triển Java.

JRE (Java Runtime Environment): JRE là viết tắt của Java Runtime Environment. Nó cung cấp môi trường chạy ứng dụng Java. JRE bao gồm Java Virtual Machine (JVM), các thư viện cần thiết và các file runtime khác để thực thi ứng dụng Java. Nếu bạn chỉ muốn chạy ứng dụng Java đã được biên dịch, bạn chỉ cần cài đặt JRE.

JVM (Java Virtual Machine): JVM là viết tắt của Java Virtual Machine. Nó là môi trường ảo trong đó các ứng dụng Java chạy. JVM là trung gian giữa ứng dụng Java và hệ điều hành, nó thực hiện việc biên dịch bytecode thành mã máy và quản lý bộ nhớ, luồng thực thi và các tài nguyên khác cho ứng dụng Java. JVM cung cấp độc lập với nền tảng, cho phép ứng dụng Java chạy trên nhiều hệ điều hành khác nhau mà không cần thay đổi mã nguồn.

Tóm lại:

* JDK là bộ công cụ phát triển Java, bao gồm JRE và các công cụ phát triển.
* JRE cung cấp môi trường chạy ứng dụng Java.
* JVM là môi trường ảo trong đó các ứng dụng Java chạy.

## Throwable

* + **Error** (lỗi mạng, lỗi phần cứng,..)
  + **Unchecked error:** NullPoint, SQL
  + **Checked error**
* Runtime Exception
* Syntax

## Thread, runnable

1. **Thread:**

* Thread là một lớp trong Java được sử dụng để tạo và quản lý một luồng thực thi.
* Để sử dụng Thread, bạn có thể tạo một lớp con kế thừa từ lớp Thread và ghi đè phương thức run() để định nghĩa công việc mà luồng sẽ thực hiện.
* Bạn có thể tạo một đối tượng Thread từ lớp con và gọi phương thức start() để bắt đầu thực thi luồng. Phương thức start() sẽ tự động gọi phương thức run() đã được định nghĩa trong lớp con.

1. **Runnable:**

* Runnable là một giao diện trong Java được sử dụng để tạo công việc cho một luồng thực thi.
* Để sử dụng Runnable, bạn có thể tạo một lớp mới implement Runnable và triển khai phương thức run() để định nghĩa công việc mà luồng sẽ thực hiện.
* Bạn cần tạo một đối tượng Thread và truyền đối tượng Runnable vào constructor của Thread.
* Sau đó, bạn có thể gọi phương thức start() của đối tượng Thread để bắt đầu thực thi luồng.

Khi tạo và chạy một luồng, quy trình chạy đa luồng sẽ được quản lý bởi JVM và có thể chạy đồng thời với các luồng khác. Việc sử dụng đa luồng có thể giúp tận dụng tối đa tài nguyên và cải thiện hiệu suất của ứng dụng trong các tình huống cần xử lý song song hoặc chia sẻ tài nguyên.

# SPRING BOOT

## Maven

* Thêm tập hợp các công nghệ Spring Hibernate
* Tạo cấu trúc dự án phù hợp

## JPA

* Cung cấp mô hình ánh xạ các Table sang Class

**Hibernate & JDBC**

[*https://hocspringboot.net/2021/05/10/tai\_sao\_nen\_dung\_hibernate\_thay\_vi\_jdbc/*](https://hocspringboot.net/2021/05/10/tai_sao_nen_dung_hibernate_thay_vi_jdbc/)

* Map giữa Object sang CSDL
* JDBC: Map giữa các trường trong bảng với các thuộc tính cách truyền thống
* Hibernate: Map tự động thông qua các annotation

## DI, IOC trong spring

* **Dependency Injection (DI):** DI là một mô hình thiết kế và một khái niệm trong Spring, nơi các đối tượng không tạo ra và quản lý các phụ thuộc của chúng mà được cung cấp bởi một bên thứ ba. Thay vì tạo và quản lý các đối tượng phụ thuộc, đối tượng chỉ quan tâm đến việc sử dụng các phụ thuộc mà nó nhận được. Điều này giúp giảm sự ràng buộc giữa các đối tượng và làm cho mã dễ dàng bảo trì, kiểm thử và mở rộng.

Spring Framework cung cấp một cách thức để thực hiện DI thông qua các công cụ như Constructor Injection, Setter Injection và Field Injection. Bằng cách sử dụng các annotation như **@Autowired**, **@Inject** hoặc **@Resource**, bạn có thể chỉ định rằng Spring nên tự động tiêm phụ thuộc vào các thành phần.

* **Inversion of Control (IoC):** IoC là một nguyên tắc trong lập trình và một phần quan trọng của Spring. Theo nguyên tắc này, quyền kiểm soát luồng của ứng dụng không nằm trong tay lập trình viên mà được chuyển giao cho một bên thứ ba (thường là framework). Điều này có nghĩa là lập trình viên không cần phải tạo ra và quản lý các đối tượng, mà chỉ cần chỉ định cách sử dụng các đối tượng thông qua cấu hình hoặc metadata.

Spring Framework thực hiện IoC bằng cách sử dụng một bộ điều khiển (container) gọi là ApplicationContext. ApplicationContext quản lý việc tạo và quản lý các đối tượng và quản lý việc tiêm phụ thuộc vào các đối tượng theo cách đã được cấu hình.

Kết hợp với DI, Spring Framework áp dụng IoC bằng cách tạo và quản lý các đối tượng trong ứng dụng và tự động tiêm phụ thuộc vào các đối

**IOC:**

* Trình quản lý bean, đối tượng. Lớp bao bọc tất cả các đối tượng
* Không tạo đối tượng, chỉ mô tả cách tạo ra chúng
* @Componet =IoC=> tạo 1 Object duy nhất (Singleton)
* @Awtowide => Tìm 1 module (tạo trước) để Inject vào

**DI - Dependency Injection**

* **Các module phụ thuộc (dependency) sẽ được inject vào module cấp cao.**
* Giảm sự phụ thuộc vào các Object (nên phụ thuộc vào Abstract Class => Inject vào lúc runtime)
* Spring sử dụng Annotation như @Componet => đánh dấu Class => Class đó tạo 1 module
* Dependency Injection là việc các Object nên phụ thuộc vào các Abstract Class và thể hiện chi tiết của nó sẽ được Inject vào đối tượng lúc runtime.

## Vòng đời của bean

Trong Spring, một bean (đối tượng) được quản lý bởi Spring IoC container. Bean trong Spring có một vòng đời (lifecycle) từ khi nó được tạo ra cho đến khi nó bị hủy bỏ. Dưới đây là các giai đoạn trong vòng đời của một bean trong Spring:

1. Instantiation (Khởi tạo): Bean được khởi tạo bằng cách sử dụng constructor hoặc factory method.
2. Populate properties (Gán giá trị cho thuộc tính): Spring gán giá trị cho các thuộc tính của bean bằng cách sử dụng Dependency Injection (DI) hoặc autowiring.
3. BeanNameAware (Nếu áp dụng): Bean được thông báo về tên của nó trong trường hợp cần thiết.
4. BeanFactoryAware hoặc ApplicationContextAware (Nếu áp dụng): Bean được thông báo về BeanFactory hoặc ApplicationContext trong trường hợp cần thiết.
5. Pre-initialization (Tiền khởi tạo): Bean có thể thực hiện các xử lý tiền khởi tạo trước khi nó sẵn sàng sử dụng. Ví dụ: Implementing InitializingBean interface, sử dụng @PostConstruct annotation hoặc cấu hình XML.
6. Initialization (Khởi tạo): Bean được khởi tạo và sẵn sàng sử dụng. Các xử lý khởi tạo bổ sung có thể được thực hiện ở đây.
7. Post-initialization (Sau khởi tạo): Bean có thể thực hiện các xử lý sau khởi tạo bổ sung. Ví dụ: Implementing BeanPostProcessor interface, sử dụng @PostConstruct annotation hoặc cấu hình XML.
8. Ready to use (Sẵn sàng sử dụng): Bean sẵn sàng được sử dụng bởi ứng dụng.
9. Destruction (Huỷ): Khi ứng dụng kết thúc hoặc bean bị hủy bỏ, bean có thể thực hiện các xử lý huỷ bổ sung. Ví dụ: Implementing DisposableBean interface, sử dụng @PreDestroy annotation hoặc cấu hình XML.
10. Destroyed (Đã huỷ): Bean đã bị hủy bỏ và không còn sử dụng.

Qua các giai đoạn trên, Spring IoC container quản lý vòng đời của các bean, cho phép chúng được tạo, cấu hình và hủy bỏ một cách linh hoạt và tự động.

## Annonation & Reflection

**Annotation**

* Đặc tả cho 1 đối tượng, giá trị nào đó (thực chất là 1 Interface - khai báo field khi khởi tạo)

**Reflection**

* Thao tác dữ liệu không thể lấy được
* Thường sử dụng để Mapping data

**@Autowire**

* Tạo các Bean được Spring sinh ra vào các Class được khai báo @Awtowide
* Cơ chế:
* Quét các lớp Annotation => tạo Bean
* Tìm trong các Bean có Autowire không => tìm kiếm Bean tương ứng => Inject

**@Configuration**

* Khai báo 1 hoặc nhiều Bean method trong class đó

**@Bean**

* Báo cho Spring Container => Sinh ra các Bean => Quản lý Spring Container

**=> Phải nằm trong Class @Configuration**

**@Component**

* Tạo 1 bean => Inject lớp nào cần dùng nó
* **Không thể Autowire nếu không dùng Component**
* **Khi muốn xác định 1 lớp để Inject vào => dùng @Component**

**@Transaction**

* Thay đổi dữ liệu trong SQL
* Khi giao tiếp với SQL, nếu lỗi => RollBack

**@Request Mapping**

* Map request đến Class xử lý request đó

**@PathVariable** là annotation để lấy tham số là một thành phần của đường dẫn **giúp bạn trích xuất thông tin trực tiếp từ URL.**

**@Request Param** là annotation để lấy tham số là biến trong query string

**@RequestBody** dùng để hứng dữ liệu dạng JSON (post request dạng JSON)

**@Request Mapping (HTTP Method)**

* Chỉ ra loại phương thức HTTP yêu cầu

## Scope trong spring

Trong Spring, có năm phạm vi (scope) chính cho các Bean, bao gồm Singleton, Prototype, Request, Session và Global Session. Dưới đây là sự khác biệt giữa chúng:

1. Singleton:

* Singleton là phạm vi mặc định trong Spring. Mỗi Bean được định nghĩa với phạm vi Singleton sẽ chỉ có một phiên bản duy nhất trong Container.
* Khi một Bean được khởi tạo, Spring Container sẽ tạo một phiên bản duy nhất của Bean và sử dụng nó cho tất cả các yêu cầu được yêu cầu.
* Điều này có nghĩa là mọi Bean khác yêu cầu cùng một Bean Singleton sẽ nhận được tham chiếu đến cùng một đối tượng.

1. Prototype:

* Phạm vi Prototype cho phép mỗi yêu cầu tạo Bean sẽ tạo ra một phiên bản mới của Bean.
* Mỗi lần Bean được yêu cầu, Spring Container sẽ tạo ra một đối tượng mới.
* Điều này có nghĩa là mọi Bean khác yêu cầu cùng một Bean Prototype sẽ nhận được các đối tượng riêng biệt.

1. Request:

* Phạm vi Request chỉ áp dụng trong môi trường web. Mỗi yêu cầu HTTP sẽ tạo ra một phiên bản mới của Bean.
* Mỗi khi một yêu cầu được gửi đến ứng dụng, Spring Container sẽ tạo một đối tượng Bean mới và liên kết nó với yêu cầu hiện tại.
* Điều này đảm bảo rằng mỗi yêu cầu sẽ có một đối tượng Bean riêng biệt.

1. Session:

* Phạm vi Session chỉ áp dụng trong môi trường web. Mỗi phiên làm việc (session) sẽ có một phiên bản riêng của Bean.
* Khi một phiên làm việc được tạo, Spring Container sẽ tạo một đối tượng Bean mới và liên kết nó với phiên làm việc hiện tại.
* Điều này đảm bảo rằng mỗi phiên làm việc sẽ có một đối tượng Bean riêng biệt.

1. Global Session:

* Phạm vi Global Session cũng chỉ áp dụng trong môi trường web. Nó tương tự như phạm vi Session, tuy nhiên, nó chỉ áp dụng cho các ứng dụng sử dụng mô hình portlet.
* Mỗi phiên làm việc portlet sẽ có một phiên bản riêng của Bean.

## Transaction Spring

Spring cung cấp một số cách hỗ trợ xử lý giao dịch trong ứng dụng, trong đó phổ biến nhất là sử dụng Transaction Management. Đây là một tính năng mạnh mẽ của Spring giúp quản lý và đồng bộ hóa các hoạt động giao dịch trong môi trường đa luồng và đảm bảo tính nhất quán và an toàn của dữ liệu. Spring hỗ trợ hai phương pháp chính để xử lý giao dịch: Annotation-based và Programmatic.

1. Annotation-based Transaction Management:

* Spring cho phép bạn sử dụng các Annotation như **@Transactional** để áp dụng giao dịch trên phương thức hoặc lớp.
* Bằng cách đánh dấu phương thức hoặc lớp với **@Transactional**, Spring sẽ tự động quản lý việc bắt đầu, commit và rollback giao dịch.
* Bạn cũng có thể sử dụng các Annotation khác như **@Transactional(propagation = Propagation.REQUIRED)** để chỉ định cách giao dịch được truyền từ các phương thức gọi đến.

1. Programmatic Transaction Management:

* Spring cung cấp các API để quản lý giao dịch bằng cách sử dụng **PlatformTransactionManager**.
* Bằng cách sử dụng API này, bạn có thể bắt đầu, commit và rollback giao dịch một cách tường minh.
* Điều này cho phép bạn có kiểm soát cao hơn và linh hoạt hơn trong việc quản lý giao dịch trong trường hợp cần thực hiện các logic phức tạp hơn.

ACID là một thuật ngữ sử dụng trong cơ sở dữ liệu để mô tả các tính chất cần được đảm bảo trong một giao dịch đúng và an toàn. ACID là viết tắt của:

1. Atomicity (Tính nguyên tử): Giao dịch được xem như một thao tác đơn lẻ và hoàn toàn thành công hoặc thất bại. Nếu bất kỳ phần nào của giao dịch gặp lỗi, toàn bộ giao dịch sẽ bị rollback và không có thay đổi nào được lưu trữ.
2. Consistency (Tính nhất quán): Giao dịch đảm bảo rằng dữ liệu trong cơ sở dữ liệu sẽ ở trạng thái nhất quán trước và sau giao dịch. Nghĩa là, dữ liệu phải tuân thủ các ràng buộc và quy tắc được định nghĩa trước.
3. Isolation (Tính độc lập): Giao dịch diễn ra một cách độc lập, không bị ảnh hưởng bởi các giao dịch khác đang xử lý cùng một thời điểm. Điều này đảm bảo rằng kết quả của mỗi giao dịch là như mong đợi.
4. Durability (Tính bền vững): Khi một giao dịch đã được xác nhận (committed), kết quả của nó sẽ được lưu trữ vĩnh viễn và không bị mất đi trong trường hợp sự cố hệ thống.

Spring Transaction Management giúp đảm bảo tính ACID trong quá trình xử lý giao dịch trong ứng dụng, đồng thời cung cấp cơ chế linh hoạt để quản lý và điều khiển giao dịch theo yêu cầu cụ thể của ứng dụng.

### Ghi log khi rollback

Để kiểm tra log khi một giao dịch bị rollback trong Spring, bạn có thể sử dụng các tính năng logging của Spring và cấu hình hệ thống logging một cách tối ưu. Dưới đây là một số bước để thực hiện điều này:

1. Cấu hình hệ thống logging:

* Spring sử dụng các thư viện logging phổ biến như Logback, Log4j hoặc JDK Logging.
* Tùy thuộc vào thư viện logging mà bạn sử dụng, hãy cấu hình đúng file cấu hình tương ứng (ví dụ: logback.xml, log4j.properties, logging.properties).
* Đảm bảo rằng mức log của giao dịch và rollback được thiết lập sao cho thông tin cần thiết được ghi log.

1. Sử dụng Annotation @Transactional:

* Đảm bảo rằng phương thức hoặc lớp mà bạn muốn theo dõi log khi rollback được đánh dấu bằng Annotation @Transactional.
* Sử dụng thuộc tính **rollbackFor** của @Transactional để chỉ định các ngoại lệ mà bạn muốn giao dịch rollback khi chúng xảy ra.

### Function A có các function con, 1 function con lỗi thì A có rollback không. Có cách nào để A không roolback không

Mặc định là rollback

1. Sử dụng Annotation **@Transactional** với propagation NOT\_SUPPORTED:

* Đánh dấu phương thức A bằng **@Transactional(propagation = Propagation.NOT\_SUPPORTED)**.
* Điều này sẽ loại bỏ việc truyền giao dịch từ phương thức gọi vào phương thức con, và khi phương thức con gây ra lỗi, giao dịch sẽ không bị rollback.

## AOP

Spring AOP (Aspect-Oriented Programming) là một khái niệm quan trọng trong Spring Framework, cho phép tách biệt các quy tắc cắt (cross-cutting concerns) khỏi logic chính của ứng dụng. Dưới đây là một hướng dẫn cơ bản về cách sử dụng Spring AOP:

1. Định nghĩa Aspect:

* Một Aspect là một đối tượng chứa các logic cắt (cross-cutting logic), chẳng hạn như logging, kiểm tra quyền truy cập, caching, hay ghi nhớ thời gian thực hiện.
* Bạn có thể định nghĩa Aspect bằng cách sử dụng Annotation **@Aspect**.

1. Xác định các Join Point:

* Join Point là các điểm trong mã ứng dụng nơi logic cắt sẽ được áp dụng. Ví dụ: phương thức, lời gọi phương thức, tạo đối tượng, hay xử lý ngoại lệ.
* Bạn có thể xác định các Join Point bằng cách sử dụng Pointcut expressions, một biểu thức mà xác định các điểm cắt.

1. Định nghĩa Advice:

* Advice đại diện cho các hành động được thực hiện bởi Aspect tại các Join Point cụ thể. Có các loại Advice như Before, After, AfterReturning, AfterThrowing và Around.
* Bạn có thể định nghĩa Advice bằng cách sử dụng các Annotation tương ứng như **@Before**, **@After**, **@AfterReturning**, **@AfterThrowing**, **@Around**.

1. Kết hợp Aspect với ứng dụng:

* Sử dụng Annotation **@EnableAspectJAutoProxy** trên lớp cấu hình của ứng dụng để bật tính năng AOP trong Spring.
* Đánh dấu Aspect và các thành phần liên quan (như Bean, phương thức) bằng các Annotation như **@Component**, **@Service**, **@Repository** để Spring quét và xử lý các Aspect và Join Point tương ứng

## Spring security

### Spring Security là gì và nó được sử dụng để làm gì trong ứng dụng?

Spring Security là một framework bảo mật mạnh mẽ và linh hoạt được tích hợp sẵn trong Spring Framework. Nó cung cấp các tính năng và công cụ để xác thực (authentication), ủy quyền (authorization) và bảo vệ ứng dụng web của bạn.

Spring Security giúp bạn xây dựng các tính năng bảo mật trong ứng dụng web, bao gồm:

1. Xác thực (Authentication): Spring Security hỗ trợ xác thực người dùng bằng cách xác định danh tính và chứng thực đối tượng sử dụng các cơ chế như form-based, basic authentication, hoặc custom authentication providers. Nó cho phép tích hợp với các cơ chế xác thực bên ngoài như LDAP, OAuth, OpenID, v.v.
2. Ủy quyền (Authorization): Spring Security cung cấp cơ chế ủy quyền để kiểm soát quyền truy cập người dùng vào các tài nguyên và chức năng trong ứng dụng. Bạn có thể xác định các quyền và vai trò người dùng, thiết lập các quy tắc ủy quyền và áp dụng chúng cho các URL, phương thức, hoặc các phần tử cụ thể khác.
3. Bảo vệ ứng dụng (Application Security): Spring Security cung cấp các lớp bảo vệ cho ứng dụng web, bao gồm bảo vệ khỏi các cuộc tấn công như cross-site scripting (XSS), cross-site request forgery (CSRF), SQL injection, v.v. Nó cung cấp các công cụ để giảm thiểu các lỗ hổng bảo mật thông qua cấu hình và quản lý phiên.

Spring Security tích hợp chặt chẽ với Spring Framework, cho phép bạn áp dụng các quy tắc bảo mật dễ dàng và linh hoạt trong ứng dụng của mình. Nó là một công cụ quan trọng để bảo vệ thông tin người dùng, kiểm soát quyền truy cập và đảm bảo tính bảo mật của ứng dụng web trong môi trường phát triển.

### Giải thích cách Spring Security hoạt động trong một ứng dụng web.

Spring Security hoạt động trong một ứng dụng web theo các bước sau:

1. Intercepting Filter: Spring Security sử dụng một servlet filter để chặn các yêu cầu đến ứng dụng web trước khi chúng đến tầng controller hoặc resource. Filter này được gọi là **DelegatingFilterProxy** và nó chịu trách nhiệm về việc xử lý các yêu cầu bảo mật.
2. Xác thực (Authentication): Khi một yêu cầu đến, Spring Security kiểm tra xem người dùng đã được xác thực hay chưa. Nếu người dùng chưa xác thực, Spring Security yêu cầu thông tin xác thực từ người dùng, chẳng hạn qua form đăng nhập. Thông tin này sau đó được sử dụng để xác thực và tạo ra một đối tượng **Authentication** đại diện cho người dùng đã xác thực.
3. Xác thực Provider (Authentication Provider): Spring Security sử dụng một Authentication Provider để xác thực thông tin người dùng. Provider này kiểm tra thông tin xác thực và trả về một đối tượng **Authentication** nếu thông tin xác thực hợp lệ.
4. Ủy quyền (Authorization): Sau khi xác thực thành công, Spring Security tiến hành quá trình ủy quyền để kiểm tra quyền truy cập của người dùng vào các tài nguyên và chức năng trong ứng dụng. Nó kiểm tra các quy tắc ủy quyền được định nghĩa và xác định vai trò và quyền của người dùng.
5. Xử lý yêu cầu: Nếu người dùng đã xác thực và được ủy quyền, yêu cầu được chuyển đến tầng controller hoặc resource tương ứng để xử lý và trả về kết quả.
6. Xử lý lỗi và chuyển hướng: Trong trường hợp xảy ra lỗi xác thực hoặc ủy quyền, Spring Security chuyển hướng người dùng đến trang lỗi tương ứng hoặc thực hiện các xử lý lỗi khác.

Spring Security sử dụng các thành phần khác nhau như Filters, Authentication Providers, UserDetailsService, AccessDecisionManager và UserDetails để thực hiện các chức năng xác thực và ủy quyền trong quá trình bảo mật ứng dụng web. Nó cung cấp một cách linh hoạt và mở rộng để điều chỉnh bảo mật ứng dụng theo nhu cầu cụ thể của dự án.

### Các thành phần có trong spring security

Spring Security bao gồm các thành phần chính sau:

1. Filters (Bộ lọc): Spring Security sử dụng các bộ lọc để chặn và xử lý các yêu cầu bảo mật trước khi chúng đến tầng controller hoặc resource. Các bộ lọc này bao gồm:
   * **DelegatingFilterProxy**: Bộ lọc chính của Spring Security, chịu trách nhiệm xử lý các yêu cầu bảo mật.
   * **UsernamePasswordAuthenticationFilter**: Xử lý yêu cầu đăng nhập dựa trên tên người dùng và mật khẩu.
   * **BasicAuthenticationFilter**: Xử lý xác thực dựa trên Authentication header của HTTP Request.
   * **JwtAuthenticationFilter**: Xử lý xác thực dựa trên JSON Web Tokens (JWT).
2. Authentication (Xác thực): Cung cấp cơ chế xác thực người dùng. Các thành phần liên quan bao gồm:
   * **AuthenticationManager**: Quản lý quá trình xác thực và xác định xem người dùng đã được xác thực hay chưa.
   * **AuthenticationProvider**: Kiểm tra thông tin xác thực và trả về một đối tượng Authentication nếu thông tin xác thực hợp lệ.
   * **UserDetailsService**: Lấy thông tin người dùng từ nguồn dữ liệu như cơ sở dữ liệu để xác thực.
3. Authorization (Ủy quyền): Quản lý quyền truy cập của người dùng vào các tài nguyên và chức năng trong ứng dụng. Các thành phần liên quan bao gồm:
   * **AccessDecisionManager**: Quyết định xem người dùng có quyền truy cập vào tài nguyên hay không dựa trên quy tắc ủy quyền.
   * **AccessDecisionVoter**: Đánh giá quyền truy cập dựa trên các đánh giá và quy tắc ủy quyền.
   * **SecurityExpressionHandler**: Xử lý các biểu thức an ninh để kiểm tra quyền truy cập.
4. Session Management (Quản lý phiên): Quản lý phiên và xử lý các hoạt động liên quan đến phiên như hết hạn phiên, kiểm tra phiên hợp lệ, v.v.
5. Remember Me (Ghi nhớ đăng nhập): Cung cấp tính năng ghi nhớ đăng nhập để người dùng không phải đăng nhập lại sau khi phiên hết hạn.
6. CSRF Protection (Bảo vệ chống tấn công CSRF): Cung cấp bảo vệ chống tấn công CSRF bằng cách tạo và kiểm tra token CSRF cho các yêu cầu.
7. Logout (Đăng xuất): Xử lý quá trình đăng xuất người dùng khỏi ứng dụng.

Các thành phần trên được cấu hình và kết hợp với nhau thông qua các cấu hình Spring Security để đảm bảo tính bảo mật trong ứng dụng.

# ADVANCE

## Design Pattern

* Singleton:  khởi tạo đối tượng và khởi tạo 1 lần
* Đảm bảo có 1 instance tạo ra => Cung cấp 1 method đến thực thi đó
* Đặc điểm:
* private constructor: hạn chế truy cập từ bên ngoài
* đặt static variable => chỉ khởi tạo trong class
* 1 method public => return instance
* Abstract: 1 lớp cha, nhiều lớp con
* Factory: lớp trừu tượng dự kiến thay đổi thường xuyên
* DTO
* Builder:
* khởi tạo đối tượng (là Abstract || Interface khai báo phương thức tạo đối tượng)
* Sử dụng với đối tượng phức tạp, nhiều tham số truyền vào
* Factory:
* Khi việc triển khai một lớp trừu tượng dự kiến ​​sẽ thay đổi thường xuyên
* Khi việc triển khai hiện tại không thể đáp ứng một cách thoải mái thay đổi mới
* Khi quá trình khởi tạo tương đối đơn giản và hàm tạo chỉ yêu cầu một số tham số

## AOP

* **là kiểu lập trình hướng khía cạnh, là kỹ thuật lập trình giúp dễ dàng hơn trong việc xử lý logic (AOP chia chương trình thành nhiều phần khác nhau)**
* là cái bổ sung những thiếu sót cho lập trình OOP, 1 kỹ thuật lập trình dùng để tách logic chương trình thành các phần riêng biệt…
* Khi hoạt động, chương trình sẽ kết hợp các module lại để thực hiện các chức năng nhưng khi sửa đổi 1 chức năng thì chỉ cần sửa 1 module.
* Trong Spring AOP, có 4 loại advice được hỗ trợ:
* Before advice: chạy trước khi method được thực thi
* After returning advice: Chạy sau khi method trả về một kết quả
* After throwing advice: Chạy khi method ném ra một exception
* Around advice: Chạy khi method được thực thi (Bao gồm cả 3 loại advice trên)

## Connection pool

Nội dung chính của connection pool bao gồm:

* Quản lý và tái sử dụng kết nối: Connection pool duy trì một tập hợp các kết nối sẵn có đến cơ sở dữ liệu. Thay vì mở và đóng kết nối cho mỗi yêu cầu từ ứng dụng, connection pool cho phép tái sử dụng các kết nối đã được mở. Khi ứng dụng cần kết nối đến cơ sở dữ liệu, nó có thể lấy một kết nối từ pool và sử dụng nó. Sau khi hoàn thành công việc, kết nối sẽ được trả lại cho pool để sử dụng lại.
* Quản lý và giới hạn số lượng kết nối: Connection pool có thể giới hạn số lượng kết nối mà ứng dụng có thể mở đến cơ sở dữ liệu. Điều này giúp tránh quá tải cơ sở dữ liệu và giới hạn sử dụng tài nguyên hệ thống. Khi số lượng kết nối trong pool đạt đến giới hạn, các yêu cầu mới sẽ phải chờ đợi cho đến khi có một kết nối trống.
* Quản lý lỗi và khắc phục: Connection pool cung cấp khả năng quản lý các lỗi xảy ra trong quá trình kết nối và thao tác với cơ sở dữ liệu. Nếu một kết nối trong pool gặp phải lỗi, nó có thể được đánh dấu là không khả dụng và một kết nối mới sẽ được tạo thay thế. Điều này giúp đảm bảo tính liên tục của ứng dụng và giảm thiểu ảnh hưởng của lỗi đến toàn bộ hệ thống.
* Tối ưu hiệu suất: Bằng cách sử dụng connection pool, ứng dụng có thể giảm thiểu thời gian kết nối đến cơ sở dữ liệu và tăng tốc độ thực hiện các truy vấn. Thay vì mở và đóng kết nối cho mỗi yêu cầu, connection pool cho phép ứng dụng sử dụng các kết nối đã sẵn sàng, giúp tiết kiệm thời gian và tài nguy

## String pool

* String pool (còn được gọi là constant pool hoặc string constant pool) là một kỹ thuật trong lập trình, thường được sử dụng trong các ngôn ngữ lập trình như Java, để tối ưu hóa việc sử dụng bộ nhớ cho các chuỗi (strings).
* Khi bạn khai báo một chuỗi trong chương trình, thông thường chuỗi đó được lưu trữ trong vùng nhớ có tên là string pool. String pool duy trì một bộ sưu tập các chuỗi duy nhất trong ứng dụng. Nếu một chuỗi mới được tạo và nó đã tồn tại trong string pool, thay vì tạo một bản sao mới, thì tham chiếu đến chuỗi đã tồn tại trong string pool sẽ được trả về. Điều này giúp tiết kiệm bộ nhớ bởi vì các chuỗi có thể được chia sẻ và tái sử dụng.
* String pool có lợi ích làm giảm sự lãng phí bộ nhớ bằng cách chỉ lưu trữ một bản sao duy nhất của mỗi chuỗi trong ứng dụng. Điều này đặc biệt hữu ích khi xử lý nhiều chuỗi có cùng nội dung, ví dụ như khi sử dụng các hằng số chuỗi trong mã nguồn.
* Tuy nhiên, điều này cũng có thể dẫn đến một số vấn đề. Vì các chuỗi trong string pool là không thay đổi (immutable), nếu bạn thực hiện các phép toán trên chuỗi và tạo ra các chuỗi mới, các chuỗi mới đó sẽ không được lưu trong string pool mà sẽ tạo ra một bản sao mới. Do đó, việc sử dụng không cẩn thận của các phép toán trên chuỗi có thể gây lãng phí bộ nhớ và làm tăng thời gian thực thi của chương trình.

## String & new String()

Khi khởi tạo String = "" và new String() có gì khác nhau không?

* Khi khởi tạo một đối tượng String bằng cách sử dụng cú pháp String str = ""; và String str = new String();, có một số khác biệt nhỏ giữa hai cách này.
* Khởi tạo bằng cú pháp String str = "";: Đây là cách thông thường được sử dụng để khởi tạo một chuỗi rỗng trong Java. Khi sử dụng cú pháp này, một đối tượng String rỗng sẽ được tạo ra và được lưu trữ trong string pool (nếu chuỗi chưa tồn tại trong pool). Các tham chiếu đến chuỗi rỗng này sẽ trỏ đến cùng một đối tượng trong string pool.
* Khởi tạo bằng cú pháp String str = new String();: Khi sử dụng cú pháp này, một đối tượng String rỗng sẽ được tạo ra, nhưng không nằm trong string pool. Thay vào đó, một đối tượng mới được cấp phát trong vùng nhớ heap, và tham chiếu str sẽ trỏ đến đối tượng này. Điều này có nghĩa là mỗi khi sử dụng cú pháp new String(), một đối tượng String mới sẽ được tạo ra, dù có chuỗi tương tự đã tồn tại trong string pool hay không.
* Vì các đối tượng String trong string pool là không thay đổi (immutable) và có thể được chia sẻ, việc sử dụng cú pháp String str = ""; thường được khuyến nghị để tạo ra các chuỗi rỗng. Điều này giúp tối ưu hóa việc sử dụng bộ nhớ bởi vì các tham chiếu đến chuỗi rỗng sẽ trỏ đến cùng một đối tượng trong string pool, giảm sự lãng phí bộ nhớ so với việc tạo ra các đối tượng String rỗng mới bằng cú pháp new String().

+ Thread là gì?

+ Có những cách nào khởi tạo 1 thread?

+ Làm sao để so sánh 2 Object với nhau?

Để so sánh hai đối tượng với nhau, có hai phương pháp phổ biến là so sánh theo địa chỉ bộ nhớ (reference comparison) và so sánh giá trị (value comparison). Cách sử dụng phụ thuộc vào loại đối tượng và yêu cầu so sánh cụ thể của bạn.

So sánh theo địa chỉ bộ nhớ (reference comparison):

Trong Java, toán tử == được sử dụng để so sánh hai đối tượng theo địa chỉ bộ nhớ của chúng. Nếu hai đối tượng có cùng địa chỉ bộ nhớ, thì chúng được coi là đồng tham chiếu (đều trỏ đến cùng một vị trí trong bộ nhớ), và phép so sánh sẽ trả về kết quả true.

Ví dụ: obj1 == obj2

So sánh giá trị (value comparison):

Đối với các đối tượng có sẵn trong Java như String, Integer, Double,... bạn có thể sử dụng phương thức equals() để so sánh giá trị của chúng.

Ví dụ: obj1.equals(obj2)

Phương thức equals() trong các lớp được xây dựng sẵn trong Java thường được ghi đè để so sánh các giá trị bên trong đối tượng. Ví dụ, trong lớp String, phương thức equals() so sánh giá trị của hai chuỗi dựa trên nội dung của chúng, không chỉ đơn thuần so sánh địa chỉ bộ nhớ.

Lưu ý rằng phép so sánh giá trị phụ thuộc vào cách cài đặt của phương thức equals() trong từng lớp đối tượng. Nếu bạn đang làm việc với đối tượng tùy chỉnh, bạn nên ghi đè phương thức equals() và hashCode() để xác định cách so sánh giá trị của đối tượng của bạn.

+ FunctionalInterface là gì?

+ Stream API là gì?

+ Dùng interface Function, phương thức apply() bào giờ chưa?

Spring:

+ Spring security là gì?

+ Cơ chế hoạt động của Spring security là gì?

+ Cơ chế oauth2 hoạt động như thế nào?

+ Dependecy Inversion là gì?

+ Dependecy Injection là gì?

+ IoC là gì?

+ Các cách inject?

+ @Autowire để làm gì?

+ Bean là gì? Các cách khởi tạo bean?

+ Nếu có 2 bean cùng type làm sao để phân biệt 2 bean đó?

## Database

+ Nếu có rất nhiều connect đến DB làm thế nào? (Nó hỏi connection pool gì đó mà đọc k hiểu gì)

+ Index là gì?

+ Tại sao dùng index lại truy vấn nhanh hơn?

+ Nhược điểm của nó là gì?

+ Làm sao để tối ưu perfomance cho câu SQL?

Để tối ưu hiệu suất của câu SQL, bạn có thể áp dụng các kỹ thuật và quy tắc sau:

* Chỉ lấy những cột cần thiết: Hạn chế lấy tất cả các cột trong bảng, chỉ lấy những cột cần thiết để giảm tải dữ liệu và tăng tốc độ truy xuất.
* Sử dụng chỉ mục (indexing): Xác định và tạo chỉ mục cho các cột thường xuyên được sử dụng trong các câu truy vấn. Chỉ mục giúp tìm kiếm và truy xuất dữ liệu nhanh hơn.
* Sử dụng câu truy vấn gọn và hiệu quả: Tối ưu câu truy vấn bằng cách sử dụng các câu truy vấn ngắn gọn và sử dụng các phép toán SQL hiệu quả như JOIN, GROUP BY, HAVING, và WHERE để giới hạn dữ liệu trước khi lấy về.
* Tránh sử dụng toán tử LIKE với ký tự đầu: Sử dụng chỉ mục tiền tố (prefix index) nếu cần sử dụng toán tử LIKE với ký tự đầu. Điều này giúp tối ưu tìm kiếm dữ liệu dựa trên tiền tố.
* Sử dụng câu truy vấn tham số hóa (parameterized queries): Sử dụng tham số thay vì các giá trị cứng trong câu truy vấn để tránh tấn công SQL injection và tăng hiệu suất câu truy vấn được lưu trong bộ nhớ cache.
* Tối ưu hóa schema cơ sở dữ liệu: Thiết kế schema cơ sở dữ liệu sao cho phù hợp với mô hình dữ liệu và nhu cầu truy xuất dữ liệu. Đảm bảo sự chuẩn hóa (normalization) và đảm bảo các quan hệ giữa các bảng được thiết lập đúng cách.
* Sử dụng transaction một cách hiệu quả: Định nghĩa và sử dụng transaction một cách chính xác để đảm bảo tính nhất quán và an toàn cho dữ liệu.
* Sử dụng cache dữ liệu: Sử dụng cache để lưu trữ các dữ liệu truy xuất thường xuyên để giảm tải cơ sở dữ liệu và tăng tốc độ truy xuất.
* Đánh giá và tối ưu lại câu truy vấn: Sử dụng các công cụ và hệ thống giám sát để phân tích và đánh giá hiệu suất của câu truy vấn. Dựa trên đánh giá, điều chỉnh câu truy vấn và cấu trúc cơ sở dữ liệu để cải thiện hiệu suất.
* Cân nhắc sử dụng các công nghệ nâng cao: Trong một số trường hợp đặc biệt, có thể sử dụng các công nghệ nâng cao như database sharding, database partitioning, hoặc replication để tăng hiệu suất và khả năng mở rộng của cơ sở dữ liệu.
* Quan trọng nhất, hãy xem xét và phân tích tình huống cụ thể của bạn, sử dụng các công cụ và kỹ thuật tối ưu phù hợp để đạt được hiệu suất tốt nhất cho câu SQL của bạn.

- Other

+ Nếu trong 1 khách sạn có còn 10 phòng mà cùng 1 lúc có 20 request cùng đặt phòng làm sao để phân chia phòng mà không bị conflict nhau?

+ Dùng rancher để làm gì?

+ Dùng jenkins để làm gì? Có setup được server không?

+ CI/CD dùng trong dự án dùng gì để chạy? Cơ chế CICD là gì?

+ Server của microservices dùng gì để config?

+ Mỗi service trong microservices dùng một node hay như nào( câu này k nhớ câu hỏi lắm)

+ Redis là gì? Dùng redis trong dự án để làm gì?

+ Live code: Input là một mảng các số nguyên. Cần in ra mảng các phần tử với giá trị là số các phần tử trước đó lớn hơn nó.

Eg: Input [5, 43, 10, 1, 55, 21] ==> Output [0, 0, 1, 3, 0, 2]

+ Một số câu hỏi về chế độ công ty thế này thế kia thì sẽ làm sao?

# RABBIT MQ

Nội dung chính của RabbitMQ bao gồm:

* Message Queue (MQ): RabbitMQ là một hệ thống Message Queue (MQ) mã nguồn mở. Nó cung cấp một cơ chế gửi và nhận thông điệp giữa các ứng dụng, đảm bảo việc truyền tải tin nhắn an toàn và đáng tin cậy giữa các thành phần của hệ thống phân tán.
* Mô hình Publish/Subscribe: RabbitMQ hỗ trợ mô hình Publish/Subscribe, cho phép một ứng dụng gửi thông điệp (publisher) tới một topic (exchange), và các ứng dụng khác (subscribers) có thể đăng ký lắng nghe topic đó để nhận thông điệp. Điều này cho phép các ứng dụng giao tiếp theo hướng không đồng bộ và có khả năng mở rộng.
* Hệ thống hàng đợi (Queue): RabbitMQ sử dụng các hàng đợi (queues) để lưu trữ các thông điệp. Khi một thông điệp được gửi tới RabbitMQ, nó sẽ được đưa vào hàng đợi và chờ đợi để được xử lý bởi một ứng dụng (consumer). Hàng đợi này giúp đảm bảo tính đồng bộ và đảm bảo rằng thông điệp không bị mất mát.
* Giao thức AMQP: RabbitMQ triển khai giao thức Advanced Message Queuing Protocol (AMQP). Giao thức này định nghĩa các quy tắc và giao tiếp giữa các thành phần của hệ thống gửi và nhận thông điệp. RabbitMQ sử dụng AMQP để đảm bảo tính tin cậy, chuyển giao tin nhắn theo thứ tự và quản lý hàng đợi.
* Xử lý công việc đồng thời: RabbitMQ cho phép xử lý công việc đồng thời bằng cách sử dụng các worker (consumer) để xử lý các thông điệp trong hàng đợi. Các worker có thể được cấu hình để đồng thời xử lý nhiều thông điệp, giúp tăng hiệu suất và khả năng mở rộng của hệ thống.
* Độ tin cậy và khả năng phục hồi: RabbitMQ cung cấp các cơ chế đảm bảo độ tin cậy và khả năng phục hồi trong trường hợp xảy ra lỗi. Nó hỗ trợ chính sách xác nhận (acknowledgment) để đảm bảo rằng một thông điệp đã được xử lý thành công trước khi được xóa khỏi hàng đợi. RabbitMQ cũng có khả năng lưu trữ thông điệp trên đĩa để đảm bảo rằng không mất dữ liệu trong trường hợp xảy ra sự cố.
* RabbitMQ cung cấp một giải pháp tin cậy và linh hoạt cho việc xây dựng các hệ thống phân tán và giao tiếp giữa các ứng dụng. Nó đã trở thành một trong những message broker phổ biến nhất trong cộng đồng phát triển phần mềm.

# KAFKA & ZOOKEEPER

## Kafka là gì

Kafka là một hệ thống thông điệp phân tán (distributed messaging system) mã nguồn mở được xây dựng bởi Apache Software Foundation. Nó được thiết kế để xử lý và truyền tải lượng lớn dữ liệu từ nhiều nguồn đến nhiều đích một cách hiệu quả.

Kafka được xây dựng với các mục tiêu chính sau:

1. Tốc độ và khả năng mở rộng: Kafka có thể xử lý hàng triệu thông điệp mỗi giây và có khả năng mở rộng dễ dàng để đáp ứng yêu cầu tăng trưởng lưu lượng.
2. Bền vững: Dữ liệu trong Kafka được lưu trữ trên đĩa và có thể được duy trì trong thời gian dài. Điều này đảm bảo tính bền vững và khả năng phục hồi sau khi xảy ra lỗi.
3. Phân tán: Kafka có thể chạy trên một cụm máy chủ phân tán và các topic (chủ đề) có thể được phân chia thành nhiều partition (phân vùng) để tăng khả năng xử lý và hiệu suất.
4. Hỗ trợ đa ngôn ngữ: Kafka hỗ trợ các khách hàng (client) viết bằng nhiều ngôn ngữ lập trình khác nhau như Java, Python, Ruby, C++, v.v.

Kafka được sử dụng rộng rãi trong các ứng dụng xử lý dữ liệu thời gian thực như hệ thống ghi nhật ký (log), hệ thống theo dõi và phân tích thống kê, xây dựng dự án streaming, hệ thống tích hợp, v.v. Nó cung cấp khả năng truyền tải thông điệp một cách tin cậy, bền vững và có hiệu suất cao.

## Kafka và ZooKeeper

Apache Kafka và Apache ZooKeeper có mối quan hệ rất chặt chẽ và ZooKeeper thường được sử dụng như một phần quan trọng trong việc triển khai và quản lý Kafka.

Apache ZooKeeper là một dịch vụ phân tán đáng tin cậy (reliable distributed service) cung cấp các tính năng quản lý, đồng bộ và theo dõi cho các ứng dụng phân tán. Kafka sử dụng ZooKeeper như một phần của hệ thống quản lý metadata và quản lý trạng thái của cụm Kafka.

Một số mối quan hệ chính giữa Kafka và ZooKeeper:

1. Quản lý metadata: ZooKeeper giữ trạng thái và metadata của cụm Kafka, bao gồm thông tin về các topic, partition, offset, thông tin về các producer và consumer.
2. Điều phối leader: ZooKeeper giúp Kafka xác định các broker (node) chịu trách nhiệm làm leader và follower cho các partition của các topic. Nó đảm bảo rằng các partition được phân chia công bằng và dữ liệu được sao chép đến các follower.
3. Quản lý tình trạng cụm: ZooKeeper theo dõi và báo cáo trạng thái của các broker trong cụm Kafka, đảm bảo sự tin cậy và sẵn sàng của cụm.
4. Xử lý failover: ZooKeeper hỗ trợ xử lý tình huống failover khi broker gặp sự cố. Nó giúp Kafka chuyển giao quyền điều khiển cho các broker khác một cách an toàn và đảm bảo tính nhất quán của dữ liệu.

Tóm lại, ZooKeeper đóng vai trò quan trọng trong việc duy trì tính nhất quán, quản lý metadata và quản lý trạng thái của cụm Kafka. Kafka sử dụng ZooKeeper như một công cụ phân tán để đảm bảo hoạt động ổn định và tin cậy của hệ thống.

## Tại sao kafka lại nhanh

Kafka có khả năng xử lý và truyền tải dữ liệu nhanh vì nó được thiết kế với mục tiêu tối ưu hiệu suất và khả năng mở rộng. Dưới đây là một số lý do giải thích tại sao Kafka nhanh:

1. Kiến trúc phân tán: Kafka được thiết kế dựa trên mô hình phân tán, cho phép nó hoạt động trên một cụm máy chủ (cluster) bao gồm nhiều broker. Các broker có thể chia sẻ công việc xử lý và truyền tải dữ liệu, giúp tăng cường khả năng xử lý và hiệu suất.
2. Partitioning: Kafka phân chia các topic thành nhiều partition, mỗi partition có thể nằm trên một broker khác nhau trong cụm. Điều này cho phép Kafka xử lý đồng thời các partition và tăng khả năng xử lý song song của hệ thống. Các partition cũng cho phép dữ liệu được lưu trữ và truyền tải song song, giúp tăng hiệu suất và khả năng mở rộng.
3. Ghi và đọc tuần tự: Kafka sử dụng cơ chế ghi tuần tự (sequential write) để ghi dữ liệu vào partition. Việc ghi tuần tự giúp tối ưu hiệu suất bởi vì nó không cần phải di chuyển đầu đọc ghi trên đĩa, và dữ liệu có thể được ghi đồng thời vào nhiều partition.
4. Lưu trữ trên đĩa: Kafka lưu trữ dữ liệu trên đĩa, cho phép nó duy trì tính bền vững và khả năng phục hồi sau khi xảy ra sự cố. Điều này đảm bảo rằng dữ liệu không bị mất khi gặp lỗi hoặc khởi động lại hệ thống.
5. Khả năng mở rộng: Kafka có khả năng mở rộng theo cả chiều ngang (horizontal scaling) và chiều sâu (vertical scaling). Bạn có thể thêm các broker mới vào cụm Kafka để tăng khả năng xử lý, và Kafka cũng hỗ trợ phân chia dữ liệu trên nhiều partition để tận dụng tối đa nguồn lực có sẵn.

Tổng quan, Kafka được thiết kế để đạt hiệu suất cao, đồng thời và khả năng mở rộng. Điều này làm cho nó trở thành một giải pháp phổ biến cho việc xử lý và truyền tải dữ liệu lớn, thời gian thực trong các hệ thống phân tán.