**BÁO CÁO VỀ SPRINGBOOT**

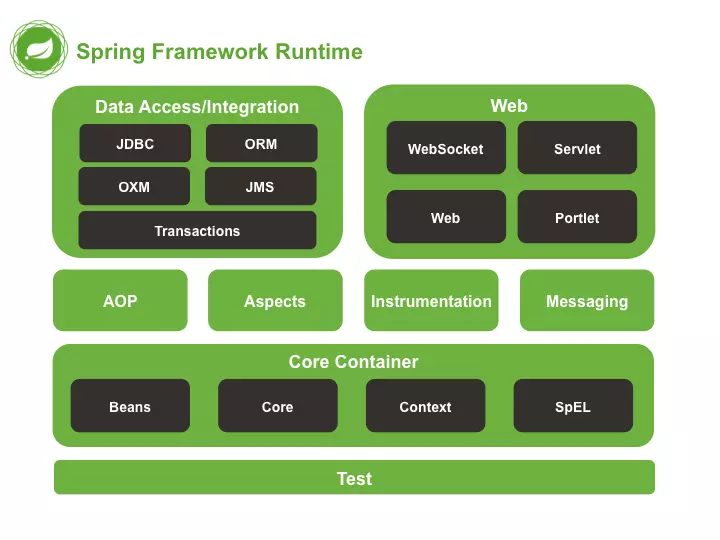
# **A/ Tìm hiểu về Spring**

***Vấn đề: một project có quá nhiều dependency => inject vất vả => Xuất hiện framework (kho chứa các dependency) sẽ tự động inject (@Autowired). Điển hình là Spring framework: mục đích tạo ra container chứa dependency => Rút ngắn thời gian lập trình và test, giảm sự rườm rà***

Framework là một bộ công cụ và thư viện phát triển được thiết kế để giúp việc xây dựng ứng dụng trở nên dễ dàng hơn. Nó cung cấp một cấu trúc, các tiêu chuẩn và các công cụ giúp lập trình viên tập trung vào việc triển khai business logic thay vì phải viết lại các thành phần cốt lõi của ứng dụng.

## **Spring framework**

Spring Framework là một trong những framework phát triển ứng dụng Java phổ biến nhất. Nó cung cấp một cách tiếp cận mô hình phát triển linh hoạt và mạnh mẽ, giúp xây dựng các ứng dụng doanh nghiệp và ứng dụng web hiệu quả.



* Core Container
* spring-core và spring-beans: Cung cấp tính năng cơ bản IoC và DI.
* spring-context: Xây dựng trên 1 nền tảng vững chắc được cung cấp bởi các module Core và Beans, được sử dụng để truy cập vào các đối tượng trong framework-style tương tự như việc đăng ký một JNDI. spring-context-support hỗ trợ tích hợp libraries của bên thứ 3 (third-party) vào ứng dụng Spring như caching (EhCache, Guava, JCache), mailing (JavaMail), schedule (CommonJ, Quatz) và các template engine (FreeMarker, JasperReports, Velocity)).
* spring-expression (Spring Expression Language): Được mở rộng từ Expresion Language trong JSP. Nó cung cấp hỗ trợ việc setting/getting giá trị, các method cải tiến cho phép truy cập collections, index, các toán tử logic…
* Data Access/Integration
* spring-jdbc: Cung cấp 1 lớp JDBC-abstraction để loại bỏ những code tẻ nhạt cả JDBC và phân tích những mã lỗi cụ thể của database-vendor.
* spring-orm: Cung cấp lớp tích hợp với các orm API phổ biến như JPA, JDO và Hibernate.
* spring-oxm: Cung cấp lớp abstraction hỗ trợ triển khai Object/XML mapping như AXB, Castor, XMLBeans, JiBX and XStream.
* spring-jms: Chứa các tính năng tạo và sử dụng các message. Từ Spring Framework 4.1, nó được tích hợp với spring-messaging.
* spring-transaction: Hỗ trợ quản lý giao dịch theo chương trình và khai báo cho các lớp mà thực hiện các giao diện đặc biệt và cho tất cả các POJO của bạn.
* Web
* spring-web: Cung cấp tính năng tích hợp web theo định hướng cơ bản như chức năng tập tin tải lên nhiều phần dữ liệu và khởi tạo các container IoC sử dụng nghe servlet.
* spring-webmvc: Bao gồm việc triển khai Model-View-Controller (MVC) của Spring cho các ứng dụng web.
* spring-websocket: Cung cấp hỗ trợ cho WebSocket-based, giao tiếp hai chiều giữa client và server trong các ứng dụng web.
* springwebmvc-portlet cung cấp việc triển khai MVC được sử dụng trong môi trường portlet và ánh xạ chức năng của module Web-Servlet.
* Các module quan trọng khác
* AOP: Hỗ trợ cài đặt lập trình hướng khía cạnh (Aspect Oriented Programming)
* Aspects: Cung cấp tích hợp với AspectJ, là 1 khuôn khổ AOP mạnh mẽ.
* Instrumentation: Cung cấp thiết bị đo đạc lớp hỗ trợ và triển khai lớp bộ nạp được sử dụng trong các máy chủ ứng dụng nhất định.
* Messaging: Cung cấp hỗ trợ cho STOMP như WebSocket sub-protocol để sử dụng trong các ứng dụng. Nó cũng hỗ trợ một mô hình lập trình chú thích cho việc định tuyến và xử lý tin nhắn STOMP từ các máy khách WebSocket.
* Test: Cung cấp khả năng hỗ trợ kiểm thử với JUnit và TestNG.
* Dựa trên các nguyên tắc thiết kế cơ bản của spring core. Spring còn phát triển nhiều project con như:
* Spring MVC: Dành cho việc xây dựng các ứng dụng nền tảng web.
* Spring Security: Cung cấp các cơ chế authentication và authorization cho ứng dụng.
* Spring Boot: Giúp phát triển cũng như chạy ứng dụng một cách nhanh chóng.
* Spring Batch: Giúp dễ dàng tạo các lịch trình (scheduling) và tiến trình (processing) cho các công việc xử lý theo mẻ (batch job).
* Spring Social: Cho phép kết nối ứng dụng với các API bên thứ ba của Facebook, Twitter, Linkedin, ...
* ...

1. **Springboot**

Spring Boot là một extension của Spring Framework, giúp loại bỏ các bước cấu hình phức tạp mà Spring bắt buộc. Spring Boot là dự án phát triển bởi Java (JAV) trong hệ sinh thái Spring framework.

Một số tính năng nổi bật của Spring Boot đó là:

* Tạo ra các ứng dụng Spring mang tính độc lập.
* Nhúng trực tiếp Tomcat, Jetty hoặc Undertow mà không cần phải triển khai ra file WAR.
* Starter dependency giúp cho việc chạy cấu hình Maven trở nên đơn giản hơn.
* Tự động chạy cấu hình Spring khi cần thiết.
* Không sinh code cấu hình, đồng thời không yêu cầu phải cấu hình bằng XML.

Spring có kích thước rất nhỏ, trong suốt và nhẹ trong quá trình chạy. Vì vậy nên, đây là một giải pháp khá gọn, nhẹ với khả năng hỗ trợ để tạo ra và phát triển các ứng dụng web có mã hiệu suất cao. Hơn hết, có thể dễ dàng kiểm tra, thử nghiệm hoặc tái sử dụng code.

Hai nguyên tắc thiết kế chính để xây dựng nên spring framework đó là:

* Dependency Injection (DI)
* Aspect Oriented Programming (AOP)

Những tính năng cốt lõi của Spring framework có thể được sử dụng trong việc phát triển java desktop, java web, … Mục tiêu chính là dễ dàng phát triển các ứng dụng J2EE dựa trên mô hình sử dụng POJO

* IoC

IoC (Inversion of Control) là một nguyên tắc trong lập trình, tuân thủ theo nguyên tắc D (SOLID), trong đó quyền kiểm soát các đối tượng và luồng của chương trình được chuyển từ phần mềm chính tới một framework hoặc container. Thay vì phần mềm gọi các hàm hoặc khởi tạo các đối tượng, hàng rào của framework sẽ gọi các hàm và khởi tạo các đối tượng để tiếp quản và điều khiển luồng của chương trình => Tăng tính linh hoạt, dễ bảo trì và tái sử dụng mã nguồn. Có thể triển khai IoC thông qua các cơ chế như Strategy design pattern, Service Locator pattern, Factory pattern, và Dependency Injection (DI).

IoC giúp tách biệt và quản lý phụ thuộc giữa các thành phần trong ứng dụng, làm cho mã của bạn linh hoạt hơn và dễ dàng thay đổi hoặc kiểm thử.

* DI

Vậy dependency là gì? dependency (phụ thuộc) đề cập đến một thành phần hoặc một đối tượng mà một thành phần khác cần để hoạt động. Đối tượng hoặc thành phần này có thể là một đối tượng, một interface, một thư viện, hoặc bất cứ thứ gì mà thành phần khác cần sử dụng để đạt được mục tiêu của nó. Thành phần phụ thuộc luôn được sử dụng hoặc được truyền vào thành phần khác để thực hiện chức năng cần thiết

DI (Dependency Injection) là một cách triển khai của IoC (giống IoC). Nó tập trung vào việc cung cấp các phụ thuộc (dependencies) cho một đối tượng từ bên ngoài. Thay vì đối tượng tự tạo hoặc tìm kiếm những đối tượng phụ thuộc của nó. Các phụ thuộc này có thể là các đối tượng, các giao diện, hoặc các giá trị cấu hình => giúp tách biệt các thành phần và làm cho chúng dễ dàng thay thế và kiểm soát.

|  |  |
| --- | --- |
| Thay vì dùng như này: | Nên dùng như này: |
| *public class UserRepository {*  *private DatabaseConnection dbConnection;*  *public UserRepository() {*  *this.dbConnection = new DatabaseConnection();*  *}*  *}*  *UserRepository trực tiếp tạo ra và nắm quyền điều khiển DatabaseConnection. Khi muốn thay đổi sang MySQLDatabaseConnection hoặc PostgreSQLDatabaseConnection thì phải thay đổi trong UserRepository => bất tiện* | *public class UserRepository {*  *private DatabaseConnection dbConnection;*  *public UserRepository(DatabaseConnection dbConnection) {*  *this.dbConnection = dbConnection;*  *}*  *}*  *UserRepository không tạo trực tiếp một đối tượng DatabaseConnection, nhưng nó được cung cấp từ bên ngoài qua constructor của UserRepository. Điều này làm linh hoạt trong việc thay đổi các giao diện và triển khai của DatabaseConnection mà không ảnh hưởng đến UserRepository.* |

* Bean

Spring boot sử dụng khái niệm "bean" để đại diện cho các đối tượng và được quản lý bởi Spring Container.

Một số loại bean:

* Singleton Bean: Là loại bean mặc định trong Spring. Một singleton bean chỉ có một phiên bản duy nhất được tạo ra và được chia sẻ cho toàn bộ ứng dụng.
* Prototype Bean: Đối với mỗi yêu cầu, một bean prototype sẽ tạo ra một phiên bản mới. Mỗi bean được tạo ra sẽ thành một đối tượng hoàn toàn độc lập.
* Request Scope Bean: Bean với request scope chỉ tồn tại trong suốt quá trình xử lý một yêu cầu HTTP cụ thể.
* Session Scope Bean: Bean với session scope chỉ tồn tại trong suốt chuỗi thời gian kết nối của một phiên (session) HTTP.
* Global Session Scope Bean: Bean với global session scope chỉ tồn tại trong suốt chuỗi thời gian kết nối của một phiên (session) toàn cầu.
* Application Scope Bean: Bean với application scope tồn tại trong toàn bộ vòng đời của ứng dụng. Chúng được tạo một lần duy nhất khi ứng dụng được khởi động.
* Container

Trong Spring Framework, Spring IoC (Inversion of Control) Container là một phần quan trọng trong Spring để quản lý vòng đời và dependencies của các đối tượng trong ứng dụng. Nó có vai trò quản lý đối tượng, có trách nhiệm tạo ra, cấu hình và quản lý các đối tượng bean của ứng dụng. Thay vì tái tạo đối tượng bằng cách gọi trực tiếp từ mã, ta chỉ cần định nghĩa các đối tượng trong cấu hình (tệp XML hoặc thông qua Java Config) và container sẽ tự động khởi tạo, khởi tạo dependencies (đảm bảo dependencies được đưa vào bean một cách tự động, dễ dàng thiết lập và thay thế), quản lý và giải phóng tài nguyên khi không cần thiết nữa.

Bean Factory và ApplicationContext đều là hai thành phần quan trọng của Spring IoC Container:

* Bean Factory là một phần nhỏ nhất của Spring IoC Container. Nó là interface cung cấp các chức năng cơ bản để tạo và quản lý các bean. Bean Factory chịu trách nhiệm cho việc tải và cấu hình các đối tượng (bean) từ các nguồn dữ liệu như tệp XML hoặc các nguồn dữ liệu khác.
* ApplicationContext (extends từ interface BeanFactory). Nó cung cấp tất cả các chức năng của Bean Factory, nhiều tính năng tiện ích bổ sung và là phiên bản nâng cao hơn. BeanFactory tạo bean khi gọi getBean(), còn ApplicationContext tự động tạo sẵn. Ngoài ra, ApplicationContext hỗ trợ cho việc quản lý giao dịch, xử lý sự kiện, quản lý thông điệp, cấu hình dựa trên annotation và tích hợp tốt với các tính năng của Spring như AOP (Aspect-Oriented Programming) và được sử dụng phổ biến hơn Bean Factory.

Hai lớp cụ thể của ApplicationContext trong Spring Framework:

* ClassPathXmlApplicationContext (đọc cấu hình từ file XML trên classpath). Nghĩa là, cần cung cấp các tệp XML cấu hình trong ứng dụng và đặt chúng trong classpath để ClassPathXmlApplicationContext có thể tìm thấy và sử dụng để cấu hình các bean
* AnnotationConfigApplicationContext (sử dụng cấu hình dựa trên Java annotations, ví dụ như: @Configuration, @ComponentScan, @Bean, …). AnnotationConfigApplicationContext quét các packages đã được chỉ định và tự động tìm kiếm các bean được đánh dấu bởi các annotations và cấu hình chúng
* Annotation

Annotation (chú thích) là một tính năng quan trọng, được sử dụng rộng rãi trong lập trình Java, cho phép thêm các thông tin bổ sung vào mã nguồn, giúp trình biên dịch và các công cụ phát triển hiểu và xử lý mã nguồn một cách thông minh.

* Annotation được sử dụng để đánh dấu và cung cấp metadata cho các lớp, phương thức, biến, hoặc gói.

Annotation không ảnh hưởng đến hoạt động chương trình khi chạy, nhưng mang thông tin quan trọng về cấu trúc và mục đích của mã nguồn => lợi cho việc tự động hóa các tác vụ, kiểm tra mã nguồn, và cung cấp hướng dẫn cho các công cụ phát triển.

Cú pháp cơ bản của Annotation: Được đặt trong một dấu @, theo sau là tên của annotation. Một số annotation có thể có giá trị được đặt trong dấu ngoặc đơn (value = ...), nhưng nếu chỉ có một giá trị và không cần gán tên, bạn có thể viết trực tiếp giá trị đó.

Một số annotation phổ biến: @Autowired, @Component, @Override, ...

* Cách IoC hoạt động trong Spring: IoC được hiện thực thông qua DI trong spring
* Trong Spring, các beans và quan hệ phụ thuộc của chúng được định nghĩa thông qua các annotation.
* Khi run, Spring tự động phân tích các đối tượng và các dependency của chúng, sau đó đưa vào container, khi cần sẽ tự động inject (@Autowired) các đối tượng thông qua DI. => giảm sự phụ thuộc mạnh mẽ giữa các đối tượng và cho phép dễ dàng thay đổi
* @Configuration là một Annotation đánh dấu trên một Class cho phép Spring Boot biết được đây là nơi định nghĩa ra các Bean.
* Tuỳ chỉnh cấu hình cho Spring Boot chỉ tìm kiếm các bean trong một package nhất định, sử dụng @ComponentScan(“<link url>”)
* Define bean với annotations
* @Component: quản lý bởi Spring và thông báo cho Spring rằng nó cần tạo một instance của class này và quản lý nó
* @Controller: Xác định rằng class này xử lý các request HTTP và trả về một response.
* @RestController: Kết hợp hai annotation @Controller và @ResponseBody. Đánh dấu một class là một controller và method trả về dữ liệu dạng JSON hoặc XML.
* @RequestMapping: Chỉ định URL mà một phương thức trong controller xử lý. Xác định HTTP method (GET, POST, PUT, DELETE...)
* @Autowired: Được sử dụng để tiêm các dependency tự động vào một bean.
* @Service: xác định lớp được sử dụng cho tầng dịch vụ (business logic layer) (xử lý logic)
* @Repository là một chú thích cho lớp được sử dụng cho tầng lưu trữ (persistence layer) (thao tác với database)
* @Configuration là một chú thích lớp được sử dụng cho tầng cấu hình (configuration layer).
* @Bean là một chú thích cho phương thức được sử dụng để định nghĩa các bean trong các lớp được đánh dấu bằng @Configuration hoặc @Component.

Nếu có nhiều bean giống nhau trong container, @Qualifier(beanName) cho Spring biết bean nào muốn injection. Có thể sử dụng @Qualifier trên các method setter, constructor, … để chỉ định rõ ràng bean cần inject.

## **Bean life cycle**

@PostConstruct được đánh dấu trên một method duy nhất bên trong Bean. IoC Container hoặc ApplicationContext sẽ gọi hàm này sau khi một Bean được tạo ra và quản lý.

Tại sao không sử dụng Constructor thông thường thay cho PostConstructor?

|  |  |
| --- | --- |
| Constructor | PostConstructor |
| Được chạy trước khi bean được khởi tạo  => các dependency chưa được inject | Chạy sau khi bean khởi tạo thành công  => Có thể sử dụng các dependency |
| Chạy nhiều lần khi bean khởi tạo nhiều lần | Chỉ chạy duy nhất một lần, dù bean khởi tạo nhiều lần |

@PreDestroy được đánh dấu trên một method duy nhất bên trong Bean. IoC Container hoặc ApplicationContext sẽ gọi hàm này trước khi một Bean bị xóa hoặc không được quản lý nữa.

Bean life cycle được hiểu là quá trình một bean được Spring Framework tạo ra cho tới khi chết đi, sẽ có những sự kiện (event) khác nhau xảy ra

* Khi IoC Container (ApplicationContext) tìm thấy một Bean cần quản lý => khởi tạo bằng Constructor
* Inject dependencies vào Bean bằng Setter
* Các method khởi tạo khác được gọi
* Tiền xử lý trước khi @PostConstruct được gọi (implements BeanPostProcessor).
* Hàm đánh dấu @PostConstruct được gọi
* Tiền xử lý sau khi @PostConstruct được gọi (implements BeanPostProcessor).
* Bean sẵn sàng để hoạt động
* Nếu IoC Container không quản lý bean nữa hoặc bị shutdown nó sẽ gọi hàm @PreDestroy trong Bean
* Xóa Bean.

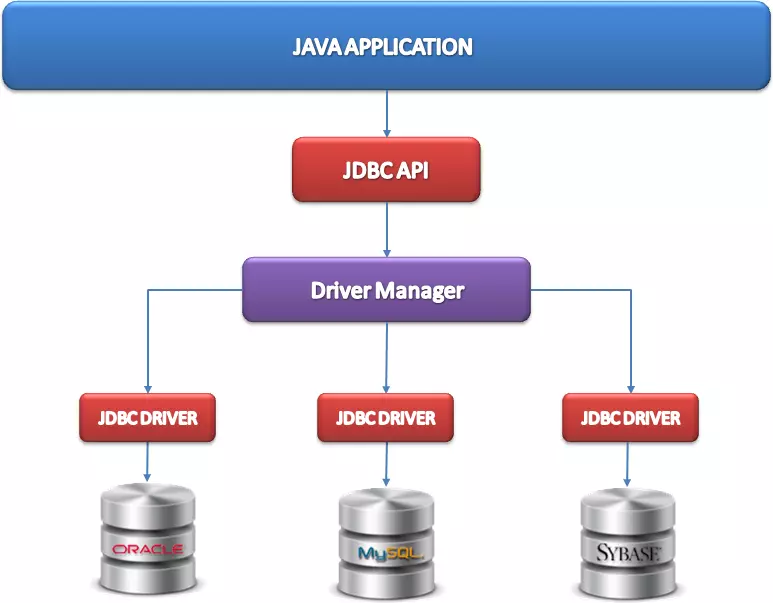
# **B/ Các dự án nổi bật của spring framework**

1. **Hibernate**

Trước khi tìm hiểu Hibernate, hãy tìm hiểu JDBC trước (hoặc có thể tìm hiểu khái quát JDBC dưới mục sau)

1. **JDBC**

JDBC (Java Database Connectivity) là ứng dụng mã nguồn mở cho Java, giúp ứng dụng Java thực hiện kết nối, làm việc với CSDL. Nó cho phép thực hiện các thao tác truy xuất, update dữ liệu với CSDL quan hệ bằng việc sử dụng các câu lệnh SQL (JDBC thường được thấy trong các dự án JSP/Servlet)



Ứng dụng Java sử dụng JDBC làm việc với CSDL thông qua trình tự 7 bước như sau:

* Tạo kết nối đến database
* Gửi SQL query đến database sử dụng JDBC driver tương ứng
* JDBC driver kết nối đến database
* Thực thi câu lệnh query để lấy kết quả trả về
* Gửi dữ liệu đến ứng dụng thông qua Driver Manager
* Xử lý dữ liệu trả về
* Đóng (giải phóng) kế nối đến database

Một số "vấn đề" gặp phải khi sử dụng JDBC:

* Phải lặp đi lặp lại những dòng code giống nhau trong ứng dụng chỉ để lấy dữ liệu từ database.
* Phải vất vả với việc map giữa Object Java với các table tương ứng trong database.
* Tốn nhiều công sức để thay đổi từ hệ quản trị CSDL này (MySQL) sang một hệ quản trị CSDL khác (Oracle).
* Khó khăn trong việc tạo các giao tiếp/liên hệ giữa các table, lập trình OOPs.
* JDBC là công cụ thô sơ nhất, mộc mạc nhất giúp kết nối CSDL trong ứng dụng Java. Và rồi Hibernate ra đời, nó mang trong mình nhiều công cụ hữu ích giúp cho việc kết nối với CSDL một cách thuận tiện, dễ dàng hơn

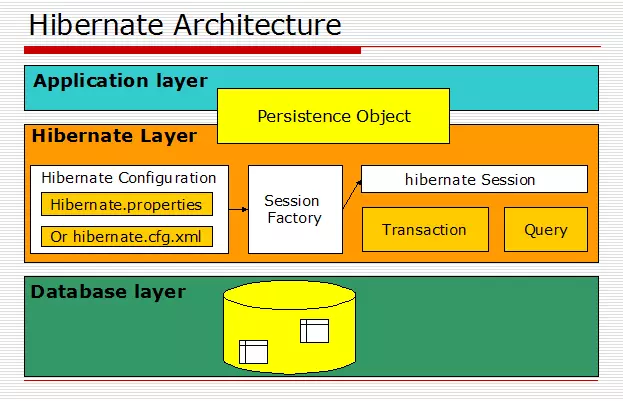
1. **Hibernate**

ORM (Object-Relational Mapping) là một kỹ thuật được sử dụng trong lập trình để ánh xạ giữa object trong mã nguồn và các table trong cơ sở dữ liệu. Nó giúp chúng ta làm việc với cơ sở dữ liệu một cách trừu tượng hóa, tức là thay vì sử dụng truy vấn SQL trực tiếp, chúng ta sử dụng các đối tượng Java để thực hiện thao tác cơ sở dữ liệu.

Lợi ích của ORM:

* Trừu tượng hóa cơ sở dữ liệu: Có thể làm việc với cơ sở dữ liệu mà không cần quan tâm đến chi tiết cụ thể của nó => giúp tách biệt logic ứng dụng và logic truy cập dữ liệu.
* Giảm việc viết SQL thủ công: Không cần phải viết nhiều truy vấn SQL thủ công, giúp giảm thiểu lỗi và tiết kiệm thời gian.
* Dễ bảo trì và mở rộng

Hibernate là một thư viện ORM (Object Relational Mapping) mã nguồn mở giúp lập trình viên viết ứng dụng Java có thể map các objects (POJO) với hệ quản trị cơ sở dữ liệu quan hệ, và hỗ trợ thực hiện các khái niệm lập trình hướng đối tượng với cơ dữ liệu quan hệ



**Persistence object** chính là các POJO object map với các table tương ứng của cơ sở dữ liệu quan hệ. Nó như là những "thùng xe" chứa dữ liệu từ ứng dụng để ghi xuống database, hay chứa dữ liệu tải lên ứng dụng từ database.

**Session Factory** là một interface giúp tạo ra session kết nối đến database bằng cách đọc các cấu hình trong Hibernate configuration. Mỗi một database phải có một session factory. Ví dụ nếu ta sử dụng MySQL, và Oracle cho ứng dụng Java của mình thì ta cần có một session factory cho MySQL, và một session factory cho Oracle.

*<?xml version='1.0' encoding='utf-8'?>*

*<!DOCTYPE hibernate-configuration PUBLIC*

*"-//Hibernate/Hibernate Configuration DTD//EN"*

*"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">*

*<hibernate-configuration>*

*<session-factory>*

*<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>*

*<property name="hibernate.connection.url">jdbc:mysql://192.168.10.13:3306/data*

*</property>*

*<property name="hibernate.connection.username">root</property>*

*<property name="hibernate.connection.password">root</property>*

*<property name="hibernate.connection.pool\_size">10</property>*

*<property name="show\_sql">true</property>*

*<property name="dialect">org.hibernate.dialect.MySQLDialect</property>*

*<property name="hibernate.current\_session\_context\_class">thread</property>*

*</session-factory>*

*</hibernate-configuration>*

**Hibernate Session:** Mỗi một đối tượng session được Session factory tạo ra sẽ tạo một kết nối đến database. Transaction là transaction đảm bảo tính toàn vẹn của phiên làm việc với cớ sở dữ liệu. Tức là nếu có một lỗi xảy ra trong transaction thì tất cả các tác vụ thực hiện sẽ thất bại. Query Hibernate cung cấp các câu truy vấn HQL (Hibernate Query Language) tới database và map kết quả trả về với đối tượng tương ứng của ứng dụng Java.

1. **Lý do lựa chọn hibernate thay vì JDBC?**

* Object mapping
* Với JDBC: Phải map các trường trong bảng với các thuộc tính của Java object một cách "thủ công".
* Với Hibernate: Hỗ trợ map một cách "tự động" thông qua các file cấu hình map XML hay sử dụng các annotation (@Entity, @Table, @Column, …)
* HQL

Hibernate cung cấp các câu lệnh truy vấn tương tự SQL, HQL của Hibernate hỗ trợ đầy đủ các truy vấn đa hình như, HQL "hiểu" các khái niệm như kế thừa (inheritance), đa hình (polymorphysm), và liên kết (association)

* Database Independent

Code sử dụng Hibernate là độc lập với hệ quản trị cơ sở dữ liệu, nghĩa là không cần thay đổi câu lệnh HQL khi chuyển đổi từ hệ quản trị CSDL MySQL sang Oracle, hay các hệ quản trị CSDL khác... Do đó rất dễ để thay đổi CSDL quan hệ, đơn giản bằng cách thay đổi thông tin cấu hình hệ quản trị CSDL trong file cấu hình

* Minimize Code Changes

Khi ta thay đổi (thêm) cột vào bảng:

* Với JDBC:
* Thêm thuộc tính vào POJO class.
* Thay đổi method chứa câu truy vấn "select", "insert", "update" để bổ sung cột mới. Có thể có rất nhiều method, nhiều class chứa các câu truy vấn như trên.
* Với Hibernate: Chỉ cần thêm thuộc tính vào POJO class. Cập nhật Hibernate XML mapping file để thêm map column – property 🡪 chỉ thay đổi duy nhất 2 file trên.
* Lazy Loading

Với những ứng dụng Java làm việc với CSDL lớn hàng trăm triệu bản ghi, việc có sử dụng Lazy loading trong truy xuất dữ liệu từ database mang lại lợi ích rất lớn.

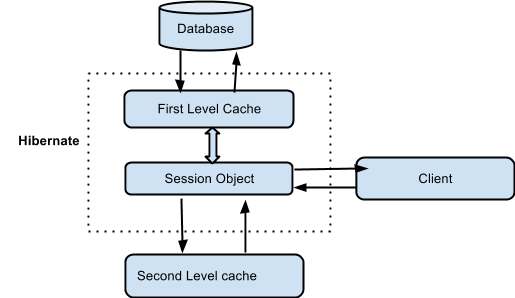
Ví dụ: Bảng user quan hệ 1-n với bảng document. Trong trường hợp này, User là class cha và Document là class con. Giả sử bảng document chứa 1 lượng lớn dữ liệu. Mỗi khi lấy thông tin user và không cần thiết phải lấy document tương ứng từ database nhằm để ứng dụng không bị chậm vì phải mất nhiều bộ nhớ để chứa toàn bộ document của user, thì áp dụng Lazy loading với FetchType.LAZY

Công dụng của FetchType.LAZY:

* Trì hoãn tải dữ liệu: Các dữ liệu liên quan đến mối quan hệ sẽ không được tải từ cơ sở dữ liệu ngay lập tức mà chỉ được tải khi cần thiết.
* Tăng hiệu suất ứng dụng: Tránh tải dữ liệu không cần thiết từ cơ sở dữ liệu, giảm tải cho máy chủ và tăng hiệu suất ứng dụng.
* Loại bỏ Try-Catch Blocks
* Với JDBC, nếu lỗi xảy khi tao tác với database thì sẽ có exception SQLexception bắn ra. Bởi vậy phải sử dụng try-catch block để xử lý ngoại lệ.
* Với Hibernate: Nó override toàn bộ JDBC exception thành Uncheck exeption, và không cần viết try-catch trong code nữa.
* Quản lý commit/rollback Transaction

Transaction là nhóm các hoạt động (với database) của một tác vụ. Nếu một hoạt động không thành công thì toàn bộ tác vụ không thành công.

* Với JDBC, lập trình viên phải chủ động thực hiện commit khi toàn bộ hoạt động của tác vụ thành công, hay phải rollback khi có một hoạt động không thành công để kết thúc tác vụ.
* Với Hibernate thì không cần quan tâm đến commit hay rollback, Hibernate đã tự quản lý rồi
* Hibernate Caching



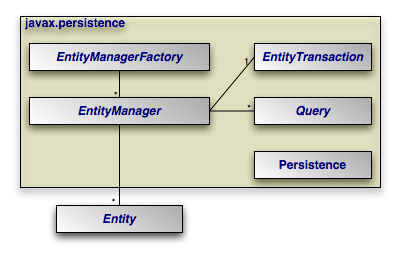
Hibernate cung cấp 1 cơ chế bộ nhớ đệm giúp giảm số lần truy cập vào database của ứng dụng và tăng performance đáng kể cho ứng dụng. Hibernate lưu trữ các đối tượng trong session khi transation được kích hoạt. Khi một query được thực hiện liên tục, giá trị được lưu trữ trong session được sử dụng lại. Khi một transaction mới bắt đầu, dữ liệu được lấy lại từ database và được lưu trữ session.

1. **Cài đặt, sử dụng Hibernate**
2. **Spring Data JPA**
3. **JPA**

JPA (Java Persistence API) là 1 interface lập trình ứng dụng Java, nó mô tả cách quản lý các mối quan hệ dữ liệu trong ứng dụng sử dụng Java Platform.

JPA cung cấp một mô hình POJO persistence cho phép ánh xạ các table/các mối quan hệ giữa các table trong database sang các class/mối quan hệ giữa các object. Vì vậy, thay vì truy vấn table hay các column bằng syntax của SQL, ta sẽ truy vấn trực tiếp trên các class, các field của class bằng JPA Query Language (JPQL). Với cách này, ta sẽ không cần quan tâm tới việc đang dùng loại database nào (mỗi loại database có thể có syntax khác nhau), dữ liệu database ra sao, …

JPA đi kèm với một tập hợp các annotations như @Table, @Entity, @Id và @Column.



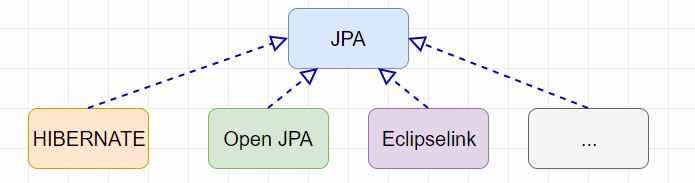
Entity là các đối tượng thể hiện tương ứng 1 table trong cơ sở dữ liệu. Khi lập trình, entity thường là các class POJO đơn giản, chỉ gồm các method getter, setter.

EntityManager là một interface cung cấp các API cho việc tương tác với các Entity như Persist (lưu một đối tượng mới), merge (cập nhật một đối tượng), remove (xóa 1 đối tượng).

EntityManagerFactory được dùng để tạo ra một thể hiện của EntityManager

* JPA là 1 tập các interface, còn Hibernate implements các interface ấy

Ngoài Hibernate ra, còn có 1 số framework khác như Open JPA, Eclipselink cũng thực hiện implements JPA nhưng Hibernate được sử dụng phổ biến hơn



* Specification
* JPQL
* JPQL cho phép định nghĩa các câu query dựa trên các entity chứ không dựa vào tên các cột, các bảng trong database.
* Cấu trúc và cú pháp của JPQL thì tương tự như cấu trúc và cú pháp của câu SQL.

Điều này giúp dễ dàng định nghĩa các câu query sử dụng JPQL nhưng nên nhớ rằng: mặc dù định nghĩa các câu query sử dụng các entity nhưng trong thực tế, lúc chạy, Hibernate hay bất kỳ thư viện nào implement JPA đều transform những câu query đó sang những câu SQL dành cho database với tên cột, tên bảng của database đó

Ví dụ:

*public interface UserRepository extends JpaRepository<User, Long> {*

*// Khi được gắn @Query, thì tên của method không còn tác dụng nữa*

*// Đây là JPQL*

*@Query("select u from User u where u.emailAddress = ?1")*

*User myCustomQuery(String emailAddress);*

*// Đây là Native SQL*

*@Query(value = "select \* from User u where u.email\_address = ?1", nativeQuery = true)*

*User myCustomQuery2(String emailAddress);*

*// JPQL*

*@Query("SELECT u FROM User u WHERE u.status = :status and u.name = :name")*

*User findUserByNamedParams(@Param("status") Integer status, @Param("name") String name);*

*// Native SQL*

*@Query(value = "SELECT \* FROM Users u WHERE u.name = :name", nativeQuery = true)*

*User findUserByNamedParamsNative( @Param("name") String name);*

*}*

1. **Spring Data**
2. **Spring Data JPA**

Nếu sử dụng JPA cùng với Spring framework trong dự án của thì hãy sử dụng Spring Data JPA.

Spring Data JPA là một module nhỏ trong một project lớn gọi là Spring Data project. Mục đích của Spring Data project là giảm thiểu các đoạn code lặp đi lặp lại liên quan đến phần thao tác với các hệ thống quản trị data khi phát triển các ứng dụng có sử dụng Spring framework. Ngoài Spring Data JPA hỗ trợ cho JPA giảm thiểu code để truy cập và thao tác với các hệ thống quản trị cơ cở dữ liệu, còn có Spring Data JDBC (cũng giống như Spring Data JPA), Spring Data LDAP (hỗ trợ Spring LDAP), Spring Data MongoDB (hỗ trợ cho MongoDB), …

Để đạt được mục đích giảm thiểu code như trên, Spring Data định nghĩa một interface chính tên là Repository nằm trong module Spring Data Common, module này sẽ được sử dụng cho tất cả các module còn lại trong Spring Data project. Nội dung của interface này đơn giản như sau:

*package org.springframework.data.repository;*

*import org.springframework.stereotype.Indexed;*

*@Indexed*

*public interface Repository<T, ID> {*

*}*

Vì interface này đơn giản như vậy nên sẽ có nhiều interface khác extend từ interface repository tuỳ thuộc vào module sử dụng. Và interface CrudRepository là một interface duy nhất extend interface Repository mà Spring Data JPA đang sử dụng.

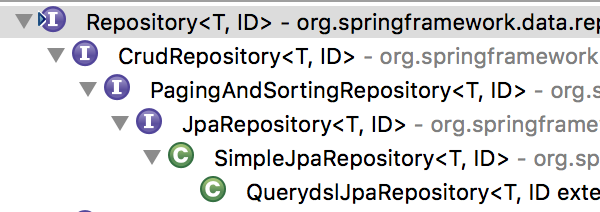
Interface CrudRepository với ý nghĩa create, read, update, delete cho phép thực hiện các thao tác cơ bản đến với các hệ thống data (không chỉ là database).

Để hỗ trợ việc phân trang và sắp xếp cho Spring Data JPA, thì còn có một interface khác là PagingAndSortingRepository.

Tất cả các interface trên đều nằm trong module Spring Data Common.

Trong Spring Data JPA, ở đây, chỉ có duy nhất một interface là JpaRepository kế thừa interface PagingAndSortingRepository. Với việc extend từ interface PagingAndSortingRepository, ta cũng có thể hình dung là Spring Data JPA có thể giúp giảm thiểu code cho các thao tác liên quan đến database.

Cấu trúc extend cho những interface:



Tham khảo:

<https://huongdanjava.com/vi/tong-quan-ve-spring-data-jpa.html>

<https://huongdanjava.com/vi/jpa-va-spring-framework.html>

<https://spring.io/projects/spring-data>

1. **Cascade**

Cascade là một tính năng giúp quản lý trạng thái của các đối tượng trong một mối quan hệ một cách tự động.

Ví dụ:

*@Entity*

*@Table(name = "company")*

*public class Company {*

*//...*

*@OneToMany(fetch = FetchType.LAZY, mappedBy = "company", cascade = CascadeType.REMOVE)*

*private Set<Employee> listEmployee = new HashSet<>();*

*//...*

*}*

*Nghĩa là, khi xóa Company thì Set<Employee> cũng bị xóa theo*

Một số loại cascade khác:

- CascadeType.PERSIST: Đồng bộ hóa các hoạt động thêm dữ liệu (INSERT) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.MERGE: Đồng bộ hóa các hoạt động cập nhật dữ liệu (UPDATE) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.REMOVE: Đồng bộ hóa các hoạt động xóa dữ liệu (DELETE) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.REFRESH: Đồng bộ hóa các hoạt động làm mới dữ liệu (REFRESH) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.DETACH: Đồng bộ hóa các hoạt động tách rời dữ liệu (DETACH) từ đối tượng gốc sang các đối tượng liên quan.

- CascadeType.ALL: Bao gồm tất cả các loại cascade đã đề cập trên (PERSIST, MERGE, REMOVE, REFRESH, DETACH).

1. **Fetch**
2. **Query DSL**

Tham khảo:

<https://www.baeldung.com/rest-api-search-language-spring-data-querydsl>

1. **Annotations**

* @OneToOne

Dùng để biểu diễn quan hệ 1-1

Ví dụ: 1 User có 1 Address duy nhất

Khi đó, từ User ta có thể lấy ra Address và từ Address ta có thể lấy ra User

*Address address = new Address ("Address 1");*

*User user = new User();*

*user.setAddress(address);*

*user.save(); // như vậy ta sẽ lưu dữ liệu xuống 2 bảng User và Address*

* @ManyToOne và @OneToMany

@ManyToOne cho phép bạn để ánh xạ cột khóa ngoại (foreign key) trong thực thể con kết nối để đối tượng con có thể có một đối tượng thực thể tham chiếu đến thực thể cha của nó.

Cho thuận tiện, để tận dụng các chuyển đổi trạng thái thực thể, nhiều lập trình viên chon ánh xạ thực thể con như một tập hợp (collection) trong đối tượng cha, vì mục đích nay JPA cung cấp chú thích @OneToMany.

|  |  |
| --- | --- |
| *public class Person {*  *@Id*  *private Long id;*  *private String name;*  *@ManyToOne*  *@JoinColumn(name = "address\_id")*  *private Address address;*  *}* | *public class Address {*  *@Id*  *private Long id;*  *private String city;*  *private String province;*  *@OneToMany(mappedBy = "address")*  *private Collection<Person> persons;*  *}* |

* @ManyToMany

Dùng để biểu diễn quan hệ n - n

|  |  |
| --- | --- |
| *public class Person {*  *@Id*  *private Long id;*  *private String name;*  *@ManyToMany(mappedBy = "persons")*  *private Collection<Address> addresses;*  *}* | *public class Address {*  *@Id*  *private Long id;*  *private String city;*  *private String province;*  *@ManyToMany(cascade = CascadeType.ALL)*  *@JoinTable(*  *name = "address\_person",*  *joinColumns = @JoinColumn(name = "address\_id"),*  *inverseJoinColumns = @JoinColumn(name = "person\_id")*  *)*  *private Collection<Person> persons;*  *}* |

* @Column

@Column là một annotation được sử dụng để ánh xạ một trường (field) của một entity (thực thể) vào một cột trong cơ sở dữ liệu. Annotation này cung cấp các thuộc tính để tùy chỉnh cách cột được tạo trong cơ sở dữ liệu.

**Fetch:**

**updatable:**

Thuộc tính updatable trong @Column annotation của JPA được sử dụng để chỉ định xem cột tương ứng có thể được sử dụng trong các câu lệnh SQL UPDATE hay không. Khi đặt giá trị của updatable là false (*@Column(name = "created\_date", updatable = false*) , nghĩa là cột đó sẽ không được cập nhật khi thực hiện các thao tác cập nhật trên entity tương ứng. Điều này hữu ích khi bạn muốn loại bỏ khả năng cập nhật một số cột trong trường hợp nào đó.

**columnDefinition:**

Được sử dụng để chỉ định kiểu dữ liệu và các thuộc tính khác cho cột tương ứng trong cơ sở dữ liệu. Thuộc tính này cho phép bạn mô tả cụ thể các đặc tính của cột, chẳng hạn như kiểu dữ liệu, độ dài, ràng buộc, và các thuộc tính khác tùy thuộc vào cơ sở dữ liệu mà bạn đang sử dụng.

Ví dụ, bạn có thể sử dụng columnDefinition để chỉ định kiểu dữ liệu VARCHAR và độ dài 100 cho một cột trong MySQL như sau:

*@Column(name = "first\_name", columnDefinition = "VARCHAR(100)")*

*private String firstName;*

Trong trường hợp này, khi entity được tạo, cột first\_name trong cơ sở dữ liệu sẽ được tạo với kiểu dữ liệu VARCHAR và độ dài 100.

Lưu ý rằng việc sử dụng columnDefinition có thể là cách cụ thể cho một cơ sở dữ liệu cụ thể và có thể không di động giữa các cơ sở dữ liệu khác nhau. Nếu bạn muốn ứng dụng của bạn có tính di động cao hơn, hãy xem xét sử dụng các annotation khác như @Length hoặc @Size để chỉ định độ dài hoặc kích thước của trường mà không liên quan trực tiếp đến cơ sở dữ liệu cụ thể.

* **@Enumerated**

@Enumerated là 1 annotation của JPA, sử dụng để chỉ định cách mà các giá trị của một enum sẽ được mapping vào db. Thường được sử dụng khi muốn lưu trữ giá trị của enum dưới dạng chuỗi (STRING) thay vì giá trị số (ORDINAL).

*@Enumerated(EnumType.STRING)*

*private ProductType type;*

ProductType là một enum và type sẽ được mapping vào column trong db có kiểu STRING. Khi lấy dữ liệu từ db và mapping vào Java object, JPA sẽ chuyển đổi giá trị từ chuỗi thành giá trị enum tương ứng.

Việc sử dụng EnumType.STRING thay vì EnumType.ORDINAL, đảm bảo tính rõ ràng và ổn định trong db. Khi sử dụng ORDINAL, mọi thay đổi trong enum có thể làm thay đổi thứ tự của các giá trị enum, gây khó khăn trong việc duy trì và cải thiện độ tin cậy của dữ liệu. STRING giúp giải quyết vấn đề này, tuy nhiên có thể làm cho db có kích thước lớn

* **@EmbeddedId**

Là 1 annotation của JPA, sử dụng để xác định 1 field hoặc 1 class nhúng (embedded) làm primary key cho 1 đối tượng entity. Nó thường được sử dụng trong các trường hợp mà khóa chính của đối tượng bao gồm nhiều field hoặc cần được ánh xạ vào 1 class đặc biệt để quản lý.

Khi sử dụng @EmbeddedId, cần tạo ra 1 class mới đại diện cho primary key, và sử dụng @Embeddable trên class này. Trong class đó, sẽ định nghĩa các field mà ta muốn sử dụng.

|  |  |
| --- | --- |
| *@Embeddable*  *public class EmployeeId implements Serializable {*  *private Long departmentId;*  *private Long employeeNumber;*  *// Constructors, getters, setters, and other methods*  *}* | *@Entity*  *public class Employee {*  *@EmbeddedId*  *private EmployeeId employeeId;*  *private String firstName;*  *private String lastName;*  *// Constructors, getters, setters, and other methods*  *}* |

# **C/ Caching dữ liệu**

1. **Giới thiệu**

Caching là một kỹ thuật trong lập trình máy tính được sử dụng để lưu trữ tạm thời các dữ liệu, kết quả của các phép toán, hoặc các tài nguyên có thể được truy cập một cách nhanh chóng. Mục tiêu của caching là giảm thời gian truy cập dữ liệu, tăng cường hiệu suất và giảm gánh nặng cho hệ thống.

Khi một ứng dụng yêu cầu dữ liệu, nó sẽ kiểm tra xem dữ liệu đã được lưu trong bộ nhớ cache hay chưa. Nếu dữ liệu đã tồn tại trong cache, ứng dụng sẽ lấy từ đó thay vì phải truy cập vào nơi lữu trữ gốc (ví dụ: cơ sở dữ liệu, API, hoặc tệp tin). Nếu dữ liệu không có trong cache, ứng dụng sẽ lấy từ nơi lưu trữ gốc và sau đó lưu vào cache để sử dụng cho các lần truy cập sau.

1. **Các phương pháp caching**

* **Spring Cache Abstraction**

Sử dụng các annotations (@Cacheable, @CacheEvict, @CachePut, @Caching) để quản lý cache một cách trừu tượng mà không cần quan tâm đến việc thực hiện cụ thể.

Nó không phải là một trình quản lý cache thực tế, mà là một API trừu tượng giúp bạn dễ dàng tích hợp với các trình quản lý cache khác như EhCache, Caffeine, Hazelcast, Guava, Redis,..

**@Cacheable:** Dùng để đánh dấu một phương thức rằng kết quả của nó có thể được cache. Khi phương thức được gọi, Spring sẽ kiểm tra xem kết quả của phương thức đó đã được cache chưa. Nếu có, kết quả từ cache sẽ được trả về thay vì thực hiện phương thức

**@CacheEvict:** Dùng để loại bỏ dữ liệu khỏi cache. Bạn có thể cấu hình nó để xóa một mục cụ thể khỏi cache hoặc để xóa tất cả các mục trong một cache cụ thể.

**@CachePut:** Luôn thực hiện phương thức và kết quả của nó sẽ được đưa vào cache theo khóa chỉ định. Điều này hữu ích khi bạn muốn cập nhật cache với một kết quả mới.

**@Caching:** Được sử dụng để nhóm nhiều chú thích caching trên một method duy nhất. Nó cho phép bạn áp dụng nhiều hành vi caching khác nhau cho một method, kết hợp hiệu ứng của @Cacheable, @CachePut, và @CacheEvict.

**@CacheConfig:** Dùng để chia sẻ một số thuộc tính cache giữa các method trong một lớp

Spring Cache rất linh hoạt. Không chỉ sử dụng các trình quản lý cache in-memory như EhCache, Caffeine mà còn có thể tích hợp với các hệ thống cache phân tán như Redis, Hazelc

Mặc dù nó được thiết kế để làm việc với bất kỳ trình quản lý cache nào nhưng không yêu cầu phải tích hợp với một trình quản lý cụ thể. Khi không cung cấp một cấu hình cache cụ thể, Spring sẽ tạo ra một SimpleCacheMemory với một ConcurrentMapChe mặc định, là một cách đơn giản để thực hiện caching in-memory mà không cần bất kỳ sự tích hợp đặc biệt nào

ConcurrentMapCache là một cách tiện lợi để bắt đầu với caching trong Spring nếu bạn không có nhu cầu phức tạp và không cần một giải pháp caching mạnh mẽ như Caffeine, Redis, EhCache, ... Đây là một cách tốt để thêm caching vào ứng dụng của bạn nếu bạn chỉ cần cải thiện hiệu suất cho các trường hợp sử dụng đơn giản và không yêu cầu tính năng đầy đủ của một trình quản lý cache chuyên nghiệp.

* In-Memory Cache

In-Memory Cache là một hình thức caching trong đó dữ liệu được lưu trữ trực tiếp trong bộ nhớ chính (RAM) của máy chủ. Giúp việc truy xuất dữ liệu nhanh hơn nhiều so với việc phải đọc từ ổ đĩa cứng hoặc một nguồn dữ liệu từ xa như cơ sở dữ liệu hoặc dịch vụ web.

Tuy nhiên, RAM có kích thước hạn chế và dữ liệu trong cache thường bị mất khi ứng dụng hoặc máy chủ bị tắt, cần phải có chiến lược để quản lý việc đặt vào và loại bỏ khỏi cache (như eviction policies), cũng như để đồng bộ hóa cache trong một hệ thống phân tán.

Công cụ phổ biến để thực hiện In-Memory Cache

* HashMap hoặc ConcurrentHashMap: Sử dụng một bản đồ đơn giản trong bộ nhớ để lưu trữ dữ liệu. Đây là cách đơn giản nhất nhưng không cung cấp các tính năng quản lý cache tự động.
* Caffeine: Một thư viện caching nhanh chóng, mạnh mẽ với các cơ chế quản lý cache như evictions, loading, và computation.
* Guava: Trước khi có Caffeine, Guava là lựa chọn phổ biến cho in-memory cache với các tính năng tương tự nhưng không hiệu suất cao như Caffeine.
* Distributed Cache

Distributed Cache là một hệ thống cache được thiết kế để hoạt động trên nhiều máy chủ hoặc nút trong một mạng. Mục đích chính của nó là để tăng tốc độ truy cập dữ liệu bằng cách lưu trữ bản sao của dữ liệu thường xuyên được truy cập hoặc tính toán tốn kém trên nhiều địa điểm trong mạng.

Các Đặc Điểm Chính:

* Phân Tán và Đồng Bộ: Dữ liệu được phân tán giữa nhiều nút hoặc máy chủ và thường được đồng bộ hóa hoặc sao chép giữa các nút để đảm bảo tính nhất quán của dữ liệu.
* Tăng Tốc Độ và Giảm Độ Trễ: Cung cấp dữ liệu từ cache gần với vị trí của người dùng giúp giảm độ trễ và tăng tốc độ truy cập dữ liệu.
* Tăng Khả Năng Mở Rộng: Hỗ trợ khả năng mở rộng ngang, cho phép hệ thống mở rộng bằng cách thêm nhiều nút cache hơn mà không ảnh hưởng đến hiệu suất.
* Tăng Khả Năng Chịu Lỗi: Một số hệ thống cache phân tán cung cấp khả năng chịu lỗi, nơi mất mát một nút không làm mất mát dữ liệu hoặc làm gián đoạn dịch vụ do dữ liệu có thể được truy cập từ các nút khác.

Các Công Cụ Distributed Cache Phổ Biến:

* Redis: Một cơ sở dữ liệu key-value nhanh chóng, hỗ trợ các cấu hình như cache phân tán, với khả năng lưu trữ dữ liệu trong bộ nhớ và cung cấp các cấu trúc dữ liệu phức tạp.
* Memcached: Một hệ thống caching phân tán nhẹ, dễ dàng cấu hình, tập trung vào việc lưu trữ dữ liệu key-value đơn giản trong bộ nhớ.
* Hazelcast: Một in-memory data grid, cung cấp các cấu trúc dữ liệu phân tán và hỗ trợ transaction, được sử dụng để tạo ra các hệ thống có khả năng chịu lỗi và mở rộng cao.

Về bản chất các công cụ này cũng hoàn toàn có thể được sử dụng cho mục đích In-memory cache tuy nhiên khi nhắc đến việc caching trên các hệ thống phân tán, chúng sẽ là những lựa chọn hàng đầu. Khi mà các công cụ In-memory cache không đáp ứng được hoặc sẽ khó để triển khai trên các ứng dụng hệ thống phân tán.

**Ưu điểm:**

* Tăng hiệu suất: giảm thời gian truy cập dữ liệu từ nơi lưu trữ gốc, giúp ứng dụng hoạt động nhanh chóng hơn, từ đó giúp cho ứng dụng của bạn tăng hiệu suất làm việc.
* Giảm gánh nặng cho cơ sở dữ liệu: giảm số lượng truy vấn trực tiếp đến cơ sở dữ liệu, giảm áp lực lên cơ sở dữ liệu và cải thiện hiệu suất hệ thống. Với những ứng dụng lớn, với hàng trăm, hàng ngàn query vào cơ sở dữ liệu mỗi giây, thì caching là một giải pháp cứu cánh.
* Cải thiện trải nghiệm người dùng: Cung cấp trải nghiệm người dùng mượt mà hơn và giảm độ trễ.
* Cải thiện độ ổn định hệ thống: Giúp cải thiện độ ổn định của hệ thống bằng cách giảm độ trễ và gánh nặng, giúp hệ thống chạy mượt mà hơn.

**Lưu ý khi sử dụng:**

* Rủi ro dữ liệu lỗi thời (Stale data): Nếu dữ liệu trong cache không được cập nhật thường xuyên, có thể dẫn đến sử dụng dữ liệu lỗi thời (stale) => cần phải kiểm tra và cập nhật dữ liệu trong cache thường xuyên để tránh cho nó rơi vào tình trạng này
* Quản lý thời gian sống (Time-to-live) cẩn thận: Thiết lập thời gian sống (time-to-live) sao cho phù hợp với tính chất của dữ liệu và mức độ thay đổi. Nhưng cũng phải cẩn thận và tính toán kỹ lưỡng cho khoảng thời gian này để tránh dữ liệu lỗi thời hoặc re-fresh quá nhiều khiến việc cache trở nên vô nghĩa.
* Quản lý bộ nhớ hiệu quả: Caching có thể làm tăng tiêu tốn bộ nhớ, đặc biệt là khi lưu trữ lớn lượng dữ liệu. Chính vì thế, kiểm soát lượng bộ nhớ sử dụng cho caching để tránh tình trạng chiếm dụng bộ nhớ quá mức.
* Xác định rõ mục tiêu và áp dụng chiến lược hợp lý: Đặt rõ mục tiêu bạn muốn đạt được với caching, liệu bạn đang tập trung vào tăng tốc độ, giảm áp lực cho cơ sở dữ liệu hay cải thiện trải nghiệm người dùng từ đó lựa chọn chiến lược caching phù hợp với yêu cầu và tính chất của ứng dụng.