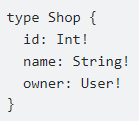
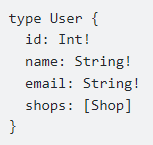
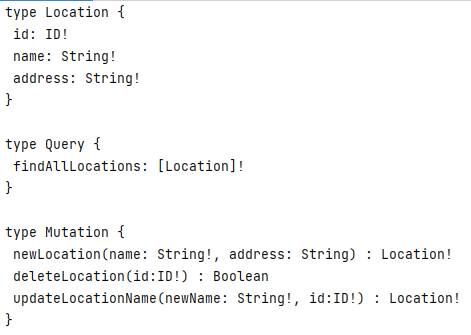
[**https://stackoverflow.com/questions/71179660/spring-boot-3-jakarta-and-javax**](https://stackoverflow.com/questions/71179660/spring-boot-3-jakarta-and-javax)

**Từ Spring Boot ver 3.x trở đi, sẽ dùng thư viện jakarta của Jakarta EE 9 thay vì javax. Và sẽ dùng Java 17.**

**-> khi import các thư viện vẫn xài thư viện của javax, thì down parent version của Spring Boot xuống ver 2.x.**

**Hoặc đơn giản hơn là cứ vào luôn solution copy file pom.xml cho nhanh.**

1. **In-memory database**
   * **IMDB:** tức là database được lưu trữ trực tiếp trên RAM của máy tính, thay vì lưu trữ trên disk cứng. IMDB sẽ cho tốc độ truy xuất nhanh hơn đáng kể cho với database lưu trữ trên disk truyền thống.
   * Nhưng nhược điểm là vì chúng lưu trên RAM nên khả năng lưu trữ có giới hạn và tính toàn vẹn dữ liệu không cao, khả năng bảo mật dữ liệu kém.
   * Một số IMDB: H2, Redis, SQLite, …
2. **Web services**
   * **Web services** **(dịch vụ web):** hiểu nôm na cũng cấp một chuẩn giao tiếp để cho 2 máy tính có thể giao tiếp với nhau (client – server) và truyền tải dữ liệu liền mạch mà không bị bó buộc vào hệ điều hành hay ngôn ngữ lập trình.
   * Ví dụ: tức là ta có thể dùng web service để cho app Java và app Python có thể giao tiếp với nhau. Hoặc cho một điện thoại có thể giao tiếp với một máy chủ server.
   * Cách thức: server định nghĩa một standard format (định dạng chuẩn) cho request và response. Khi user gửi request thông qua giao thức HTTP hoặc HTTPS, sử dụng ngôn ngữ thường là JSON hoặc XML, sau đó server xử lí request đó và trả về response.
   * **SOAP** (Simple Object Access Protocol): là phương thức gửi tin nhắn để giao tiếp giữa 2 hệ thống bằng ngôn ngữ XML. SOAP dùng **WSDL: Web Services Description Language** (tóm lại là syntax của nó giống như XML)
     + <https://tubean.github.io/2018/12/web-service-s%E1%BB%B1-kh%C3%A1c-nhau-gi%E1%BB%AFa-soap-v%C3%A0-restful-web-service-trong-java/>
     + **SOAP** khá giống **REST:** đều là phương thức để giao tiếp giữa 2 hệ thống khác nhau. Đều dùng một phương thức na ná HTTP, cũng có header và body
     + **SOAP** khác với **REST:**
       - SOAP chỉ trả về kiểu XML, còn REST có thể trả về plain text, HTML, JSON, XML.
       - SOAP chậm hơn REST vì SOAP cần parsing nhiều hơn, do đó có độ dài và dung lượng cao hơn.
       - SOAP bảo mật hơn REST, giúp truyền tải dữ liệu đáng tin cậy hơn
       - SOAP ko hề có caching, REST thì có.
       - Ví dụ: giả sử bạn muốn truy xuất thời tiết hôm nay cho một thành phố cụ thể từ một máy chủ cung cấp thông tin thời tiết, URL RESTful của bạn sẽ trông giống như
         * http://weatherdata.org/data/weather/uk/london
       - Bên cạnh đó thì để có được cùng một dữ liệu bằng SOAP, bạn cần tạo một thông báo XML có tiêu đề và nội dung và gửi nó
         * <http://www.webservicex.net/globalweather.asmx?op=GetWeather>
       - Tóm lại, RESTful đơn giản, linh hoạt và dễ chịu hơn nhiều so với SOAP trong Java.
       - **Tóm cái váy lại là SOAP lỗi thời, các ông lớn đéo ai dùng nữa, học cho biết.**
   * **API** (Application Programming Interface): nhẹ hơn và sắp xếp hợp lý hơn web services
   * **Microservices:** giống với API. Tức là microservice bao gồm hoàn toàn những service riêng biệt. Tức là những service đó khi gom lại với nhau sẽ tạo thành một hệ thống hoàn chỉnh có đầy đủ chức năng,
3. **REST**
   * **REST (Representational State Transfer) :** REST hiểu đơn giản nó là một bộ các ràng buộc và quy định để thiết kế hệ thống. Hệ thống tuân theo REST, gọi là hệ thống RESTful, đặc trưng bởi việc chúng ko có trạng thái (stateless) và hoạt động theo mô hình client-server tách biệt.
   * Hệ thống hoạt động theo mô hình client-server, trong đó server là tập hợp các service nhỏ lắng nghe các request từ client. Với từng request khác nhau thì có thể một hoặc nhiều service xử lý.
   * Stateless: nghĩa là server và client ko lưu trạng thái của nhau => mỗi request lên server thì client phải đóng gói thông tin đầy đủ rồi gửi lên để server hiểu được.
   * Khả năng caching: các response có thể lấy ra từ cache. Bằng cách cache vào response, client gửi request mà lặp lại (kiểu như phân trang) thì có thể lấy dữ liệu từ cache => giảm tải việc server phải xử lí request, client cũng nhận được thông tin nhanh hơn.
   * Phân lớp hệ thống: tách biệt hoàn toàn các lớp (request handler, service handler, database handler), mỗi lớp chỉ giao tiếp với lớp ở trên hoặc dưới nó => tách biệt các thành phần => dễ mở rộng.
   * Trong kiến trúc REST, client gửi request tới server để modify resource và sau đó server sẽ gửi response tới client.
4. **GraphQL**
   * Nhược điểm của REST API là chúng luôn luôn trả về cấu trúc data cố định trong response mà dev define ra từ trước, mà điều này sẽ làm tốn thời gian khi ta gửi request lên và response trả về nhiều data thừa ko cần thiết, hoặc là thiếu những data mà ta cần.
   * VD: khi ta gửi cần request để get data bao gồm user, cart, product, ta sẽ phải gửi 3 request tới 3 endpoint khác nhau: /user, /cart, /product (bởi vì dev đã design API như vậy). GraphQL có thể linh hoạt hơn bằng cách gom cả 3 data cần get đó vào 1 request duy nhất và gửi lên server và nhận về response bao gồm cả user, cart, product.
   * GraphQL ra đời với mục đích có thể linh hoạt hơn trong việc lựa chọn ta muốn nhận data gì từ response.
   * GraphQL là query language cho API, dùng để request chính xác những data mà ta muốn từ server (có thể nhiều hơn hoặc ít hơn). Điều này sẽ làm tăng sự linh hoạt và performance.
   * **Schema:** hiểu nôm na giống như request body của REST API. Được viết bởi **SDL -** **Schema Definition Language.** Khá hao hao JSON.
   * GraphQL có một **hệ thống type** riêng để xác định **schema** của chúng. Schema bao gồm data point, data types, quan hệ giữa chúng và **Query** để select dữ liệu hay **Mutation** thao tác dữ liệu.
     + ****
     + ****
     + **!** là trường bắt buộc
     + owner: User để thể hiện foreign key owner của Shop là User.
     + shops: [Shop] để thể hiện quan hệ many rằng User có nhiều Shop. Array được kí hiệu bởi []
     + 1 schema chỉ có thể có 1 root Query và 1 root Mutation.
   * **GraphiQL** là một web app để giao tiếp với graphQL server và execute query hay mutation.
   * **Resolver:** có chức năng khá tương đồng với RestController, giúp cho app Spring có thể nhận diện và call đúng method ta muốn trong request cho GraphQL query. Chỉ khác là ta sẽ ko cần phải mapping endpoint như RestController mà ta chỉ cần map với tên method trong Resolver là được.
     + **QueryResolver** sẽ dùng để resolve query. Khi code ta sẽ tạo 1 component implement class **GraphQLQueryResolver,** và tên method trong đó sẽ match tới tên query trong schema.
     + **MutationResolver** sẽ dùng để resolve mutation. Khi code ta sẽ tạo 1 component implement class **GraphQLMuationResolver,** và tên method trong đó sẽ match tới tên mutation trong schema.
   * **GraphQLError:** khi có exception được throw ra, reponse trả về sẽ có thêm field là error và bên trong chứa message lỗi. Trong này sẽ chứa **extentions** có dạng key value giống như trên.
5. Annotation
   * **@RestController = @Controller + @ResponseBody**
     + Mặc định @Controller sẽ trả về **View** (tức là HTML thuần, chỉ dùng cho **HTML template engine** như **Thymleaf**…). Nếu muốn trả về kiểu JSON/XML thì thêm annotation @ResponseBody.
     + Còn @RestController là đã bao gồm @Controller + @ResponseBody rồi. Mặc định sẽ trả về JSON/XML thay vì HTML.
   * **@RequestMapping**
     + Có thể dùng bên trên class controller, để chỉ định endpoint gốc cho toàn bộ method bên trong nó
     + Hoặc vẫn có thể dùng bên trên method trong controller (ít dùng)
       - @RequestMapping(value = "/users", method = RequestMethod.GET)
   * **@Entity**
     + Đánh dấu 1 class java thành một entity để có thể thao tác với database.
   * **Tại sao bỏ annotation @Repository mà app vẫn run được?** Vì bản chất repository khi viết trong code là interface, nhưng khi app SpringBoot chạy thì sẽ tự tạo ra class implementation của interface đó -> tức là đã tạo ra bean rồi. @Repository lúc này chỉ mang tính chất là marker anotation.
6. **Microservices Architecture (MSA)**
   * Trước khi kiến trúc microservice ra đời, các app được build theo **kiến trúc monolithic.**
   * **Monolithic architecture** nghĩa là toàn bộ app đó được build thành 1 file duy nhất là được deploy lên một server duy nhất. Nhược điểm của kiến trúc monolith là: code quá phụ thuộc vào nhau, chỉ cần một thay đổi nhỏ ở service này sẽ ảnh hưởng tới toàn bộ các service khác liên quan tới nó, dẫn tới việc phải test, build và deploy lại toàn bộ project đó. Việc này sẽ càng trở nên khó khăn khi app càng mở rộng
   * **Kiến trúc microservices** được ra đời để khắc phục những nhược điểm của kiến trúc monolith: hệ thống này sẽ phân tách các service thành các component hoàn toàn riêng biệt mà vẫn có thể giao tiếp với nhau.
   * Ưu điểm của microservice thì cũng nhiều:
     + Mỗi một service sẽ được tách thành một app riêng, được build trên một server riêng, và có database riêng biệt. Điều này sẽ giảm tải quá trình test, build và deploy khi code thay đổi.
     + Các service được tách thành các app riêng biệt, vậy nên ta có thể dùng các ngôn ngữ khác nhau cho từng service mà vẫn có thể giao tiếp được giữa các service với nhau.
     + Khi hệ thống mở rộng thì khi ta chỉnh sửa service nào sẽ ko ảnh hưởng tới các service còn lại vì vốn dĩ các service được build ở các server khác nhau.
     + Khi ta cần deploy một service lên 1 server khoẻ hơn thì với cloud computing ta chỉ cần thực hiện vài cú click.
   * Nhược điểm của kiến trúc này thì cũng có:
     + Sẽ làm hệ thống trở nên phức tạp hơn, code khó đọc hơn -> yêu cầu trình độ dev cao hơn để có thể hiểu luồng đi của code hay cách các service gọi nhau như thế nào.
     + Khi truy vấn mà phải đi qua nhiều service sẽ khiến code chạy chậm hơn.
     + Nếu hệ thống ko quá to mà build theo kiến trúc microservice thì hệ thống sẽ rối rắm một cách ko cần thiết và tốn thời gian dev.
7. **Eureka** **server**
   * Eureka server (được tạo ra bởi Netflix) là một server để đăng kí service trong hệ thống microservices. Tức là khi ta đăng kí service đó dưới một cái tên (name) thì khi cần gọi nó ta chỉ cần gọi thông qua name thay vì IP address và port.
   * Để apply hệ thống này ta cần deploy 2 thứ :
     + **Eureka server** : là server để các microservice register những service của mình vào
     + **Eureka client** : khi các service của microservices được register vào eureka serser, chúng sẽ được gọi là các eureka client.
8. **Spring Data REST**
   * Trước khi có Spring Data REST, viết REST web service rất nhàm chán vì những boilerplate code lặp đi lặp lại (viết controller, service rồi mới chọc tới repository) mà chỉ để thao tác CRUD.
   * Spring Data REST được build dựa trên Spring Data Repository, sẽ tự động export thành các Rest resource. Vì khi
   * build app Spring Boot, Spring Data Rest sẽ được cấu hình tự động.
   * Spring Data REST sẽ quét toàn bộ repository trong app và tạo ra các end point cho các thao tác CRUD cơ bản. Các endpoint này map với tên Entity và thêm chữ ‘s’.
   * Từ đó, ta đã có sẵn các endpoint cho các thao tác CRUD cơ bản mà ko cần phải tạo các controller hay service một cách lặp đi lặp lại nữa.
9. **REST Template**
   * Khi một API nào đó trả về response những data mà ta đã cần, ta ko nhất thiết phải tạo lại 1 object mới từ response đó để có thể sử dụng. Thay vào đó ta sẽ sử dụng REST Template cho tiện.
   * Reponse trả về từ một REST API ko phải là JSON, mà chúng là một **REST Template**, JSON là chỉ là format của response đó.
   * REST template dùng để tiêu thụ (sử dụng) kết quả trả về từ REST API. Tức là REST Template sẽ lấy kết quả (response) từ REST API trả về, convert từ JSON sang kiểu entity (mapping sang object) trong java để ta có thể sử dụng trực tiếp.
10. **Spring Security**
    * **Authenication? Authorization?**
      + <https://techmaster.vn/posts/36295/spring-security-ban-sau-ve-authentication-va-authorization-p1>
      + **Authentication (xác thực)**
        - <https://kipalog.com/posts/Authentication-story-part-1--Authentication-la-lam-gi>
      + **Authentication** là quá trình kiểm tra danh tính của một tài khoản đang truy cập vào hệ thống thông qua một hệ thống xác thực. Hiểu nôm na là **quá trình xác thực ai đang truy cập vào hệ thống.** 
        - Nếu ko có authentication, hệ thống sẽ ko biết ai đang truy cập hệ thống để có phản hồi phù hợp.
        - **Authentication diễn ra trước khi Authorization.**
        - Nếu hệ thống ko thể authenticate request gửi lên, sẽ trả về lỗi 401 Unauthorized.
      + **Authorization (uỷ quyền)**
        - <https://viblo.asia/p/phan-biet-su-khac-nhau-giua-authentication-va-authorization-Eb85oad4Z2G>
      + **Authorization** là quá trình kiểm tra xem người dùng đã authenticated có quyền truy cập vào các tài nguyên nào của hệ thống. Hiểu nôm na là **quá trình cấp quyền cho ai đó thực hiện những chức năng gì, truy cập vào tài nguyên gì của hệ thống.**
        - Nếu ko có authorization, user đó sẽ ko có quyền thực hiện chức năng gì (ngoại trừ những chức năng được permit all).
        - **Authorization diễn ra sau Authentication.**
        - Nếu hệ thống nhận request đã authenticated nhưng ko được authorized để thực hiện chức năng nào đó, sẽ trả về lỗi lỗi 403 Forbidden
11. **Swagger**
    * Khi code thì dev FE và dev BE cần thống nhất API sẽ dùng method gì, endpoint là gì, parameters gì, header gì, input và output gì, authentication method gì,… để có thể mapping với nhau. Hoặc hiểu đơn giản là khi ta cần định nghĩa trước API đó trông sẽ như thế nào, thì ta cần một tài liệu API document chung để thống nhất giữa 2 bên.
    * **Swagger** là một tool để build OpenAPI specification. API specification này được viết bằng YAML hoặc JSON. Swagger sẽ tự động update API document mỗi khi run project. Và ta có thể call API trực tiếp qua Swagger UI mà ko cần Postman.
    * Swagger bao gồm 3 tool chính:
      + Swagger Editor: sử dụng để thiết kế và xây dựng các APIs theo cách hoàn toàn mới hoặc sửa đổi các APIs hiện có với việc tận dụng một file config.
      + Swagger UI: cho phép từ file config để tạo ra API Documentation dưới dạng HTML, CSS, JS… để hiển thị lên browser.
      + Swagger codegen: dùng để generate ra code từ file config có sẵn.
    * **SpringFox** là một Swagger intergration (tích hợp) cho Spring Boot.
12. **JPA/Hibernate**
    * **Java Persistence API** là cơ sở để quản lí dữ liệu quan hệ trong app Java. JPA ko phải là tool hoặc framework, JPA là 1 tập hợp các interface để truy cập, duy trì và quản lí data giữa object Java và CSDL
      + JPA chỉ là specification (định nghĩa), chứ ko phải implementation
      + Nó là một tập hợp các quy tắc và hướng dấn để set các interface để triển khai ORM
    * **Hiểu hôm na JPA là một tập hợp các interface chứa những method để có thể thao tác với data giữa app Java và database**
    * JPA định nghĩa những API để thực hiện những việc : data conversion, querying, validation, schema generation

**=> vì JPA là tập hợp các interface, nên sẽ cần implementation để triển khai ORM (JPA providers/ORM frameworks).**

* + **Persistence data (dữ liệu bền vững)** là một khái niệm nói đến vấn đề rằng nếu dữ liệu không được lưu trữ khi hệ thống ngưng hoạt động cho đến khi nó hoạt động lại thì tất cả các dữ liệu phải được khởi tạo và xử lý lại từ đầu. Vậy nên ta cần đảm bảo rằng dữ liệu vẫn có thể sống kể cả khi ứng dụng ngưng hoạt động và có thể tải lên để xử lý tiếp khi hệ thống hoạt động trở lại. **Hiểu nôm na là cứ dữ liệu gì cần lưu trữ lâu dài thì phải lưu vào database.**
  + Entity design
    - Values
    - Entity types
  + Primary key:
    - Identifier: @Id
    - Genterated Strategy:
    - Compose identifier: @EmbbedId
  + Relationship
    - OneToOne
    - OneToMany
    - ManyToMany
  + Inheritance:
    - Single table
    - Joined
    - Table per class
    - Mapped superclass:
  + Data Conversion:
    - DTO
    - JSONView
  + <https://techmaster.vn/posts/36269/trang-thai-cua-entity-thuc-the-trong-hibernate-va-jpa>
  + JPA/Hibernate thay đổi tư duy lập trình viên từ câu lệnh SQL sang chuyển đổi trạng thái entity. Khi entity được quản lý bằng JPA/Hibernate thì mọi thay đổi của entity sẽ được tự động truyền vào trong cơ sở dữ liệu (database). Hiểu đơn giản là nếu dùng Hibernate thì ta sẽ ko phải xài các câu lệnh SQL nữa mà ta sẽ thực hiện những thay đổi trực tiếp object được ORM ánh xạ sang record trong table.
  + Thao tác với các entity (cùng với liên kết của chúng) dễ dàng hơn nhiều so với việc viết bằng câu lệnh SQL. Không có thư viện ORM thì mỗi khi thêm một cột mới vào sẽ đòi hỏi sửa đổi tất cả các câu lệnh INSERT/UPDATE liên quan.
  + Nhưng đi kèm theo đó là vì Hibernate sẽ tự động làm hết mọi thứ cho chúng ta nên chúng ta sẽ khó kiểm soát các câu lệnh SQL được thực thi khi nào và như thế nào. Có nghĩa là sau khi ta code xong ta vẫn phải debug để xem câu lệnh SQL được hibernate sinh ra có đúng theo ý ta không.
  + Muốn biết sự thay đổi entity đã được apply vào database chưa, ta cần phải biết mối quan hệ giữa entity và table đang lưu trữ nó là gì. **Pesistence context** sẽ lưu những thông tin đó.
  + **Persitence context** mô tả mỗi quan hệ giữa tất cả các entity instance trong app và mối quan hệ của chúng ở dưới database.
  + Có 4 state của entity instance trong persistence context:
    - **Transient/New** (tạm thời) là một object vừa mới được tạo ra mà chưa được kết nối với persistence context và cũng chưa được ánh xạ vào bất cứ bảng nào trong database. Vậy nên persistence context sẽ ko biết sự tồn tại của object này.
    - **Persistent/Managed** (bền vững) là một object được tạo ra và đã được kết nối với persistence context. Bất kì thay đổi nào lên object ở state này sẽ đều tự động truyền xuống database mà ko cần gọi persist, merge, remove.
    - **Detached/Unmanaged** (đã bị tách riêng ra) là một object đã bị tách khỏi persistence context khi persistence context đang chạy đó bị đóng lại. Những thay đổi này lên object sẽ ko còn tự động truyền xuống database nữa.
    - **Removed** (đã bị xoá): là một object đã bị detached nhưng record ứng với object này đã bị xoá khỏi database.
  + Để ta kiểm soát được việc modify object đó có được truyền xuống database không, ta cần thay đổi state của các entity đó. Khi đó ta sẽ dùng **EntityManager**.
  + EntityManager có một số method:
    - Persist: save entity ở state transient xuống database và chuyển state entity thành persistent.
    - Find: lấy 1 object ở dưới database, và entity trả về sẽ có sẵn state là persistent. Vì vậy những thay đổi lên object đó sẽ được truyền xuống database mà ko cần gọi lại method persist.
    - Merge: save entity ở state detached xuống database nhưng entity đó vẫn sẽ ở state detached.
    - Remove: detach một entity ở state persistent và xoá khỏi database.
  + **Note:** tất cả những thay đổi này chỉ áp dụng trên persistence context và sẽ chỉ được truyền xuống database khi Persistent context đó kết thúc.
  + EntityManager là của JPA. Hibernate thì có Session, là implementation của EntityManager, vây nên Session của hibernate sẽ có nhiều method hơn.
  + Khi có những entity associations nằm trong entity. Việc lúc nào cũng lấy những entity con nằm trong entity cha ta cần lấy sẽ hơi thừa thãi và tiêu tốn tài nguyên cho database. **Fetch type** của Hibernate sẽ xử lí việc này:
    - **EAGER**: lấy toàn bộ value mà nằm trong entity mà ta fetch. Hibernate mặc định sẽ dùng fetch type này cho quan hệ **OneToOne** và **ManyToOne**.
    - **LAZY**: chờ việc lấy value nằm trong entity cho tới khi chúng được refer. Tức là khi chưa được refer, entity type này sẽ không có giá trị, sau khi được refer thì chúng sẽ lấy dữ liệu dưới database và gán vào nên sẽ có giá trị. Hibernate mặc định sẽ dùng fetch type này cho quan hệ **OneToMany** và **ManyToMany**.
  + Entity relationship thường phụ thuộc vào sự tồn tại của entity khác, ví dụ Person – Address. Nếu ko có person, address entity sẽ ko có ý nghĩa. Vậy nên khi ta xoá Person, Address nên bị xoá theo. **Cascade** là cách để đạt được điều này.
  + **CascadeType** cho phép ta thực hiện hành động persistence nào đó trên Entity thì những entity assiociations cũng sẽ được thực hiện theo.
    - **ALL:** tương ứng với tất cả các loại cascade. cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}
    - **PERSIST:** Nếu đối tượng cha được persisted vào persistence context, thì các đối tượng tham chiếu tới nó cũng được persisted.
    - **DETACH:** tương tự
    - **MERGE:** tương tự
    - **PERSIST:** tương tự
    - **REMOVE:** tương tự
  + **JPQL (Java Persistence Query Language)** là query được viết cho Entity manager để có thể trả về entity. Syntax khá giống SQL và sẽ tự động convert sang câu SQL phù hợp với database mà app đang dùng. (trông có vẻ ít dùng nên ko cần học)
    - Typed query
    - Named query
  + **Hibernate** là một trong những JPA Providers/ORM framework (**Object Relational Mapping**) phổ biến giúp mapping giữa record dưới database và object trong Java. Hiểu đơn giản đây là một implementation của JPA API
  + **Spring Data JPA** cung cấp code và code generation tools để ta có thể dễ dàng sử dụng JPA. Nó cung cấp cho ta những đoạn code đơn giản như find, save, remove… cho những thao tác select, insert, delete đơn giản và có thể tạo ra những method cho những thao tác phức tạp hơn (bằng cách tự động generate JPQL). Việc này sẽ do **Spring Data common** làm.
  + Interface **JPARepository** thuộc library **Spring Data JPA**, extends từ interface **CrudRepository** thuộc library **Spring Data Common => Spring Data JPA** ko liên quan gì tới **Hibernate**