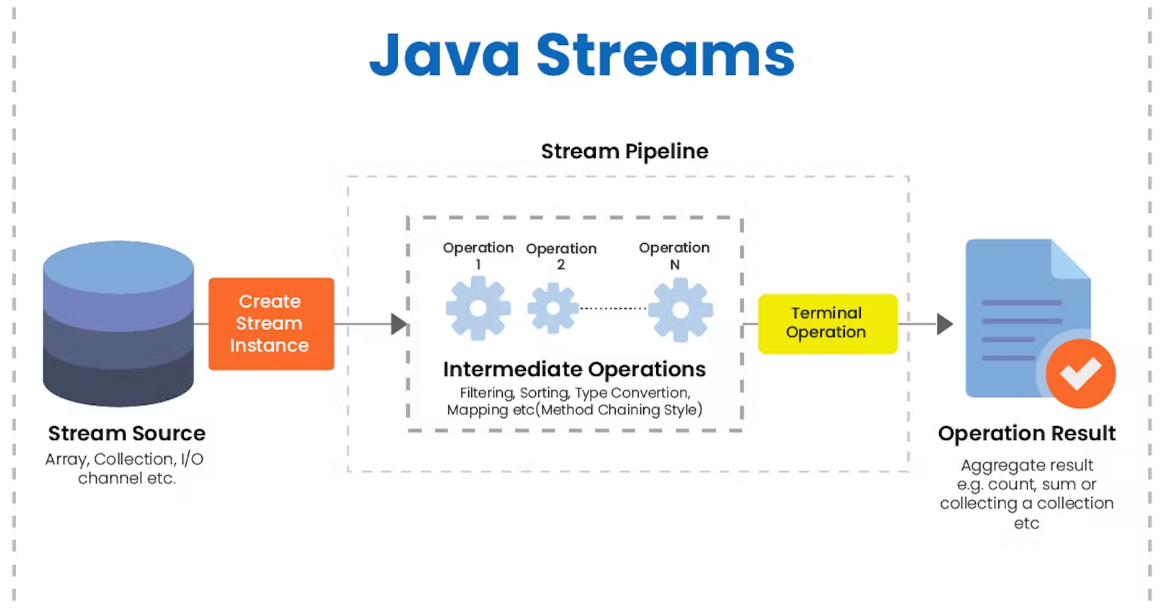
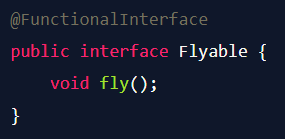
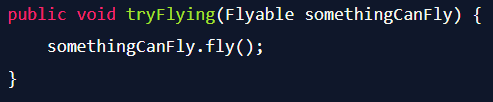
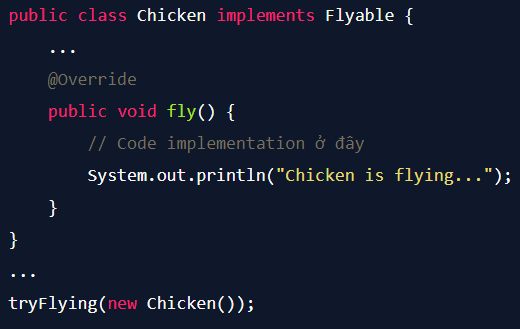
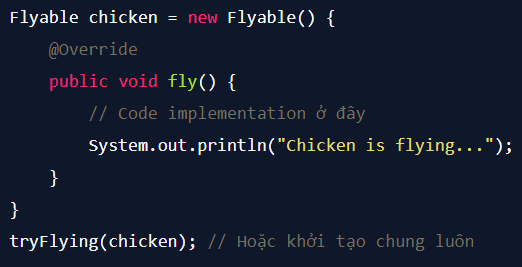
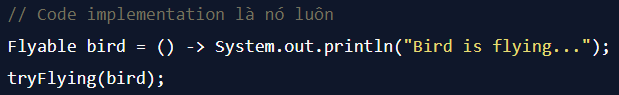
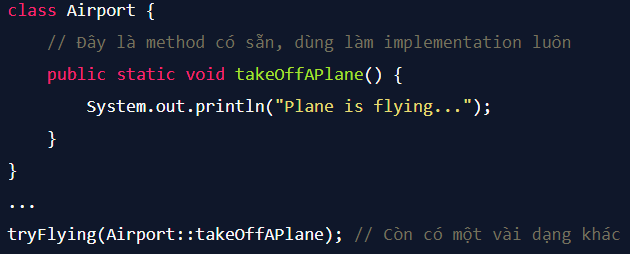
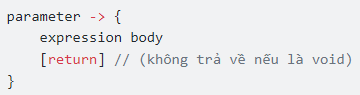
Contents

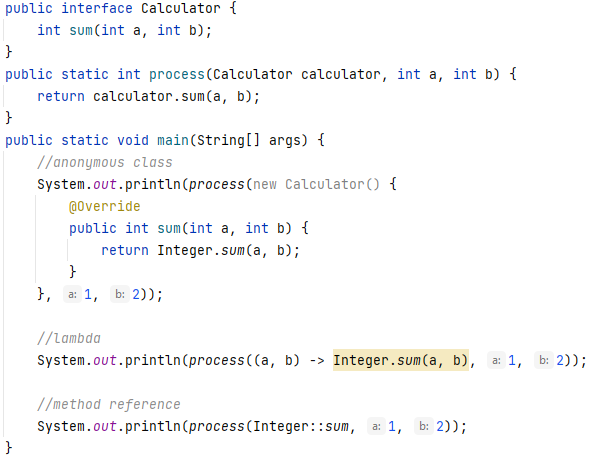
[I. Keyword 1](#_Toc125981765)

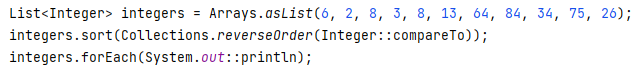
[1. Static 1](#_Toc125981766)

[a. Biến static 1](#_Toc125981767)

1. Keyword
   1. Static
      1. Biến static
         1. Chỉ lấy bộ nhớ 1 lần khi khởi tạo class
      2. Method static
         1. Thuộc class chứ ko thuộc object.
         2. Có thể gọi thông qua class mà không cần tạo instance
         3. Chỉ có thể truy cập biến static và thay đổi giá trị của nó
         4. Ko thể truy cập biến non-static và call method non-static
   2. Final
      1. Biến là final thì ko thể thay đổi giá trị
      2. Method là final thì ko thể override
      3. Class là final thì ko thể kế thừa
   3. Extends, implements
   4. Super
      1. Là một biến tham chiếu đến object của class cha gần nhất
      2. Bất cứ khi nào tạo ra instance của class con, 1 instance của class cha được tạo ra ngầm định
   5. This
      1. Là một biến tham chiếu đến instance của lớp hiện tại
   6. New
      1. Dùng tạo 1 ô nhớ trong memory và chứa 1 object mới
2. Biến primitive vs biến wrapper
   1. Biến primitive chỉ lưu giá trị, ko có API
   2. Biến wrapper là object, có API để thao tác
3. Class, Object, Interface
   1. Class
      1. Inner class?
         1. Class nằm trong class
      2. **Anomymous class?**
         1. Kĩ thuật này cho phép define và sử dụng class ngay tại thời điểm define
         2. Anonymous class sẽ được JVM tạo ra ở thời điểm run-time.
         3. **Chỉ xài được 1 lần**
      3. Abstract class?
         1. **Access modifier: public, default, protected, private. Biến: final, non-final.**
         2. **Method: abstract, non-abstract.**
         3. Có thể triển khai implementation cho interface
      4. Hashcode? Equal()?
   2. Interface
      1. Mục đích sử dụng?
         1. Loose coupling
         2. Đạt được abstraction
         3. Đạt được multiple inheritance
      2. Tính chất
         1. **Access modifier: public, default. Biến: final. Method: abstract**
         2. Ko thể tạo instance nhưng có thể tạo reference refers tới 1 object khác
         3. Các inteface có thể kế thừa các interface khác (tăng tính maintain)
         4. 1 class implements 1 interface phải implement tất cả methods trong interface đó
   3. Object
      1. **.equal()?** 
         1. <https://shareprogramming.net/moi-quan-he-giua-equals-va-hashcode-trong-java/>
         2. **Khi so sánh 2 object, hàm equal() mặc định sẽ so sánh địa chỉ 2 object thay vì so sánh các giá trị của properties trong object => kể cả chúng có properties có giá trị bằng nhau thì kết quả vẫn trả về false, vì đơn giản 2 object khác nhau sẽ lưu ở 2 địa chỉ khác nhau => phải override lại hàm equal() define lại thành so sánh giá trị properties của 2 object đó**
         3. Phản xạ: Một object phải bằng chính nó
         4. Đối xứng: x.equals(y) trả về kết quả giống với y.equals(x)
         5. Bắc cầu: Nếu x.equals(y), y.equals(z) thì x.equals(z)
      2. **hashCode()?**
         1. **Mỗi object sẽ có một định danh riêng, gọi là hashcode => 2 object là bằng nhau khi có cùng hashcode**
         2. **2 object bằng nhau => sẽ có cùng hashcode. Nhưng 2 object có cùng hashcode chưa chắc bằng nhau (vì thuật toán hashing ko thật sự hoàn hảo, nên vẫn có tỉ lệ nhỏ 2 object khác nhau băm ra cùng một hashcode)**
         3. **Khi override hàm equal(), hashCode() cũng phải được override.**
      3. **.equal() vs ==?**
         1. **“equal()” so sánh giá trị, còn “==” so sánh địa chỉ bộ nhớ**
         2. **Vì primitive type là immutable, nên so sánh == sẽ luôn đúng.**
      4. **Immutable?**
         1. **Nghĩa là không thể thay đổi**. Giá trị được lưu trong ô nhớ ko thể bị thay đổi, chỉ có thể được gán lúc khởi tạo. Muốn thay đổi giá trị của biến reference đang refer vào ô nhớ đó, ta buộc phải tạo 1 ô nhớ mới và refer vào ô đó. Biến primitive trong java luôn là immutable.
         2. **Đại khái 1 immutable class là 1 class chỉ có constructor, ko có setter.**
4. 4 tính chất java OOP
   1. Inheritance
      1. Keyword super?
      2. **Không thể có multiple inheritance => diamond problem (method cùng tên ở 2 class cha khác nhau thì class con sẽ ko biết thực hiện method của class cha nào)**
   2. Polymorphism
      1. **Là việc thực hiện 1 hành động theo nhiều cách khác nhau**
      2. Upcasting?
         1. Là biến reference của class cha refers tới class con
      3. **Overload ?**
         1. Đa hình complie-time
         2. Cùng tên method nhưng khác số lượng, data type của đối số truyền vào
      4. **Override?**
         1. Đa hình run-time
         2. Cùng tên method nhưng khác cách triển khai
         3. Nếu method không bị ghi đè thì thì sẽ call method của cha
   3. Abstraction
      1. **Khi nào dùng interface hay abstract class?**
         1. Khi chúng thực hiện những hành động giống nhau => interface
         2. Khi chúng có cùng bản chất => abstract class
   4. Encapsulation
      1. Đảm bảo tính che giấu thông tin. Muốn lấy thông tin hay chỉnh sửa thì phải thông qua getter, setter
5. **JAVA 8**
   1. **Stream API?**
      1. Tham khảo
         1. <https://codelearn.io/sharing/stream-xu-ly-du-lieu-the-nao-trong-java>
         2. <https://huongdanjava.com/vi/gioi-thieu-ve-stream-trong-java.html>
         3. <https://topdev.vn/blog/cung-tim-hieu-ve-stream-trong-java-8/>
         4. <https://tubean.github.io/2018/11/java-stream/>
      2. Thành phần
         1. [**https://blog.tonghoangvu.dev/gioi-thieu-ve-stream-api**](https://blog.tonghoangvu.dev/gioi-thieu-ve-stream-api)
         2. 
         3. **Source**: được hiểu là dữ liệu đầu vào: có thể là một mảng (array), một danh sách (list) hoặc một stream được tạo ra bằng phương thức generate() , đầu vào ở đây có thể linh động theo yêu cầu cụ thể.
         4. **Intermediate operation** là một hoạt động trung gian, hoạt động này gọi trên từng items của Stream. Stream có 5 phương thức trung gian là: Stream.map(), Stream.filter() , Stream.limit(). Sở dĩ gọi là trung gian vì sau bước này, ta chưa thể có stream output, các phương thức này chỉ là phương thức hỗ trợ xử lí để có output. Ví dụ như 2 method filter() và sorted() chỉ giúp lọc và sắp xếp chuỗi stream đầu vào, nếu không có bước cuối cùng (terminal), ta chưa thể có một stream hoàn chỉnh.
         5. **Terminal operation**, là hoạt động cuối cùng của đường ống (pipelined). Sở dĩ gọi nó là terminal vì sau khi thực hiện, ta không thể sử dụng lại stream. Bước cuối cùng này có trách nhiệm trả về ‘output’. Bước cuối cùng này có thể trả về từng đối tượng (forEach()), độ dài stream (count()) hoặc một collections (collect()). Một khi đã thực hiện tới bước cuối cùng (terminal), **không thể quay trở lại thực hiện một hoạt động trung gian nào khác.**
      3. Tính chất
         1. Dùng để thay thế for, while truyền thống.
         2. **Dùng để duyệt qua phần tử mà không làm thay đổi dữ liệu gốc đưa vào.**
         3. **Không lưu trữ: Stream không phải là một cấu trúc dữ liệu**, đầu vào của Stream có thể là các Colllections (Arraylist, Set, LinkedList,…), Arrays và các kênh Input/Output.
         4. **Không chính sửa: Stream không làm thay đổi dữ liệu gốc (chỉ có địa chỉ nhớ của từng đối tượng được lấy ra, các đối tượng được sao chép, và đưa vào streams từng phần tử một)** mà chỉ trả về kết quả thông qua các methods. Giải nghĩa: Collection là một cấu trúc dữ liệu (có thể là ArrayList, LinekedList), còn stream thì chỉ như là công cụ xử lí. Nếu ta có danh sách 10 tên tội phạm thì collection sẽ trực tiếp lưu các phần tử vào ArrayList, còn stream chỉ giúp ta filter() những tên tội phạm ma túy, sort() theo thứ tự ngày phạm tội, … **Stream không trực tiếp lưu trữ những phần tử này, nó luôn cần source (dữ liệu đầu vào) để xử lí.**
         5. Syntax đa số xài lambda expression
         6. **Stream là immutable object.**
         7. **Consumable: Stream không dùng lại được.** Sau khi đã sử dụng nó xong thì ko thể gọi lại để sử dụng lần nữa. Nếu muốn xài lại thì đa số method của Stream sẽ trả về 1 stream mới
         8. Không thể dùng index để access các element trong Stream.
      4. Một vài API
         1. .map: thay đổi phần tử trong stream theo function truyển vào
         2. .filter: giữ lại những phần tử trong stream thoả mãn điều kiện truyền vào
         3. .collect: trả về kết quả của stream dưới dạng List hoặc Set.
         4. forEach: Dùng để duyệt qua mọi phần tử trong stream.
         5. .distinct: Được dùng để loại bỏ các phần tử trùng lặp.
         6. .sorted: Dùng cho việc sắp xếp các phần tử.
   2. Datetime API?
   3. **Functional interface?** 
      1. [**https://blog.tonghoangvu.dev/hieu-hon-ve-functional-interface-trong-java**](https://blog.tonghoangvu.dev/hieu-hon-ve-functional-interface-trong-java)
      2. <https://blog.tonghoangvu.dev/cac-dang-implementation-cua-functional-interface>
      3. <https://viblo.asia/p/java-8functional-interfaces-lambda-expressions-cuc-de-hieu-m68Z0L96ZkG>
      4. [**https://blog.tonghoangvu.dev/4-functional-interface-co-san-va-vi-du-voi-stream-api**](https://blog.tonghoangvu.dev/4-functional-interface-co-san-va-vi-du-voi-stream-api)
      5. *Nói sơ qua về Functional Programming*
         1. *Lập trình hàm, hiểu nôm na là lập trình chú trọng kết quả, chứ ko chú trọng quá trình*
         2. ***Như bên Javascript, ta có thể truyền function này làm arguments của một function khác*** *(thay đổi cách xử lí của một function bằng cách truyền function khác làm arguments của function đó) => linh hoạt*
         3. ***Java ko linh hoạt như vậy => Functional Interface được sinh ra để đạt được sự linh hoạt đó:*** *có nghĩa là ta cứ khai báo function này sẽ có parameters là bao gồm abstract function nào trước, rồi khi cần gọi ta mới* ***truyền implementation của abstract function đó vào.***
      6. **Functional interface là SAM interface (Single Abstract Method)**, nghĩa là **có duy nhất 1 abstract method =>** chỉ có thể extends functional interface khác ko có abstract method nào
      7. Ko bắt buộc nhưng nên có annotation **@FunctionalInterface** (để tránh sai sót trong quá trình complie kiểu như nếu có nhiều abstract method)
      8. Có thể không có hoặc nhiều default method, static method
         1. 
         2. 
      9. Có 4 cách để implement functional interface:
         1. Dùng 1 class implements interface đó, rồi override lại abstract method.
         2. 
         3. Dùng anonymous class
         4. 
         5. Dùng **lambda expression (**giống arrow function bên Javascript)
         6. ****
         7. Dùng **method reference** (cũng là 1 kiểu lambda expression)
         8. ****
   4. **Default method? Static method?**
      1. <https://viblo.asia/p/interface-trong-java-8-gioi-thieu-default-method-va-static-method-PDOkqLqjejx>
      2. *Vấn đề của interface trước Java 8*
         1. ***Khi một interface thay đổi thì các class implements interface đó đều phải thay đổi theo*** *=> kém flexible*
         2. *Interface chỉ có thể khai báo abstract method (ko có body xử lí)*

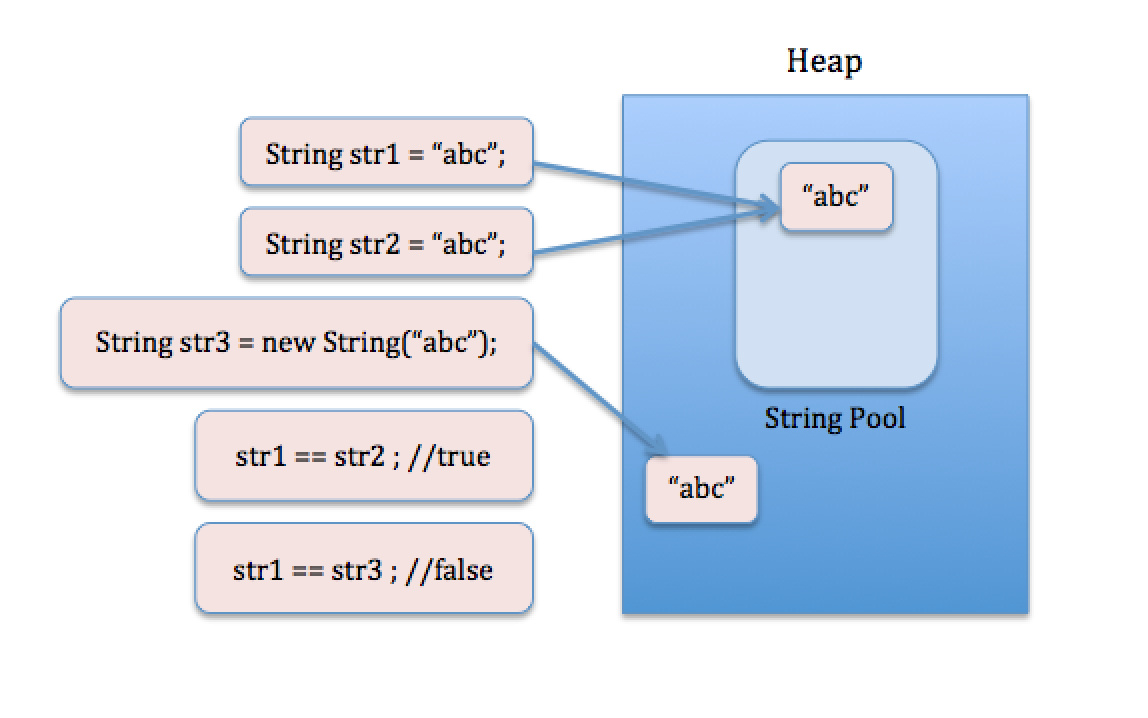
* *Functional interface sẽ giải quyết 2 vấn đề này bằng default method và static method. Default method và static method sẽ có body xử lí và các class implements functional interface sẽ ko bắt buộc phải implement default method.* 
  + 1. Default method
       1. Class implement interface sẽ ko bắt buộc phải implement default method. Ta chỉ cần viết thêm default method vào interface
       2. Nếu 1 class implement 2 interface có default method tên giống nhau sẽ xảy ra **diamond problem** (vì default method có body xử lí, giống việc implement 2 class có method tên giống nhau) => class đó sẽ phải override lại method đó.
    2. Static method
       1. Gần giống như default method. Chỉ khác là:
          1. Có thể call mà ko cần tạo một class implement interface đó.
          2. Class implements từ nó không thể override
  1. **Lambda expression?**
     1. [**https://viblo.asia/p/nam-ro-java-lambda-expression-cho-nguoi-moi-bat-dau-RQqKLNwbl7z**](https://viblo.asia/p/nam-ro-java-lambda-expression-cho-nguoi-moi-bat-dau-RQqKLNwbl7z)
     2. **Lí do ra đời rất đơn giản, giúp viết code ngắn gọn hơn.**
     3. **Lambda expression cung cấp implementation cho abstract method được định nghĩa ở functional interface**
     4. Đơn giản mà nói, lambda expression giống như một function thu nhỏ (gọi là **anonymous function**): có parameters, có body xử lí, có thể có return nhưng ko có tên function.
     5. Cấu trúc của một lambda như sau:
        1. ****
        2. ****
        3. Parameter là những tham số đầu vào của hàm (một hoặc nhiều)
        4. -> là toán tử mũi tên, dùng để liên kết params và body.
        5. Expression body là phần xử lý parameter (nếu lambda này là một xử lí ta truyền vào một function, ta cần return đúng kiểu dữ liệu đã khai báo trong Functional Interface
  2. Method reference?
     1. <https://viblo.asia/p/java-8-method-reference-phan-1-V3m5WObE5O7>
     2. Là cú pháp ngắn của Lambda expression sử dụng 1 method (tức là lambda đó có phần body xử lí chỉ call tới 1 method duy nhất)
     3. Để dùng method reference, ta cần có lambda expression -> để có lambda expression, ta cần có functional interface -> để là một functional interface, interface đó chỉ có 1 abstract method.
* **Method reference để gọi 1 và chỉ 1 method -> vì biết method đó đã bao gồm những arguments nào nên method reference sẽ bỏ qua phần arguments.**

****

****

* 1. Optional?
     1. [**https://loda.me/articles/java-8optional**](https://loda.me/articles/java-8optional)
     2. Bản chất Optional<T> là một đối tượng Generic, nhiệm vụ chính của nó là bọc hay wrapper lấy một object khác. Nó chỉ chứa được một object duy nhất bên trong.
     3. **Mục đích sinh ra rất đơn giản: kiểm soát tình trạng null của một object, giúp hạn thế NullPointerException.**
     4. Về cơ bản thì không khác gì “if (object != null)” là mấy, nhưng Optional sẽ có thêm một số hàm dùng lambda expression giúp code linh hoạt hơn (giống Stream, cũng có một số hàm kiểu map(), filter(),…).

1. Exception & Error
   1. Error
      1. Không nên catch, kệ bố nó
   2. Exception
      1. User-defined exception (custom exception, tạo 1 class extends class Exception)
      2. Built-in exception
         1. Checked exception (complie-time exception)
            1. Bắt buộc phải catch khi complie
         2. Un-checked exception (run-time exception)
            1. Không bắt buộc phải catch khi complie
   3. Throw vs Throws
      1. Throw
         1. Để ném ra 1 exception rõ ràng
         2. Sau throw là một instance Exception
         3. Được sử dụng trong method
         4. Không thể throw nhiều exception
      2. Throws
         1. Để báo rằng method này có thể throws Exception
         2. Sau throws là một hoặc nhiều class Exceptiond
         3. Được khai bao sau paramters của method
         4. Có thể khai báo nhiều exceptions
2. Generics
   1. Generics là tham số hoá type => khi nào cần xài thì đưa type mong muốn vào.
   2. Trước khi có generic, collection có thể chứa bất kì kiểu dữ liệu nào, khi muốn lấy ra thì phải typecasting. Khi có generic, collection chỉ có thể chứa kiểu 1 dữ liệu cụ thể.
   3. <https://www.geeksforgeeks.org/generics-in-java/>
      1. T – Type
      2. E – Element
      3. K – Key
      4. N – Number
      5. V – Value
3. **Data structure and algorithm**
   1. **Sort algorithm**
   2. **Collection framework**
      1. Tính chất
         1. **Không cần khai báo trước số lượng phần tử, collection có khả năng tự co dãn size khi thay đổi.**
         2. Trước JDK 1.5, collection là non-generic. Các collection có thể chứa nhiều kiểu dữ liệu => phải type casting khi lấy giá trị ra. Sau JDK 1.5 collection là generic => collection chỉ chứa 1 kiểu dữ liệu => Ko cần type casting.
      2. **Iterator?**
         1. **Dùng để duyệt phẩn tử trong collection (.hasNext())**
      3. **Comparator interface**
         1. Dùng để triển khai hàm so sánh compare(object1, object2) khi muốn sort collection theo cách user tự define
      4. List
         1. ArrayList
            1. <https://www.javatpoint.com/java-linkedlist>
            2. **Cơ chế như dynamic array**
            3. **Lưu trữ phần tử có thứ tự**
            4. Lưu trữ theo index.
            5. Có thể truy xuất phẩn tử ngẫu nhiên theo index.
            6. **Performance tốt cho việc thao tác truy xuất dữ liệu, nhưng performance tệ cho việc manipulate dữ liệu (phải tốn công dịch chuyển vị trí phần tử)**
         2. LinkedList
            1. <https://www.javatpoint.com/java-linkedlist>
            2. **Cơ chế như doubly linked list**
            3. **Lưu trữ phần tử có thứ tự**
            4. Lưu trữ dưới dạng các node. Các node bao gồm prev, value, và next.
            5. Không thể truy xuất phần tử ngẫu nhiên => bắt buộc phải duyệt tuần tự
            6. **Performance tốt cho việc manipulate dữ liệu, performance tệ cho việc truy xuất dữ liệu (ko tốn công dịch chuyển vị trí phần tử)**
            7. Có thể sử dụng như list, stack hoặc queue
         3. Stack (sub-class of Vector)
            1. **LIFO**
      5. Set
         1. **Lưu trữ phần tử ko theo thứ tự**
         2. **Lưu trữ các phần tử ko trùng lặp**
         3. **Có thể lưu trữ giá trị null**
         4. HashSet
            1. <https://www.javatpoint.com/java-hashset>
            2. Lưu trữ các phẩn tử bằng thuật toán **hashing**
         5. TreeSet
      6. Queue
         1. **FIFO**
      7. Stack
      8. Map (ko nằm trong hierarchy của Collection interface nhưng vẫn dc coi là collection, map chỉ thao tác với key/value, còn các collection khác chỉ thao tác với value)
         1. Lưu trữ theo cặp key/value
         2. **Không cho lưu key trùng lặp, nhưng value có thể trùng lặp**
         3. **Nếu insert 1 entry mà key đã tồn tại trong map, value sẽ được replace**
         4. **Mỗi 1 cặp key/value được gọi là một Entry**
         5. **Lưu trữ phần tử ko theo thứ tự**
         6. HashTable
         7. HashMap
         8. TreeMap
      9. **Hash: Collection thuộc nhánh Hash lưu trữ phần tử theo thuật toán Hash. Collection sẽ được khởi tạo với size mặc định là 16. Vị trí các phần tử được chèn vào dựa theo thuật toán Hashing.**
      10. **Tree: Collection thuộc nhánh Tree không lưu trữ giá trị null và luôn lưu trữ phần tử theo thứ tự sắp xếp tăng dần => truy xuất phần tử nhanh (binary search thì chả nhanh)**
4. JVM
   1. Khái quát
      1. <https://www.geeksforgeeks.org/why-is-java-write-once-and-run-anywhere/>
      2. JVM là 1 máy trừu tượng. Mà 1 thứ cung cấp run-time environment nơi bytecode Java có thể execute.
      3. **Viết 1 lần, chạy nhiều nơi (write one, run anywhere):** khác với C, Java có thể được viết ở 1 thiết bị và execute trên bất cứ thiết bị nào hỗ trợ Java, nhờ JVM. Lí do: Java ko trực tiếp convert code thành mã máy, chúng chỉ convert sang bytecode (file .class) khi JVM intepret. Khi chúng được complie, bytecode mới chuyển sang thành mã máy.
   2. Architecture?
      1. <https://www.geeksforgeeks.org/java-memory-management/>
      2. **Stack area?**
         1. Stack được tạo ra khi thread được tạo, và thread-safe.
         2. Có thể có size fixed hoặc dynamic
         3. Stack memory được tạo ra cho mỗi thread
         4. **Nó chứa biến giá trị kiểu primitive**
         5. **Nó chứa reference tới object lưu trong heap**
         6. **Stack memory được tham chiếu theo thứ tự LIFO => vậy nên được gọi là stack.**
         7. **Nếu stack memory đầym sẽ gây ra lỗi StackOverFlow.**
         8. Dung lượng stack thường nhỏ
         9. Stackframe:
            1. Là một cấu trúc dữ liệu chứa dữ liệu của thread.
            2. Khi 1 method được call, một frame sẽ được tạo ra để chứa các biến nguyên thủy local và các tham chiếu tới các object. Frame sẽ bị phá huỷ khi việc method hoàn tất hoặc call method khác.
      3. **Heap area?**
         1. Được tạo ra khi JVM khởi chạy, và không thread-safe.
         2. Heap có thể có size cố định hoặc dynamic tuỳ cấu hình.
         3. **Dùng để lưu trữ object (khi thực hiện toán tử new), và các biến static hoặc biến instance (toàn cục), và String**
         4. Heap được cấp phát cho class instances và array.
         5. JVM cung cấp user control cho việc khởi tạo hoặc thay đổi kích thước heap theo yêu cầu. Khi sử dụng từ khoá **new**, object được gán ở 1 không gian trong heap, nhưng reference của nó được tạo trên stack.
         6. **Nếu stack memory đầy sẽ gây ra lỗi OutOfMemoryError.**
         7. Dung lượng heap thường lớn hơn stack.
      4. Method area?
         1. Là một phần của heap memory, được share giữa tất cả thread.
         2. Lưu trữ cấu trúc class, class cha, interface và constructor. Được tạo ra khi JVM khởi chạy.
         3. Có thể có size fixed hoặc dynamic
      5. Native method stack?
         1. Còn được gọi là C stacks, vì nó ko được viết thằng Java. Memory này dược cấp phát cho mỗi thread khi nó được tạo. Performance của nó phụ thuộc vào OS.
      6. PC register (program counter register)?
         1. Mỗi thread có một PC register liên kết với nó. PC register lưu trữ địa chỉ return hoặc native pointer.
   3. Garbage collector?
      1. <https://viblo.asia/p/garbage-collection-quan-ly-bo-nho-heap-trong-jvm-wjAM7aJrvmWe>
      2. <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>
      3. Ko giống ngôn ngữ C, lập trình viên ko thể can thiệp trực tiếp vào memory, JVM hay cụ thể hơn la Garbage collector sẽ có nhiệm vụ quản lý memory => lập trình viên ko cần làm gì.
      4. Để giảm gánh nặng cho lập trình viên, JVM sẽ tự kích hoạt quá trình cấp phát hoặc thu hồi bộ nhớ. Khác với C, việc tối ưu sử dụng bộ nhớ lập trình viên phải can thiệp rất nhiều.
      5. Heap là nơi object được lưu trữ, và heap là nơi duy nhất Garbage collector can thiệp vào.
      6. **JVM sẽ tự động kích hoạt GC khi bộ nhớ sắp hết. Chúng ta cũng có thể tự request JVM để chạy GC, nhưng chưa chắc JVM sẽ chấp nhận.**
      7. **Cơ chế GC khá phức tạp, giải thích đơn giản là:**
         1. Mọi object được khởi tạo sẽ được lưu vào Young generation.
         2. Khi vùng nhớ Young generation đầy thì garbage collector là Minor GC hoạt động, quét hết tất cả object ko được sử dụng.
         3. Sau "nhiều" chu kỳ quét mà Object vẫn còn được sử dụng thì chúng được chuyển sang Old generation.
         4. Old generation sẽ được Major GC quản lí. Major GC rất ít khi được chạy
   4. String pool?
      1. <https://viblo.asia/p/string-pool-la-gi-4dbZNL3L5YM>
      2. String pool là một vùng nhớ đặc biệt nằm trong vùng nhớ Heap (Heap memory), dùng để lưu trữ các biến được khai báo theo kiểu String literal.
      3. String pool giúp tối ưu hoá việc lưu trữ và sử dụng vùng nhớ khi khai báo biến String, giúp hạn chế tình trạng tràn bộ nhớ Java Heap Space.
      4. **String là một immutable class**
      5. Biến khai báo theo kiểu String literal khi được tạo ra sẽ được lưu vào String Pool.



1. File IO? Serialization?
   1. Serialization?
      1. <https://www.geeksforgeeks.org/serialization-in-java/>
      2. <https://topdev.vn/blog/noi-ve-serialization-trong-java/>
      3. Khái niệm
         1. Serialization đơn giản là cơ chế chuyển từ một object vào 1 byte stream (ghi ra file). Deserialization là ngược lại, từ byte stream tạo thành object trong memory.
         2. Vì byte stream được tạo ra độc lập với platform => object có thể serialization và deserialization trên các platform khác nhau đều dùng Java (giữa thiết bị khác nhau, giữa DB, web…)
         3. Để 1 object có thể serialization được, object phải implement interface Serializable.
      4. Lưu ý
         1. **Interface Serializable**: đây là **marker interface**: ko method, ko properties, dùng để đánh dấu class này có thể có một khả năng gì đó. Nếu ko implement interface này khi serialization sẽ xảy ra NotSerializableException.
         2. Chỉ **non-static** property mới được serialize, property là **static**, **transient** sẽ ko được serialize (return 0)
         3. Constructor sẽ ko bao giờ được gọi
         4. serialVersionUID để đảm bảo chắc chắn rằng đối tượng trước và sau khi serialization là một
   2. File IO (dễ, làm tới đâu search code tới đó)
2. **Thread? Concurrency program?**
3. **Synchronization? Thread-safe?**
4. Cookie? Session?
5. **Web Socket?**
6. **Framework**
   1. **Spring**
      1. <https://www.marcobehler.com/guides/spring-framework>
   2. **Spring boot**
      1. **Phân biệt Spring và Springboot?**
         1. <https://www.ibm.com/cloud/learn/java-spring-boot>
         2. <https://viblo.asia/p/hoc-spring-boot-bat-dau-tu-dau-6J3ZgN7WKmB>
         3. [**https://www.marcobehler.com/guides/spring-framework#difference-spring-spring-boot**](https://www.marcobehler.com/guides/spring-framework#difference-spring-spring-boot)
         4. Spring là một Java framework siêu to và khổng lồ, làm được đủ mọi thứ. Nó được chia thành nhiều module, mỗi module làm một chức năng, ví dụ Spring Core, Web, Data access, AOP,... Spring được xây dựng dựa trên 2 khái niệm nền tảng là Dependency injection và AOP (Aspect Oriented Programming).
         5. Một rắc rối khi dùng Spring là việc cấu hình (config) dự án quá phức tạp (toàn phải cấu hình XML).
            1. Tạo Maven hoặc Gradle project
            2. Thêm các thư viện cần thiết
            3. Tạo XML để cấu hình project, cấu hình các bean
            4. Code và build thành file WAR
            5. Cấu hình Tomcat server để chạy được file WAR vừa build
         6. Đấy, Spring khá mạnh mẽ nhưng việc cấu hình nghe thôi cũng mệt rồi. Do đó Spring boot ra đời, Springboot là 1 module trong Spring framekwork, với các ưu điểm:
            1. Auto config: tự động cấu hình thay cho bạn, chỉ cần bắt đầu code và chạy là được
            2. Xây dựng các bean dựa trên annotation thay vì XML
            3. Server Tomcat được nhúng ngay trong file JAR build ra, chỉ cần chạy ở bất kì đâu java chạy được (tạo ứng dụng standalone)
         7. So sánh với Spring, thì Spring Boot bạn chỉ cần:
            1. Dùng Spring Initializr, nhập các info của project, chọn thư viện rồi down code về (hoặc là tạo project Springboot luôn nếu dùng IntelliJ Ultimate :D)
            2. Mở source code ra và bắt đầu code
            3. Chạy ngay trong IDE, hoặc build thành file JAR để chạy được ngay, không cần cấu hình server

* Tóm lại: bớt phải làm đi làm lại những công việc config (config file properties, nhúng server Tomcat, cấu hình rest controller, config bean bằng XML…) => code đơn giản hơn, tập trung vào business logic. Mặc dù phải đánh đổi sự linh hoạt khi sử dụng trực tiếp Spring framework, nhưng Springboot cũng có khả năng làm những điều mà Spring framework làm được.
* Hiểu nôm na Springboot được xây dựng trên Spring framework và Spring MVC. Dùng Springboot để code web thực chất nghĩa là ta đã dùng Spring framework, Spring MVC rồi.
  + 1. **RESTful API? HTTP Method?**
       1. **REST (REpresentational State Transfer)**
          1. RESThiểu 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.

* + - * 1. Trong kiến trúc REST, client gửi request tới server để modify resource và sau đó server sẽ gửi response tới client.
      1. **HTTP (Hyper Text Transfer Protocol)?**
         1. HTTP hiểu đơn giản là một giao thức để một giao thức để các hệ thống có thể giao tiếp với nhau qua internet, bằng việc truyền đi các dữ liệu siêu văn bản (hyper text)
      2. **RESTful API?**
         1. <https://aws.amazon.com/vi/what-is/restful-api/>
         2. <https://www.codecademy.com/article/what-is-rest>
         3. <https://200lab.io/blog/rest-api-la-gi-cach-thiet-ke-rest-api/>
         4. là một giao diện lập trình ứng dụng (API) tuân thủ các ràng buộc và quy ước kiến trúc REST được sử dụng trong việc giao tiếp giữa client và server.
         5. Cách hoạt động

Client gửi request tới server.

Server authen và author để xác định client có permisson thực hiện request đó không

Server nhận request và handle request đó.

Server trả về response tới client. Response chứa thông tin và cho biết request đó có thành công hay không.

* + - * 1. REST API gửi request tới server thông qua các HTTP method:

GET: trả về một Resource hoặc một danh sách Resource.

POST: Tạo mới một Resource.

PUT: Cập nhật thông tin cho Resource (toàn bộ resource).

PATCH: Cật nhật thông tin cho resourse (một phần resource).

DELETE: Xoá một Resource.

* + - * 1. Header: Authentication và quy định kiểu dữ liệu trả về

Hệ thống tuân theo REST sẽ stateless, hệ thống ko biết bất kì thông tin gì trước đó. Khác với hệ thống thông thường sẽ có session và cookie để hỗ trợ phân biệt request đấy là của ai, thông tin trước đó là gì.

Trong REST, nếu một request cần xác thực quyền truy cập, chúng sẽ phải dùng thêm thông tin trong header. Ví dụ như thông tin Authorization sẽ mang theo một user token (JWT, OAuth, …)

Ngoài ra Header còn giúp client chỉ định được loại content cần trả về từ server – content type. Việc này được thực hiện thông qua phần Accept trong header. Giá trị của nó thường là MIME type:

image — image/png, image/jpeg, image/gif

audio — audio/wav, audio/mpeg

video — video/mp4, video/ogg

application — application/json, application/pdf, application/xml, application/octet-stream

Status Code <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

* 200 OK – Trả về thành công cho những phương thức GET, PUT, PATCH hoặc DELETE.
* 201 Created – Trả về khi một Resouce vừa được tạo thành công.
* 400 Bad Request – Request không hợp lệ
* 401 Unauthorized – Request cần có xác thực.
* 403 Forbidden – bị từ chối không cho phép.
* 404 Not Found – Không tìm thấy resource từ URI
* 405 Method Not Allowed – Phương thức không cho phép với user hiện tại.
* 415 Unsupported Media Type – Không hỗ trợ kiểu Resource này.
  + - 1. **Request param? Path variable? Request body? (dễ rồi)**
    1. **IOC? DI?**
       1. <https://viblo.asia/p/dependency-injection-ap-dung-vao-spring-boot-nhu-the-nao-phan-1-WAyK8AWWZxX>
       2. <https://www.marcobehler.com/guides/spring-framework>
       3. **Module coupling**
          1. Trước tiên cần nói sơ qua về module coupling. Coupling gọi là quan hệ giữa 2 module, 2 object, có sự phụ thuộc với nhau. Có 2 loại:

**Tight coupling:** hai module liên kết chặt chẽ, khó tách rời

**Loose coupling:** liên kết yếu, rời rạc

* + - * 1. Để code dễ bảo trì và sửa đổi, thì nguyên tắc là phải **giảm sự phụ thuộc** giữa các module. Nghĩa là biến mối quan hệ giữa chúng từ **tight coupling** thành **loose** **coupling**.

Ví dụ: Giả sử class Car chứa 1 property là class ChinaEngine => class Car **phụ thuộc** vào class ChinaEngine, vậy ChinaEngine là một **dependency** của Car. Khi tạo object Car ta buộc phải tạo object ChinaEngine. Muốn thay đổi property ChinaEngine trong Car, ta buộc phải sửa class Car => tight coupling (các module quá dính vào nhau) => sinh ra vấn đề khó maintain hệ thống.

* + - 1. **Dependency inversion principle (nguyên tắc đảo ngược sự phụ thuộc)**
         1. Là nguyên lí số 5 trong SOLID principles. Đưa ra 1 kiểu thiết kế module, để làm **giảm sự phụ thuộc** nhất có thể. Có 2 ý chính:

Các module cấp cao không **nên phụ thuộc** vào module cấp thấp, cả hai nên phụ thuộc vào interface. Các class giao tiếp với nhau qua interface, ko thông qua implementation.

Hiểu đơn giản là ta nên thay class ChinaEngine bằng interface Engine trong class Car, và tạo một class ChinaEngine implement class Engine đó => Lúc khởi tạo class Car thì mới gán ChinaEngine vào khi khởi tạo => Dễ maintain, nếu sau hệ thống cần tạo class Car với VNEngine thì chỉ cần tạo 1 class VNEngine implement interface Engine rồi gán vào Car là xong

Abstraction không nên phụ thuộc vào chi tiết, mà ngược lại.

Hiểu đơn giản giống tính chất abstraction trong OOP. Các class implement interface Engine sẽ chỉ cần biết có những method gì mà ko cần quan tâm chúng chạy như nào.

* + - 1. **Inversion of Control** 
         1. **Đây là một dạng design pattern:** dev sẽ chỉ việc tạo ra component và dev sẽ dùng framework để connect các components đó lại với nhau, thay vì viết code chay
         2. **Đảo ngược điều khiển:** nghĩa là việc kiểm soát sự phụ thuộc của các object sẽ bị đảo ngược, nó sẽ ko được quyền quản lí, mà sẽ đẩy cho thằng **IoC** **container** làm.
         3. Nhằm đơn giản hoá quá trình tạo object và liên kết giữa chúng bằng cách tuân theo quy tắc: **Không tạo đối tượng, chỉ mô tả cách chúng sẽ được tạo ra.**
         4. Hiểu nôm na nó là 1 cơ chế giúp mình ko phải chỉ chính xác thằng nào sẽ được tiêm vào object, mà chỉ cần mô tả class đó sẽ được tiêm những dependency gì, rồi **IoC** **container** sẽ tự tiêm nó vào.
      2. **IoC container**
         1. IoC container của Spring gọi là **Application context**
         2. Các module chứa trong IoC container được Spring gọi là các **Bean**
         3. Là một container chứa các dependency: khi app chạy, tất cả các bean sẽ được tạo (default là **scope** **singleton**), lúc nào cần thì lấy ra xài => đảm bảo chỉ được tạo ra một object duy nhât => tiết kiệm bộ nhớ và dễ quản lý.
      3. **Dependency injection**
         1. Là một cơ chế để thực hiện IoC. Bằng cách inject 1 module nhỏ vào 1 module to.
         2. Khi app chạy, IoC container sẽ tạo tất cả các **dependency** và tìm kiếm **sự phụ thuộc** giữa các module, nếu IoC container tìm thấy 1 module to cần tới 1 module nhỏ, IoC container sẽ kiểm tra, nếu chưa có thì tạo mới, có rồi thì sẽ tiêm module nhỏ (**dependency**) vào module to => đó gọi là **Dependency injection**
         3. Có 3 cách:

**Constructor** **injection**: tiêm luôn lúc tạo object thông qua gọi constructor.

**Note**: Những phiên bản Springboot mới thì ko cần annotation @Autowired trên constructor

**Setter injection**: tiêm thông qua method setter

Field injection: tiêm trực tiếp vào field, sử dụng **@Autowired**:

<https://stackoverflow.com/questions/39890849/what-exactly-is-field-injection-and-how-to-avoid-it>

**không khuyến kích, vì cách này sẽ sử dụng Java Reflection => giảm performance**

vì sử dụng Java Reflection nên nếu ko sử dụng @Autowired thì sẽ phải có constructor hoặc setter thay thế

dễ vi phạm **single responsibility principle** nếu inject quá nhiều module

ko thể tạo immutable object (vì ko thể autowired cho biến final)

*Interface injection: class cần tiêm sẽ implement thông qua interface, interface chứa hàm tên inject. IoC sẽ tiêm dependency thông qua việc call method inject của interface đó.*

* + - * 1. **Lưu ý:** chỉ áp dụng DI khi hệ thống flexible, cần sự linh hoạt, mở rộng. Nếu hệ thống đơn giản thì ko nên áp dụng, vì:

Sẽ làm tăng độ phức tạp code

Giảm performance vì tất cả các object đều được khởi tạo khi bật app

Khó debug vì không biết implementation nào của interface được gọi đến

* + 1. **Bean? Application context?**
       1. <https://viblo.asia/p/bean-va-applicationcontext-la-gi-trong-spring-boot-Ljy5Vjwj5ra>
       2. **Bean**
          1. Chính là những module được tạo ra trong lúc run app, được quản lí với IoC container.
          2. Để đánh dấu 1 class là 1 bean thì dùng annotation (@Component, @Controller, @Service, …)
       3. **ApplicationContext** (đéo ai dùng BeanFactory cả, vì phải config bằng XML, rườm rà)
          1. Dùng để chỉ IoC container. Nó là component scan, khi run app, IoC container sẽ quét toàn bộ packages, tìm ra các bean và đưa vào **ApplicationContext**.
          2. **ApplicationContext** được tạo ra ngay khi run app, ở trong hàm main (SpringApplication.run(Application.class, args)).
       4. **Khi Spring Boot ko tìm được bean phù hợp**
          1. Hoặc khi có nhiều implementations của 1 interface là dependency của 1 class => không biết chọn implementation nào để tạo bean (class VNEngine và class ChinaEngine đều implement interface Engine)
          2. Giải pháp:

Xài annotation @Primary: đánh dấu 1 class sẽ được ưu tiên chọn hơn

Xài annotation @Qualifier: chỉ chính xác class nào sẽ khởi tạo bean

* + 1. Bean life cycle? **Bean Scope?**
       1. Bean life cycle
          1. <https://viblo.asia/p/vong-doi-cac-loai-bean-va-co-che-component-scan-L4x5x6BrZBM>
          2. Đại loại sẽ gồm các bước sau:

IoC container tạo bean bằng cách gọi constructor (có thể inject các bean dependency vào đây)

Gọi các setter method để inject các bean vào bằng setter based injection

Các method khởi tạo khác được gọi (không cần quan tâm nhiều)

**@PostConstructor** (dùng để thực hiện một số task khi khởi tạo bean) được gọi

Init method được gọi

* + - * 1. Sau đó bean sẽ sẵn sàng hoạt động. Nếu sau đó bean không dùng nữa thì nó sẽ được hủy:

Gọi **@PreDestroy** (thực hiện các task để dọn dẹp bean sau khi dùng xong)

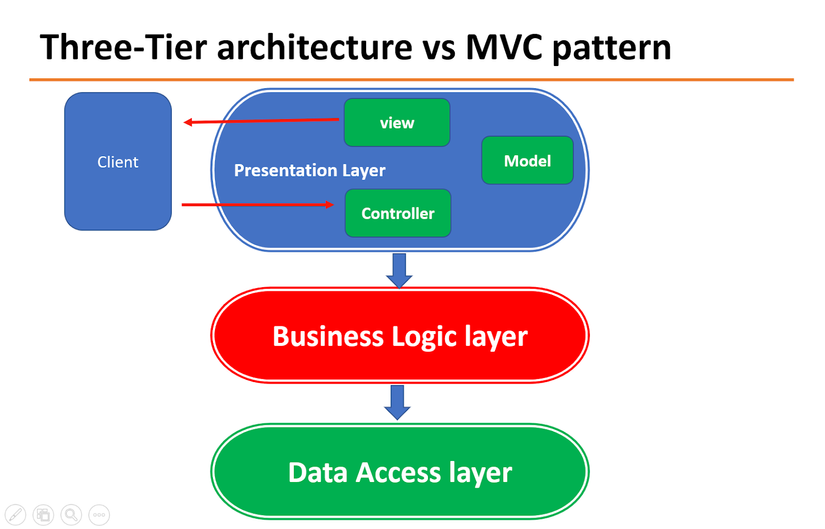
Hủy bean như các object thông thường

* + - 1. [**Bean scope (@Scope)**](https://docs.spring.io/spring-framework/docs/5.2.x/spring-framework-reference/core.html#beans-factory-scopes)
         1. **Singleton**: IoC container chỉ tạo đúng duy nhất 1 instance từ class bean này, mỗi lần request sẽ vẫn xài lại cái bean đó. Mặc định bean sẽ xài scope này.
         2. **Prototype**: mỗi lần được lấy ra sử dụng thì sẽ tạo 1 bean mới
         3. Request: tạo mỗi bean cho mỗi HttpRequest
         4. Session: tạo mỗi bean cho mỗi HttpSession
      2. Các cách tạo bean
         1. Cấu hình bằng file XML
         2. Dùng annotation trên tên class
         3. **@Configuration** và **@Bean**

Dùng trong trường hợp các bean cần thực hiện nhiều thao tác phức tạp để khởi tạo, và liên quan với nhau (thường mục đích là để cấu hình app).

Ta gom chung các bean cần khởi tạo và ném vào class config có annotation là **@Configuration**. Các này này thường có hậu tố là “config” (AppConfig, SecurityConfig)

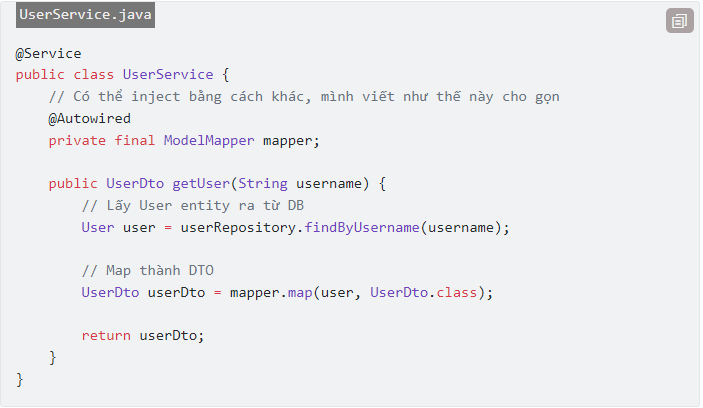
* + 1. Component scan (@ComponentScan)
       1. Component scan sẽ tìm toàn bộ class ở package cùng cấp hoặc các package thấp hơn. Khi ứng dụng Spring bắt đầu chạy, thì nó sẽ tìm hết các class đánh dấu là bean (tức là có gán annotaion @Component, @Controller, @Service,…) trong chương trình và tạo bean
       2. Do đó, class đánh dấu @SpringBootApplication có chứa main method sẽ là nơi bắt đầu. Spring sẽ tìm từ package này (package gốc) tìm xuống để tạo các bean.
       3. Do cấu trúc thư mục mặc định của Spring Boot, class chứa hàm main nằm ở package default => mọi class được khai báo là bean đều có thể được tìm được.
    2. **Cấu trúc dự án SpringBoot**
       1. <https://viblo.asia/p/luong-di-trong-spring-boot-ORNZqdELK0n>
       2. <https://viblo.asia/p/cau-truc-du-an-spring-boot-the-nao-cho-chuan-ORNZqdwbK0n>



* + - 1. Đại khái là bao gồm 3 tầng:
         1. Presentation layer

Các web cũ layer này thường trả thẳng View, tức là 1 file HTML thuần đã có sẵn data (Spring MVC, JSP Servlet, Thymfeleaf,…) => mỗi lần thực hiện action sẽ phải refresh page lại để hiển thị data mới => đã cũ

Các web bây giờ sẽ có View tách riêng (viết bằng Angular, React,…). Phía backend sẽ chỉ trả data vào trong response và để React, Angular nhận và bind data lên View để hiển thị => Single page application

* + - * 1. Business logic layer (Service)
        2. Data access layer (Repository)
    1. Entity, DTO, domain model
       1. Data trong ứng dụng Spring Boot chia thành 2 loại chính:
          1. **Public**: nghĩa là để trao đổi, chia sẻ với bên ngoài qua REST API hoặc giao tiếp với các service khác trong microservice. Data lúc này ở dạng DTO.
          2. **Private**: các data dùng trong nội bộ ứng dụng, bên ngoài không nên biết. Data lúc này nằm trong các Domain model hoặc Entity.
       2. Từ 2 loại public và private trên, chúng ta có 3 dạng data:
          1. **DTO (Data transfer object):** là các class đóng gói data để chuyển giữa client - server hoặc giữa các service trong microservice. Mục đích tạo ra DTO là để giảm bớt lượng info không cần thiết phải chuyển đi, và cũng tăng cường độ bảo mật.
          2. **Domain model:** là các class đại diện cho các domain, hiểu là các đối tượng thuộc business như Client, Report, Department,... chẳng hạn. Trong ứng dụng thực, các class đại diện cho kết quả tính toán, các class làm tham số đầu vào cho service tính toán,... được coi là domain model.
          3. **Entity:** cũng là domain model nhưng tương ứng với table trong DB, có thể map vào DB được. Lưu ý chỉ có entity mới có thể đại diện cho data trong DB.
       3. Việc convert giữa DTO, domain model, entity với nhau gọi là **Model mapping**, thường dùng thư viện **ModelMapper** (hoặc có thể làm thủ công bằng cách viết constructor hoặc method để từng field cũng được)**.**
          1. <https://viblo.asia/p/su-dung-modelmapper-trong-spring-boot-63vKj1Vd52R>
          2. **ModelMapper** là 1 thư viện Java, giúp đơn giản hoá code mapping các object, nếu các object đó có cấu trúc gần giống nhau.
          3. **ModelMapper** sẽ có performance chậm hơn 1 chút so với code copy từng field thủ công, nhưng ko đáng kể.
          4. 
    2. **Application Config và Property injection (@Value)**
       1. Ko phải lúc nào cũng nên để mọi thứ trong code. Có những thông số nên tốt hơn là truyền từ ngoài vào, để giúp app dễ thay đổi giữa các môi trường khác nhau => để phục vụ điều này, xài application.properties (Maven) hoặc application.yml (Gradle).
       2. Trong trường hợp, ta muốn tự config những giá trị của riêng mình, thì Spring Boot hỗ trợ với annotation **@Value**
          1. 
          2. Ta chỉ cần thêm annotation **@Value** trên field hoặc trước args của constructor, Spring sẽ đi qua environment (những file config application.properties (Maven) hoặc application.yml (Gradle)) để tìm property phù hợp, nếu ko tìm được sẽ throw message.
    3. **Cách controller hoạt động**
       1. [**https://www.marcobehler.com/guides/spring-mvc#\_what\_is\_spring\_mvcs\_dispatcherservlet**](https://www.marcobehler.com/guides/spring-mvc#_what_is_spring_mvcs_dispatcherservlet)
       2. Client gửi request tới server Springboot, nó sẽ đi qua **DispatcherServlet** (front controller) trước. Controller này có tác dụng:
          1. Tìm coi request này gọi tới method nào của controller nào để call đúng method đó (dựa vào URL và method đó là get, post, put, delete,…) E.g.: POST /register?name=john&age33
          2. Các data của request sẽ được parse ra và mapping tương ứng vào các tham số controller method (có @RequestParam, @PathVariable, @Header,... tương ứng).
          3. Nếu data không thể parse được, front controller sẽ trả về bad request (hoặc có cơ chế khác để chúng ta ghi đè lại việc này).
          4. Sau khi xử lí trong controller, service thì front controller sẽ nhận output từ controller rồi convert sang HTML, JSON,…
       3. Với chiều ngược lại cũng tương tự như vậy. Dữ liệu trả về từ controller sẽ được build thành response và trả cho client.
       4. 1 request có thể gồm:
          1. Request param
          2. Path variable
          3. Request body

Chỉ có trong method PUT, POST. Là nơi để chứa data để gửi lên

* + - * 1. Có thể ở dạng JSON hoặc form-data (để gửi file)
        2. Header

Thường chứa thông tin liên quan tới authen, author, cookie, session

* + - 1. Có 3 cách trả về response
         1. *@ResponseBody và @ResponseStatus*
         2. *Dùng đối tượng HttpServletResponse param*
         3. **Dùng class ResponseEntity<T> (nên dùng)**

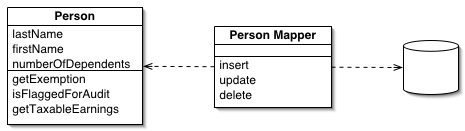
Không cần code dài dòng, class này có sử dụng builder pattern nên dùng khá tiện.

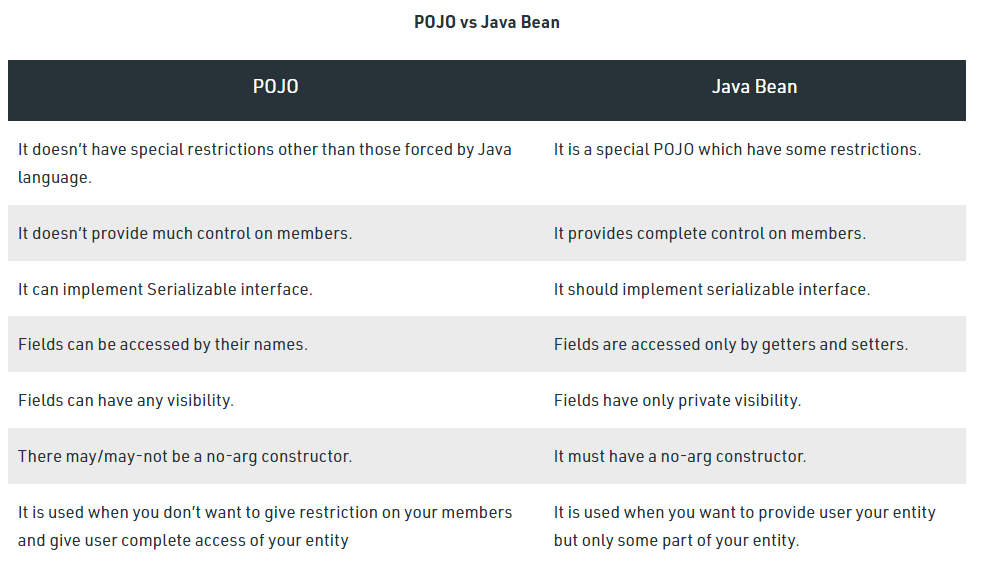
Với @ResponseStatus thì chỉ set cứng status code, tuy nhiên nếu mình muốn status code khác nhau tùy vào điều kiện thì sao. ResponseEntity<T> sẽ giải quyết được hết.

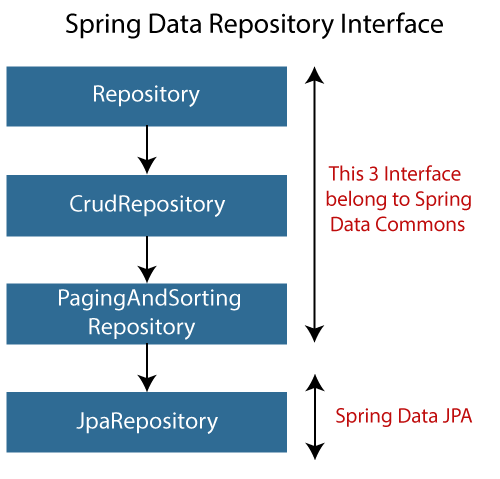
Có thể vừa đặt data trong body, vừa tùy chỉnh header, vừa,... cùng lúc được luôn.

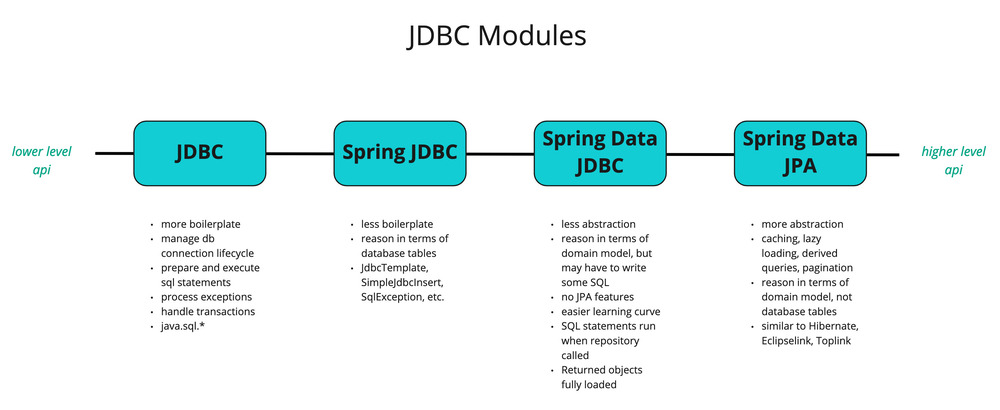
* + 1. Lombok
    2. **Exception handler**
       1. <https://viblo.asia/p/xu-ly-exception-phat-sinh-trong-ung-dung-spring-boot-6J3ZgWkLZmB>
       2. Khái niệm
          1. Exception handler sử dụng kĩ thuật AOP, dùng để bắt mọi exception xuất hiện trong app.
          2. Nếu có exception ở bất cứ đâu, sẽ được chuyển tới method có **@ExceptionHandler** tương ứng.
          3. Thứ tự đặt chúng không quan trọng, Spring sẽ tự động tìm cái phù hợp nhất, nếu không có thì chuyển dần lên các exception class cha (do đó, nên có một **@ExceptionHandler** để bắt Exception.class, dành cho các exception con còn lại).
          4. Các method này viết tương ứng với method của Controller, nhưng thay vì trả data về thì chúng ta trả về message lỗi.
          5. Thường để handle Custom Exception
       3. Cách dùng
          1. Nếu xài luôn trong controller thì chỉ cần thêm method handle exception và annotaion **@ExceptionHandler** ở trên là được
          2. Nếu tách 1 class riêng để xử lí exception thì thêm annotation **@RestControllerAdvice** phía trên tên class, còn trên method handle exception thì thêm annotaion **@ExceptionHandler**
    3. Validation
       1. <https://viblo.asia/p/dung-validation-bang-tay-trong-spring-boot-phan-1-3P0lPGyoZox>
       2. **Luôn validate bằng thư viện, code sẽ đơn giản và ngắn gọn hơn. Chỉ validate bằng tay nếu ko thể dùng thư viện để validate (check username đã tồn tại, …)**
       3. **Tại sao phải validate ở client lẫn server?**
          1. Validate ở client thì có thể nhanh chóng thông báo cho user, giúp giảm việc chờ đợi server phản hồi
          2. Validate ở server cung cấp thêm một lớp nữa để đảm bảo rằng dữ liệu luôn hợp lệ.
          3. Nếu không có validation ở server, thì hacker có thể gửi những request độc hại đến, gây lỗi, sập server,... (không thông qua web, app,... mà thông qua một vài cách tấn công như XSS, SQL Injection thì không bị dính validate ở client).
       4. Quy trình validation
          1. Thêm các annotation ràng buộc trên các field của class nào đó. Mỗi annotation có ý nghĩa riêng, ví dụ @NotNull, @NotEmpty, @Email,...
          2. Class đó được dùng ở vị trí nào cần đảm bảo hợp lệ, ví dụ truyền cho method làm tham số, mà tham số phải hợp lệ rồi mới thực hiện method, thì thêm @Valid hoặc @Validated trên tham số (thuộc class đó).
       5. Xử lý lỗi khi validation fail
          1. Nếu dữ liệu ko hợp lệ, method mà có tham số kèm theo annotation @Valid sẽ ko được gọi, Springboot sẽ ném ra BindException => Chỉ cần bắt lỗi bằng **@ExceptionHandler(BindException.class)** rồi xử lí là dc
    4. **Persistence data? (tính toàn vẹn dữ liệu)**
       1. Nói về vấn đề khi run app, app khởi tạo data và thao tác trên data đó, sau đó tắt app đi thì mọi data đó sẽ mất hết và data sẽ được tạo mới lại từ đầu khi run lại app => vậy nếu muốn dữ liệu được vẹn toàn, thì phải lưu lại vào database.
       2. **Đại khái là mọi dữ liệu đều phải được lưu vào database để ko bị mất khi thoát app hoặc app bị error.**
    5. **JPA?**
       1. <https://www.geeksforgeeks.org/java-jpa-vs-hibernate/>
       2. **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
          1. JPA chỉ là specification (định nghĩa), chứ ko phải implementation
          2. Nó là một tập hợp các quy tắc và hướng dấn để set các interface để triển khai ORM
       3. **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 => vì JPA là tập hợp các interface, nên sẽ cần implementation để triển khai ORM (JPA providers/ORM frameworks)**
    6. **ORM?**
       1. <https://viblo.asia/p/object-relational-mapping-djeZ1PQ3KWz>
       2. <https://www.altexsoft.com/blog/object-relational-mapping/>
       3. **Object Relational Mapping** là một kĩ thuật giúp mapping các record trong database sang dạng object được định nghĩa trong class. Hiểu nôm na là convert data giữa CSDL với ngôn ngữ OOP như Java,… => cho phép ta có thể thao tác với database mà ko cần biết đang thao tác với database nào, kiểu dữ liệu trong database.
       4. Có 2 pattern cho cách thiết kế ORM
          1. Active record (éo hiểu lắm, đại khái là chỉ hợp cho việc phát triển các app nhỏ)
          2. **Data mapper (Hibernate sử dụng cái này,** hợp cho phát triển app lớn, dễ maintain và mở rộng**)**

Đó là 1 layer mapper đưa dữ liệu qua lại giữa object và Database mà vẫn giữ được tính độc lập giữa chúng và cả giữa chính nó (lớp mapper). Như vậy là có layer đặc biệt tách biệt với bộ nhớ của database và nhiệm vụ chính là chuyển dữ liệu qua lại giữa 2 lớp

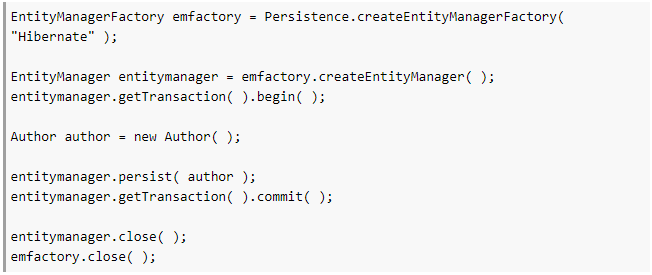


* + - 1. Cơ chế hoạt động ORM
         1. Đặc trưng cơ bản của ORM là gói gọn CSDL trong 1 object. 1 phần của object sẽ chứa data, và phần còn lại lo việc data xử lý như nào và biến nó thành CSDL quan hệ.
         2. ORM giải quyết vấn đề đồng bộ giữa kiểu dữ liệu trả về khác nhau. Một bên là cơ sở dữ liệu, **ở đó dữ liệu được thể hiện dưới dạng tập hợp các bản ghi**. Một bên là các đối tượng, **ở đó dữ liệu đc hiện thị dưới dạng object**
      2. Ưu điểm
         1. Tuân thủ nguyên tắc DRY, tập trung tại một chỗ, và việc chỉnh sửa, maintain, tái sử dụng lại code sẽ dễ dàng hơn
         2. Mọi thứ được **tự động hoá**. Ko cần biết đang chọc tới CSDL nào, phải viết câu SQL như nào. ORM sẽ tự generate câu SQL với hệ CSDL tương ứng. Kiểu data CSDL sẽ tự convert đang kiểu data của Java. => tập trung hơn vào OOP, viết code ít hơn.
         3. Flexible: Ko phụ thuộc vào hệ CSDL nên dễ thay đổi database.
      3. Nhược điểm
         1. Vì là thư viện, nên sẽ hơi **nặng**
         2. Khả năng truy vấn hạn chế, vài trường hợp vẫn phải dùng raw sql.
         3. Vì query SQL được ORM generate một cách tự động (vì trước khi biên dịch ra các câu truy vấn SQL thì cần có quá trình map data giữ các lớp đối tượng), nên có thể sẽ ko tối ưu => Performance ổn đối với truy cấp thông thường, nhưng đối với truy vấn phức tạp thì sẽ bị chậm => Raw SQL (**native query**) sẽ tối ưu hơn về mặt tốc độ.
* Khác biệt chỉ thực sự rõ khi select, update vài nghìn record => không đáng kể => nên sử dụng linh hoạt giữa ORM và Raw SQL.
  + 1. POJO object?
       1. <https://www.geeksforgeeks.org/pojo-vs-java-beans/>
       2. **Plain Old Java 0bject** là object Java **thuần tuý.** Chỉ có Java, ko có implement, extends class nào, ko dính dáng gì tới framework. Chỉ bao gồm các field và getter, setter (ko nhất thiết phải có constructor) => **đơn giản là define một entity**
       3. Phân biệt POJO và Java Bean
          1. 
    2. Spring Data Repository
       1. <https://www.javatpoint.com/spring-boot-crud-operations>



* + - 1. **Repository<T, ID>** là **marker interface** (interface ko chứa method gì, chỉ có tác dụng đánh dấu interface này có mục đích gì đó). **T là kiểu entity mà repository này quản lí. ID là kiểu dữ liệu của ID**
      2. CrudRepository extends interface Repository, cung cấp các method thao tác CRUD
      3. PagingAndSortingRepository extends interface CrusRepository, thêm các method để phân trang và sort.
      4. **JpaRepository** extends interface PagingAndSortingRepository, thêm các method như flushing hay deleteInBatch. => **JpaRepository chứa tất cả các method của CrudRepository và PagingAndSortingRepository => nếu ko cần xài tới các function của JpaRepository và PagingAndSortingRepository, chỉ cần xài CrudRepository là đủ**
      5. 
    1. **JPA Providers/ORM frameworks**
       1. Hibernate là một trong những JPA Providers/ORM framework phổ biến (vì JPA chỉ là một tập các interface, thì Hibernate và Spring Data JPA là implementation của nó)

=> Hiểu nôm na **JPA là 1 tầng abstraction**, cung cấp các định nghĩa (interface) để thao tác giữa object Java và database . **ORM là một cơ chế để thực hiện nó**. Và **ORM framework là cách triển khai chi tiết nó làm như nào.**

* + - 1. **Spring Data JPA**
         1. <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>
         2. <https://stackoverflow.com/questions/23862994/whats-the-difference-between-hibernate-and-spring-data-jpa>
         3. <https://www.stackchief.com/blog/Spring%20Data%20JPA%20vs%20Hibernate>
         4. Spring Data có thể generate JPA query thay cho chúng ta thông qua method name convention (ví dụ findByNameAndAge…)
         5. Nếu ko xài Spring Data JPA, thì mỗi lần muốn chọc vào database thì sẽ phải code hết đống này
         6. 
      2. **Lưu ý:**
         1. Phân biệt Spring Data JPA vs Hibernate: **Hibernate là một JPA implementation, còn Spring Data JPA thì không**.
         2. Spring Data JPA chỉ đệm thêm 1 lớp JPA abstraction cho việc truy cập dữ liệu trong database. Spring Data JPA ko thể hoạt động nếu ko có JPA Provider. Hibernate và Spring Data bổ sung cho nhau. => project có sử dụng Spring Data JPA thì có thể switch sang khác JPA Provider khác nhau.
    1. Lombok?
       1. <https://stackjava.com/library/lombok-la-gi-sinh-code-tu-dong-tren-eclipse-intellij.html>
       2. <https://daothaison.me/general-huong-dan-su-dung-lombok-giup-code-java-nhanh-hon-69-loda1552789752787>
       3. Lombok là một thư viện Java giúp giảm thiểu các đoạn code thừa (boilerplate) bằng cách generate các đoạn code getter & setter,… một cách tự động. Bên cạnh đó còn hỗ trợ sinh các hàm khởi tạo (constructor) với tham số, hoặc không có tham số.
       4. Hiểu đơn giản là **lombok sẽ có 1 vài annotation sẽ giúp cho mình có thể rút gọn mấy method thường có trong class POJO => rút gọn code, trông đỡ rườm rà**
          1. @NoArgsConstructor
          2. @AllArgsConstructor
          3. @RequiredArgsConstructor

Hàm khởi tạo theo yêu cầu. Bạn chỉ muốn hàm khởi tạo có vài thuộc tính do bạn chọn thôi, thì bạn thêm final trước thuộc tính trong class, nó sẽ tự sinh ra Constructor như thế.

* + - * 1. @Getter
        2. @Setter
        3. @toString
        4. @EqualsAndHashCode
        5. @Builder

Tự generate các hàm setter của class theo builder pattern: Student me = Student.builder().name(“Trung”).age(23).build();

* + - * 1. **@Data = @Getter + @Setter + @ToString + @EqualsAndHashCode + @NoArgsConstructor**
    1. Mail
    2. **Những annotation phố biến**
       1. **@SpringBootApplication**
       2. **@Data vs @Entity?**
       3. **@Autowired**
          1. Những phiên bản Springboot mới thì ko cần annotation @Autowired trên constructor
       4. @Primary, @Qualifier
       5. **@Component, @Controller, @Service, @Repository**
          1. <https://stackoverflow.com/questions/35449808/what-will-happen-if-we-interchange-service-and-repository-annotation-in-the-sp>
          2. @Controller, @Service, @Repository: **3 annotation này cơ bản là giống nhau, đều giống annotation @Component.** Có thể thay thế bằng @Component, app vẫn run bình thường, nhưng ko nên vì để annotation đó class sẽ meaningful hơn, ta sẽ biết được class đó thực hiện nhiệm vụ gì.
          3. **Tại sao bỏ annotation @Repository mà app vẫn run được?**
       6. @Transactional
          1. Khai báo ở trên method hoặc trên class (áp dụng cho toàn bộ method trong class). Khi đặt annotation này, mọi xử lí trong method sẽ được nhét vào trong 1 transaction.
          2. Mặc định: chỉ khi có **Error** hoặc **Un-check exception (RuntimeException)** xảy ra, transaction sẽ rollback lại, **Checked exception (CompileTimeException) thì không** => giải pháp: thêm điều kiện rollback @Transactional(rollbackFor = {Exception.class, Throwable.class})
       7. @Configuration, @Bean
       8. **@RestController = @Controller + @ResponseBody**
          1. 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.
          2. Còn @RestController là đã bao gồm @Controller + @ResponseBody rồi. Mặc định sẽ trả về JSON/XML thay vì HTML.
       9. **@RequestMapping**
          1. Có thể dùng bên trên class controller, để chỉ định endpoint gốc cho toàn bộ method bên trong nó
          2. Hoặc vấn có thể dùng bên trên method trong controller (ít dùng)

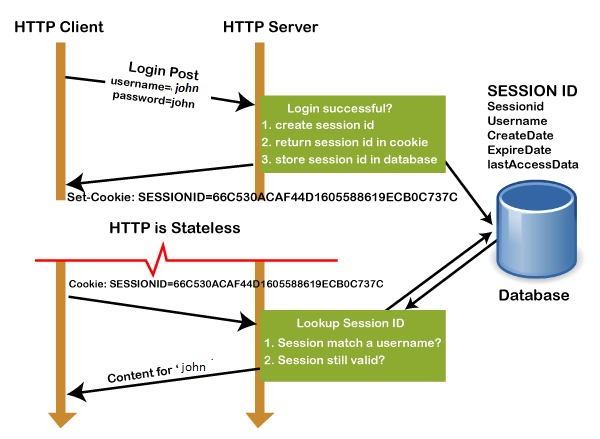
@RequestMapping(value = "/users", method = RequestMethod.GET)

* + - 1. @GetMapping, @PostMapping, @PutMapping, @PatchMapping, @DeleteMapping
      2. @RequestBody
      3. **Phân biệt @Valid vs @Validated?**
      4. **@EnableGlobalMethodSecurity**
         1. Spring cung cấp 1 số annotations để kiểm tra ủy quyền trước và sau khi check, filter các đối số đã gửi hoặc giá trị trả về: @PreAuthorize, @PreFilter, @PostAuthorize và @PostFilter. => Để xài các annotation này thì phải có annotation @EnableGlobalMethodSecurity
  1. **Spring Security** 
     1. **Session? Cookie?**
        1. <https://viblo.asia/p/session-cookie-va-cache-ban-da-thuc-su-hieu-chung-3P0lPv3PKox>
        2. <https://www.javatpoint.com/session-vs-cookies>
        3. **Session**
           1. Session hiểu đơn giản là một phiên làm việc giữa 2 thiết bị và 2 hệ thống tương tác với nhau.
           2. **Session dùng để lưu trữ tạm thời những thông tin trên server mà sẽ được sử dụng trên nhiều page khác nhau của website. Session bắt đầu khi user truy cập vào hệ thống (lần đầu tiên) và kết thúc khi user thoát khỏi hệ thống => dùng để lưu trữ thông tin ngắn hạn.**
           3. Web server bản chất ko biết user đã làm gì (vì chúng ko có bộ nhớ), vì giao thức HTTP ko lưu trạng thái (**stateless).** Giả sử thông tin mà user tương tác ở 1 trang (giả sử ấn thêm vào giỏ hàng) sẽ ko được transfer sang trang khác (trang giỏ hàng) => vì vậy phải dùng session để lưu những thông tin này lại. (một số trang web người dùng ko cần đăng nhập mà vẫn có thể add vào giỏ hàng, chỉ khi nào ấn nút thanh toán thì mới bắt đăng nhập => dùng session là có thể lưu được rồi chứ chưa cần đăng nhập)
           4. **Cơ chế hoạt động:**

Mỗi khi user bắt đầu truy cập vào trang web sẽ được cấp một mã sessionID. Và ID này sẽ bắt đầu với một session, chứa tất cả những dữ liệu của người truy cập từ đầu phiên đến cuối phiên.

**Với mỗi session được tạo ra, server sẽ tạo ra một tệp session lưu ở phía server và một tệp cookie CHỨA sessionID và gửi ngược về trình duyệt của user => với mỗi lần user gửi request thì chỉ cần so sánh tệp cookie bên phía client gửi lên server và tệp session được lưu trên server là sẽ phân biệt được session đó của của client nào.**

SessionID có một đặc điểm: khi nhiều tab của cùng một website được mở, chúng được chỉ định cho một sessionID duy nhất.



Session có thời hạn. Khi session expired thì user sẽ được cấp một sessionID mới và cứ như vậy tới khi user thoát trang web (đóng trình duyệt).

Session được lưu ở 1 tệp tin trên server (lưu ở đâu thì tuỳ vào web được code bằng framework gì), và khả năng lưu trữ là vô hạn => nhưng nếu dùng vô tội vạ, lưu trữ quá nhiều sẽ khiến server tốn tài nguyên

Vì session được lưu ở phía server => khó bị đánh cắp hoặc sửa đổi

* + - 1. **Cookie**
         1. Cookie cũng như session, được dùng để lưu những thông tin tạm thời.
         2. **Nhưng khác session ở chỗ, tệp cookie được tạo ra và truyền từ server tới browser và được lưu trữ trên máy tính của user khi user truy cập vào web (lần đầu tiên).**
         3. **Cookie có thể set thời gian expire => dùng để lưu trữ thông tin dài hạn (giả sử user login rồi tắt máy trong vài ngày, hôm sau user mở máy lên account vẫn đã được login sẵn)**
         4. **Mỗi khi user gửi request, browser sẽ gửi kèm theo cookie để thông báo cho ứng dụng về hoạt động trước đó của user.**
         5. **Cookie được gửi cùng request thông qua HTTP header.**
         6. Cookie có kích thước khá nhỏ, chỉ 4KB và chỉ lưu trữ data dưới dạng String.
         7. Theo luật của Liên Minh Châu Âu (GDPR: General Data Protection Regulation - Quy định chung về bảo mật thông tin), website mà có trụ sở đặt tại EU thì muốn xài cookie thì phải thông báo cho user => user phải accept xài cookie thì web server mới gửi về được.
         8. **Vì cookie được lưu ở browser (lưu ở đâu thì tuỳ xài browser gì) => ko an toàn, có thể sẽ dàng bị đánh cắp hoặc thay đổi (dùng extension) => có thể gửi request độc hại tới server.**
      2. Tóm cái váy lại
         1. **Đặt ra câu hỏi: data có cần được lưu lại sau khi user tắt trình duyệt không?**

Nếu **có,** dùng **cookie**.

Nếu **ko,** dùng **session**.

* + 1. **Web Storage API (LocalStorage và SessionStorage) vs Cookie?**
       1. <https://viblo.asia/p/local-storage-session-storage-va-cookie-ORNZqN3bl0n>
       2. <https://stackoverflow.com/questions/19867599/what-is-the-difference-between-localstorage-sessionstorage-session-and-cookies>
       3. Cả 3 đều lưu ở phía client => đều có thể bị user chỉnh sửa, kém bảo mật. User có thể xem hoặc chỉnh sửa bằng cách F12 ở chrome
       4. Được lưu trữ dưới dạng key-value.
       5. **Web Storage API?**
          1. Lưu trữ được nhiều data (khoảng 5MB)
          2. Chỉ lưu trên client, không gửi lên server
          3. Cách lưu object: chỉ lưu được kiểu String => cần convert object sang string bằng hàm JSON.stringify, sau khi get dc data thì convert string về object bằng hàm JSON.parse

LocalStorage

Khả năng lưu trữ vô thời hạn: chỉ có thể xoá bằng javascript hoặc xoá cache trình duyệt

Phạm v i: có thể truy xuất lẫn nhau giữa các cửa sổ trình duyệt.

SessionStorage

Khả năng lưu trữ: sẽ mất khi đóng tab hoặc cửa sổ browser

Phạm vi: giới hạn trong một window hoặc một tab của browser. Một trang web được mở trong 2 tab trong cùng một window ko thể truy xuất dữ liệu lẫn nhau.

* + - 1. Cookie
         1. Có thời gian timeout do lập trình viên xác định trước
         2. Lưu trữ được ít data (4KB)
         3. Có thể đọc ở cả client và server
    1. **Authenication? Authorization?**
       1. <https://techmaster.vn/posts/36295/spring-security-ban-sau-ve-authentication-va-authorization-p1>
       2. **Authentication (xác thực)**
          1. <https://kipalog.com/posts/Authentication-story-part-1--Authentication-la-lam-gi>
          2. **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.**
          3. Nếu ko có authentication, hệ thống sẽ ko biết ai đang truy cập hệ thống để có phản hồi phù hợp.
          4. **Authentication diễn ra trước khi Authorization.**
          5. Nếu hệ thống ko thể authenticate request gửi lên, sẽ trả về lỗi 401 Unauthorized.
       3. **Authorization (uỷ quyền)**
          1. <https://viblo.asia/p/phan-biet-su-khac-nhau-giua-authentication-va-authorization-Eb85oad4Z2G>
          2. **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.**
          3. Nếu ko có authorization, user đó sẽ ko có quyền thực hiện chức năng gì (ngoại trừ những chức năng được permit all).
          4. **Authorization diễn ra sau Authentication.**
          5. 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
    2. **Basic auth? Session-based auth? Token-based auth?**
       1. <https://kipalog.com/posts/Authentication-story-part-2--Authentication-co-ban>
    3. **ServletFilter? FilterChains?**
       1. <https://techmaster.vn/posts/36295/spring-security-ban-sau-ve-authentication-va-authorization-p1>
       2. <https://www.marcobehler.com/guides/spring-security>
          1. **ServletFilter**

Cơ bản bất kì web Spring nào cũng đơn giản là xài **Servlet**, có tác dụng giúp **redirect HTTP request tới @Controller hoặc @RestController**

Vì chúng ta ko muốn hardcode phần security vào tất cả header trong controller. Nên để tối ưu, authentication và authorization nên làm trước khi request chạy vào controller.

Có một cách để làm việc này: thêm một **Filter** vào trước servlet, có nghĩa là ta sẽ viết **SecurityFilter** và config nó để lọc tất cả HTTP request trước khi chạm vào servlet.

Một **SecurityFilter** đơn giản bao gồm 4 task cơ bản:

Đầu tiên filter sẽ extract username/password từ request, chúng có thể từ form fields, cookie…

Sau đó filter sẽ validate username/password với một cái gì đó, ví dụ như database

Sau khi authenicate thành công, filter sẽ authorize xem user có được phép truy cập tới request URI không

Nếu request vượt qua sau tất cả lần kiểm tra này, filter sẽ cho request di chuyển tới **DispatcherServlet**, tức là @Controller

* + - * 1. **FilterChain**

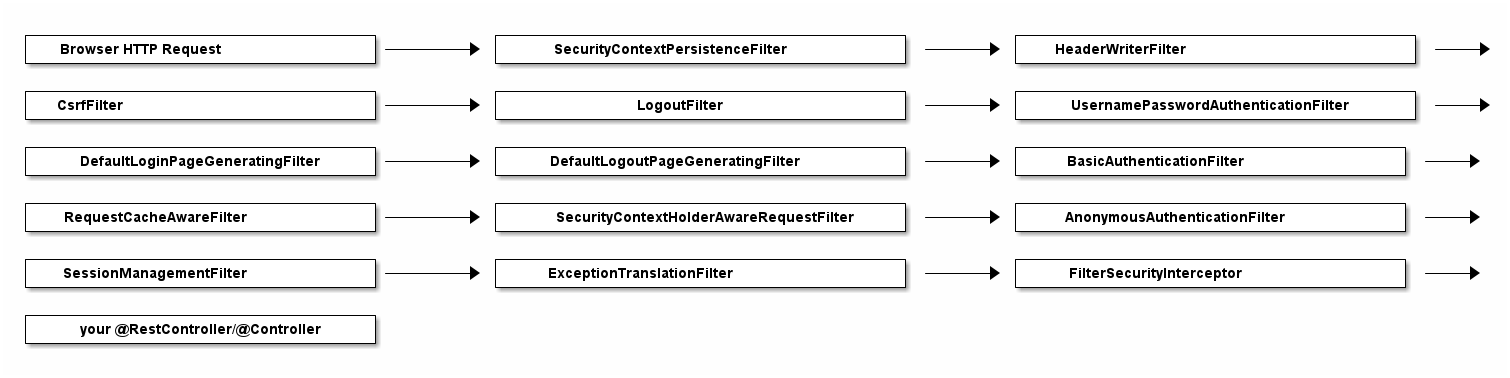
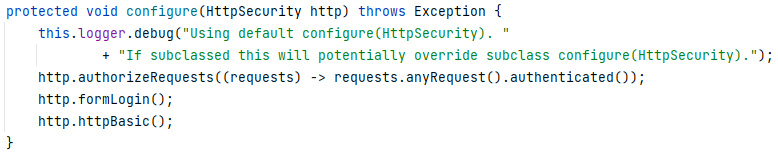
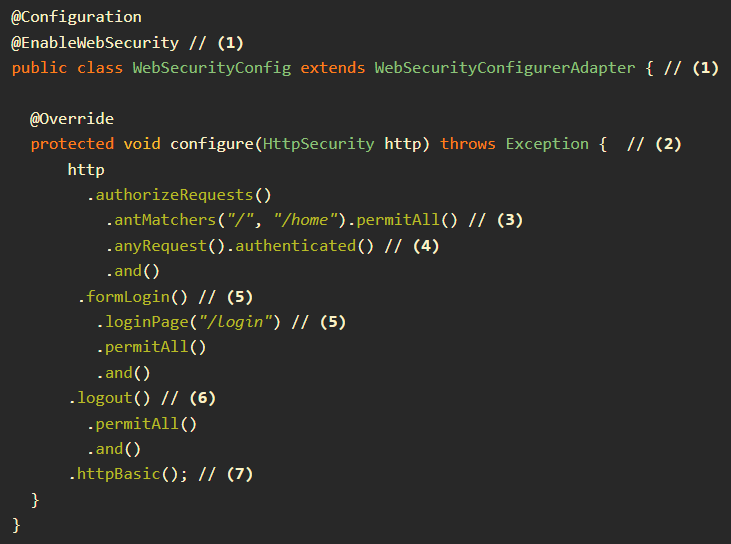
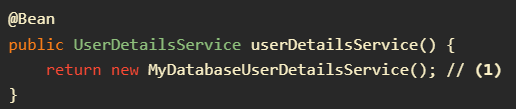
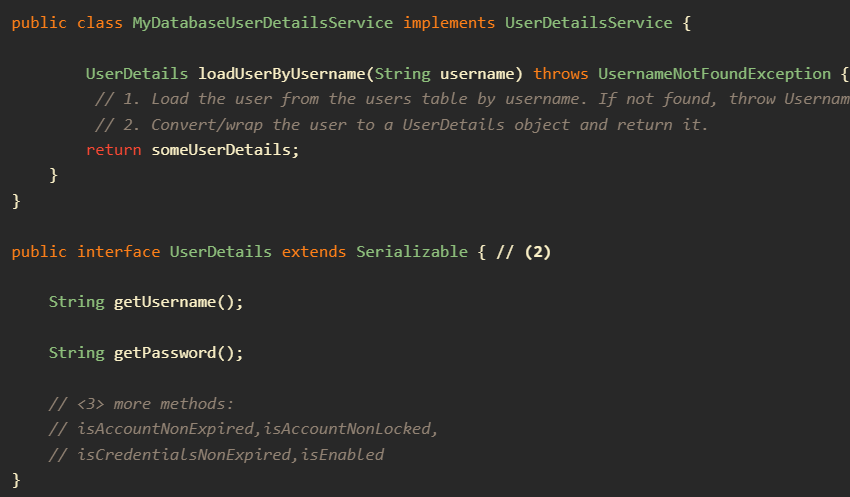
Trong khi code, sẽ sinh ra rất nhiều code liên quan tới authentication và authorization. Thực tế, filter này chia làm nhiều filter, sau đó **“xâu chuỗi” (chain)** lại với nhau

Đầu tiên, đi qua LoginMethodFilter

Sau đó đi qua AuthenticationFilter

Sau đó đi qua AuthorizationFilter

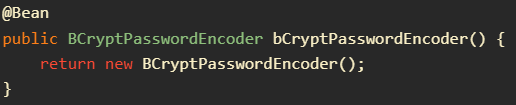
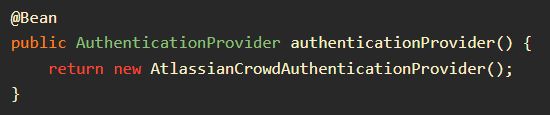
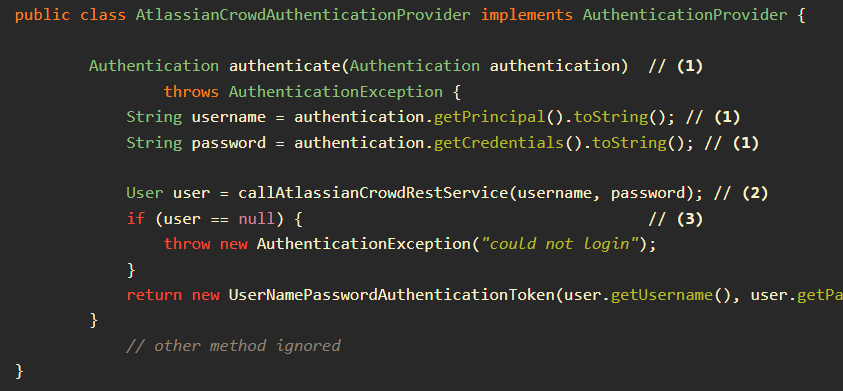
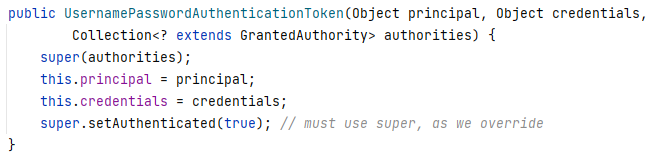
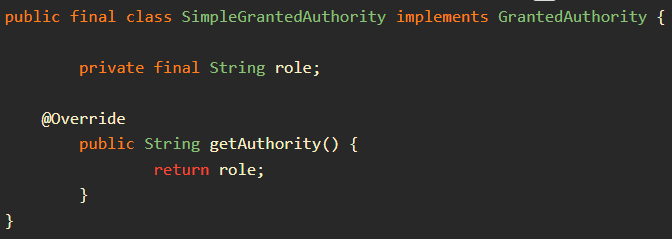
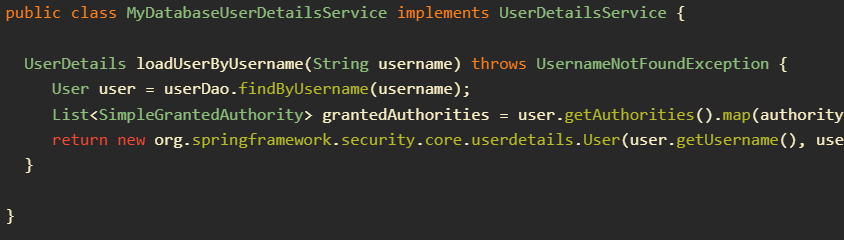
Cuối cùng tới servlet

* + Concept này gọi là **FilterChain.** Nhờ cái này, ta có thể xử lí mọi vấn đề liên quan tới authentication và authorization mà ko cần chọc tới @Controller hay @RestController (lập trình AOP)
  + ****
* Khi ta setup Spring Security đúng cách và run app. Mặc định Spring Security sẽ install cả filterchain, bao gồm đâu đó tận 15 filters khác nhau. Giải thích thì khá dài dòng, nhưng đại khái là mục đích của Spring security chỉ có thế, nó tạo ra thêm nhiều filter để authen và author request. Kiến trúc mặc định là như vậy, những gì ta cần thì là config nó hoạt động như thế nào.
  + 1. **WebSecurityConfigurerAdapter**
       1. Với phiên bản Spring Boot và Spring Security mới nhất, cách để config Spring Security là **có một class @Configuration có annotation @EnableWebSecurity và extends abstract class WebSecurityConfigurerAdapter (**class này sẽ cho ta nhưng method để định nghĩa URI nào phải login, phải có role gì mới có thể call, …)
       2. **Note: mặc định khi add Spring security, nó sẽ cấp một trang HttpLogin mặc định và FilterChain sẽ authenticated tất cả các request (tất cả các request đều phải login mới có thể call được) => muốn config lại FilterChain phải override lại method configure(HttpSecurity) để config.**
       3. 
       4. 
          1. authorizeRequests(): để lấy ra AuthorizationConfigurer
          2. [antMatcher()](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/AntPathMatcher.html): chỉ những Matcher match với request URI này (hiểu nôm na là để chỉ những request URI mà match theo regex này). antMatcher() với mvcMatcher() cơ bản là giống nhau
          3. permitAll(): chấp nhận request (có thể call mà ko cần user phải authenticated). Nhớ là chính trang /login cũng phải permitAll để user có thể vào trang login mà ko cần login trước :D
          4. anyRequest(): tất cả request URI còn lại ngoại trừ những request URI viết trong antMatcher()
          5. authenticate(): bắt buộc user phải login mới call được
    2. **UserDetailService và PasswordEncoder**
       1. Về cơ bản thì trong database luôn có 1 bảng user chứa 2 cột chính là username và password, mà password thì thường phải hash.
       2. Spring Security cần ta define 2 bean **UserDetailService** và **PasswordEncoder** để Authenticate.
       3. **UserDetailService**
          1. 
          2. 
          3. Class **UserDetailService** thì chỉ có 1 method loadUserByUsername, ta sẽ override lại method này và định nghĩa cách load user lên từ database (lưu ý là method chỉ truyền username vào, chứ ko có password)
          4. Interface **UserDetail** sẽ có method getUsername và getPassword, ta sẽ override lại 2 method này để trả về username và password.
          5. **Flow của UserDetailService:**

Extract username/password từ header trong một filter.

Call tới UserDetailService để load user từ database lên, gói nó vào 1 object UserDetail, đã bao gồm username và password đã hash

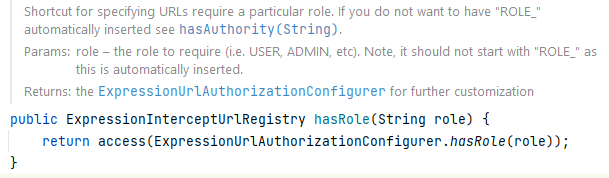
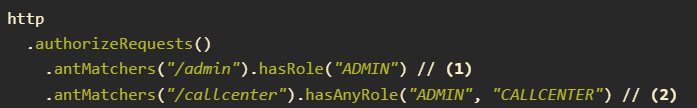
Lấy password từ header, hash nó và so sánh với pasword đã hash trong object UserDetail, nếu match thì user được authenticated.

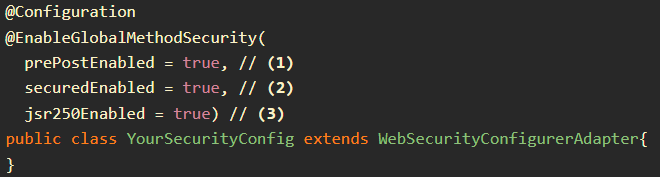
* + - 1. **PasswordEncoder**
         1. Spring Security sẽ cần ta định nghĩa thêm một @Bean nữa là PasswordEncoder để chọn thuật toán hash password. Mặc định Spring Security sẽ xài **Bcrypt**.
         2. 
         3. Nếu ko muốn hash password thì dùng thằng **NoOpPasswordEncoder()** (tất nhiên là ko khuyến khích, chỉ dùng cho mục đích test)
    1. **AuthenticationProvider**
       - 1. ****
         2. ****
       1. Về nôm na thì nếu hệ thống của ta việc authen ko phải do bên mình làm (trong database không có username password) mà do bên thứ 3 làm. Thì ta sẽ xài thằng AuthenticationProvider => ta sẽ ko xài UserDetailService nữa
       2. So với method loadUserByUsername() của thằng UserDetailService, cái mà ta chỉ access được vào username, giờ ra có thể access vào cả username và password
       3. Nếu authentication fail thì throw exception
       4. Nếu authentication thành công, ta sẽ return về 1 thằng UsernamePasswordAuthenticationToken, đây là class implement interface Authentication và field **authenticated** set bằng **true** (vì đã authenticated nên sẽ bao gồm cả authorities)
          1. ****
       5. **Flow của AnthenticationProvider:**
          1. Extract username/password từ header trong một filter.
          2. Call tới thằng AuthenticationProvider cùng với username và password, mình sẽ tự implement việc authorize như nào.
* Nôm na nếu ta dùng Spring Security và không có quyền access vào user password, ta sẽ xài thằng @Bean **AuthenticationProvider**. Vì ta cho bên thứ 3 toàn quyền quản lí username password, nên sẽ ko có thuật toán hashing nào ở đây.
  + 1. **Authorization**
       1. **GrantedAuthorities? SimpleGrantedAuthorities?**
          1. Spring Security đơn giản ko để ta chỉ xài mỗi String để lưu authority. Class **SimpleGrantedAuthority**, implement interface **GrantedAuthority** sẽ lưu String authority này.
          2. 
          3. 
          4. Ta sẽ load authority từ DB lên và cho vào list SimpleGrantedAuthority để nhét vào UserDetail trả về
       2. **Authorize**
          1. Ta chỉ cần config lại hàm configuure(HttpSecurity) là authorize được
          2. ****

Muốn vào trang /admin thì user phải authenticated và có authority ROLE\_ADMIN

Muốn vào trang /callcenter thì user phải authenticated và có authority ROLE\_ADMIN hoặc ROLE\_CALLCENTER

Còn các request còn lại user **vẫn phải authenticated nhưng ko cần role**

* + - 1. **hasAuthority()? hasRole()?**
         1. <https://stackoverflow.com/questions/19525380/difference-between-role-and-grantedauthority-in-spring-security>
         2. Khái niệm **authority** và **role** khá giống nhau. Đều có nghĩa là “permission” (quyền). Đều là String
         3. Từ phiên bản Spring Security 4, method **hasAuthority(“ROLE\_USER”)** giống với method **hasRole(“USER”)** vì Spring Security sẽ tự thêm tiền tố **“ROLE\_”** vào
         4. ****
         5. **Đoạn code trên và đoạn code này tương đương nhau:**
         6. ****
    1. **CSRF?**
    2. **Oauth 2.0?**
    3. **Spring Integration**
       1. Ngoài việc bảo vệ các URL bằng antMatcher() với DSL của WebSecurityConfigurerAdapter, ta còn có thể bảo vệ sâu hơn, cả logic nghiệp vụ, có nghĩa là Spring bean (@Controllers, @Services, …)
       2. **Method security**
          1. Method security hoạt động thông qua annotation mà ta đặt trên method public. Nhưng ta phải bật method security trước bằng cách thêm annotation **@EnableGlobalMethodSecurity**() trong file security config

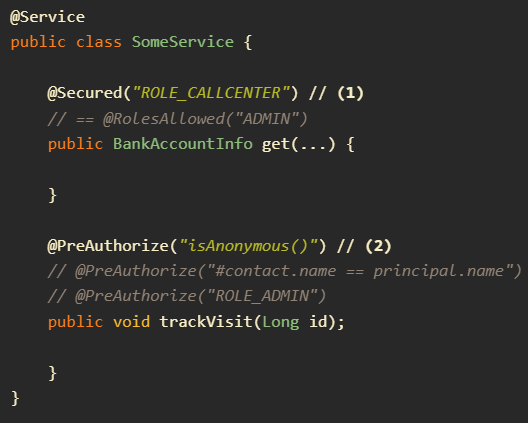


**prePostEnabled** hỗ trợ annotation **@PreAuthorize** và **@PostAuthorize**

securedEnabled hỗ trợ annotation **@Secured**

jsr250Enabled hỗ trợ annotation **@RolesAllowed**

* + - * 1. Nếu ko set những thuộc tính này bằng true, Spring sẽ bỏ qua những annotation này.
        2. 2 annotation **@Secured** và **@RolesAllowed** cơ bản là cũ hơn, value là string authority/role. **@PreAuthorize** và **@PostAuthorize** mới hơn, nó ngoài có thể chứa value là string authority/role, mà còn có thể xài **SpEL expession** (Spring Expression Language).
        3. Những annotation này sẽ đều throw **AccessDeniedException** nếu ta cố gắng call API nếu role của a ko đủ quyền hạn.



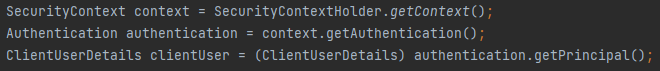
**@Secured(“ROLE\_ADMIN”)** sẽ tương tự với **@RolesAllowed(“ADMIN”).**

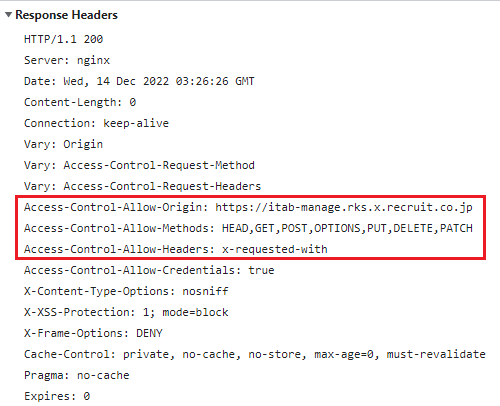
**@PreAuthorize** ngoài dùng String authority, mà còn có thể xài [SpEL expession](https://docs.spring.io/spring-security/reference/servlet/authorization/expression-based.html), ví dụ: **@PreAuthorize(hasRole(“admin”))**

Về cơ bản thì xài annotation nào cũng được, chỉ cần đồng nhất với dự án.

* + - 1. **@AuthenticationPrincipal**
         1. Vì Spring Security tích hợp với Spring WebMVC, nên ta cũng có thể Inject **authenticated principal vào trong @Controller/@RestController** method.
         2. Bằng cách xài annotation **@AuthenticationPrincipal.** Nó sẽ inject **UserDetail** nếu user đã authenticated, hoặc null nếu ko có user authenticated.
         3. Nếu ko xài **@AuthenticationPrincipal**, ta sẽ phải lấy principal ra bằng tay trong **SecurityContextHolder**:





* + 1. **CORS filter (Cross-origin resource sharing)?**
       1. <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
       2. [**https://viblo.asia/p/tim-hieu-ve-cross-origin-resource-sharing-cors-Az45bGWqKxY**](https://viblo.asia/p/tim-hieu-ve-cross-origin-resource-sharing-cors-Az45bGWqKxY)
       3. <https://topdev.vn/blog/cors-la-gi/>
       4. Trước khi có tiêu chuẩn về CORS, không có cách nào để một trang web gửi request thông qua browser tới một domain khác.
       5. Hiểu nôm na đây là một cơ chế cho phép nhiều resource khác nhau của một trang web có thể được chọc vào từ các request tới từ domain khác với domain của trang đó. Hiểu nôm na là mình có thể config server có thể bị chọc vào từ request tới từ domain nào, method gì… (VD: trang web mysite.com có thể config để ko accept request get list user tới từ request GET othersite.com/getuser)
       6. Cơ chế này nhằm hạn chế các cuộc tấn công **Cross-site scripting (XSS),** nôm na là 1 kiểu đánh cắp thông tin mà attacker sẽ nhúng đoạn code vào website (thường là Js hoặc HTML) để gửi các thông tin đánh cắp được về server khác.
       7. **Cơ chế hoạt động:**
          1. Client tạo request GET, POST, PUT, HEAD lên server. Những request này sẽ kèm theo một header tên là Origin (chính là domain của trang web)
          2. 
          3. Server sẽ xem xét Origin để biết được nguồn này có phải là nguồn hợp lệ hay không. Nếu hợp lệ, server sẽ trả về response kèm với header Access-Control-Allow-Origin. Header này sẽ cho biết xem client có phải là nguồn hợp lệ để browser tiếp tục thực hiện quá trình request.
          4. 
       8. **CORS HTTP headers**
          1. Về cơ bản thì server sẽ thông báo cho browser biết là server chỉ chấp nhận request từ origin nào và HTTP request method nào.

**Access-Control-Allow-Origin:** Cho browser biết domain nào được phép truy xuất resource từ server đó

**Access-Control-Allow-Headers:** Cho browser biết những request header nào được phía server hỗ trợ (x-authentication-token, Authorization, …). Nếu clienet gửi những header khác sẽ bị server bỏ qua

**Access-Control-Allow-Methods:** Cho browser biết những HTTP method cho phép client sử dụng (GET, POST, PUT,…)

* + 1. **Phân biệt method GET và POST?**
    2. **HTTP vs HTTPS? SSL certificate?**
    3. Token
       1. JWT?
    4. Encrypt method?
    5. User details
    6. Role
    7. Authentication
    8. Authority
    9. Encrypt
  1. Spring batch

1. OOP? AOP
   1. **AOP** 
      1. <https://viblo.asia/p/su-dung-aop-trong-spring-boot-va-aspectj-vyDZOkbaZwj>
      2. **Aspect Oriented Programming (AOP)** – lập trình hướng khía cạnh: là một kỹ thuật lập trình nhằm phân tách chương trình thành các module riêng rẽ, phân biệt, không phụ thuộc nhau.
      3. 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.

**=> Đại khái AOP là 1 kĩ thuật code mà mình có thể thêm những đoạn code xử lí mới vào các đoạn code đã tồn tại mà ko cẩn phải sửa những đoạn code đó (Exception Handler của Spring là 1 ví dụ, hoặc là để chèn Log vào các method).**

* + 1. AOP không phải dùng để thay thế OOP mà để bổ sung cho OOP.
    2. Ưu điểm:
       1. Tái sử dụng code một cách tối đa
       2. Cách thức hoạt động tốt: module nào làm việc của module đó. Gặp lỗi, cần sửa module nào thì chỉ cần chọc vào module đó, ko ảnh hưởng module khác.
       3. Code tổ chức tốt hơn. Mọi module, chức năng đều độc lập. Dễ phân biệt phần xử lí chính, phần xử lí phụ. Có thể bật tắt các phần xử lí chính, xử lí phụ tuỳ ý => Giảm thời gian thiết kế, maintain hệ thống
    3. Nhược điểm
       1. Tăng sự trừu tượng của app
       2. Tăng độ phức tạp của app, khó nắm bắt flow code chạy như nào
       3. Yêu cầu trình độ cao vì khó thiết kế
    4. AOP trong Springboot
       1. Để chèn 1 đoạn code nằm giữa 1 luồng xử lí, ta tạo 1 file @Aspect
          1. **@Aspect**: đánh dấu đây là 1 file aspect
          2. **@Before**(“execution…)): đánh dấu method này sẽ cần chạy trước method cần chèn
          3. **@After**(“execution…)): method này sẽ cần chạy sau method cần chèn (kể cả run thành công hay có exception)
          4. **@AfterReturning**(“execution…)): chỉ chạy sau khi method run thành công
          5. **@AfterThrowing**(“execution…)): chỉ chạy sau khi method có exception

1. Tấn công qua mạng
   1. SQL Injection
   2. XSS
2. Firebase
3. Build tools, package manager
   1. <https://topdev.vn/blog/tai-sao-lai-dung-gradle-thay-the-maven-va-ant/>
   2. Gradle
      1. Build.gradle
   3. Maven
   4. So sánh Gradle vs Maven
      1. Build script viết bằng Gradle ngắn gọn và trực quan dễ đọc hơn
      2. Thời gian build của gradle nhanh hơn
4. SQL
   1. Relational DB
      1. MySQL
      2. Oracle
      3. MS SQL
      4. PostgreSQL
   2. Non-relational DB
      1. MongoDB
      2. Redis
      3. Cassandra
5. **Software development principles?**
   1. **SOLID?**
      1. **Single responsibility principle** 
         1. Hiểu nôm na là mỗi 1 module chỉ nên thực hiện 1 nhiệm vụ duy nhất (ví dụ đơn giản là code cần sửa đổi ở 1 chỗ nhưng chỗ đó đang phụ thuộc vào code ở nhiều chỗ khác, sửa chỗ này sẽ phải sửa nhiều chỗ khác => fix bug này lòi bug kia)
         2. Áp dụng Single responsibility principle nghĩa là tách 1 module đang làm nhiều việc thành nhiều module nhỏ => hỏng cái nào sửa cái đấy, ko ảnh hưởng tới nhau (đơn giản là tách 1 xử lí lớn ra làm nhiều xử lí nhỏ, rồi gom nhóm với nhau)
   2. **DRY (Don’t repeat yourself)**
      1. Hiểu nôm na là đừng có viết lặp bất kỳ lại một đoạn mã nào mà hãy đóng gói nó thành phương thức riêng. Đến khi cần thì chỉ cần gọi tên nó ra => dễ maintain, tái sử dụng code. Sửa code ở 1 chỗ, các chỗ khác sẽ đều áp dụng theo, mọi thứ sẽ được đồng bộ.
6. **Database/transaction principles**
   1. **ACID (Atomicity, Consistency, Isolation và Durability)**
      1. [**https://vietnix.vn/acid-la-gi/**](https://vietnix.vn/acid-la-gi/)
      2. Là bốn thuộc tính quan trọng cần đảm bảo khi thực hiện bất kỳ transaction nào với database.
      3. Là các yêu cầu về tính an toàn, tính bền vững và tính trọn vẹn cho dữ liệu
      4. Nếu một tiến trình không đảm bảo được các thuộc tính trên thì sẽ không thể thực hiện được các thao tác với data và sẽ bị hủy ngay lập tức.
         1. **Atomicity (tính nguyên tử)**
            1. Hiểu nôm na là tính đảm bảo trọn vẹn dữ liệu. Transaction đã thực hiện xong là phải thành công hoặc nếu có lỗi xảy ra thì mọi dữ liệu trong database phải giữ nguyên ko có gì thay đổi.
         2. **Consistency (tính nhất quán)**
            1. Hiểu nôm na là khi transaction được hoàn thành, mọi dữ liệu đều phải bảo toàn các mối liên kết dù transaction thành công hay thất bại. Tức là data phải hợp lệ tại bất kì thời điểm nào.
         3. **Isolation (tính độc lập)**
            1. Nói về tính độc lập của các transaction khi thực hiện đồng thời (parallel) trên hệ thống. Nếu cùng một lúc, có nhiều transaction diễn ra đồng thời thì cần có một cơ chế đưa ra để đảm bảo các transaction này có thể hoạt động song song mà ko ảnh hưởng đến nhau.
         4. **Durability (tính bền vững)**
            1. Nôm na là phải luôn đảm bảo rằng hệ thống nếu xảy ra bất kì lỗi gì thì có thể khôi phục lại dữ liệu về nguyên trạng (ví dụ là luôn backup dữ liệu để có thể khôi phục khi cần)
7. Testing
   1. Unit test
      1. Junit
   2. JMeter
8. Mô hình phát triển phần mềm
   1. Waterfall
   2. V-model
   3. Agile/scrum
9. Tools
   1. Git
      1. .gitignore?
   2. SVN
   3. Jira
   4. Postman
10. CI-CD?
    1. AWS
11. **Certificate?**
    1. **OCP**
    2. **AWS**