Spring Framework

Spring Framework là một trong những Java EE frameworks phổ biến:

* Spring framework architecture
* Core components of Spring
* Various projects under the spring framework umbrella.

Spring Framework là gì?

Spring is one of the most popular frameworks for Java enterprise edition.

Spring Core Concepts - DI, IoC, AOP

Bạn không thể hiểu Spring Framework nếu không hiểu [Dependency Injection](https://www.digitalocean.com/community/tutorials/java-dependency-injection-design-pattern-example-tutorial) và [Inversion of Control](https://www.digitalocean.com/community/tutorials/spring-ioc-bean-example-tutorial) là gì.

<https://loda.me/articles/core-gii-thch-dependency-injection-di-v-ioc-bng-ngc-trinh>

Trước khi tìm hiểu ta nên hiểu thế nào là **tight-coupling** hay “liên kết ràng buộc”.

Là một khái niệm ám chỉ việc liên kết giữa các class quá chặt chẽ. Khi thay đổi yêu cầu thay đổi logic hoặc một class bị lỗi sẽ dẫn tới ảnh hưởng tới toàn bộ các Class khác.

**lossely-coupled:** là cách ám chỉ việc làm giảm bớt sự phụ thuộc giữa các Class với nhau.

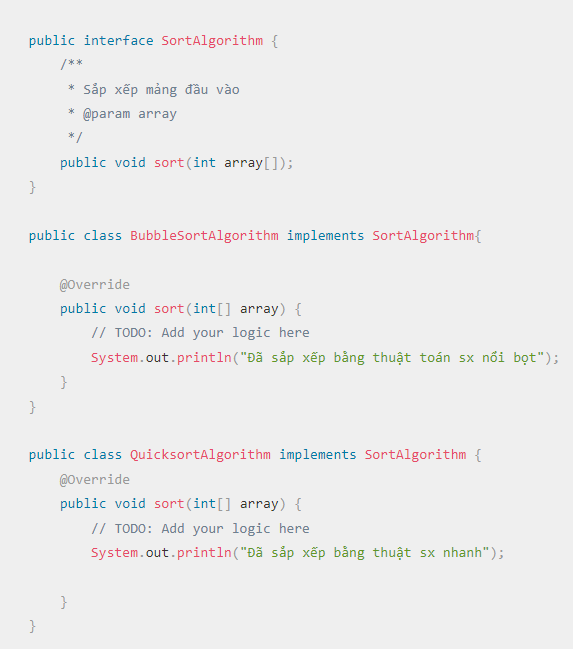


Nếu thay đổi thuật toán thành QuickSort thì ta có thể phải sửa lại hoàn toàn cả 2 Class trên.

Ngoài ra BubbleSortAlgorithm sẽ chỉ tồn tại nếu VeryComplexService tồn tại, vì VeryComplexService tạo đối tượng BubbleSortAlgorithm bên trong nó.

Theo như cách implement này, nó liên kết rất chặt với nhau.

Cách dưới đây sẽ giảm sự phụ thuộc giữa các Class:



VeryComplexService sẽ không quan tâm tới việc thuật toán sắp xếp là gì nữa, mà chỉ cần t ập trung vào nghiệp vụ.

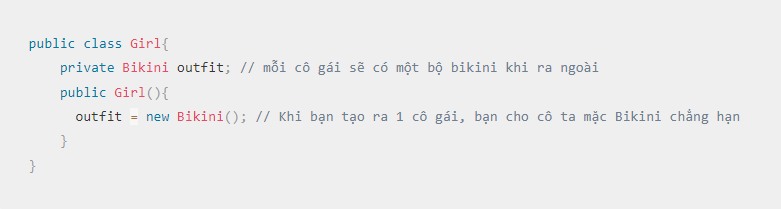
Còn SortAlgorithm sẽ được đưa vào từ bên ngoài tùy theo nhu cầu sử dụng



Dependency Injection (DI)

Dependency Injection is a design pattern, …

Hiểu nôm na là một phương pháp lập trình, một thiết kế để bạn có được hiệu quả cao hơn khi code. Sẽ giúp ích nhiều hơn cho việc lập trình của bạn.



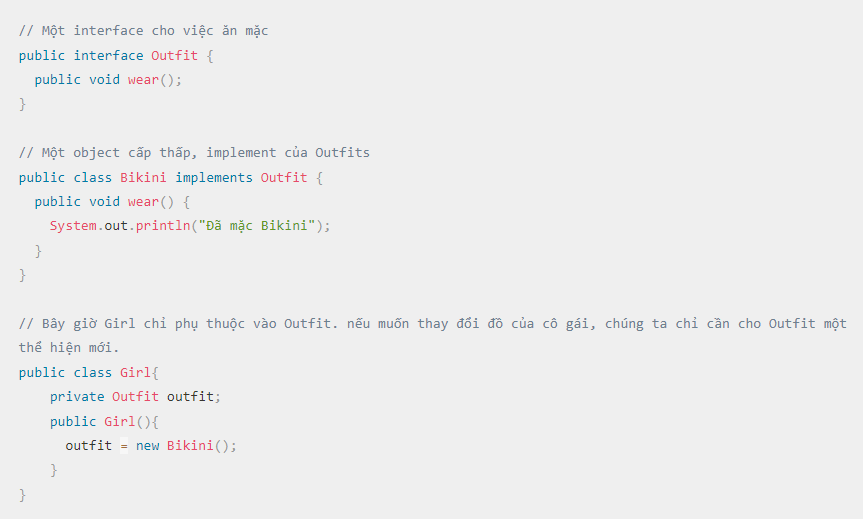
Trước hết, qua đoạn code này, bạn sẽ thấy là khi bạn tạo một Girl, bạn sẽ tạo ra thêm 1 bộ Bikini đi kèm với cô gái đó. Lúc này, Bikini tồn tại mang ý nghĩa là dependecy của Girl

Khi khởi tạo thuộc tính như này, bạn vô tình tạo ra một điểm thắt nút trong chương trình của mình. Giả sử Girl muốn mặc cái khác thì sao?. Bạn phải thay class Bikini thành SkirtWithTshirt (Váy với áo T-shirt) hay Naked (Trần như nhộng) ư?

Hay nguy hiểm hơn, bộ đồ Bikini bị hỏng => nó sẽ ảnh hưởng trực tiếp tới Girl

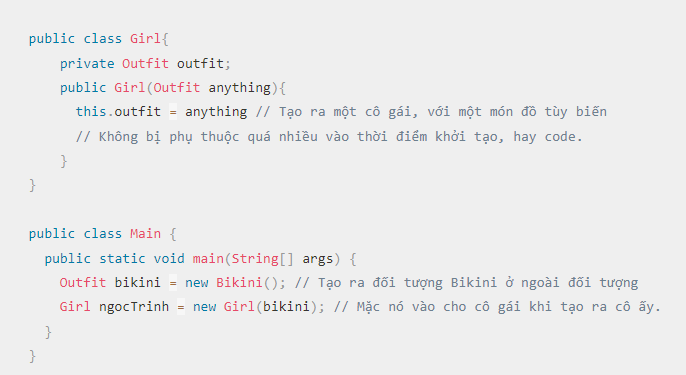
Vấn đề là ở đó: nguyên tắc là:

**Các Class không nên phụ thuộc vào các kế thừa cấp thấp, mà nên phụ thuộc vào Abstraction (lớp trừu tượng)**

Bây giờ thay đoạn code như này:

Tới đây chúng ta mới chỉ Abstract hóa thuộc tính Girl mà thôi, còn thực tế Girl vẫn đang bị gắn với một bộ Bikini duy nhất. Vậy muốn thay đồ cho cô gái phải làm như nào.

Phải sửa code thêm chút nữa:



Với đoạn code ở trên, chúng ta đã gần như tách được Bikini ra hoàn toàn khỏi Girl, điều này làm giảm sự phụ thuộc giữa Girl và Bikini. Mà tăng tính tùy biến, linh hoạt cho code.

Bây giờ Girl sẽ hoạt động với Outfit mà thôi. Và Outfit ở đâu ra? Chúng ta **tạo ra** và **Inject** vào Girl.

Khái niệm Dependency Injection từ đây mà ra:

**Dependency Injection là việc các Object nên phụ thuộc vào các Abstract Class và thể hiện chi tiết của nó sẽ được Inject vào đối tượng lúc runtime.**

Bây giờ muốn Girl mặc gì khác, bạn chỉ cần tạo một Class kế thừa Outfit và ***Inject*** (dịch là Tiêm vào cũng được) nó vào Girl là xong

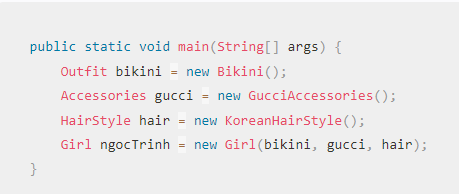
Các cách để *Inject dependency* vào một đối tượng có thể kể đến như sau:

* Constructor Injection: Cái này chính là ví dụ của mình, tiêm dependency ngay vào Contructor cho tiện.
* Setter Injection: Ồ, sao không chứ 😗 chúng ta học về Setter từ những bài học vỡ lòng rồi, quá hợp lý. Xài girl.setOutfit(new Naked()) 😈
* Interface Injection: Mỗi Class muốn inject cái gì, thì phải implement một Interface có chứa một hàm inject(xx) (Gần như thay thế cho setter ý bạn). Rồi bạn muốn inject gì đó thì gọi cái hàm inject(xx) ra. Cách này hơi dài và khó cho người mới.

Inversion of Control

Dependency Injection giúp chúng ta dễ dàng mở rộng code và giảm sự phụ thuộc giữa các dependency với nhau. Tuy nhiên lúc này khi code bạn sẽ phải kiêm thêm nhiệm vụ Inject dependency (tiêm sự phụ thuộc).

Thử tưởng tượng một Class có hàng chục dependency thì bạn sẽ phải tự tay inject từng ý cái. Việc này lại dẫn tối khó khăn trong việc code, quản lý code và dependency.

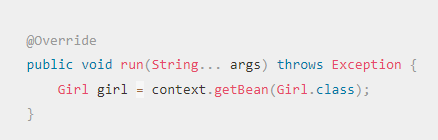


That it, chính nó, đó cũng chính là nguyên lý chính của Inversion of Control (IOC) - Đảo chiều sự điều khiển

**Inversion of Control is a programming principle. flow of control within the application is not controlled by the application itself, but rather by the underlying framework.**

Đảo chiều sự điều khiển là một nguyên tắc lập trình, luồng kiểm soát trong ứng dụng không được kiểm soát bởi chính ứng dụng, mà là bởi framework cơ bản.

Khi đó, code chúng ta sẽ chỉ cần như này, để lấy ra 1 đối tượng:



Đối với Java thì có một số Framework hỗ trợ chúng ta Inversion of Control (IOC), trong đó nổi bật có:

* Spring framework
* Google Guice

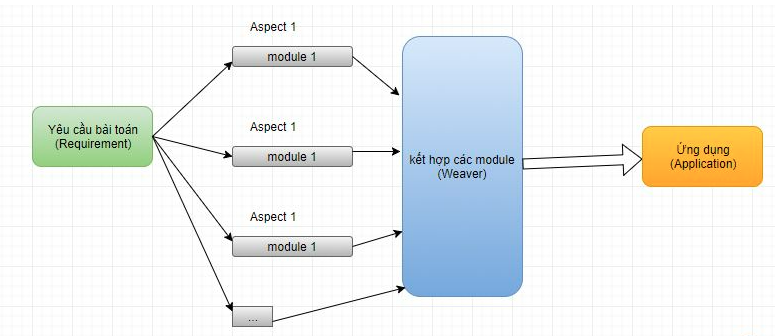
Spring framework là một framework từ những ngày đầu, ra đời để hiện thực ý tưởng *Inversion of Control (IOC)*, tuy nhiên, theo thời gian, Spring lớn mạnh và trở thành một hệ sinh thái rộng lớn phục vụ rất nhiều chức năng trên nền tảng IoC này.

**AOP là gì? Aspect Oriented Programming trong Java**

Aspect Oriented Programming (AOP) – lập trình hướng khía cạnh: là một kỹ thuật lập trình (kiểu như lập trình hướng đối tượng) nhằm phân tách chương trình thành cách moudule riêng rẽ, phân biệt, không phụ thuộc nhau.

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.

AOP không phải dùng để thay thế OOP mà để bổ sung cho OOP.



Một số khái nhiệm và thuật ngữ trong AOP với Spring Framework:

* Join point: là các điểm trong chương trình ví dụ điểm thực thi method (method execution), điểm xử lý excpetion, field access… Spring chỉ hỗ trợ method execution join point
* Advice: một hành động thực hiện ở joint point
* Pointcut: Là expression language giúp khớp nối với joint point
* Introduction: Cho phép introduce các new interface tới bất kì object adviced nào.
* Target Object: Object sẽ được adviced
* Aspect: là một class bao gồm các advice và các joint point
* Interceptor: là một aspect chỉ có duy nhất một advice
* AOP Proxy: dùng để cài đặt các Aspect

**Ví du 1:**

1 công ty được chia thành nhiều phòng ban, phòng kỹ thuật, phòng kế toán, phòng kinh doanh (mỗi phòng giống như một module trong chương trình)…

Mỗi phòng thực hiện một nhiệm vụ riêng, nhưng cần kết hợp lại để vận hành một công ty.

– Trong AOP có khái niệm “lát cắt”, “điểm cắt”, tạm hiểu là module A sẽ xen vào module B để thực hiện 1 chức năng nào đó mà không làm ảnh hưởng module B, “điểm cắt” là vị trí mà module A xen vào module B

Ví dụ, trong công ty, tất cả các nhân viên đều cần phải quyết toán thuế, nếu từng người 1 thực hiện sẽ tốn rất nhiều thời gian và ảnh hưởng tới công ty. Phòng kế toán sẽ gộp lại và thực hiện quyết toán cho tất cả nhân viên. Bây giờ luật thuế thay đổi, thì cũng chỉ cần mỗi phòng kế toán chỉnh sửa lại cho phù hợp, không ảnh hưởng gì tới các phòng ban khác. (Module phòng kế toán thực hiện lát cắt với các phòng còn lại ở nhiệm vụ quyết toán thuế)

Điểm cắt ở đây tùy vào xếp muốn, luật yêu cầu, ví dụ luật yêu cầu là phải quyết toán thuế cuối năm thì phòng kế toán sẽ thực hiện lát cắt quyết toán thuế cho các phòng ban còn lại vào cuối năm.

**Ví dụ 2:**

Trong code bạn thường thực hiện viết log cho các method (method nào chạy, lỗi gì xảy ra…). Rõ ràng việc viết log chẳng liên quan gì đến chức năng của method.

Bây giờ yêu cầu chuyển sang chỉ viết log khi bắt đầu các method chẳng hạn, ta lại phải sửa tất cả các method đó.

Bây giờ áp dụng AOP, ta sẽ phân tách chức năng log ra một module riêng và dùng nó thực hiện các lát cắt với các method cần thực hiện log.

Muốn log ở đầu method, điểm cắt sẽ là ở đầu các method, muốn cắt ở cuối method thì điểm cắt sẽ là cuối method.

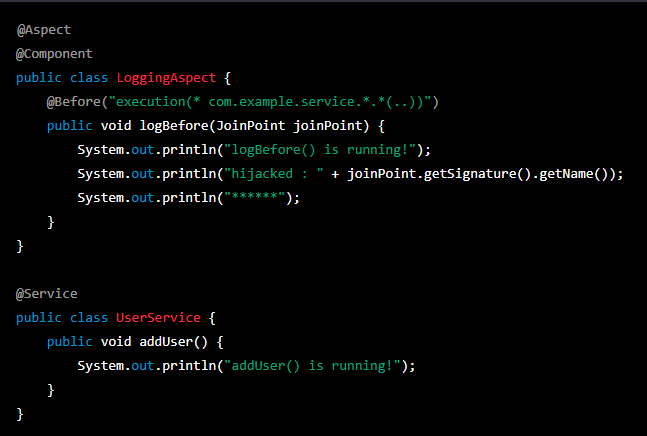
**Ví dụ 3:**

Trong lập trình web, có 1 khái nhiệm filter, tức là tất cả các request được chọn sẽ đều chạy qua filter đó.

filter ở đây cũng được coi như 1 module độc lập riêng rẽ. Chẳng hạn mỗi request gửi đến cần xác thực (kiểm tra login) ta chỉ cần cho nó đi qua 1 filter thực hiện xác thực là được, muốn sửa đổi cách xác thực thì ta chỉ cần sửa filter là xong.

Spring AOP (Aspect-Oriented Programming) là một tính năng của Spring Framework cho phép bạn thêm các cross-cutting concerns (như bảo mật, ghi nhận, v.v.) vào các lớp bằng cách sử dụng các "aspects" (đối tượng đặc biệt).

Đây là một ví dụ về cách sử dụng Spring AOP trong Java:



Trong ví dụ này, chúng ta đã khai báo một aspect LoggingAspect với một "advice" (hành động sẽ được thực hiện khi một "pointcut" (điểm cắt) được thực hiện). Trong trường hợp này, chúng ta muốn in ra một thông báo trước khi bất kỳ phương thức nào trong lớp UserService được gọi.

Sau khi khai báo aspect, bạn cần cấu hình Spring AOP bằng cách thêm @EnableAspectJAutoProxy vào lớp cấu hình của bạn. Sau đó, bạn có thể sử dụng nó trong các lớp khác bằng cách khai báo @Autowired và gọi các phương thức của nó.

Cross-cutting concerns là những thứ liên quan đến nhiều lớp khác nhau trong một hệ thống hoặc ứng dụng. Ví dụ, bảo mật, ghi nhận, và quản lý transactions là cross-cutting concerns thường xuyên xuất hiện trong các hệ thống phần mềm.

Trong OOP, cross-cutting concerns thường khó khăn để xử lý vì chúng thường không thuộc về một lớp nào cả. Tuy nhiên, với Spring AOP, bạn có thể sử dụng các "aspects" để xử lý các cross-cutting concerns này theo cách dễ dàng hơn.

**Spring IoC, Spring Bean Example Tutorial**

Spring Framework được xây dựng trên nguyên tắc Inversion of Control. [Dependency injection](https://www.digitalocean.com/community/tutorials/java-dependency-injection-design-pattern-example-tutorial) là kỹ thuật triển khai IoC trong các ứng dụng.

**Spring IoC Container**

Spring IoC (Inversion of Control) là một tính năng của Spring Framework cho phép bạn khai báo các "bean" (đối tượng) trong một file cấu hình, và Spring sẽ quản lý việc tạo ra và quản lý các bean này cho bạn. Điều này giúp cho việc phát triển ứng dụng của bạn trở nên dễ dàng hơn vì bạn không cần phải quản lý việc khởi tạo và quản lý các đối tượng trong ứng dụng của mình.

Spring IoC Container sử dụng các "bean definition" (định nghĩa bean) để biết cách tạo ra và quản lý các bean. Bean definition có thể được khai báo trong một file cấu hình XML hoặc bằng cách sử dụng annotation trong lớp Java.

Đây là một ví dụ về cách sử dụng Spring IoC Container trong Java:



Trong ví dụ này, chúng ta đã khai báo một lớp cấu hình **AppConfig** với một "bean" là **UserService**. Sau đó, chúng ta sử dụng **ApplicationContext** để tạo ra một đối tượng **UserService** và gọi phương thức của nó. **ApplicationContext** là một interface đại diện cho Spring IoC Container.

**1. Bean và ApplicationContext là gì?**

Trong documentation của Spring framework, thì bean được định nghĩa như sau:

Trong spring, các đối tượng tạo thành xương sống cho ứng dụng của bạn và được quản lý bởi Spring IoC container được gọi là beans. Một Bean là một đối tượng được khởi tạo, lắp ráp và quản lý bởi Spring IoC container.

*In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.*

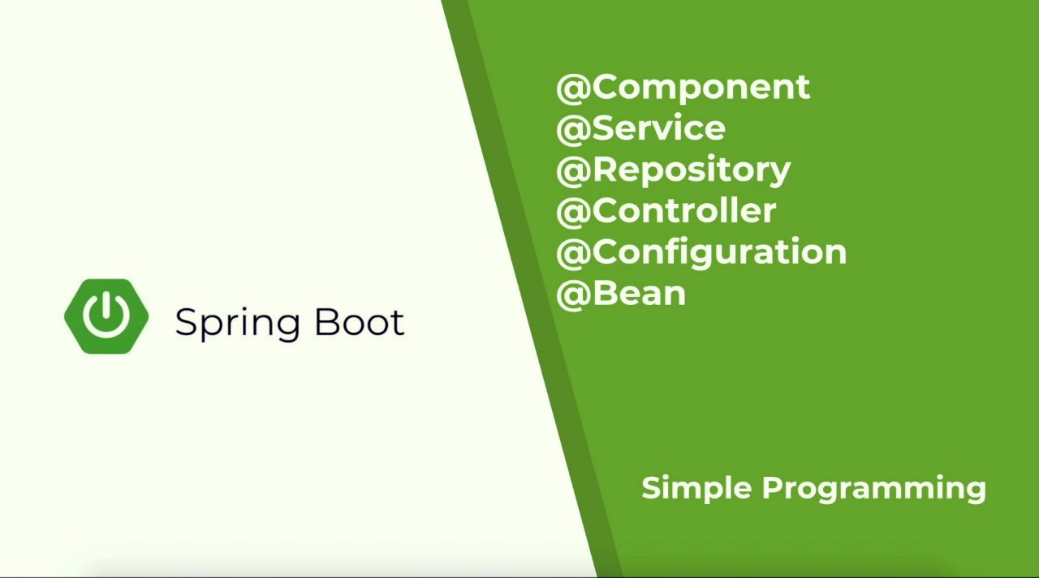
Nói một cách đơn giản, bean là những module chính của chương trình, được tạo ra và quản lý bởi Spring IoC container.

Các bean có thể phụ thuộc lẫn nhau, như ví dụ về **Car**, **Engin** và **ChinaEngine**. Sự phục thuộc này được mô tả cho IoC biết nhờ cơ chế Dependency injection.

Chỉ cần biết đơn giản nhất là dùng @Component lên class là class đó là một bean.

**1.2. ApplicationContext là gì?**

ApplicationContext là khái niệm Spring Boot dùng để chỉ Spring IoC container, tương tự như bean là đại diện cho các dependency.

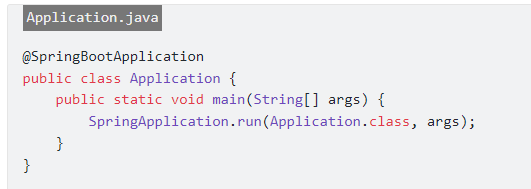
Ngoài ra bạn có thể sẽ nghe nói về BeanFactory. Nó cũng đại loại như ApplicationContext, đại diện cho Spring IoC container nhưng ở mức cơ bản. ApplicationContext thì ở mức cao hơn, cung cấp nhiều tính năng hơn BeanFactory như i18n, resolving messages, publishing events,...

Khi ứng dụng Spring chạy, Spring IoC container sẽ quét toàn bộ packages, tìm ra các bean và đưa vào ApplicationContext. Cơ chế đó là Component scan.

**1.3. Cách lấy bean ra từ Context**

Tất nhiên trước khi lấy bean ra từ context thì phải có context rồi. 🙁 Câu hỏi đặt ra là biến context ở đâu?

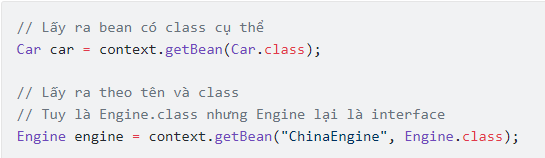
Đó là ngay dòng bắt đầu chương trình Spring Boot.



Dòng method SpringApplication.run() sẽ return về một object ApplicationContext interface, đại diện cho IoC container

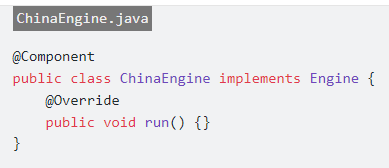
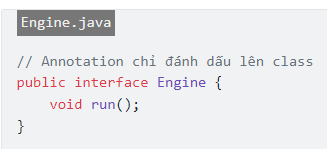
https://lh4.googleusercontent.com/PGm3q3ZdpvrNLGa2u3X2TdU5QGxCKfKQlHO-rb9qJEe_56w60_gVV8CNx-XLfrMswfpgspIfvZbpm2nlFuJweIFagWqPHWApAQPXOtBeBa6MMazV0AMeE43piljFOn6Dl1ehm2qJO34C7S5YtUol0A

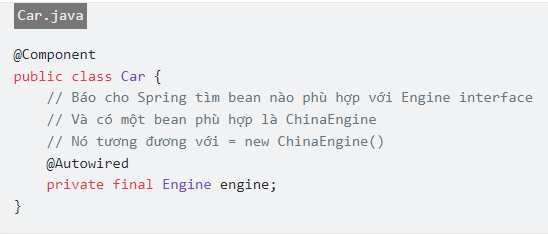
Chúng ta có thể lấy ra bean từ đây, dùng method getBean().



**2. Kĩ thuật inject bean vào bean khác**

Ví dụ bạn có hai bean là Car  và Engine. Và Car  thì phụ thuộc vào Engine, do đó theo Dependency injection thì chúng ta cần Engine  inject vào trong Car.

**2.1. Sử dụng @Autowired**

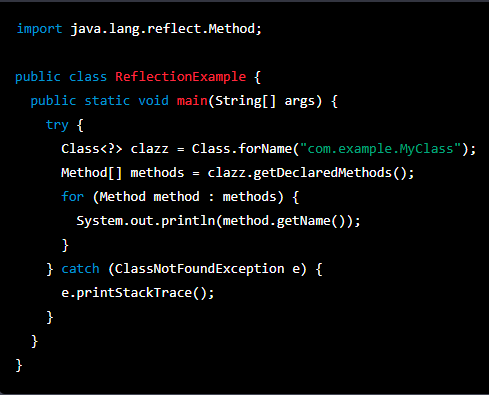


Cách dùng @Autowired trên field là không được khuyến khích, do nó sử dụng [*Java reflection*](https://loda.me/articles/huong-dan-java-reflection) (ấn vào để tìm hiểu thêm) để inject. Chúng ta nên cân nhắc đổi qua dùng inject theo kiểu constructor hoặc setter.

*Java Reflection là một tính năng trong ngôn ngữ lập trình Java cho phép mã khám phá và điều khiển các thuộc tính của một lớp, giao diện, hàm tạo hoặc phương thức tại thời điểm chạy, mà không biết tên của các thành viên mà nó đang điều khiển tại thời điểm biên dịch.*

*Ví dụ, bạn có thể sử dụng Java Reflection để lấy danh sách tất cả các phương thức của một lớp, đặt tên của một trường hoặc gọi một phương thức tại thời điểm chạy, mà không biết tên của các phương thức hoặc trường này tại thời điểm biên dịch.*

*Reflection có thể là một công cụ mạnh mẽ, nhưng nó cũng cần được sử dụng cẩn thận, vì nó có thể làm cho mã khó hiểu hơn và khó bảo trì hơn, và cũng có thể gây ảnh hưởng xấu đến hiệu suất.*



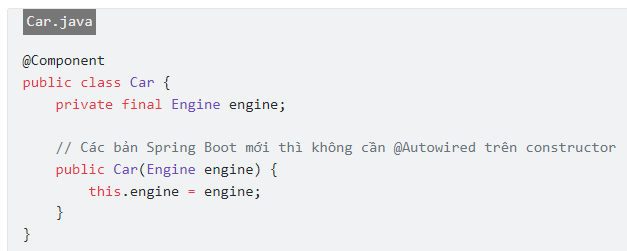
Trong ví dụ này, chúng ta sử dụng hàm Class.forName() để tải một lớp từ tên của nó, sau đó sử dụng phương thức getDeclaredMethods() của lớp Class để lấy danh sách các phương thức đã định nghĩa trong lớp đó. Cuối cùng, chúng ta in tên của từng phương thức ra màn hình.

Bạn có thể tham khảo tài liệu Java Reflection để biết thêm chi tiết về cách sử dụng tính năng này trong Java.

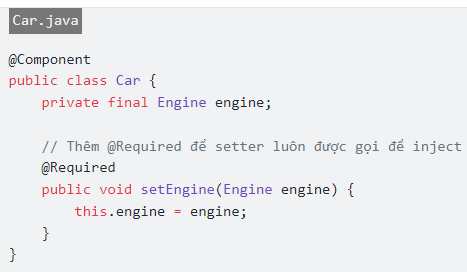
**2.2. Inject qua constructor hoặc setter**

Code inject theo kiểu constructor-based nên dùng khi các module là bắt buộc. Khi đó Spring Boot khi tạo bean (cũng chỉ là tạo object, gọi constructor thì sẽ đưa các phụ thuộc vào constructor khi gọi).

Ví dụ class Car đã được sửa lại để inject Engine vào qua constructor.



Hoặc dùng kiểu setter-based như sau. Spring Boot sau khi tạo xong bean Car sẽ gọi thêm method setEngine() sau đó.



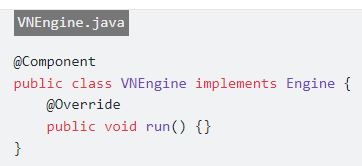
Cách dùng setter để inject thường dùng trong trường hợp phụ thuộc vòng, module A phụ thuộc vào B và ngược lại. Do đó, nếu cả hai đều sử dụng constructor based injection thì Spring Boot sẽ không biết nên tạo bean nào trước. Vì thế, giải pháp là một bean sẽ dùng constructor, một bean dùng setter như trên.

<https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-dependencies>

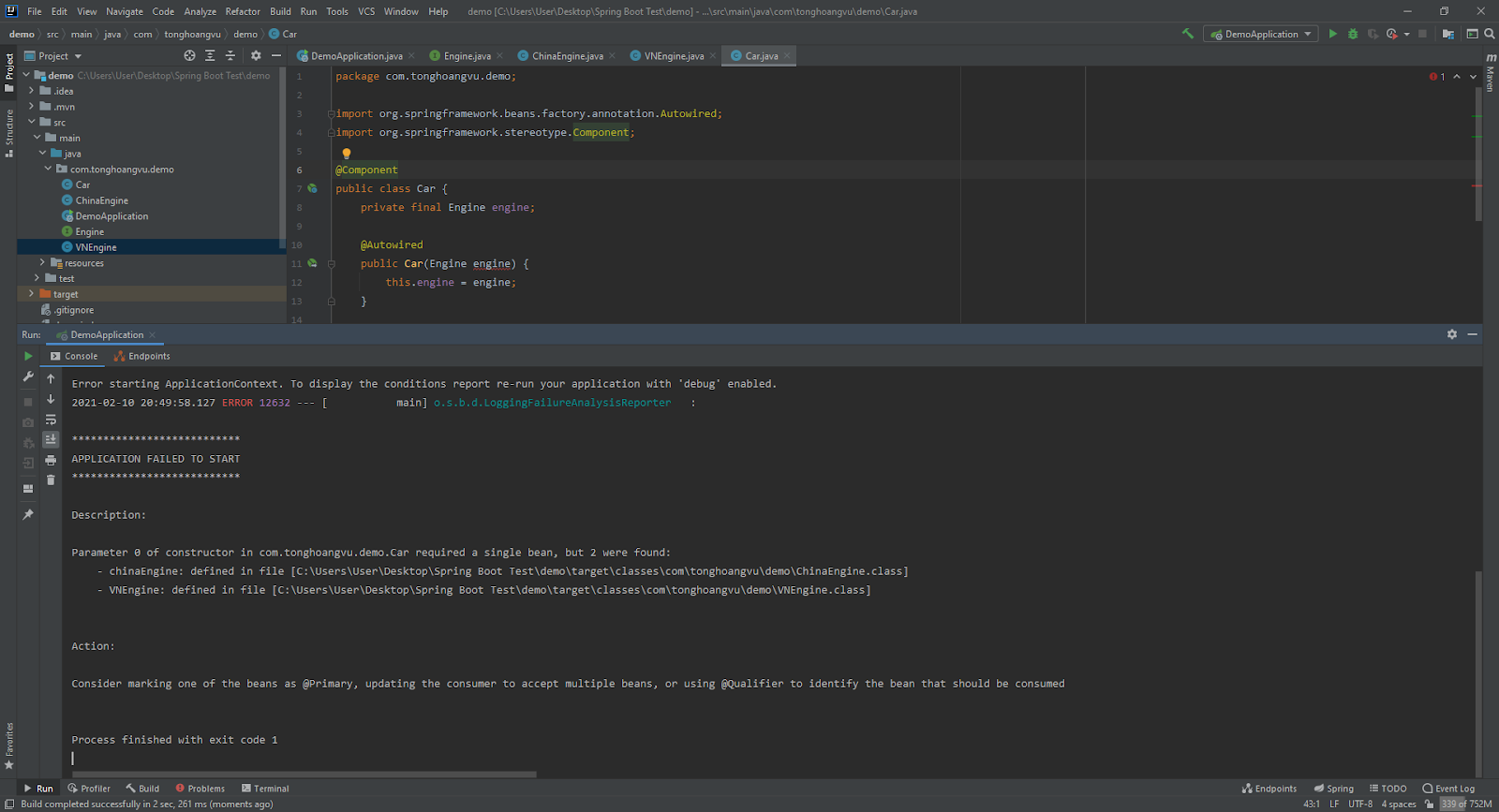
**3. Khi Spring Boot không biết chọn bean nào?**

**3.1. Khi tìm thấy nhiều bean phù hợp**

Cũng lấy ví dụ trên, nếu chúng ta tạo thêm class VNEngine có chức năng tương tự ChinaEngine.



Thì Spring Boot sẽ báo lỗi như sau (báo khi chạy và cả trong IDE nữa.



Có thể hiểu do Spring Boot đã tìm thấy hai bean phù hợp để inject vào Car. Do cả hai VNEngine và ChinaEngine đều implements Engine, mà Car cần Engine nên không biết nên chọn cái nào.

**3.2. Giải pháp**

Có hai cách giải quyết vấn đề này. Thứ nhất là dùng @Primary đánh dấu lên một bean. Khi đó bean này sẽ được ưu tiên chọn hơn, trong trường hợp có nhiều bean phù hợp trong context.



Cách 2 là chỉ định rõ tên bean (tên class) cụ thể được inject bằng @Qualifier.



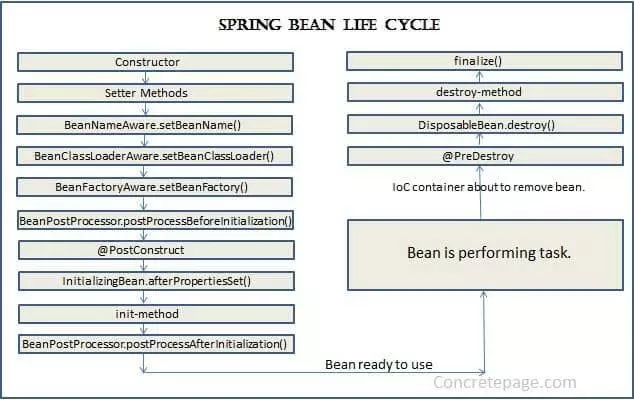
Đối với constructor hay setter based cũng tương tự, chỉ cần có @Qualifier trước tên field cần inject vào là được.

**Vòng đời, các loại bean và cơ chế Component scan**

**1. Vòng đời của bean**

**1.1. Bean life cycle**

Vòng đời (life cycle) của bean được hiểu là từ khi bean được tạo ra cho tới khi chết đi, sẽ có những sự kiện (event) khác nhau xảy ra. Về vòng đời của bean có thể mô tả bởi sơ đồ sau.



Nhìn có vẻ dài và khó hiểu, nhưng 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 được gọi
* Init method được gọi

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

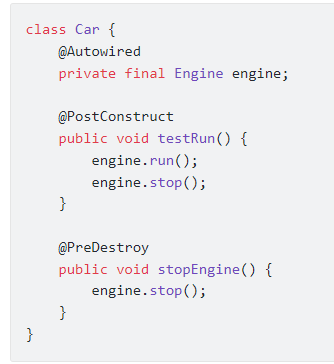
* Gọi @PreDestroy
* Hủy bean như các object thông thường

**1.2. @PostConstructor và @PreDestroy**

Đây là hai event khá quan trọng với bean, bạn có thể hook một method vào đó để thực thi khi event xảy ra:

* @PostConstruct là sau khi bean đã khởi tạo xong
* @PreDestroy là trước khi bean bị phá hủy

Chúng ta dùng hai annotation trên đánh dấu lên method nào đó, method đó sẽ được tự động gọi khi sự kiện bean xảy ra.



Như code ví dụ trên, mình gắn @PostConstruct cho method testRun(). Method này được gọi khi bean Car được tạo ra và khởi tạo hoàn chỉnh. Và trước khi Car bị phá hủy, thì cần gọi stopEngine tương tự như trên.

Dùng trong thực tế thì hai annotation trên làm các nhiệm vụ như:

* @PostConstruct dùng để thực hiện một số task khi khởi tạo bean
* @PreDestroy thực hiện các task để dọn dẹp bean sau khi dùng xong

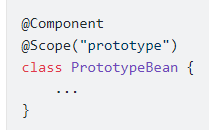
**2. Các loại bean**

Nói đúng hơn thì gọi là các scope, phân loại dựa trên số lượng bean được tạo ra. Bean gồm có 5 scope:

* Singleton (mặc định): IoC container chỉ tạo đúng duy nhất 1 object từ class bean này
* Prototype: return một bean object riêng biệt cho mỗi lần sử dụng.
* Request: tạo mỗi bean cho mỗi request
* Session: tạo mỗi bean cho mỗi session
* Global session: tạo mỗi bean cho mỗi global session (cái này không hiểu lắm.

Trong 5 scope trên chúng ta chỉ quan tâm tới hai scope đầu. Thường thì các bạn sẽ ít đụng tới prototype bean, nhưng mình cũng viết ra tại đây luôn.

Đối với singleton bean thì khỏi cần đánh dấu gì hết, nó là mặc định rồi. Còn nếu muốn chỉ định một class là prototype bean thì dùng @Scope như sau.



Nói rõ hơn về prototype bean, ví dụ bean X được sử dụng bởi hai bean khác là A, B:

* Nếu X là singleton bean, thì chỉ có một object X được tạo ra. A và B dùng chung X.
* Nếu X là prototype bean, thì có 2 X được tạo ra cho 2 bean khác sử dụng là X cho A và X cho B.

**3. Cách định nghĩa bean**

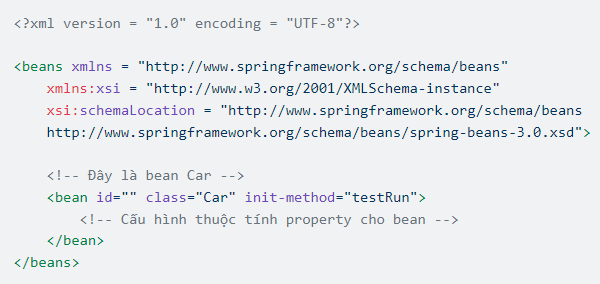
Có 3 cách định nghĩa class là một bean:

* Khai báo trong file XML
* Dùng annotation trên class
* Dùng @Configuration và @Bean

Tùy từng trường hợp cụ thể mà dùng cho phù hợp. Ví dụ trong series này mình không bàn sâu về cấu hình bean bằng XML (do Spring Boot sinh ra không phải để cấu hình.

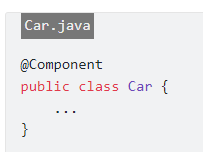
**3.1. Dùng XML, annotations**

Nhưng đơn giản, các bạn cũng nên biết trước đây Spring dùng XML để cấu hình các bean như sau. Nó khá là cực nên người ta dùng cách khác hay hơn.



**3.2. Dùng Component annotations**

Do đó người ta mới dùng cách cấu hình bean dựa trên các annotation như @Component. Cụ thể thì như bài trước có nói, chỉ cần đánh dấu @Component lên trên class thì IoC sẽ biết và tạo bean từ class đó.



Ngoài ra có các annotations khác cụ thể hơn như @Service, @Repository, @Controller,... cũng bao gồm @Component, nên tác dụng của chúng cũng là tạo bean.

**3.3. Dùng @Bean bên trong @Configuration**

Cách này dùng cho trường hợp bean cần thực hiện nhiều thao tác phức tạp để khởi tạo, hoặc có nhiều bean liên quan với nhau. Do đó, thay vì khởi tạo riêng rẽ từng class là từng bean, thì gom chung các bean cần khởi tạo lại bỏ vào class chứa là @Configuration.

Khi Spring tìm thấy class @Configuration, nó sẽ tạo bean của class này trước (do @Configuration cũng là @Component). Trong khi tạo thì các logic khởi tạo cũng được thực thi, để chuẩn bị sẵn sàng tạo các @Bean bên trong.

Sau đó Spring Boot sẽ tìm các method được đánh dấu @Bean bên trong @Configuration để tạo bean. Thường thì các bean dạng này ngắn và return ngay object chứ không phải để Spring Boot tạo ra.

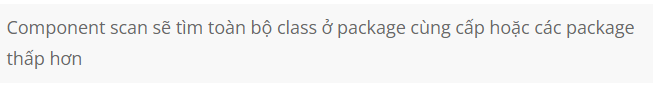
Các bean cũng được đưa vào ApplicationContext như bình thường.

Tuy nhiên, không phải class nào đánh dấu cũng được tạo bean. Mà phải có điều kiện quá trình component scan của IoC phải tìm thấy nó. Chúng ta sẽ đi tiếp về component scan ngay sau đây.

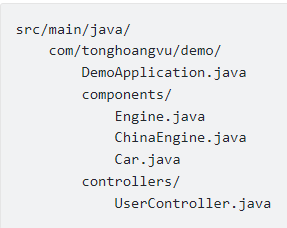
**4. Component scan**

**4.1. Cách component scan hoạt động**

Khi ứng dụng Spring Boot bắt đầu chạy, thì nó sẽ tìm hết các class đánh dấu là bean trong chương trình và tạo bean. Quá trình tìm kiếm các bean này gọi là component scan.



Do đó, class đánh dấu @SpringBootApplication có chứa main method sẽ là nơi bắt đầu. Spring Boot sẽ tìm từ package này (package gốc) tìm xuống để tạo các bean.



Do cấu trúc thư mục mặc định của Spring Boot nó thế, nên từ package gốc có DemoApplication.java là com.tonghoangvu.demo, nó sẽ tìm:

* Các class cùng cấp, tìm được DemoApplication class, tạo bean
* Tìm xuống các package thấp hơn như com.tonghoangvu.demo.components và com.tonghoangvu.demo.controllers, tìm thêm được các class như ChinaEngine, Car, UserController (Engine.java là interface nhé).

Do đó, mặc định mọi class được khai báo là bean đều có thể được tìm được.

**4.2. Tùy chỉnh package tìm kiếm**

Trong trường hợp bạn chỉ muốn Spring Boot tìm các bean trong một package cụ thể, ví dụ chỉ tìm trong thư mục components thì có 2 cách như sau.



**Spring Boot - Xây dựng ứng dụng Reactive với Spring 5 Webflux**

**Tổng quan**

Spring Webflux Framework là một phần của Spring 5 và cung cấp [Reactive Programming] nhằm hỗ trợ cho việc xây dựng ứng dụng web.

**Reactive Streams API**

được tạo bởi các kỹ sư từ Netflix, Pivotal, Lightbend, RedHat, Twitter, and Oracle và bây giờ là một phần của Java 9. Nó định nghĩa 4 interface:

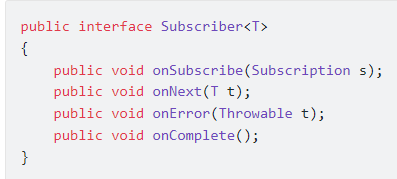
Publisher: Phát ra một chuỗi các sự kiện đến subscriber theo yêu cầu của người mà subscriber đến nó. Một Publisher có thể phục vụ nhiều subscriber. Interface này chỉ có một phương thức:

*Publisher.java*



Subscriber: Nhận và xử lý sự kiện được phát ra bởi Publisher. Chú ý rằng không có gì xảy ra cho tới khi Subscription – nó được gọi là báo hiệu yêu cầu cho Publisher.

*Subscriber.java*

**

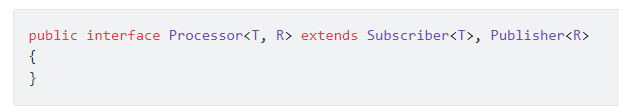
Subscription: Định nghĩa mỗi quan hệ 1-1 giữa Publisher và Subscriber. Nó chỉ có thể được sử dụng bởi một Subsriber duy nhất và được sử dụng để báo hiệu yêu cầu (request) hoặc hủy (cancel) data.

*Subscription.java*

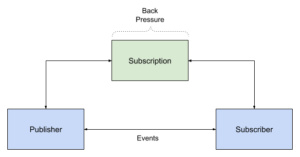
**

Processor: Đại diện cho giai đoạn xử lý gồm cả Publisher và Subscriber đồng thời tuân thủ nguyên tắc của cả 2.

*Processor.java*

**

Bản chất, một Subscriber tạo một Subscription tới Publisher, sau đó Publisher gửi một sự kiện cho Subsriber với một luồng các phần tử.



**Spring WebFlux**

**Spring Webflux** là một phiên bản song song với **Spring MVC** và hỗ trợ non-blocking reactive streams. Nó hỗ trợ khái niệm **back pressure** và sử dụng Server Netty để run hệ thống reactive. Nếu bạn đã quen thuộc với style **Spring MVC** thì bạn cũng dễ dàng làm việc với **Spring Webflux.**

Spring webflux sử dụng project reactor ( Thư viện implements phỗ biến nhất) như một thư viện reactive, vì thế nó cung cấp 2 kiểu Publisher:

Mono: Phát ra 0 hoặc 1 phần tử.

Flux : Phát ra 0..N phần tử.

**Dependencies**

Dươi đây là dependency của Webflux, nó đã kéo theo các dependencies khác gồm:

1. spring-boot và spring boot-starter
2. spring-webflux framework
3. reactor-core

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-webflux</artifactId>

    <version>2.0.5.RELEASE</version>

</dependency>

**Demo String Boot WebFlux**

Chúng tôi sẽ xây dựng một ứng dụng đơn giản sử dụng Spring Webflux.

* Để đơn giản chúng tôi tạo một đối tượng đơn giản là Employee với 2 thuộc tính đơn giản
* Tạo một Rest APIS cho việc query danh sách Employees sử dụng [@RestController](https://viblo.asia/u/RestController).
* Cuối cùng là tạo 1 DB đơn giản và hỗ trợ Reactive bằng việc trả về kiểu Flux mà Webflux cung cấp. Các bạn có thể sử dụng DB lữu trữ khác hỗ trợ Reative như là **MongoDB**.

*pom.xml*

<dependencies>

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-webflux</artifactId>

        <version>2.0.5.RELEASE</version>

    </dependency>

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-test</artifactId>

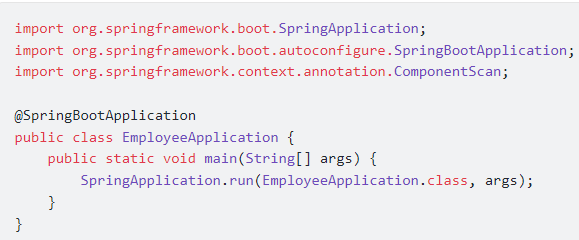
        <version>2.0.5.RELEASE</version>

        <scope>test</scope>

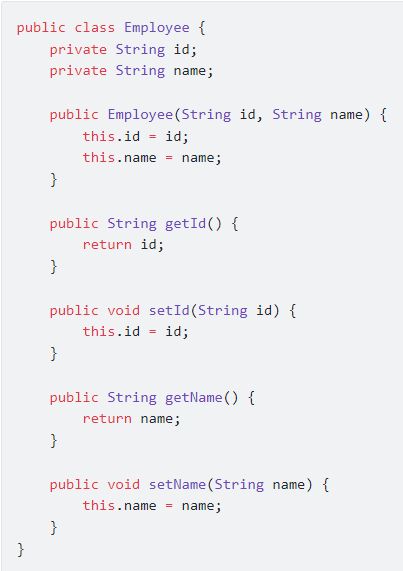
    </dependency>

</dependencies>

*EmployeeApplication.class*



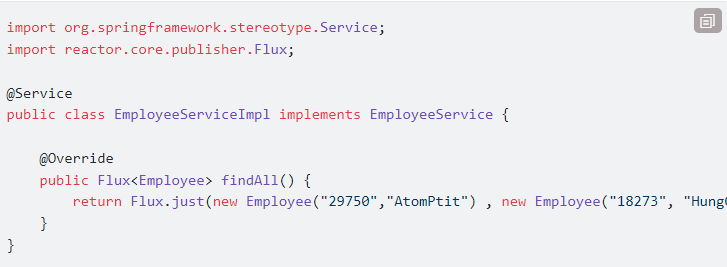
*Employee.class*



*EmployeeService.class*



*EmployeeServiceImpl.class*



*EmployeeController.class*



OUTPUT:

[

    {

        "id": "29750",

        "name": "AtomPtit"

    },

    {

        "id": "18273",

        "name": "HungCD"

    }

]

**Kết luận**

Cả Spring MVC và Spring Webflux để hỗ trợ kiến trúc Client-Server nhưng điểm khác nhau chính là mô hình concurrency và hành động mặc định trong tính chất non-blocking và threads.

Trong Spring MVC, nó mặc định rằng ứng dụng có thể bị block tại thread hiện tại, trong khi webflux thì mặc định threads là non-blocking .

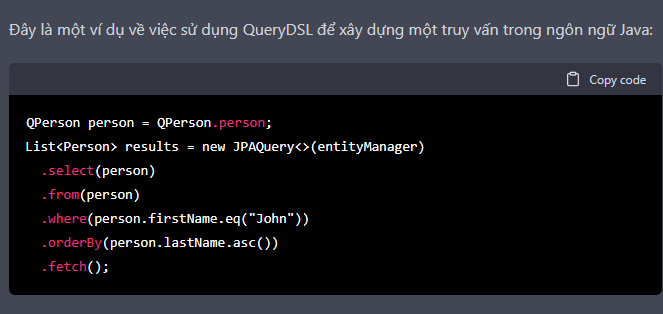
Reactive và non-blocking nhìn chung thì không làm cho ứng dụng chạy nhanh hơn. Lợi ích mà nó được kỳ vọng là mở rộng ứng dụng với số luồng nhỏ và yêu cầu ít bộ nhớ hơn. Nó làm cho các ứng dụng trở nên linh hoạt hơn khi tải.

**Spring Data JPA**

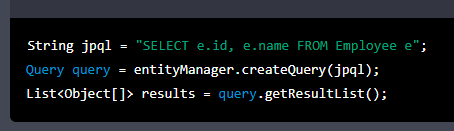
**Spring Boot JPA** là một phần trong hệ sinh thái Spring Data, nó tạo ra một layer ở giữa tầng service và database, giúp chúng ta thao tác với database một cách dễ dàng hơn, tự động config và giảm thiểu code thừa thãi.

**Spring Boot JPA** đã wrapper Hibernate và tạo ra một interface mạnh mẽ. Nếu như bạn gặp khó khăn khi làm việc với Hibernate thì đừng lo, bạn hãy để **Spring JPA** làm hộ.

Support QueryDSL and JPA queries



JPA queries: JPQL (Java Persistence Query Language)



**Cài đặt**

<https://loda.me/articles/sb11-huong-dan-spring-boot-jpa-mysql>

**Tạo Model User**

Khi đã có dữ liệu trong Database. Chúng ta sẽ tạo một Class trong Java để mapping thông tin.

*User.java*



**Vấn đề của Hibernate truyền thống**

Thông thường, khi bạn đã định nghĩa Entity tương ứng với Table trong DB thông qua Hibernate. Thì nhiệm vụ tiếp theo sẽ là tạo ra các class thao tác với DB.

Ví dụ muốn query lấy tất cả User bằng Hibernate truyền thống sẽ như sau:



Mặc dù Hibernate đã làm rất tốt và giảm thiểu code cho việc thao tác với Database xuống rồi, những nó vẫn chưa hẳn là dễ dàng :(

Mục đích ban đầu của Hibernate là giúp người lập trình dễ sử dụng, tuy nhiên, trên thực tế, nhiều người gặp khó khăn trong việc sử dụng với Hibernate hơn cả jdbc.

Nắm được vấn đề này, **Spring Data** đã wrapper lên Hibernate một lớp nữa gọi là **Spring JPA**, giúp cho mọi thao tác với DB của chúng ta rút ngắn xuống còn 1 dòng và tất nhiên là làm mờ Hibernate xuống đáng kể để tránh rắc rối cho người lập trình.

**JpaRepository**

Để sử dụng Spring JPA, bạn cần sử dụng interface JpaRepository.

Yêu cầu của interface này đó là bạn phải cung cấp 2 thông tin:

1. Entity (Đối tượng tương ứng với Table trong DB)
2. Kiểu dữ liệu của khóa chính (primary key)

Ví dụ: Tôi muốn lấy thông tin của bảng User thì làm như sau:



Vậy thôi, @Repository đánh dấu UserRepository là một Bean và chịu trách nhiệm giao tiếp với DB.

Spring Boot sẽ tự tìm thấy và khởi tạo ra đối tượng UserRepository trong Context. Việc tạo ra UserRepository hoàn toàn tự động và tự config, vì chúng ta đã kế thừa JpaRepository.

Bây giờ, việc lấy ra toàn bộ User sẽ như sau:



Đơn giản và ngắn gọn hơn rất nhiều.

Nếu bạn tìm kiếm thì sẽ thấy UserRepository có hàng chục method mà chúng ta không cần viết lại nữa. Vì nó kế thừa JpaRepository rồi.

Spring Annotations

Một số spring core framework [annotations](https://www.digitalocean.com/community/tutorials/java-annotations) là:

1. **@Configuration**: một annotation trong Spring Framework, nó cho biết rằng một class là một class cấu hình. Các class cấu hình chứa các bean definition và có thể sử dụng các annotation khác như @Bean để tạo các bean. Các class cấu hình cũng có thể sử dụng các annotation khác như @Import để nhập các class cấu hình khác hoặc @Value để lấy giá trị từ file cấu hình.
2. **@Bean** là một annotation trong Spring Framework, nó cho biết rằng một method sẽ tạo ra một bean để quản lý bởi Spring IoC container. Khi một method được annotated với @Bean, Spring sẽ gọi method đó khi cần tạo ra bean và sẽ quản lý lifecycle của bean đó. Các tham số của method sẽ được tiêm vào từ các bean khác mà Spring quản lý. @Bean annotation có thể được sử dụng trong class cấu hình (@Configuration) hoặc trong class bình thường.
3. **@PreDestroy** và **@PostConstruct**

* @PreDestroy cho biết rằng một method sẽ được gọi trước khi một bean được hủy bởi Spring IoC container.
* @PostConstruct cho biết rằng một method sẽ được gọi sau khi một bean được tạo và các thuộc tính được tiêm vào.

1. **@ComponentScan** nó cho phép Spring tìm kiếm các bean trong package và các package con của package được chỉ định. Nó sử dụng để tìm các class có annotation như ***@Component, @Service, @Repository,*** và ***@Controller***. Khi một class được tìm thấy, nó sẽ được tạo thành một bean và quản lý bởi Spring IoC container.
2. **@Component** là cho biết rằng một class là một bean cần được quản lý bởi Spring IoC container. Annotation này thường được sử dụng cho các class tiêu chuẩn của ứng dụng, ví dụ như service và repository.

@Component là một annotation chung, có thể sử dụng cho các class khác nhau, nhưng có các annotation khác như @Service, @Repository, và @Controller, cho phép bạn chỉ rõ ràng hơn về mục đích của class.

1. **@PropertySource** nó cho phép bạn nạp một hoặc nhiều file properties vào Spring Environment. Nó cho phép bạn sử dụng các giá trị trong file properties đó trong các bean cấu hình bằng cách sử dụng @Value annotation hoặc để sử dụng trong các bean khác.
2. **@Service** một class là một service bean thường được sử dụng cho các class chứa các logic xử lý nghiệp vụ của ứng dụng, ví dụ như lấy dữ liệu từ database hoặc gọi api.
3. **@Repository** là một repository bean thường được sử dụng cho các class chứa các logic truy vấn dữ liệu từ database
4. **@Autowired** cho phép Spring tự động tiêm (inject) một thành phần cần thiết vào một bean khác. Nó có thể được sử dụng trên các thuộc tính (property), phương thức (method) hoặc constructor.

Nếu Spring không tìm thấy bean phù hợp để tiêm vào thuộc tính, phương thức hoặc constructor, nó sẽ ném một NoSuchBeanDefinitionException.

Lưu ý: @Autowired có thể được sử dụng với các annotation khác như @Resource, @Inject để tiêm thành phần vào bean.