

Spring framework

● Spring boot

- Là một framework của Java nhằm phát triển một hệ quản lý object cũng như triển khai các mô hình website

Table of contents

[1. Kiến trúc \(#1-ki%E1%BA%BFn-tr%C3%BAc\)](#)

[2. IOC container \(#2-ioc-container\)](#)

[3. Beans \(#3-beans\)](#)

[3.1. Định nghĩa \(#31-%C4%91%E1%BB%8Bnh-ngh%C4%A9a\)](#)

[3.2. Vòng đời của beans \(#32-v%C3%B2ng-%C4%91%E1%BB%9Di-c%E1%BB%A7a-beans\)](#)

[3.3. Spring Beans Scope \(#33-spring-beans-scope\)](#)

[4. Profiles \(#4-profiles\)](#)

[5. Autowired \(#5-autowired\)](#)

[6. JDBC \(#6-jdbc\)](#)

[7. Quản lý giao dịch \(#7-qu%E1%BA%A3n-l%C3%BD-giao-d%E1%BB%8Bch\)](#)

[8. Spring với rest api \(#8-spring-v%E1%BB%9Bi-rest-api\)](#)

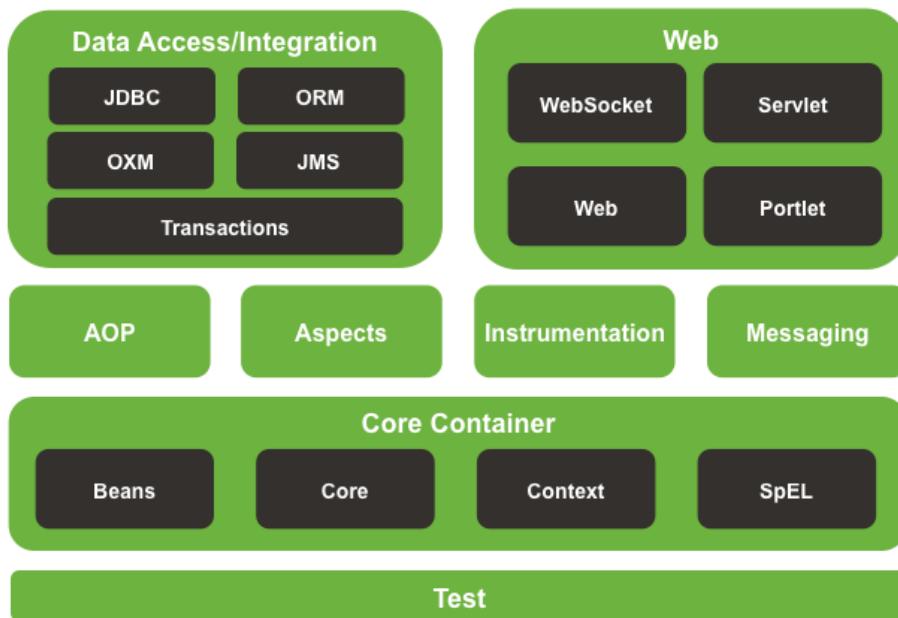
[9. Spring MVC \(#10-spring-mvc\)](#)

1. KIẾN TRÚC

- Spring framework có kiến trúc được mô tả trong hình bên dưới, với thành phần cốt lõi là Container Core.



Spring Framework Runtime



Hình 1: Kiến trúc của Spring framework

Trong đó, kiến trúc của Spring framework bao gồm:

- **Core container:** bao gồm các module spring-core, spring beans, spring context, springcontext-support và spring-expression:
 - Module spring-core và spring-beans cung cấp những phần cơ bản cho framework bao gồm: Dependency Injection và IoC
 - Module spring-context xây dựng trên một nền tảng vững chắc được cung cấp bởi spring-core và spring-beans, được sử dụng để truy cập vào các đối tượng trong framework-style tương tự như việc đăng ký một JNDI. spring-context-support giúp cho việc tích hợp ứng dụng bên thứ ba vào trong ứng dụng như caching, mailing, schedule, ...
 - Spring expression cung cấp một Expression Language mạnh mẽ cho việc truy cập và tính toán đồ thị đối tượng trong runtime được kế thừa từ unified expression language. Ngôn ngữ này hỗ trợ setting và getting giá trị các property, gọi phương thức, truy cập vào nội dung của mảng, tập hợp và chỉ mục, toán tử logic và tính toán, đặt tên biến, và truy xuất các đối tượng theo tên từ IoC Container của Spring.
- **Data Access/Integration:** bao gồm các module JDBC, ORM, OXM, JMS, Transactions trong đó:
 - spring-jdbc cung cấp một lớp JDBC-abstraction để loại bỏ những code tẻ nhạt. Cả JDBC và phân tích những mã lỗi cụ thể của database-vendor.
 - spring-orm cung cấp lớp tích hợp với các orm API phổ biến như JPA, JDO, Hibernate.
 - spring-oxm cung cấp lớp abstraction hỗ trợ triển khai Object/XMLmapping như AXB, Castor, XMLBeans, JiBX và XStream.
 - spring-jms chứa các tính năng tạo và sử dụng các message. Từ Spring Framework 4.1, nó đã được tích hợp spring-messaging.
 - spring-transaction hỗ trợ quản lý giao dịch theo chương trình và khai báo cho các lớp mà thực hiện các giao diện đặc biệt và cho tất cả POJO của bạn.
- **WEB:** bao gồm các module spring-web, spring-webmvc, spring-websocket, springwebmvc-porlet. Trong đó:
 - spring-web cung cấp các tính năng tích hợp web theo những chức năng theo hướng cơ bản như tải tập tin lên nhiều phần dữ liệu và khởi tạo các container IoC sử dụng nghe servlet và một bối cảnh ứng dụng web theo định hướng.
 - spring-webmvc bao gồm việc triển khai Model-View-Controller (MVC) của Spring cho ứng dụng web.
 - spring-websocket cung cấp hỗ trợ cho Websocket-based, giao tiếp hai chiều giữa client và server trong các ứng dụng web.

- **springwebmvc-portlet** cung cấp việc triển khai MVC được sử dụng trong môi trường portlet và ánh xạ chức năng của module Web-Servlet.

- Những module khác bao gồm:

- Module **AOP** cung cấp một thực hiện lập trình hướng khía cạnh cho phép bạn xác định phương pháp-chặn và pointcuts để sạch tách mã thực hiện chức năng đó nên được tách ra.
- Module **Aspects** cung cấp tích hợp với AspectJ, mà lại là một khuôn khổ AOP mạnh mẽ và trưởng thành.
- Module **Instrumentation** cung cấp thiết bị đo đạc lớp hỗ trợ và triển khai lớp bộ nạp được sử dụng trong các máy chủ ứng dụng nhất định.
- Module **Messaging** cung cấp hỗ trợ cho STOMP như WebSocket sub-protocol để sử dụng trong các ứng dụng. Nó cũng hỗ trợ một mô hình lập trình chú thích cho việc định tuyến và xử lý tin nhắn STOMP từ các máy khách WebSocket.
- Module **Test** hỗ trợ việc kiểm tra các thành phần Spring với JUnit hoặc TestNG khuôn khổ.

EXPLAIN

[↑ back \(#table-of-contents\)](#)

2. IOC CONTAINER

● Inversion of Control (IoC)

- **IoC** là một nguyên lý lập trình được thiết kế để đảo ngược qui trình điều khiển của **object** dựa trên một **object** khác. IoC bao gồm ba hướng triển khai:
 - Dependencies Injection
 - Service Locator
 - Events

Service Locator

- Đây đơn giản là việc lưu trữ một **service** (hay **dependencies**) vào trong **cache** và sẽ gọi **service** (hay **dependencies**) khi được yêu cầu.
- Hai **method** chính của **Service Locator** bao gồm:
 - **register**: thực hiện đăng ký, thêm **service** vào trong **cache**, dùng để lưu trữ và gọi **service** sau này.
 - **getService**: yêu cầu và thực hiện gọi **method**.

Trong **java**, việc sử dụng **Service Locator** không thường xuyên được quan tâm đến so với **Dependencies Injection**. Do đó, ở [phần sau \(#dependencies-injection\)](#), ta sẽ nói kĩ hơn về vấn đề **Dependencies Injection**.

NOTE

Dependencies Injection

- Trong khai báo thông thường, khi muốn tham chiếu/gọi một đối tượng được thực hiện bằng hàm contructor và gọi đối tượng đó một cách trực tiếp:

```
public class Cat{  
    private Hair hair;  
    public Cat(){  
        this.hair = new Hair();  
    }  
}
```

- Với kiểu khai báo như trên, trong trường hợp ta thay đổi **object** Hair (thêm một đối số chẵng hạn), thì lúc này ta buộc phải thay đổi giá trị của biến hair được khởi tạo bên trong hàm **constructor** của **object** Cat. Dòng code trên sẽ được thay đổi như sau.

```
public class Cat{  
    private Hair hair;  
    public Cat(){  
        # blue là màu sắc của Hair, hay là đối số của Hair.  
        this.hair = new Hair("blue");  
    }  
}
```

- Khi có sự thay đổi sẽ dẫn đến thay đổi bên trong nội tại dòng code. Về lâu dài khi dự án phình to ra, sẽ gây bất tiện cho quá trình truyền biến. Khi đó khái niệm **Dependencies Injection** được sử dụng để đề cập đến đoạn code bên dưới:

Dependencies Injection ↓

```
public class Cat{  
    private Hair hair;  
    public Cat(Hair hair){  
        # blue là màu sắc của Hair, hay là đối số của Hair.  
        this.hair = hair;  
    }  
}
```

Dependencies Inversion (đảo ngược điều khiển)

- Như đã đề cập ở trên, việc sử dụng **Dependencies Injection** giúp đảo ngược hành động điều khiển của **class** phụ thuộc nó. Quay lại với ví dụ ở trên trong hai **class** Cat và Hair:

- Khi sử dụng **constructor** mà không sử dụng cùng với Dependencies Injection thì khi này **class** Cat sẽ phụ thuộc vào Hair. Điều này sẽ dẫn đến việc khi dòng code của Hair thay đổi (như đã đề cập từ trước), thì Cat cũng sẽ thay đổi theo sự phụ thuộc của **class** này với Hair.
- Ngược lại, khi sử dụng **constructor** cùng với Dependencies Injection thì **class** Hair sẽ phụ thuộc vào Cat. Khi này sẽ được gọi là **Inversion of control** (đảo ngược điều khiển).

EXPLAIN

● IoC Container (Spring container)

- Trong Spring, khái niệm IoC Container (hay Spring Container) chỉ một **interface** quản lý việc truy xuất **đối tượng** trong **[java]** thông qua khái niệm Dependencies Injection. Spring sử dụng một đối tượng được gọi là **Beans** (sẽ được nêu rõ hơn ở [hần 3 \(#3-beans\)](#)).
- Trong Spring, **interface** ApplicationContext đại diện cho IoC Container. Spring container chịu trách nhiệm khởi tạo, config, cài đặt và lắp ráp các **dependencies** trong các đối tượng được gọi là **bean**, cũng như quản lý vòng đời của chúng.
- Spring cung cấp những **implements** của ApplicationContext: ClassPathXmlApplicationContext, FileSystemXmlApplicationContext cho những ứng dụng độc lập và WebApplicationContext cho những ứng dụng web.
- Tập hợp các beans là tập hợp những containers sử dụng siêu dữ liệu **xml** hoặc dưới dạng **annotation** trong **[java]**.
- Đoạn **code** dưới đây thể hiện cách khởi tạo vùng chứa theo cách thủ công:

```
ApplicationContext context
= new ClassPathXmlApplicationContext("applicationContext.xml");
```

- Vùng trên sẽ khởi tạo vùng chứa, nó sẽ đọc siêu dữ liệu chứa beans và sử dụng nó để tập hợp các bean trong thời gian chạy.

Ưu và nhược điểm của IoC Container

[↑ back \(#table-of-contents\)](#)

3. BEANS

● 3.1. Định nghĩa

- Spring Beans chính là những **Java Object** mà từ đó tạo nên khung sườn của một Spring application. Chúng được cài đặt, lắp ráp và quản lý bởi Spring IoC container. Những bean này được tạo ra bởi **configuration metadata** được cung cấp từ container, ví dụ, trong tag nằm trong **[file XML]**.
- Các bean được define trong spring framework là singleton bean. Có một thuộc tính trong bean với tên là “singleton” nếu được gán giá trị là **true** thì bean đó sẽ trở thành singleton, nếu là **false** thì bean đó sẽ trở thành **prototype** bean. Mặc định nếu không được định nghĩa giá trị của nó sẽ là **true**. Vì thế tất cả các bean trong spring framework mặc định sẽ là singleton bean.

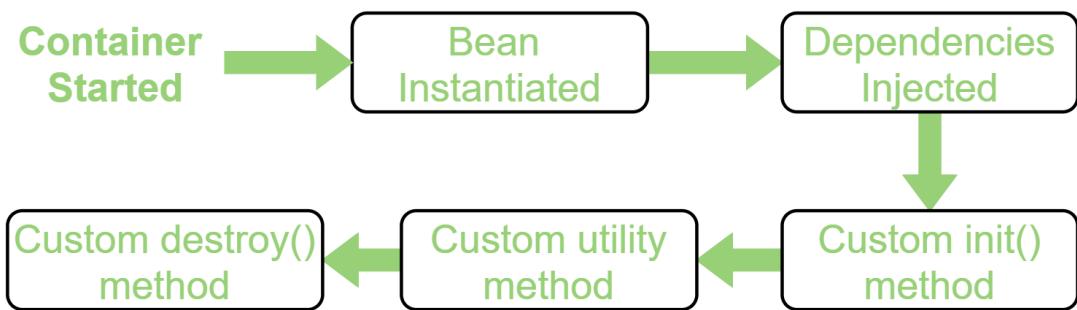
Singleton: là một **scope** mặc định của **bean scope** mang ý nghĩa khi một **bean** được khởi tạo bởi **IoC Container** sẽ được khởi tạo duy nhất một **instance** cho một **bean** đó và được sử dụng trong tất cả các yêu cầu đến **bean** này.

NOTE

[↑ back \(#table-of-contents\)](#)

● 3.2. Vòng đời của Beans

- Vòng đời của **bean** được mô tả như hình bên dưới:



Trong đó vòng đời của trải qua 2 giai đoạn chính: sau khi được khởi tạo và trước khi bị huỷ. Cả 2 giai đoạn này sẽ được mô tả ở bên dưới như sau:

- @PostConstructor**: là đánh dấu được sử dụng sau khi **bean** được khởi tạo. Đánh dấu này được đặt duy nhất trên một **method**. Đoạn code bên dưới sẽ mô tả một **method** được đánh dấu.
- @PreDestroy**: tương tự như **@PostConstructor**, **@PreDestroy** **annotation** sẽ được gán duy nhất một lần trong một **method**. Hàm sẽ được gọi trước khi **bị huỷ**.

EXPLAIN

Ví dụ về vòng đời của bean

```
// Girl.java
import org.springframework.stereotype.Component;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Component
```

```

public class Girl {

    @PostConstruct
    public void postConstruct() {
        System.out.println("\t>> Đôi tượng Girl sau khi khởi tạo xong sẽ chạy hàm này");
    }

    @PreDestroy
    public void preDestroy() {
        System.out.println("\t>> Đôi tượng Girl trước khi bị destroy thi chạy hàm này");
    }
}

// App.java

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class App {
    public static void main(String[] args) {
        // ApplicationContext chính là container, chứa toàn bộ các Bean
        System.out.println("> Trước khi IoC Container được khởi tạo");
        ApplicationContext context = SpringApplication.run(App.class, args);
        System.out.println("> Sau khi IoC Container được khởi tạo");

        // Khi chạy xong, lúc này context sẽ chứa các Bean có đánh
        // dấu @Component.

        Girl girl = context.getBean(Girl.class);

        System.out.println("> Trước khi IoC Container destroy Girl");
        ((ConfigurableApplicationContext) context).getBeanFactory().destroyBean(girl);
        System.out.println("> Sau khi IoC Container destroy Girl");

    }
}

```

Output:

- > Trước khi IoC Container được khởi tạo
- > Trước khi IoC Container được khởi tạo
 - >> Đối tượng Girl sau khi khởi tạo xong sẽ chạy hàm này
- > Sau khi IoC Container được khởi tạo
- > Trước khi IoC Container destroy Girl
 - >> Đối tượng Girl trước khi bị destroy thì chạy hàm này
- > Sau khi IoC Container destroy Girl

Bạn sẽ thấy dòng `Trước khi IoC Container được khởi tạo` được chạy 2 lần. Điều này xảy ra bởi vì hàm `App.main(args)` được chạy 2 lần! Lần đầu là do chúng ta chạy. Lần thứ hai là do Spring Boot chạy sau khi nó được gọi `SpringApplication.run(App.class, args)`. Đây là lúc mà **IoC Container** (`ApplicationContext`) được tạo ra và đi tìm Bean.

NOTE

IMPORTANT

Ý nghĩa

`@PostConstruct` và `@PreDestroy` là 2 **Annotation** cực kỳ ý nghĩa, nếu bạn nắm được vòng đời của một **Bean**, bạn có thể tận dụng nó để làm các nhiệm vụ riêng như setting, thêm giá trị mặc định trong thuộc tính sau khi tạo, xóa dữ liệu trước khi xóa, v.v.. Rất nhiều chức năng khác tùy theo nhu cầu.

[⬆ back \(#table-of-contents\)](#)

● 3.3. Spring Beans Scope

Định nghĩa scope

- Trong Spring phạm vi của **beans** được qui định như sau:
 - Application Scope
 - **Singleton Scope:** như đã nêu rõ ở [phản trước \(#note-singleton\)](#), **singleton** là phạm vi mặc định của **Spring Bean Scope**. Trong đó việc khởi tạo **beans** sẽ có ảnh hưởng một lần duy nhất cho một **bean** và mỗi lần được thay đổi giá trị sẽ tác động đến những vùng đã tham chiếu đến **beans**. Nó được định nghĩa trong **Spring** như đoạn code sau [code 1 \(#code-exam-singleton\)](#)
 - **Prototype Scope:** trong phạm vi này, mỗi khi **beans** được gọi bởi **container**, nó sẽ trả về một bản copy của mình. Khác với **singleton scope**, tại **prototype scope**, mỗi khi tác động giá trị đến **bean** này sẽ không ảnh hưởng đến những **bean** khác, miễn là ta không có tác động đến **bean** đó theo cách thủ công. Đoạn code sau sẽ mô tả về cách config **scope** trong **bean** [code 2 \(#code-exam-prototype\)](#)
 - Web aware scope:

Khác so với 2 **scope** ở trên, những **scope** trong phần này chỉ có phạm vi trong **Ứng dụng web**. Do đó, xét về độ thông dụng những **scope** trong phần này sẽ ít được sử dụng nhiều trong `[java]` cũng như tầm ảnh hưởng của nó.

 - **Request Scope:** nhìn vào đoạn [code \(#code-exam-request\)](#), ta có thể thấy đoạn `[proxyMode]`. Đoạn này rất cần thiết vì trong giai đoạn khi ứng dụng web được khởi tạo sẽ không có yêu cầu nào được kích hoạt. Khi này **Spring** sẽ dựa vào `[proxyMode]` mà khởi tạo một **proxy** như một **dependency** và khởi tạo **bean** mục tiêu khi nó được yêu cầu cho **request**.

- **Session Scope:** tương tự như Request Scope, Session Scope khởi tạo Scope như [code 4 \(#code-exam-session\)](#). Khi một bean được thay đổi giá trị thì sẽ tác động đến những bean khác bên trong session.
- **Application Scope:** tương tự như phạm vi singleton, tuy nhiên giữa chúng có một vài khác biệt quan trọng: khi được khởi tạo, beans trong application scope có phạm vi với tất cả ứng dụng được xây dựng dựa trên servlet-based và cùng chạy trên một ServletContext, trong khi singleton scope chỉ có phạm vi trong ngữ cảnh của một ứng dụng duy nhất. Đoạn code bên dưới sẽ mô tả cách qui định scope cho application scope [code 5 \(#code-exam-application\)](#).
- **Websocket scope:** khi beans được gọi lần đầu, Websocket beans sẽ lưu trữ nó trong websocket store. Một phiên bản tương tự của bean sẽ được lấy ra bất cứ khi nào beans được truy cập trong suốt quá trình diễn ra Websocket. Có thể nói, Websocket Scope là một dạng Singleton Scope nhưng chỉ trong phạm vi Websocket. Đoạn code bên dưới sẽ mô tả Spring khi **config** Socket scope [code 6 \(#code-exam-websocket\)](#)

Code Singleton Scope

[↑ return section \(#section-definite\)](#)

```
// Scope Singleton

@Bean
@Scope("singleton")
public Person personSingleton() {
    return new Person();
}

// hoặc

@Bean
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
public Person personSingleton() {
    return new Person();
}
```

Code Prototype Scope

[↑ return section \(#section-definite\)](#)

```
// Scope prototype

@Bean
@Scope("prototype")
public Person personPrototype() {
    return new Person();
}
```

Code Request Scope

[↑ return section \(#section-definite\)](#)

```

@Bean
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator requestScopedBean() {
    return new HelloMessageGenerator();
}

// hoặc

@Bean
@RequestScope
public HelloMessageGenerator requestScopedBean() {
    return new HelloMessageGenerator();
}

// Định nghĩa controller.

// Mỗi bean là đại diện cho một request được gửi đến
@Controller
public class ScopesController {
    @Resource(name = "requestScopedBean")
    HelloMessageGenerator requestScopedBean;

    @RequestMapping("/scopes/request")
    public String getRequestScopeMessage(final Model model) {
        model.addAttribute("previousMessage", requestScopedBean.getMessage());
        requestScopedBean.setMessage("Good morning!");
        model.addAttribute("currentMessage", requestScopedBean.getMessage());
        return "scopesExample";
    }
}

```

Code Session Scope

[⬆ return section \(#section-definite\)](#)

```

@Bean
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator sessionScopedBean() {
    return new HelloMessageGenerator();
}

// hoặc

@Bean

```

```

@SessionScope

public HelloMessageGenerator sessionScopedBean() {
    return new HelloMessageGenerator();
}

// Định nghĩa controller
// Một bean đại diện cho những bean khác trong session.

@Controller
public class ScopesController {

    @Resource(name = "sessionScopedBean")
    HelloMessageGenerator sessionScopedBean;

    @RequestMapping("/scopes/session")
    public String getSessionScopeMessage(final Model model) {
        model.addAttribute("previousMessage", sessionScopedBean.getMessage());
        sessionScopedBean.setMessage("Good afternoon!");
        model.addAttribute("currentMessage", sessionScopedBean.getMessage());
        return "scopesExample";
    }
}

```

Code Application Scope

[↑ return section \(#section-definite\)](#)

```

@Bean
@Scope(
    value = WebApplicationContext.SCOPE_APPLICATION, proxyMode =
    ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator applicationScopedBean() {
    return new HelloMessageGenerator();
}

// hoặc

@Bean
@ApplicationScope
public HelloMessageGenerator applicationScopedBean() {
    return new HelloMessageGenerator();
}

```

```

// Khởi tạo controller

// Trong trường hợp này, mỗi khi applicationScopedBean được gán, giá trị message
// sẽ được tác dụng cho tất cả những chuỗi con của request, session và event.
// và cả những ứng dụng servlet truy cập vào trong bean,
// miễn là chúng chạy chung trong ngữ cảnh ServletContext.

@Controller
public class ScopesController {
    @Resource(name = "applicationScopedBean")
    HelloMessageGenerator applicationScopedBean;

    @RequestMapping("/scopes/application")
    public String getApplicationScopeMessage(final Model model) {
        model.addAttribute("previousMessage", applicationScopedBean.getMessage());
        applicationScopedBean.setMessage("Good afternoon!");
        model.addAttribute("currentMessage", applicationScopedBean.getMessage());
        return "scopesExample";
    }
}

```

Code Websocket Scope

```

@Bean
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator websocketScopedBean() {
    return new HelloMessageGenerator();
}

```

[⬆ back \(#table-of-contents\)](#)

4. PROFILES

● 4.1 Định nghĩa

- Spring profiles là một feature của Spring framework, cho phép chúng ta cấu hình ứng dụng, active/deactive bean tùy theo môi trường. Sử dụng Profiles giúp cho việc config môi trường được quản lý tốt hơn và dễ dàng hơn.
- Để xây dựng profiles, ta tạo files config trong thư mục resources project. Mặc định spring nhận các file có tên như sau:

```

application.properties
application.yml
application-{profile-name}.yml // .properties

```

- Ví dụ:

```
application.yml  
application-local.yml  
application-aws.yml  
application-common.yml
```

Trong đó:

- `application`: là `file config` chính khai báo các `environment`.
- `application-local`: chỉ sử dụng khi chạy chương trình ở local.
- `application-aws`: chỉ sử dụng khi chạy ở AWS.
- `application-common`: là `config` dùng chung môi trường nào cũng cần.

EXPLAIN

● 4.2 Khai báo profiles

- Ta sẽ xem qua những đoạn khai báo sau để thấy rõ cách `profiles` hoạt động.

Khai báo `application.yml`

```
#application.yml  
---  
spring.profiles: local  
spring.profiles.include: common, local  
---  
spring.profiles: aws  
spring.profiles.include: common, aws  
---
```

Khai báo `application-aws.yml`

```
spring:  
  datasource:  
    -----
```

```
username: xxx

password: xxx

url: jdbc:mysql://10.127.24.12:2030/news?useSSL=false&characterEncoding=UTF-8
```

Khai báo `application-local.yml`

```
spring:
  datasource:
    username: root
    password:
    url: jdbc:mysql://localhost:3306/news?useSSL=false&characterEncoding=UTF-8

  logging:
    level:
      org:
        hibernate:
          SQL: debug
```

Khai báo `application-common.yml`

```
spring:
  jpa:
    properties:
      hibernate:
        jdbc:
          batch_size: 50
          batch_versioned_data: true
      hibernate:
        ddl-auto: none
```

● 4.3 Kích hoạt Profiles

Ta có những cách sau để kích hoạt config:

1. Sử dụng `spring.profiles.active` trong file `application.properties` hoặc `application.yml`

```
spring.profiles.active=aws
```

2. Active trong code, trước khi chạy chương trình.

```
@Configuration  
public class ApplicationInitializer  
    implements WebApplicationInitializer {  
  
    @Override  
    public void onStartup(ServletContext servletContext) throws ServletException {  
        servletContext.setInitParameter(  
            "spring.profiles.active", "aws");  
    }  
}
```

hoặc

```
@Autowired  
private ConfigurableEnvironment env;  
...  
env setActiveProfiles("aws");
```

hoặc

```
SpringApplication application = new SpringApplication(SpringBootProfilesApplication.class);  
ConfigurableEnvironment environment = new StandardEnvironment();  
environment.setActiveProfiles("aws");  
application.setEnvironment(environment);  
application.run(args);
```

→ cách này không thường xuyên được sử dụng và nó khiến cho chương trình trở nên phát sinh thêm một đối tượng ngoại lai.

3. Sử dụng JVM System Parameter (nên dùng)

```
-Dspring.profiles.active=aws
```

4. Environment Variable (Unix) (nên dùng)

```
export SPRING_PROFILES_ACTIVE=aws
```

● 4.4 Sử dụng với **@Profile**

- Khi đã sử dụng **profile**, ngoài biến toàn cục được thay đổi theo môi trường theo như **file** **config**, ta cũng có thể toàn quyền quyết định những **beans** hay **class** nào được quyền chạy ở môi trường nào thông qua Annotation **@Profile**

```
// Bean này Spring chỉ khởi tạo và quản lý khi môi trường là 'local'  
@Component  
@Profile("local")  
public class LocalDatasourceConfig
```

- Bên cạnh sử dụng như cách thông thường, người ta có thể sử dụng toán tử **logic** ở đây

```
// Bean này Spring chỉ khởi tạo và quản lý khi môi trường
// là những môi trường không phải là 'local'
@Component
@Profile("!local")
public class LocalDatasourceConfig
```

[↑back \(#table-of-contents\)](#)

5. AUTOWIRED

Như đã nêu ở phần **Dependencies Injection**, mỗi **bean** được khai báo trong **file beans.yml** phải chỉ rõ **beans** nào được **tiêm** vào bên trong **beans** nào. Tuy nhiên, **Spring Container** vẫn có thể tự tìm kiếm vào **tiêm beans** vào bên trong **beans** khác thông qua **Autowiring**.

● 5.1 Các dạng Autowired

Autowired sẽ tự định nghĩa các **beans** và thêm vào **beans** nguồn dựa vào các dạng sau.

5.1.1 With 'no' autowiring

Đây là dạng mặc định của việc tiêm **beans** khi **beans** được tiêm vào **beans** khác thông qua **[ref]**.

5.1.2 Autowire 'byName'

Tại dạng này, Autowire sẽ tự tìm kiếm những **beans** có **[id]** trùng với biến được khai báo trong **beans** cha. Khi đó những **beans** **[id]** này sẽ được tự động **tiêm** vào trong **beans** cha. Autowire này sẽ **tiêm beans** qua **[method]** **[setter]**.

Class Person với biến **[address]**

```
public class Person {
    private String name;
    private int age;
    private Address address; 
    public Person() {
    }

    public Person(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }
}
```

tìm kiếm bean có id = 'address'

Beans chứa **[id='address']** được **autowire="byName"**

```
<!-- Inject by setter -->
<bean id="person" class="stackjava.com.springdiobject.demo.Person" autowire="byName">
    <property name="name" value="stackjava.com"></property>
    <property name="age" value="25"></property>
</bean>

<bean id="address" class="stackjava.com.springdiobject.demo.Address">
    <property name="country" value="Viet Nam"></property>
    <property name="province" value="Ha Noi"></property>
    <property name="district" value="Thanh Xuan"></property>
</bean>
```

5.1.3 Autowire "byType"

Autowire này sẽ tiêm beans thông qua **method** **setter**. Theo đó, autowired sẽ tìm kiếm những beans có **type** trùng với **type** được khai báo trong **class** của beans cha:

Class Person với đối tượng/type **Address**

```
public class Person {  
    private String name;  
    private int age;  
    private Address address;  
  
    public Person() {}
```

tìm kiếm bean có type = Address

Beans chứa **class='Address'** được **autowire="byType"**

```
<!-- Inject by setter -->  
<bean id="person" class="stackjava.com.springdiobject.demo.Person" autowire="byType">  
    <property name="name" value="stackjava.com"></property>  
    <property name="age" value="25"></property>  
</bean>  
  
<bean id="address" class="stackjava.com.springdiobject.demo.Address">  
    <property name="country" value="Viet Nam"></property>  
    <property name="province" value="Ha Noi"></property>  
    <property name="district" value="Thanh Xuan"></property>  
</bean>
```

5.1.4 Autowired 'byConstructor'

Autowired sẽ được sử dụng thông qua **method** **constructor**. Theo đó autowired sẽ gọi đến những beans trùng **type** với beans được khai báo trong cha.

Class Person với đối tượng/type **Address**

```
public class Person {  
    private String name;  
    private int age;  
    private Address address;  
  
    public Person() {}  
  
    public Person(Address address) {  
        this.address = address;  
    }
```

Tìm bean có type = Address

Beans chứa **class='Address'** được **autowire="byConstructor"**

```
<!-- Inject by setter -->  
<bean id="person" class="stackjava.com.springdiobject.demo.Person" autowire="constructor">  
    <property name="name" value="stackjava.com"></property>  
    <property name="age" value="25"></property>  
</bean>  
  
<bean id="address" class="stackjava.com.springdiobject.demo.Address">  
    <property name="country" value="Viet Nam"></property>  
    <property name="province" value="Ha Noi"></property>  
    <property name="district" value="Thanh Xuan"></property>  
</bean>
```

● 5.2 Annotation **@Autowired** trong Spring

- Trong Spring, annotation này sẽ định nghĩa biểu thị các thuộc tính sẽ được autowired:

Để auto wired byType ta khai báo **@Autowired** ở trước phần khai báo thuộc tính hoặc trước

method **setter**

```
@Autowired(required = false)  
private Address address;  
//hoặc  
@Autowired(required = false)  
public void setAddress(Address address) {  
    this.address = address;  
}
```

Để auto wired byConstructor ta khai báo @Autowired ở trước method Constructor

```
@Autowired(required=true)  
public Person(Address address) {  
    this.address = address;  
}
```

Lưu ý: thuộc tính mặc định của **@AutoWired** là **true**:

- **[required=true]**: nếu spring container không tìm thấy bean address để inject vào thì nó sẽ báo lỗi.
- **[required=false]**: nếu spring container không tìm thấy bean address để inject vào thì nó sẽ inject **null**

NOTE

Demo

XML file

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns:context="http://www.springframework.org/schema/context"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

<context:annotation-config />
<!-- Inject by setter -->
<bean id="person" class="stackjava.com.springdiobject.demo.Person">
    <property name="name" value="stackjava.com"></property>
    <property name="age" value="25"></property>
</bean>
<bean id="address" class="stackjava.com.springdiobject.demo.Address">
    <property name="country" value="Viet Nam"></property>
    <property name="province" value="Ha Noi"></property>
    <property name="district" value="Thanh Xuan"></property>
</bean>
</beans>
```

Class Person

```
package stackjava.com.springdiobject.demo;

import org.springframework.beans.factory.annotation.Autowired;

public class Person {
    private String name;
    private int age;
    @Autowired(required = false)
    private Address address;

    public Person() {
    }

    public Person(Address address) {
        this.address = address;
    }

    public Person(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }
}
```

```

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Address getAddress() {
    return address;
}

public void setAddress(Address address) {
    this.address = address;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public void print() {
    System.out.println("Person: " + this.name + " Age: " + this.age + " Address: "
        + (this.address == null ? "null" : this.address.toString()));
}

```

Class MainApp

```

package stackjava.com.springdiobject.demo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new

```

```

ClassPathXmlApplicationContext("applicationContext.xml");

Person person = (Person) context.getBean("person");
person.print();
}
}

```

● Lưu ý với AutoWired

- Khả năng ghi đè (Overriding possibility): bạn vẫn có thể chỉ rõ dependency bằng cách sử dụng `<constructor-arg>` và `<property>` nó sẽ ghi đè lại autowiring.
- Kiểu dữ liệu nguyên thủy (Primitive data types): Bạn không thể thực hiện autowire với các dữ liệu nguyên thủy như `int`, `String`...
- Confusing nature: việc autowiring thực hiện tự động, đôi khi nó có thể link tới những bean không tồn tại, nếu có thể thì bạn hãy link nó một cách rõ ràng và dùng `required = true` để chắc chắn bean được prefer tới có tồn tại.
[↑back \(#table-of-contents\)](#)

6. JDBC

● 6.1. Định nghĩa

JDBC (Java Database Connectivity), là một API chuẩn để kết nối giữa ngôn ngữ lập trình Java và cơ sở dữ liệu. JDBC có thể thực hiện nhiều tác vụ đa dạng để tương tác với cơ sở dữ liệu như: tạo, sửa, xoá dữ liệu trong cơ sở dữ liệu.

● 6.2. Các thành phần của JDBC

JDBC API cung cấp các class và interface sau:

- DriverManager:** Lớp này quản lý các Database Driver. Ánh xạ các yêu cầu kết nối từ ứng dụng Java với Data driver thích hợp bởi sử dụng giao thức kết nối phụ.
- Driver:** Interface này xử lý các kết nối với Database Server. Hiếm khi, bạn tương tác trực tiếp với các đối tượng Driver này. Thay vào đó, bạn sử dụng các đối tượng DriverManager để quản lý các đối tượng kiểu này.
- Connection:** Đối tượng Connection biểu diễn ngữ cảnh giao tiếp. Interface này chứa nhiều phương thức đa dạng để tạo kết nối với một Database.
- Statement:** Bạn sử dụng các đối tượng được tạo từ Interface này để đệ trình các lệnh SQL tới Database. Ngoài ra, một số Interface kế thừa từ nó cung cấp nhận thêm các tham số để thực thi các thủ tục đã được lưu trữ.
- ResultSet:** Các đối tượng này giữ dữ liệu được thu nhận từ một Database sau khi bạn thực thi một truy vấn SQL. Nó đóng vai trò như một Iterator để cho phép bạn đọc qua dữ liệu của nó.
- SQLException:** Lớp này xử lý bất cứ lỗi nào xuất hiện trong khi làm việc với Database.
[↑back \(#table-of-contents\)](#)

7. QUẢN LÝ GIAO DỊCH

● 7.1 Transaction là gì?

- Transaction (giao dịch) là một tiến trình xử lý, có điểm bắt đầu và điểm kết thúc, gồm nhiều phép thực thi nhỏ, trong đó mỗi phép thực thi được thực thi một cách tuần tự và độc lập theo nguyên tắc là tất cả thành công hoặc một phép thực thi thất bại

thì cả tiến trình thất bại.

- Các thuộc tính ACID miêu tả rõ ràng nhất về Transaction. 4 thuộc tính này bao gồm Atomicity, Consistency, Isolation và Durability, trong đó:
 - Atomicity nghĩa là tất cả thành công hoặc không.
 - Consistency bảo đảm rằng tính đồng nhất của dữ liệu.
 - Isolation bảo đảm rằng Transaction này là độc lập với Transaction khác.
 - Durability nghĩa là khi một Transaction đã được ký thác thì nó sẽ vẫn tồn tại như thế cho dù xảy ra các lỗi.

● 7.2 Những method được sử dụng cho quản lý giao dịch

`void setAutoCommit(boolean status)`

Là `true` theo mặc định. Để thao tác với Transaction, bạn nên thiết lập về `false`.

`void commit()`

Để ký thác các thay đổi bạn đã thực hiện.

`void rollback()`

Xóa tất cả các thay đổi đã được thực hiện trước đó và quay về trạng thái trước khi thực hiện thay đổi.

`setSavepoint(String ten_cua_savepoint)`

Định nghĩa một savepoint mới. Phương thức này cũng trả về một đối tượng Savepoint.

`releaseSavepoint(Savepoint ten_cua_savepoint)`

Xóa một savepoint. Phương thức này nhận một đối tượng Savepoint làm tham số.

`rollback (String ten_cua_savepoint)`

Quay về trạng thái của savepoint đã cho

[⬆ back \(#table-of-contents\)](#)

● 8. Spring với Rest api

8.1 Định nghĩa