**Spring Core:**

* Spring framework là một nền tảng Java mã nguồn mở cung cấp cơ sở hạ tầng toàn diện hỗ trợ việc phát triển các ứng dụng Java dễ dàng và nhanh chóng. Hướng dẫn sau đây được viết dựa trên Spring phiên bản 4.1.6.
* Spring là một framework phát triển ứng dụng Java tốt nhất cho doanh nghiệp. Tối ưu cao , dễ dàng kiểm tra và tái sử dụng code.
* Tính năng của Spring Framework có thể sử dụng để phát triển bất kì ứng dụng Java nhưng chúng mở rộng cho việc xây dựng ứng dụng Web dựa trên nền tảng Java EE. Mục đích của Spring framework để làm cho việc phát triển J2EE dễ dàng hơn để sử dụng và thúc đẩy luyện tập phương pháp lập trình tốt bằng cách kích hoạt mô hình lập trình POJO cơ bản.

ứng dụng của Spring:

* **POJO Based**: Spring cho phép các nhà phát triển phát triển các ứng dụng cấp doanh nghiệp bằng cách sử dụng POJO. Lợi ích của việc chỉ sử dụng POJO là bạn không cần EJB container product chẳng hạn như máy chủ ứng dụng nhưng bạn có thể tùy chọn chỉ sử dụng vùng chứa Servlet mạnh mẽ như Tomcat hoặc một số sản phẩm thương mại.
* **Modular**: Spring được tổ chức theo kiểu modul. Mặc dù số lượng packages và class là đáng kể , bạn chỉ cần quan tâm về 1 số thứ bạn cần và bỏ qua phần còn lại.
* **Integration with existing frameworks**: Spring sử dụng một vài công nghệ đã tồn tại như là several ORM framworks, logging frameworks, JEE , Quartz , và JDK timers và một vài công nghệ khác.
* **Testablity:** Việc kiểm tra một ứng dụng được viết với Spring là đơn giản bởi vì code dựa vào môi trường được chuyển vào trong framework này. Hơn nữa , bằng việc sử dụng JavaBeanstyle POJOs sẽ khiến cho dễ dàng hơn để sử dụng injection dependency cho việc inject kiểm tra dữ liệu.
* **Web MVC:** Spring’s web framework là một thiết kế web MVC, nó cung cấp một giải pháp tuyệt vời tới web frameworks như là Struts hoặc các web framework ít phổ biến hơn.
* **Central Exception Handling :** Spring cung cấp một API tiện lợi để dịch các ngoại lệ giành riêng cho công nghệ (được throw bởi JDBC , Hibernate or JDO)thành các ngoại lệ nhất quán chưa được kiểm tra.
* **Lightweight :** Lightweight IoC containers thường “nhẹ” đặc biệt “nhẹ” hơn EJB containers, Điều này có lợi cho việc phát triển và triển khai các ứng dụng trên máy tính có bộ nhớ và tài nguyên CPU hạn chế.
* **Transaction management :** Spring cung cấp một giao diện quản lý transaction nhất quán thứ có thể mở rộng xuống một transaction cục bộ (sử dụng một database) và mở rộng lên tới transaction cục bộ (Sử dụng JTA).

**OVERVIEW:**

* **Dependency Injection (DI) :** Công nghệ mà Spring được định nghĩa nhiều nhất là Dependency Injection (DI) của Inversion of Control. Inversion of Control (IOC) là một khái niệm chung, và nó có thể diễn đạt theo nhiều cách khác nhau. Dependency Injection chỉ là một ví dụ cụ thể về Inversion of Control. Khi viết một ứng dụng Java phức tạp, các lớp ứng dụng phải càng độc lập với các lớp Java khác để tăng khả năng sử dụng lại các lớp khác và kiểm tra tính độc lập của các lớp khác trong khi kiểm tra đơn vị. Dependency Injection giúp dán các lớp này lại với nhau và đồng thời giữ cho chúng độc lập.

Dependency Injection chính xác là gì? Chúng ta hãy xem xét hai từ này riêng biệt. Ở đây phần phụ thuộc chuyển thành sự liên kết giữa 2 lớp. Ví dụ, Lớp A phụ thuộc của lớp B. Bây giờ , injection có nghĩa là Lớp B sẽ được IoC đưa vào lớp A.

Dependency Injection có thể xảy ra theo cách truyền các tham số cho constructor hoặc bằng cách sử dụng các phương thức setter sau khi xây dựng. Vì Dependency Injection là trung tâm của Spring Framework , nên sẽ có chương riêng nói về nó.

* **Aspect Oriented Programming (AOP):** Một trong những thành phần quan trọng của Spring là Lập trình khía cạnh AOP Framework. Các chức năng trải dài nhiều điểm của một ứng dụng được gọi là “Mối quan tâm xuyên suốt” và những mối quan tâm này tách biệt về mặt khái niệm với logic nghiệp vụ của ứng dụng. Ví dụ: logging, declarative transactions, security, caching, etc..

Đơn vị quan trọng của modul trong OOP là lớp, trong khi AOP, đơn vị mô đun là khía cạnh. DI giúp bạn tách các đối tượng ứng dụng của mình khỏi nhau, trong khi AOP giúp bạn tách các mối quan tâm xuyên suốt khỏi các đối mà chúng ảnh hưởng.

Mô đun AOP của Spring Framework cung cấp một triển khai lập trình hướng khía cạnh cho phép bạn định nghĩa các phương thức chặn “interceptors” và các điểm dừng để phân tách rõ ràng code thực thi chức năng cần được tách biệt.

**Architecture:**

* Spring có thể là một điểm dừng tiềm năng cho toàn bộ ứng dụng doanh nghiệp của bạn. Tuy nhiên, Spring là modul cho phép bạn lấy và chọn modul nào có thể phù hợp cho bạn mà không cần phải chọn phần còn lại.
* Spring Framework cung cấp khoảng 20 modul có thể được sử dụng dựa trên yêu cầu ứng dụng.



Core Container:

* Core Container bao gồm Core , Beans , Context và Expression Language:

Modul Core cung cấp các phần cơ bản của framework, bao gồm IOC và tính năng Dependency Injection.

* Modul Bean cung cấp BeanFactory , là một triển khai phức tạp của factory pattern.
* Modul Context xây dựng dựa trên nền tảng vững chắc được cung cấp bởi modul Core và Beans và nó là một phương tiện để truy cập bất kì đối tượng nào được xác định và cấu hình nó. ApplicationContext là tâm điểm của modul Context.
* Modul SpEL cung cấp một expression language để truy vấn và thao tác một object graph trong thời gian chạy.

Miscellaneous:

Có một vài modul quan trọng khác như là AOP, Aspects , Instrumentation, Web và Test:

* Modul AOP cung cấp một triển khai lập trình hướng khía cạnh cho phép bạn định nghĩa các phương thức interceptors và điểm cụt để code triển khai rõ ràng tách biệt các chức năng.
* Aspects modul cung cấp tích hợp với AspectJ là một AOP framework mạnh mẽ.
* Modul Instrumentation cung cấp hỗ trợ đo đạc các class và triển khai nạp các lớp để được sử dụng trong các ứng dụng servers nhất định.
* Modul Messaging cung cấp hỗ trợ cho STOMP như một bản sao của WebSocket để sử dụng trong ứng dụng. Nó cũng hỗ trợ một mẫu lập trình annotation dành cho việc định tuyến và tiến trình tin nhắn STOMP từ WebSocket clients.
* Modul Test hỗ trợ việc kiểm tra của thành phần Spring với Junit hoặc TestNG frameworks.

**IoC Containers :**

* Spring container là cốt lõi của Spring Framework. Vùng chứa sẽ tạo các đối tượng, kết nối chúng lại với nhau, cấu hình chúng và quản lý vòng đời hoàn chỉnh của chúng từ khi tạo ra cho đến khi bị phá hủy. Spring container sử dụng DI để quản lý các thành phần tạo nên một ứng dụng. Những đối tượng này được gọi là Spring Beans, chúng ta sẽ thảo luận trong chương sau.

Vùng chứa nhận hướng dẫn về những đối tượng nào cần khởi tạo, cấu hình và lắp ráp bằng cách đọc siêu dữ liệu cấu hình được cung cấp. Siêu dữ liệu cấu hình có thể được biểu diễn bằng XML, annotation Java hoặc code Java. Biểu đồ sau đây thể hiện chế độ xem cấp cao về cách hoạt động của Spring. Vùng chứa Spring IoC sử dụng các lớp Java POJO và siêu dữ liệu cấu hình để tạo ra một cấu hình đầy đủ và hệ thống hoặc ứng dụng thực thi.



Spring cung cấp 2 loại containers sau đây:

1 - Spring BeanFactory Container

Đây là container đơn giản nhất cung cấp hỗ trợ cơ bản cho DI và

Được định nghĩa bởi giao diện org.springframework.beans.factory.BeanFactory.

BeanFactory và các giao diện liên quan, chẳng hặn như BeanFactoryAware, InitializingBean, DisposableBean, vẫn hiện diện trong Spring với mục đích tương thích lại với một số lượng lớn các framework thứ ba tích hợp vs Spring.

2- Spring ApplicationContext Container

Container này thêm nhiều các tính năng dành cho doanh nghiệp như là khả năng giải quyết các thông báo văn bản từ một tệp thuộc tính và khả năng xuất bản các sự kiện ứng dụng cho những người lắng nghe sự kiện đó. Container này được định nghĩa bởi org.springframework.context.ApplicationContext Interface.

Container ApplicationContext bao gồm tất cả các chức năng của BeanFactory container vì vậy nó thường được khuyến nghị sử dụng trên BeanFactory. BeanFactory vẫn có thể được sử dụng cho các ứng dụng nhẹ như thiết bị di động hoặc các ứng dụng dựa trên applet, nơi lưu lượng và tốc độ dữ liệu là đáng kể.

Spring BeanFactory Container

Đây là vùng chứa đơn giản nhất cung cấp hỗ trợ cơ bản cho DI và được định nghĩa bởi interface org.springframework.beans.factory.BeanFactory. BeanFactory và các giao diện liên quan , chẳng hạn như BeanFactoryAware, InitializingBean, DisposableBean, vẫn hiện diện trong Spring với mục đích tương thích ngược với một số lượng lớn các framework bên thứ ba tích hợp với Spring.

Có một số triển khai của interface BeanFactory được cung cấp ngay lập tức với Spring. Việc triển khai BeanFactory được sử dụng phổ biến nhất là lớp XMLBeanFactory. Vùng chứa này đọc siêu dữ liệu cấu hình từ tệp XML và sử dụng nó để tạo hệ thống hoặc ứng dụng được định cấu hình đầy đủ.

BeanFactory thường được ưu tiên khi tài nguyên bị hạn chế như thiết bị di động hoặc ứng dụng dựa trên applet, Vì vậy , hãy sử dụng ApplicationContext trừ khi bạn có lý do chính đáng để không làm như vậy.

Example:

HelloWorld.java:

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

MainApp.java:

package com.tutorialspoint;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

public class MainApp {

public static void main(String[] args) {

XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("Beans.xml"));

HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");

obj.getMessage();

}

}

2 điểm quan tọng nên được lưu ý về chương trình main:

1. bước đầu tiên là để tạo một đối tượng factory nơi chúng sử dụng APIXmlBeanFactory() để tạo bean factory và API ClassPathResource() để load file cấu hình bean có sẵn trong CLASSPATH. API XmlBeanFactory() đảm nhận việc tạo và khởi tạo toàn bộ đối tượng, tức là các bean được đề cập trong file cấu hình.
2. Bước hai được sử dụng để lấy bean được yêu cầu sử dụng phương thức getBean() của đối tượng bean factory được tạo. Phương thức này sử dụng bean ID để trả về một đối tượng generic cuối cùng có thể được cast tới đối tượng thực tế. Sau khi bạn có đối tượng , bạn có thể sử dụng đối tượng này để gọi bất kỳ phương thức nào.

Beans.xml:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>

Kết quả:

Your Message : Hello World!

Spring ApplicationContext Container

Application Context là container cấp cao hơn của Spring. Tương tự như BeanFactory, nó có thể load định nghĩa bean, wire các bean với nhau, và phân phối bean theo yêu cầu. Ngoài ra , nó thêm nhiều chức năng chỉ dành cho doanh nghiệp như là khả năng giải quyết các thông báo văn bản từ tệp thuộc tính và khả năng xuất bản các sự kiện ứng dụng cho những người nghe sự kiện quan tâm. Container này được xác định bởi interface org.springframework.context.ApplicationContext.

ApplicationContext bao gồm toàn bộ chức năng của BeanFactory, Nó thường được đề xuất trên BeanFactory. BeanFactory vẫn có thể được sử dụng cho các ứng dụng nhẹ như thiết bị di động hoặc applet.

Các triển khai ApplicationContext thường được sử dụng nhất là:

1. FileSystemXmlApplicationContext:

Container này load các định nghĩa của các bean từ một file XML. Bạn cần cung cấp đường dẫn đầy đủ của file cấu hình bean XML tới hàm khởi tạo.

1. ClassPathXmlApplicationContext:

Container này load các định nghĩa của các bean từ một file XML. Bạn không cần cung cấp đường dẫn đầy đủ của file XML nhưng bạn cần set CLASSPATH bởi vì container này sẽ tìm kiếm file XML cấu hình bean trong CLASSPATH.

1. WebXmlApplicationContext:

Container này load file XML với các định nghĩa của toàn bộ bean từ một ứng dụng web

Chúng ta đã hoàn thành một ví dụ trên ClassPathXmlApplicationContext container trong Spring Hello World Example, và chung ta sẽ thảo luận nhiều hơn về WebXmlApplicationContext trong một chương riêng khi chung ta sẽ thảo luận ứng dụng Spring dựa trên web. Hãy xem một ví dụ trên FileSystemXmlApplicationContext.

Example:

HelloWorld.java:

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

MainApp.java:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new FileSystemXmlApplicationContext

("C:/Users/ZARA/workspace/HelloSpring/src/Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

}

Có 2 điểm quan trọng nên được lưu ý về chương trình main:

1. Bước 1 để tạo đối tượng factory khi chúng ta sử dụng API FileSystemXmlApplicationContext để tạo bean factory sau khi load file cấu hình bean khi nhận đường dẫn . Api FileSystemXmlApplicationContext() đảm nhận việc tạo và khởi tạo toàn bộ đối tượng tức là bean được đề cập trong file cấu hình bean XML.
2. Bước 2 được sử dụng để lấy bean yêu cầu sử dụng phương thức getBean() của context đã tạo. Phương thức này sử dụng bean ID để trả về một đối tượng generic cuối cùng có thể được cast tới đối tượng thực tế. Sau khi bạn có một đối tượng , bạn có thể sử dụng đối tượng này để gọi bất kì phương thức nào.

Beans.xml:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>

Kết quả:

Your Message : Hello World!

**Bean Definition :**

* Các đối tượng tạo thành xương sống của ứng dụng của bạn và chúng được quản lý bởi Spring IoC container được gọi là các beans. Một bean là một đối tượng được khởi tạo , lắp ráp và quản lý bởi một Spring IoC container. Những beans được tạo với cấu hình siêu dữ liệu cung cấp cho container.Ví dụ: trong mẫu của XML <bean/> định nghĩa:
* Bean định nghĩa chứa thông tin được gọi là cấu hình siêu dữ liệu, chúng được cần thiết cho container biết:
  + - * Cách tạo một bean
      * Chi tiết vòng đời của bean
      * Bean’s dependencies
* Toàn bộ việc cấu hình siêu dữ liệu chuyển thành một tập hợp các thuộc tính sau đây tạo nên mỗi định nghĩa bean:
  + Class: thuộc tính này là bắt buộc và cần thiết cho lớp bean được sử dụng để tạo bean.
  + Name: thuộc tính này dành cho định nghĩa bean duy nhất. Trong việc cấu hình siêu dữ liệu XML , bạn sử dụng id và/hoặc thuộc tính tên để chỉ định định nghĩa bean.
  + Scope : thuộc tính này chỉ định phạm vi của đối tượng được tạo từ một định nghĩa bean cụ thể
  + Constructor-arg: nó được sử dụng để inject dependencies
  + Properties: nó được sử dụng để inject dependencies
  + Autowiring mode: nó được sử dụng để inject dependencies
  + Lazy-initialization mode: một lazy-initialized bean yêu cầu IoC container để tạo một thành phần bean khi nó là yêu cầu đầu tiên thay vì khởi chạy.
  + Initialization method: một callback được gọi chỉ sau khi toàn bộ properties cần thiết trên bean đã hoàn thành cài đặt bởi container.
  + Destruction method: một callback được sử dụng khi container chứa bean bị phá hủy.
* Spring Configuration Metadata
* Spring IoC container hoàn toàn được tách biệt khỏi định dạng mà siêu dữ liệu cấu hình thực sự được viết. Sau đây là ba phương thức quan trọng để cung cấp siêu dữ liệu cấu hình tới Spring Container.
  + File cấu hình dựa trên XML
  + Cấu hình dựa trên Annotation
  + Cấu hình dựa trên Java
* Bạn đã thấy cách siêu dữ liệu cấu hình dựa trên XML được cung cấp cho container, nhưng chúng ta hãy xem một file cấu hình dựa trên XML khác với các định nghĩa bean khác nhau bao gồm lazy initialization, initialization method và destruction method

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- A simple bean definition -->

<bean id = "..." class = "...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with lazy init set on -->

<bean id = "..." class = "..." lazy-init = "true">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with initialization method -->

<bean id = "..." class = "..." init-method = "...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with destruction method -->

<bean id = "..." class = "..." destroy-method = "...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- more bean definitions go here -->

</beans>

Bạn có thể kiểm ví dụ Spring Hello World để hiểu cách định nghĩa , cấu hình và tạo Spring Beans. Chúng ta sẽ thảo luận về Annotation Based Configuration trong một chương riêng.Chúng tôi muốn bạn nắm bắt một vài khái niệm Spring quan trọng khác trước khi bạn bắt đầu lập trình với Spring Dependency Injection với Annotations.

**Bean Scopes :**

Khi định nghĩa một <bean> bạn có lựa chọn việc khai báo một scope dành cho bean. Ví dụ , để buộc Spring tạo ra một thành phần bean mới mỗi lần khi cần, bạn nên khai báo thuộc tính scope của bean để Prototype. Tương tự vậy , nếu bạn muốn Spring trả về cùng một thành phần bean giống nhau mỗi lần khi cần, bạn nên khai báo thuộc tính scope của bean là Singleton.

Spring Framework hỗ trợ 5 scopes sau , 3 trong số chúng là khả thi chỉ nếu bạn sử dụng một web-aware ApplicationContext.

1. Singleton
   1. Scopes này định nghĩa bean là một thành phần duy nhất cho mỗi Spring IoC container.(default)
2. Prototype
   1. Scopes này định nghĩa một bean duy nhất có bất kì số lượng thành phần đối tượng.
3. Request
   1. Scopes này định nghĩa một bean tới một HTTP request. Chỉ đúng trong context của một web-aware Spring ApplicationContext.
4. Session
   1. Scopes này định nghĩa một bean tới một HTTP session. Chỉ đúng trong context của một web-aware Spring ApplicationContext.
5. Global-Session
   1. Scopes này định nghĩa một bean tới một HTTP session toàn cục . Chỉ đúng trong context của một web-aware Spring ApplicationContext.

Trong chương này , chúng ta sẽ chỉ thảo luận về 2 scopes đầu tiên và con lại sẽ được thảo luận khi chúng ta thảo luận về web-aware Spring ApplicationContext.

Singleton Scope:

Nếu một scope được set là singleton, Spring IoC container tạo chính xác một thành phần của đối tượng được định nghĩa bởi chính việc định nghĩa bean. Thành phần duy nhất này được chứa trong một bộ nhớ như là singleton beans, và toàn bộ các request và tham chiếu tiếp theo cho bean có tên đó trả về đối tượng lưu trong bộ nhớ.

Scope mặc định luôn luôn là singleton. Tuy nhiên, khi bạn cần một và chỉ một thành phần của một bean, bạn có thể set thuộc tính scope là singleton trong file cấu hình bean, được đưa ra dưới đây:

<!-- A bean definition with singleton scope -->

<bean id = "..." class = "..." scope = "singleton">

<!-- collaborators and configuration for this bean go here -->

</bean>

Prototype Scope:

Nếu scope được set là prototype, Spring IoC container tạo một thành phần bean mới của object mỗi khi yêu cầu cụ thể bean được tạo. Theo nguyên tắc , sử dụng prototype scope cho toàn bộ các bean trạng thái đầy đủ và phạm vi singleton cho các bean không trạng thái.

Để định nghĩa một scope prototype , bạn có thể set thuộc tính scope là prototype trong file cấu hình bean:

<!-- A bean definition with prototype scope -->

<bean id = "..." class = "..." scope = "prototype">

<!-- collaborators and configuration for this bean go here -->

</bean>

**Bean Life Cycle :**

Vòng đời của một Spring bean rất dễ hiểu. Khi 1 bean được khởi tạo, nó có thể yêu cầu một số khởi tạo để đưa nó vào trạng thái sử dụng. Tương tự, khi bean không còn cần thiết và được xóa khỏi container, một vài việc dọn dẹp có thể cần thiết.

Mặc dù , có danh sách các hoạt động ở phía sau giữa thời điểm của việc khởi tạo bean đến khi nó phát hủy, chương này sẽ thảo luận về 2 phương thức callback vòng đời của bean quan trọng, được yêu cầu tại thời điểm khởi tạo bean đến khi nó phá hủy.

Để định nghĩa cài đặt và chia nhỏ một bean, chúng ta khai báo đơn giản <bean> với tham số initmethod và/ hoặc destroy-method. Thuộc tính init-method chỉ định một phương thức được gọi ngay sau khi khởi tạo. Tương tự , Destroymethod chỉ định một phương thức được gọi sau khi một bean được xóa khỏi container.

Initialization Callbacks

org.springframework.beans.factory.InitializingBean interface chỉ định một phương thức đơn-

void afterPropertiesSet() throws Exception;

bạn có thể triển khai đơn giản giữa interface và việc khởi tạo có thể được hoàn thành bên trong phương thức afterProperties() :

public class ExampleBean implements InitializingBean {

public void afterPropertiesSet() {

// do some initialization work

}

}

Trong trường hợp siêu dữ liệu cấu hình dựa trên XML , bạn có thể sử dụng thuộc tính init-method để chỉ định tên của phương thức có kí hiệu một void không tham số:

<bean id = "exampleBean" class = "examples.ExampleBean" init-method = "init"/>

Giống như định nghĩa class dưới đây:

public class ExampleBean {

public void init() {

// do some initialization work

}

}

Destruction Callbacks

Org.springframework.beans.factory.DisposableBean interface chỉ định một phương thức duy nhất

void destroy() throws Exception;

Bạn có thể triển khai đơn giản giữa interface và việc kết thúc có thể hoàn thành bên trong phương thức destroy():

public class ExampleBean implements DisposableBean {

public void destroy() {

// do some destruction work

}

}

Trong trường hoặc siêu dữ liệu cấu hình dựa trên XML , bạn có thể sử dụng thuộc tính destroy-method để chỉ định tên của phương thức có kí hiệu void không tham số :

<bean id = "exampleBean" class = "examples.ExampleBean" destroy-method = "destroy"/>

Giống như định nghĩa class dưới đây:

public class ExampleBean {

public void destroy() {

// do some destruction work

}

}

Nếu bạn đang sử dụng Spring’s IoC container trong một môi trường ứng dụng non-web , ví dụ trong một môi trường client desktop , bạn đăng ký một móc shutdown với JVM. Làm như vậy đảm bảo việc tắt một cách dễ dàng và gọi các phương thức hủy có liên quan trên các singleton beans để giải phóng tất cả các tài nguyên.

Nó được khuyến kích bạn không sử dụng IntializingBean hoặc DisposableBean callbacks bởi vì cấu hình XML mang lại nhiều tính linh hoạt trong việc đặt tên các phương thức của bạn.

Default Initialization And Destroy Methods

Nếu bạn có quá nhiều phương thức khởi tạo và/hoặc hủy beans với tên giống nhau, bạn không cần khai báo init-method và destroy-method trên mỗi bean.

Thay thế , framework cung cấp tính linh hoạt để cấu hình tình huống đó sử dụng thuộc tính default-init-method và default-destroy-method trên phần tử <beans>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"

default-init-method = "init"

default-destroy-method = "destroy">

<bean id = "..." class = "...">

<!-- collaborators and configuration for this bean go here -->

</bean>

</beans>

**Bean Post Processors :**

BeanPostProcessor interface định nghĩa phương thức callback bạn có thể triển khai để cung cấp logic khởi tạo của riêng bạn, logic giải quyết phụ thuộc , etc. Bạn cũng có thể triển khai một vài logic tự tạo sau khi việc tạo ra Spring container hoàn thành, cấu hình và khởi tạo một bean bằng thực hiện một hoặc nhiều triển khai BeanPostProcessor.

Bạn có thể cấu hình nhiều BeanPostProcessor interfaces và bạn có thể kiểm soát thứ tự mà các BeanPostProcessor interfaces thực thi bằng cách đặt thứ tự thuộc tính đã cung cấp bởi BeanPostProcessor triển khai Ordered interface.

BeanPostProcessors hoạt động trên cá thể bean hoặc đối tượng, có nghĩa là Spring IoC container khởi tạo một cá thể bean và sau đó BeanPostProcessor interface thực hiện công việc của chúng.

ApplicationContext tự động phát hiện bất kỳ bean nào được xác định với việc triển khai BeanPostProcessor interface và đăng ký các bean này như là postprocessors, sau đó được gọi thích hợp bởi container khi tạo bean.

Ví dụ:

Ví dụ dưới đây sẽ chỉ cách viết , đăng kí , và sử dụng BeanPostProcessors trong context của một ApplicationContext.

HelloWorld.java:

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

public void init(){

System.out.println("Bean is going through init.");

}

public void destroy(){

System.out.println("Bean will destroy now.");

}

}

Đây là một ví dụ rất cơ bản của việc triển khai BeanPostProcessor , nó in tên một bean trước và sau khởi tạo bất kể bean nào. Bạn có thể triển khai nhiều logic phức tạp trước và sau khi khởi tạo một bean bởi vì bạn có thể truy cập vào đối tượng bean bên trong cả hai phương thức post processor.

InitHelloWorld.java:

package com.tutorialspoint;

import org.springframework.beans.factory.config.BeanPostProcessor;

import org.springframework.beans.BeansException;

public class InitHelloWorld implements BeanPostProcessor {

public Object postProcessBeforeInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("BeforeInitialization : " + beanName);

return bean; // you can return any other object as well

}

public Object postProcessAfterInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("AfterInitialization : " + beanName);

return bean; // you can return any other object as well

}

}

Bạn cần đăng kí một shutdown hook registerShutdownHook() method nó được khai báo trên AbstractApplicationContext class. Nó sẽ đảm bảo shutdown và gọi các phương thức destroy liên quan.

MainApp.java

package com.tutorialspoint;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

AbstractApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

context.registerShutdownHook();

}

}

Beans.xml:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld"

init-method = "init" destroy-method = "destroy">

<property name = "message" value = "Hello World!"/>

</bean>

<bean class = "com.tutorialspoint.InitHelloWorld" />

</beans>

Khi bạn hoàn thanh việc tạo file nguồn và file cấu hình bean, chạy ứng dụng và in ra thông báo sau:

BeforeInitialization : helloWorld

Bean is going through init.

AfterInitialization : helloWorld

Your Message : Hello World!

Bean will destroy now.

**Bean Definition Inheritance**

Một bean định nghĩa có thể chứa nhiều thông tin cấu hình , bao gồm tham số hàm khởi tạo , giá trị thuộc tính , và thông tin container cụ thể như là phương thức khởi tạo , tên phương thức factory static.

Định nghĩa 1 bean con kế thừa dữ liệu cấu hình từ định nghĩa cha. Định nghĩa con có thể override một vài giá trị hoặc thêm nếu cần thiết.

Spring Bean định nghĩa kế thừa không liên quan gì đến kế thừa lớp Java nhưng khái niệm kế thừa giống nhau.Bạn có thể định nghĩa một định nghĩa bean cha như một khuôn mẫu và các bean con khác có thể kế thừa cấu hình bắt buộc từ bean cha.

Khi bạn sử dụng Siêu dữ liệu cấu hình dựa trên XML , bạn chỉ định một định nghĩa bean con bằng cách sử dụng thuộc tính parent, chỉ định bean cha làm giá trị thuộc tính này.

Sau đây là file cấu hình Beans.xml nơi chúng ta định nghĩa bean “helloWorld” có hai thuộc tính message1 và message2. Tiếp theo bean “helloIndia” đã được định nghĩa là con của bean “helloWorld” bằng cách sử dụng thuộc tính parent. Bean con kế thừa thuộc tính message2 như hiện tại, và overrides thuộc tính message1 và giới thiệu thêm một thuộc tính message3.

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">

<property name = "message1" value = "Hello World!"/>

<property name = "message2" value = "Hello Second World!"/>

</bean>

<bean id ="helloIndia" class = "com.tutorialspoint.HelloIndia" parent = "helloWorld">

<property name = "message1" value = "Hello India!"/>

<property name = "message3" value = "Namaste India!"/>

</bean>

</beans>

HelloWorld.java

package com.tutorialspoint;

public class HelloWorld {

private String message1;

private String message2;

public void setMessage1(String message){

this.message1 = message;

}

public void setMessage2(String message){

this.message2 = message;

}

public void getMessage1(){

System.out.println("World Message1 : " + message1);

}

public void getMessage2(){

System.out.println("World Message2 : " + message2);

}

}

HelloIndia.java

package com.tutorialspoint;

public class HelloIndia {

private String message1;

private String message2;

private String message3;

public void setMessage1(String message){

this.message1 = message;

}

public void setMessage2(String message){

this.message2 = message;

}

public void setMessage3(String message){

this.message3 = message;

}

public void getMessage1(){

System.out.println("India Message1 : " + message1);

}

public void getMessage2(){

System.out.println("India Message2 : " + message2);

}

public void getMessage3(){

System.out.println("India Message3 : " + message3);

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

objA.getMessage1();

objA.getMessage2();

HelloIndia objB = (HelloIndia) context.getBean("helloIndia");

objB.getMessage1();

objB.getMessage2();

objB.getMessage3();

}

}

Chạy application:

World Message1 : Hello World!

World Message2 : Hello Second World!

India Message1 : Hello India!

India Message2 : Hello Second World!

India Message3 : Namaste India!

Bean Definition Template

Bạn có thể tạo một mẫu định nghĩa Bean, có thể được sử dụng bởi định nghĩa bean con khác mà không cần nỗ lực nhiều. Trong khi định nghĩa một mẫu định nghĩa Bean , bạn không nên chỉ định thuộc tính class và nên chỉ định thuộc tính abstract và nên chỉ định thuộc tính abstract với một giá trị của true chỉ ra dưới đây:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "beanTeamplate" abstract = "true">

<property name = "message1" value = "Hello World!"/>

<property name = "message2" value = "Hello Second World!"/>

<property name = "message3" value = "Namaste India!"/>

</bean>

<bean id = "helloIndia" class = "com.tutorialspoint.HelloIndia" parent = "beanTeamplate">

<property name = "message1" value = "Hello India!"/>

<property name = "message3" value = "Namaste India!"/>

</bean>

</beans>

Không thể khởi tạo bean cha một mình vì nó chưa hoàn chỉnh và nó cũng được đánh dấu rõ ràng là abstract. Khi a định nghĩa là abstract như đây , nó chỉ sử dụng như là một khuôn mẫu thuần định nghĩa bean thuần túy dùng làm định nghĩa cha cho các định nghĩa con.

**Dependency Injection :**

Mỗi ứng dụng dựa trên Java có một vài đối tượng làm việc cùng nhau để trình bày những gì người dùng kết thúc coi như là một ứng dụng đang hoạt động. Khi viết một ứng dụng Java phức tạp , các lớp ứng dụng nên là độc lập khả thi của các lớp Java khác để tăng khả năng tái sử dụng lại các lớp này và kiểm tra chúng độc lập với các lớp khác trong khi kiểm tra đơn vị. Dependency Injection (hoặc đôi khi gọi là wiring) giúp gắn kết các lớp lại với nhau và đồng thời giữ chúng độc lập.

Quan sát bạn có một ứng dụng có một thành phần sửa văn bản và bạn muốn cung cấp một kiểm tra chính tả. Code bình thường của bạn sẽ trông như này:

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor() {

spellChecker = new SpellChecker();

}

}

Những gì chúng ta đã làm ở đây là tạo sự phụ thuộc giữa TextEditor và SpellChecker. Trong một kịch bản điều khiển đảo ngược thay vào đó chúng ta sẽ làm như này:

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

this.spellChecker = spellChecker;

}

}

TextEditor không lo lắng về triển khai SpellChecker. SpellChecker sẽ được triển khai độc lập và sẽ được cung cấp tới TextEditor tại thời điểm TextEditor khởi tạo. Toàn bộ quy trình này được kiểm soát bởi Spring Framework.

Chúng ta đã xóa toàn bộ quyền kiểm soát từ TextEditor và giữ nó ở đâu đó (ví dụ file cấu hình XML) và phụ thuộc (lớp SpellChecker) đang được inject tới lớp TextEditor bằng một Class Constructor. Do đó luống kiểm soát đã bị đảo ngược bởi Dependency Injection (DI) bởi vì bạn đã ủy quyền một cách hiệu quả các phụ thuộc cho một số hệ thống bên ngoài.

Phương thức thứ hai để đưa sự phụ thuộc vào là thông qua phương thức Setter của lớp TextEditor, nơi chúng ta sẽ tạo một cá thể SpellChecker. Thành phần này sẽ được sử dụng để gọi phương thức Setter để khởi tạo các thuộc tính của TextEditor. Do đó , DI tồn tại trong hai biến thể chính:

1 Dependency Injection dựa trên Constructor

DI dựa trên Constructor được thực hiện khi container gọi một constructor của lớp với một tham số , mỗi đại diện cho một sự phụ thuộc vào lớp khác.

2 Dependency Injection dựa trên Setter

DI dựa trên Setter được thực hiện khi container đang gọi phương thức setter trên bean của bạn sau khi gọi một constructor không tham số hoặc một phương thức factory static không tham số để khởi tạo bean của bạn.

Bạn có thể trộn cả hai DI dựa trên Constructor và dựa trên Setter nhưng tốt nhất nguyên tắc chung là sử dụng đối số constructor cho các phụ thuộc bắt buộc và setters cho phụ thuộc không bắt buộc.

Code rõ ràng hơn với nguyên tắc DI và việc tách sẽ hiệu quả hơn khi các đối tượng được cung cấp các phụ thuộc của chúng. Đối tượng không tra cứu các phụ thuộc của nó và không biết vị trí hoặc lớp của các phụ thuộc , thay vào đó mọi thứ đều do Spring Framework đảm nhận.

Constructor-based Dependency Injection

DI dựa trên Constructor được hoàn thành khi container gọi một constructor của lớp với một số đối số , mỗi đối số đại diện cho một sự phụ thuộc vào lớp kia.

Example:

TextEditor.java

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

SpellChecker.java

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Beans.xml

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<constructor-arg ref = "spellChecker"/>

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

Kết quả:

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

Constructor arguments resolution

Có thể có sự không rõ ràng khi truyền các đối số cho constructor, trong trường hợp có nhiều hơn một tham số. Để giải quyết vấn đề này , thứ tự trong đó các tham số của constructor được định nghĩa trong định nghĩa một bean là thứ tự mà các tham số đó được cung cấp cho constructor thích hợp.

package x.y;

public class Foo {

public Foo(Bar bar, Baz baz) {

// ...

}

}

File cấu hình:

<beans>

<bean id = "foo" class = "x.y.Foo">

<constructor-arg ref = "bar"/>

<constructor-arg ref = "baz"/>

</bean>

<bean id = "bar" class = "x.y.Bar"/>

<bean id = "baz" class = "x.y.Baz"/>

</beans>

Một trường hợp trong đó chúng ta truyền các kiểu khác nhau cho constructor:

package x.y;

public class Foo {

public Foo(int year, String name) {

// ...

}

}

Container cũng có thể sử dụng đối xứng kiểu với các kiểu đơn giản, nếu bạn chỉ định rõ ràng kiểu của đối số phương thức khởi tạo bằng cách sử dụng thuộc tính type.

<beans>

<bean id = "exampleBean" class = "examples.ExampleBean">

<constructor-arg type = "int" value = "2001"/>

<constructor-arg type = "java.lang.String" value = "Zara"/>

</bean>

</beans>

Cuối cùng , cách tốt nhất để truyền tham số constructor, sử dụng thuộc tính index để chỉ định rõ ràng index của tham số constructor. Ở đây index dựa trên 0:

<beans>

<bean id = "exampleBean" class = "examples.ExampleBean">

<constructor-arg index = "0" value = "2001"/>

<constructor-arg index = "1" value = "Zara"/>

</bean>

</beans>

Một chú ý cuối cùng , trong trường hợp bạn đang truyền một tham chiếu tới một đối tượng, bạn cần sử dụng thuộc tính ref của <constructor-arg> và nếu bạn đang truyền một giá trị trực tiếp khi đó bạn nên sử dụng thuộc tính value.

Spring Setter-based Dependency Injection

DI dựa trên Setter được thực hiện bởi container gọi các phương thức setter trên bean của bạn sau khi gọi một phương thức khởi tạo không tham số hoặc phương thức static fatory không tham số để khởi tạo bean.

Example:

TextEditor.java

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

// a setter method to inject the dependency.

public void setSpellChecker(SpellChecker spellChecker) {

System.out.println("Inside setSpellChecker." );

this.spellChecker = spellChecker;

}

// a getter method to return spellChecker

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Bạn cần kiểm tra quy chuẩn đặt tên của phương thức setter. Để set một biến spellChecker chúng ta đang sử dụng phương thức setSpellChecker() rất giống với các POJO . Hãy để chúng tôi tạo nội dung của file lớp phụ thuộc khác SpellChecker.java:

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Beans.xml

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<property name = "spellChecker" ref = "spellChecker"/>

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

Bạn nên lưu ý sự khác nhau trong Beans.xml trong việc inject dựa trên constructor và injection dựa trên setter. Khác nhau ở phía trong phần tử <bean> nơi bạn sử dụng <constructor-arg> dành cho inject dựa trên constructor và <property> cho injection dựa trên setter.

Điểm quan trọng thứ hai để lưu ý là trong trường hợp bạn đang truyền một tham chiếu tới một đối tượng, bạn cần sử dụng thuộc tính ref của <property> và nếu bạn đang truyền một giá trị trực tiếp bạn nên sử dụng thuộc tính value.

Sau khi bạn hoàn thành việc tạo nguồn và file cấu hình bean, Kết quả:

Inside SpellChecker constructor.

Inside setSpellChecker.

Inside checkSpelling.

XML Configuration sử dụng p-namespace

Nếu bạn có nhiều phương thức setter , thì việc sử dụng p-namespace trong file cấu hình XML sẽ rất tiện lợi. Quan sát ví dụ của file cấu hình XML đơn giản với <property>:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "john-classic" class = "com.example.Person">

<property name = "name" value = "John Doe"/>

<property name = "spouse" ref = "jane"/>

</bean>

<bean name = "jane" class = "com.example.Person">

<property name = "name" value = "John Doe"/>

</bean>

</beans>

Cấu hình XML có thể được viết lại một cách sạch sẽ hơn sử dụng p-namespace như sau:

<?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:p = "http://www.springframework.org/schema/p"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "john-classic" class = "com.example.Person"

p:name = "John Doe"

p:spouse-ref = "jane"/>

</bean>

<bean name =" jane" class = "com.example.Person"

p:name = "John Doe"/>

</bean>

</beans>

Bạn nên lưu ý sự khác nhau trong giá trị nguyên thủy và tham chiếu đối tượng với p-namespace. –ref chỉ ra rằng đây không phải là một giá trị mà là một tham chiếu đến một bean khác.

**Injecting Inner Beans**

Bạn biết Java inner classes được định nghĩa với scope của classes khác, tương tự, inner beans là beans được xác định trong phạm vi của một bean khác. Do đó , một phần tử <bean/> bên trong các phần từ <property/> hoặc <constructor-arg/> được gọi là bên trong bean và nó được hiển thị bên dưới:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "outerBean" class = "...">

<property name = "target">

<bean id = "innerBean" class = "..."/>

</property>

</bean>

</beans>

Ví dụ:

TextEditor.java

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

// a setter method to inject the dependency.

public void setSpellChecker(SpellChecker spellChecker) {

System.out.println("Inside setSpellChecker." );

this.spellChecker = spellChecker;

}

// a getter method to return spellChecker

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

SpellChecker.java

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Beans.xml:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean using inner bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<property name = "spellChecker">

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"/>

</property>

</bean>

</beans>

Kết quả:

Inside SpellChecker constructor.

Inside setSpellChecker.

Inside checkSpelling.

**Injecting Collection**

Bạn đã biết cách cấu hình dữ liệu nguyên thủy sử dụng thuộc tính value và tham chiếu đối tượng sử dụng thuộc tính ref của <property> trong file cấu hình Bean của bạn. Cả 2 trường hợp đều chuyển giá trị duy nhất cho 1 bean.

Bây giờ , Nếu bạn muốn chuyển các giá trị số nhiều như Collection Java như List , Set , Map và Properties.

Để xử lý tình huống , Spring đề nghị 4 kiểu cấu hình collection:

1. <list> :

Nó giúp wiring đưa vào danh sách các giá trị, chấp nhận trùng lặp

1. <set>:

Nó giúp wiring một set của giá trị nhưng không chấp nhận trùng lặp

1. <map>

Nó có thể được sử dụng để inject một collection của cặp name-value khi name và value có thể là bất kì kiểu gì.

1. <props>

Nó có thể được sử dụng để inject một collection của cặp name-value khi name và value là Strings.

Bạn có thể sử dụng <list> hoặc <set> để kết nối bất kì triển khai của Java.util.Collection hoặc một mảng.

Bạn sẽ gặp 2 tình huống (a) Truyển các giá trị trực tiếp của collection và (b) Truyền một tham chiếu của bean làm một trong các phần tử của collection.

Ví dụ:

JavaCollection.java

package com.tutorialspoint;

import java.util.\*;

public class JavaCollection {

List addressList;

Set addressSet;

Map addressMap;

Properties addressProp;

// a setter method to set List

public void setAddressList(List addressList) {

this.addressList = addressList;

}

// prints and returns all the elements of the list.

public List getAddressList() {

System.out.println("List Elements :" + addressList);

return addressList;

}

// a setter method to set Set

public void setAddressSet(Set addressSet) {

this.addressSet = addressSet;

}

// prints and returns all the elements of the Set.

public Set getAddressSet() {

System.out.println("Set Elements :" + addressSet);

return addressSet;

}

// a setter method to set Map

public void setAddressMap(Map addressMap) {

this.addressMap = addressMap;

}

// prints and returns all the elements of the Map.

public Map getAddressMap() {

System.out.println("Map Elements :" + addressMap);

return addressMap;

}

// a setter method to set Property

public void setAddressProp(Properties addressProp) {

this.addressProp = addressProp;

}

// prints and returns all the elements of the Property.

public Properties getAddressProp() {

System.out.println("Property Elements :" + addressProp);

return addressProp;

}

}

MainApp.java:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

JavaCollection jc=(JavaCollection)context.getBean("javaCollection");

jc.getAddressList();

jc.getAddressSet();

jc.getAddressMap();

jc.getAddressProp();

}

}

Beans.xml:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for javaCollection -->

<bean id = "javaCollection" class = "com.tutorialspoint.JavaCollection">

<!-- results in a setAddressList(java.util.List) call -->

<property name = "addressList">

<list>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</list>

</property>

<!-- results in a setAddressSet(java.util.Set) call -->

<property name = "addressSet">

<set>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</set>

</property>

<!-- results in a setAddressMap(java.util.Map) call -->

<property name = "addressMap">

<map>

<entry key = "1" value = "INDIA"/>

<entry key = "2" value = "Pakistan"/>

<entry key = "3" value = "USA"/>

<entry key = "4" value = "USA"/>

</map>

</property>

<!-- results in a setAddressProp(java.util.Properties) call -->

<property name = "addressProp">

<props>

<prop key = "one">INDIA</prop>

<prop key = "one">INDIA</prop>

<prop key = "two">Pakistan</prop>

<prop key = "three">USA</prop>

<prop key = "four">USA</prop>

</props>

</property>

</bean>

</beans>

Kết quả:

List Elements :[INDIA, Pakistan, USA, USA]

Set Elements :[INDIA, Pakistan, USA]

ap Elements :{1 = INDIA, 2 = Pakistan, 3 = USA, 4 = USA}

Property Elements :{two = Pakistan, one = INDIA, three = USA, four = USA}

Injecting Bean References:

Định nghĩa Bean sau đây sẽ giúp bạn hiểu cách đưa các tham chiếu bean như là một phần tử của collection. Thậm chí , bạn có thể kết hợp tất cả các tham chiếu và giá trị với nhau:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Bean Definition to handle references and values -->

<bean id = "..." class = "...">

<!-- Passing bean reference for java.util.List -->

<property name = "addressList">

<list>

<ref bean = "address1"/>

<ref bean = "address2"/>

<value>Pakistan</value>

</list>

</property>

<!-- Passing bean reference for java.util.Set -->

<property name = "addressSet">

<set>

<ref bean = "address1"/>

<ref bean = "address2"/>

<value>Pakistan</value>

</set>

</property>

<!-- Passing bean reference for java.util.Map -->

<property name = "addressMap">

<map>

<entry key = "one" value = "INDIA"/>

<entry key = "two" value-ref = "address1"/>

<entry key = "three" value-ref = "address2"/>

</map>

</property>

</bean>

</beans>

Để sử dụng định nghĩa bean ở trên , bạn cần định nghĩa các phương thức setter của mình theo cách mà chúng cũng có thể xử lý các tham chiếu:

Injecting null and empty String values:

Nếu bạn cần để pass một string empty như một giá trị :

<bean id = "..." class = "exampleBean">

<property name = "email" value = ""/>

</bean>

Nếu bạn cần pass một giá trị null:

<bean id = "..." class = "exampleBean">

<property name = "email"><null/></property>

</bean>

**Beans Auto-Wiring**

Bạn đã học cách khai báo beans sử dụng phần tử <bean> và inject <bean> sử dụng <constructor-arg> và <property> trong cấu hình file XML.

Spring container có thể autowire các mối quan hệ giữa các beans với nhau mà không cần sử dụng <constructor-arg> và <property> , giúp cắt giảm lượng cấu hình XML bạn viết cho một ứng dụng lớn dựa trên Spring.

Autowiring Modes

Sau đây là các chế độ autowiring có thể được sử dụng để chỉ Spring container sử dụng autowiring cho dependency injection.Bạn sử dụng thuộc tính autowire của phần tử <bean/> để chỉ định chế độ autowire cho định nghĩa bean.

1. no

đây là cài đặt mặc định có nghĩa không autowiring và bạn nên sử dụng tham chiếu bean rõ ràng cho wiring. Bạn không có gì để làm đặc biệt wiring này. Đây là những gì đã có trong chương Dependency Injection.

1. byName

Autowiring bằng thuộc tính name. Spring container tìm kiếm các thuộc tính của beans mà thuộc tính autowire được đặt là byName trong file cấu hình XML. Sau đó , nó cố gắng khớp và nối các thuộc tính của nó với các bean được định nghĩa bởi cùng tên trong file cấu hình.

1. byType

Autowiring bằng thuộc tính kiểu dữ liệu. Spring container tìm kiếm tại các thuộc tính của bean mà thuộc tính autowire được set là byType trong file cấu hình XML. Sau đó nó cố gắng khớp và nối một thuộc tính nếu kiểu của nó khớp với chính xác một tên beans trong file cấu hình. Nếu nhiều hơn một beans như vậy tồn tại, một exception được thrown ra.

1. Constructor

Tương tự như byType , nhưng kiểu áp dụng cho các đối số của hàm khởi tạo. Nếu không có chính xác một bean của kiểu đối số hàm khởi tạo trong container, một lỗi xảy ra.

1. Autodetect

Đầu tiên Spring sẽ cố gắng để wire sử dụng autowire bằng constructor , nếu nó không hoạt động, Spring sẽ cố gắng để autowire bằng byType.

Bạn có thể sử dụng byType hoặc Constructor autowiring mode để wire mảng và các kiểu collections khác.

Limitations with autowiring

Autowiring hoạt động tốt nhất khi nó được sử dụng nhất quán trong một project. Nếu autowiring không được sử dụng chung. Nó có thể gây nhầm lẫn cho các nhà phát triển khi sử dụng nó để chỉ một hoặc hai định nghĩa bean. Dù vậy , autowiring có thể làm giảm đáng kể nhu cầu chỉ định thuộc tính hoặc đối số constructor nhưng bạn nên xem xét các hạn chế và bất lợi của autowiring trước khi sử dụng chúng.

1. Overriding possibility

Bạn có thể chỉ định các phần phụ thuộc bằng cách sử dụng cài đặt <constructor-arg> và <property> sẽ luôn ghi đè autowiring.

1. Primitive data types

Bạn không thể autowire thuộc tính đơn giản như nguyên thủy, Strings và Classes

1. Confusing nature

Autowiring kém chính xác hơn so với wiring rõ ràng, vì vậy nếu có thể , bạn nên sử dụng wiring rõ ràng.

Autowiring – byName

Mode này chỉ định autowiring thuộc tính name. Spring container tìm kiếm beans trên thuộc tính auto-wire được set tới byName trong file cấu hình XML. Nó cố gắng match và wire thuộc tính của nó với các bean được định nghĩa cùng tên trong file cấu hình.Nếu các match được tìm thấy , nó sẽ inject các beans đó . Ngược lại , các bean sẽ ko được wire.

Ví dụ, nếu một bean định nghĩa được set tới autowire byName trong file cấu hình, và nó chứa thuộc tính spellChecker điều đó có nghĩa nó có phương thức setSpellChecker() , Spring tìm kiếm một bean định nghĩa tên spellChecker và sử dụng nó để set thuộc tính. Tuy nhiên bạn vẫn có thể kết nối các thuộc tính còn lại bằng cách sử dụng <property>:

TextEditor.java

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

private String name;

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

SpellChecker.java

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker() {

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Beans.xml

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<property name = "spellChecker" ref = "spellChecker" />

<property name = "name" value = "Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

Nhưng nếu bạn sẽ sử dụng autowiring ‘byName’ , khi đó file cấu hình XML của bạn sẽ như sau:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor" autowire = "byName">

<property name = "name" value = "Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

Kết quả:

Inside SpellChecker constructor.

Inside checkSpelling.

Autowiring – byType

Mode này chỉ định autowiring theo loại thuộc tính. Spring container tìm kiếm các bean mà thuộc tính autowire được set tới byType trong file cấu hình XML. Nó cố gắng match và wire một thuộc tính nếu kiểu của nó khớp chính xác một trong các tên bean trong file cấu hình. Nếu match được tìm thấy nó sẽ inject beans đó. Nếu không bean sẽ không được wire.

Ví dụ: nếu định nghĩa bean được đặt thành autowire byType trong file caaush ình và nó chứa thuộc tính spellChecker của loại SpellChecker, Spring sẽ tìm định nghĩa bean có tên SpellChecker và sử dụng nó để đặt thuộc tính. Tuy nhiên , bạn vẫn có thể kết nối các thuộc tính còn lại bằng cách sử dụng <property>:

TextEditor.java

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

private String name;

public void setSpellChecker( SpellChecker spellChecker ) {

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

SpellChecker.java

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Beans.xml

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<property name = "spellChecker" ref = "spellChecker" />

<property name = "name" value = "Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

Sử dụng autowiring byType:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor" autowire = "byType">

<property name = "name" value = "Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "SpellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

Kết quả:

Inside SpellChecker constructor.

Inside checkSpelling.

Autowiring by Constructor

Mode này rất giống với byType, nhưng nó áp dụng cho các tham số của hàm tạo. Spring container xem xét các bean mà trên đó thuộc tính autowire được thiết lập constructor trong file cấu hình XML. Sau đó, nó cố gắng match và wire tham số của phương thức constructor với chính xác một trong các tên bean trong file cấu hình. Nếu match được tìm thấy , nó sẽ inject các bean đó. Nếu không, các bean không được wire.

Ví dụ: nếu định nghĩa bean được đặt thành autowire bởi constructor trong file cấu hình và nó có một constructor với một trong các tham số của kiểu SpellChecker, Spring sẽ tìm định nghĩa bean có tên SpellChecker và sử dụng nó để đặt đối số của hàm tạo. Tuy nhiên bạn vẫn có thể nối các đối số còn lại bằng cách sử dụng <constructor-arg>:

TextEditor.java

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

private String name;

public TextEditor( SpellChecker spellChecker, String name ) {

this.spellChecker = spellChecker;

this.name = name;

}

public SpellChecker getSpellChecker() {

return spellChecker;

}

public String getName() {

return name;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

SpellChecker.java

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Beans.xml

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<constructor-arg ref = "spellChecker" />

<constructor-arg value = "Generic Text Editor"/>

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

Sử dụng autowiring byConstructor :

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor"

autowire = "constructor">

<constructor-arg value = "Generic Text Editor"/>

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "SpellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

Kết quả:

Inside SpellChecker constructor.

Inside checkSpelling.

**Annotation Based Configuration**

Bắt đầu từ Spring 2.5 có thể cấu hình dependency injection sử dụng annotations.

Vì vậy, thay vì sử dụng XML để mô tả wiring một bean, bạn có thể đưa cấu hình bean thành các thành phần lớp chính nó bằng cách sử dụng annotations trên khai báo lớp , phương thức hoặc trường có liên quan.

Annotation injection được thực hiện trước khi XML injection. Do đó , cấu hình sau sẽ ghi đè cấu hình cũ cho các thuộc tính có wire thông qua cả hai cách tiếp cận.

Annotation wiring không được mặc định bật trong Spring container. Vì thế , trước khi chúng ta có thể sử dụng wiring dựa trên annotation, chúng ta cần kích hoạt nó trong file cấu hình Spring. Vì thế quan sát file cấu hình sau đây trong trường hợp bạn muốn sử dụng bất kì annotation trong ứng dụng Spring.

<?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-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- bean definitions go here -->

</beans>

Sau khi<context:annotation-config/> được cấu hình , bạn có thể bắt đầu annotating code của bạn để chỉ ra rằng Spring sẽ tự động wire các giá trị vào các thuộc tính , phương thức và hàm khởi tạo.

Một vài annotation quan trọng sau đây:

1. @Required

@Required annotation áp dụng cho các phương thức setter thuộc tính bean.

1. @Autowired

@Autowired annotation có thể áp dụng để phương thức setter thuộc tính bean , phương thức không phải setter, hàm khởi tạo và thuộc tính.

1. @Qualifier

@Qualifier annotation cùng với @Autowired có thể được sử dụng để loại bỏ sự nhầm lẫn bằng cách chỉ định chính xác bean sẽ được wire.

1. JSR-250 Annotations

Spring hỗ trợ JSR-250 dựa trên annotations bao gồm @Resource

@PostConstruct và @PreDestroy annotations.

@Required Annotation

@Required annotation áp dụng cho các phương thức setter thuộc tính bean và nó chỉ ra rằng thuộc tính bean bị ảnh hưởng phải đc điền vào tệp cấu hình XML tại thời điểm cấu hình. Nếu không , container sẽ ném ra một ngoại lệ BeanInitializationException:

Student.java:

package com.tutorialspoint;

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

public class Student {

private Integer age;

private String name;

@Required

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

@Required

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

Student student = (Student) context.getBean("student");

System.out.println("Name : " + student.getName() );

System.out.println("Age : " + student.getAge() );

}

}

Beans.xml:

<?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-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for student bean -->

<bean id = "student" class = "com.tutorialspoint.Student">

<property name = "name" value = "Zara" />

<!-- try without passing age and check the result -->

<!-- property name = "age" value = "11"-->

</bean>

</beans>

Kết quả sẽ là BeanInitializationException và in ra messages:

Property 'age' is required for bean 'student'

Tiếp theo bạn có thể thay đổi beans.xml:

<?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-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for student bean -->

<bean id = "student" class = "com.tutorialspoint.Student">

<property name = "name" value = "Zara" />

<property name = "age" value = "11"/>

</bean>

</beans>

Kết quả:

Name : Zara

Age : 11

@Autowired Annotation

@Autowired cung cấp khả năng kiểm soát chi tiết hơn đối với vị trí và cách thức autowiring nên được thực hiện. @Autowired có thể được sử dụng để autowire bean trên phương thức setter giống như @Required , constructor, một thuộc tính hoặc các phương thức có tên tùy ý và hoặc nhiều tham số.

@Autowired trên phương thức Setter:

Bạn có thể sử dụng @Autowired trên phương thức setter để loại bỏ <property> trong file cấu hình XML. Khi Spring tìm thấy @Autowired được sử dụng với các phương thức setter, nó sẽ cố gắng thực hiện tự động tạo byType trên phương thức.

TextEditor.java

package com.tutorialspoint;

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

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker( ) {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

SpellChecker.java

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Beans.xml:

<?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.xsd">

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker">

</bean>

</beans>

Kết quả:

Inside SpellChecker constructor.

Inside checkSpelling.

@Autowired trên Properties

Bạn có thể sử dụng @Autowired trên properties để loại bỏ các phương thức setter. Khi bạn chuyển các giá trị của thuộc tính autowired bằng cách sử dụng <property> , Spring sẽ tự động gán các thuộc tính đó với các giá trị hoặc tham chiếu đã truyền. Vì vậy , việc sử dụng @Autowired trên thuộc tính sẽ như sau:

TextEditor.java

package com.tutorialspoint;

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

public class TextEditor {

@Autowired

private SpellChecker spellChecker;

public TextEditor() {

System.out.println("Inside TextEditor constructor." );

}

public SpellChecker getSpellChecker( ){

return spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

Beans.xml:

<?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.xsd">

<context:annotation-config/>

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker">

</bean>

</beans>

Kết quả:

Inside TextEditor constructor.

Inside SpellChecker constructor.

Inside checkSpelling.

@Autowired trên Constructor

Bạn có thể áp dụng @Autowired cho constructor. @Autowired trên constructor chỉ ra rằng constructor phải được tự động load khi tạo bean, ngày cả khi không có <constructor-arg> nào được sử dụng trong file cấu hình bean XML:

TextEditor.java

package com.tutorialspoint;

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

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

Beans.xml

<?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.xsd">

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker">

</bean>

</beans>

Kết quả:

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

@Autowired với (requied = false)

Mặc định , @Autowired ngụ ý rằng phụ thuộc được yêu cầu tương tự như @Required, tuy nhiên , bạn có thể tắt hành vi mặc định bằng cách sử dụng required = false với @Autowired.

Ví dụ sau sẽ hoạt động ngày cả khi bạn không chuyển bất kì giá trị nào cho thuộc tính age nhưng nó vẫn sẽ yêu cầu thuộc tính name.

Student.java:

package com.tutorialspoint;

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

public class Student {

private Integer age;

private String name;

@Autowired(required=false)

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

@Autowired

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

@Qualifier Annotation

Có thể có tình huống khi bạn tạo nhiều bean cùng loại và chỉ muốn nối một trong số chúng với một thuộc tính. Trong trường hợp như vậy, bạn có thể sử dụng @Qualifier cùng với @Autowired để loại bỏ sự nhầm lẫn bằng cách chỉ định bean chính xác nào sẽ đc wire. Sau đây là một ví dụ:

Student.java

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

Profile.java

package com.tutorialspoint;

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

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

public class Profile {

@Autowired

@Qualifier("student1")

private Student student;

public Profile(){

System.out.println("Inside Profile constructor." );

}

public void printAge() {

System.out.println("Age : " + student.getAge() );

}

public void printName() {

System.out.println("Name : " + student.getName() );

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

Profile profile = (Profile) context.getBean("profile");

profile.printAge();

profile.printName();

}

}

Beans.xml

<?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-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for profile bean -->

<bean id = "profile" class = "com.tutorialspoint.Profile"></bean>

<!-- Definition for student1 bean -->

<bean id = "student1" class = "com.tutorialspoint.Student">

<property name = "name" value = "Zara" />

<property name = "age" value = "11"/>

</bean>

<!-- Definition for student2 bean -->

<bean id = "student2" class = "com.tutorialspoint.Student">

<property name = "name" value = "Nuha" />

<property name = "age" value = "2"/>

</bean>

</beans>

Kết quả:

Inside Profile constructor.

Age : 11

Name : Zara

Spring JSR-250 Annotations

Spring cũng hỗ trợ JSR-250 dựa trên annotation bao gồm @PostConstruct, @PreDestroy và @Resource. Mặc dù những annotation này không thực sự bắt buộc vì bạn đã có những annotation khác:

@PostConstruct và @PreDestroy

Để định nghĩa thiết lập và chia nhỏ cho một bean, chúng ta chỉ cần khai báo <bean> với các tham số init-method và / hoặc destroy-method. Thuộc tính init-method chỉ đinh một phương thức sẽ được gọi trên bean ngay sau khi khởi tạo. Tương tự , destroy-method chỉ định một phương thức được gọi ngay trước khi một bean bị xóa khỏi container.

Bạn có thể sử dụng @PostConstruct làm sự thay thế callback khởi tạo và @PreDestroy như một sự thay thế của callback hủy :

HelloWorld.java

package com.tutorialspoint;

import javax.annotation.\*;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public String getMessage(){

System.out.println("Your Message : " + message);

return message;

}

@PostConstruct

public void init(){

System.out.println("Bean is going through init.");

}

@PreDestroy

public void destroy(){

System.out.println("Bean will destroy now.");

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

AbstractApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

context.registerShutdownHook();

}

}

Beans.xml:

<?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-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld"

init-method = "init" destroy-method = "destroy">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>

Kết quả:

Bean is going through init.

Your Message : Hello World!

Bean will destroy now.

@Resource

Bạn có thể sử udjng @Resource trên trường hoặc phương thức setter và nó hoạt động tương tự. @Resource lấy thuộc tính ‘name’ sẽ được giải thích như tên của bean để inject. Nó tuân theo byName autowiring

package com.tutorialspoint;

import javax.annotation.Resource;

public class TextEditor {

private SpellChecker spellChecker;

@Resource(name = "spellChecker")

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker(){

return spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

Nếu không ‘name’ được chỉ định rõ ràng, tên mặc định được chuyển đến trường name hoặc phương thức setter. Trong trường hợp của một trường , nó lấy tên trường , trong trường hợp phương thức setter, nó lấy thuộc tính tên bean.

**Java Based Configuration**

Đến đây bạn đã thấy cách chúng ta cấu hình Spring beans sử dụng file cấu hình XML. Nếu bạn cảm thấy thỏa mái với cấu hình XML. Sau đó nó thực sự không bắt buộc để học cách tiến trình cấu hình dựa trên Java bạn sẽ đạt được kết quả tương tự bằng cách sử dụng một trong các cấu hình có sẵn.

Lựa chọn cấu hình dựa trên Java cho phép bạn viết hầu hết cấu hình Spring của mình mà không cần XML nhưng với sự trợ giúp của một số annotations dựa trên Java được giải thích như sau:

@Configuration và @Bean Annotations

Annotating một class với @Configuration chỉ ra rằng lớp có thể được sử dụng bởi Spring IoC container làm nguồn định nghĩa bean. @Bean annotation cho Spring biết rằng một phương thức được chú thích bằng @Bean sẽ trả về một đối tượng nên được đăng ký dưới dạng bean trong Spring applicationcontext. Lớp @Configuration đơn giản nhất có thể sẽ như sau:

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

Đoạn code trên tương đương cấu hình XML sau:

<beans>

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" />

</beans>

Tên phương thức được chú thích với @Bean hoạt động như ID bean và nó tạo và trả về bean thực tế. Lớp cấu hình của bạn có thể có một khai báo nhiều hơn một @Bean. Sau khi lớp cấu hình của bạn được định nghĩa, bạn có thể load và cung cấp chúng tới Spring container sử dụng AnnotationConfigApplicationContext như sau:

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

helloWorld.getMessage();

}

Bạn có thể load các lớp cấu hình khác nhau như sau:

public static void main(String[] args) {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

Ví dụ:

HelloWorldConfig.java

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

HelloWorld.java

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApp {

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

helloWorld.getMessage();

}

}

Kết quả:

Your Message : Hello World!

Injecting Bean Dependencies

Khi @Beans có nhiều phụ thuộc vs bean khác, diễn đạt rằng sự phụ thuộc đơn giản như có một phương thức bean gọi một phương thức khác:

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class AppConfig {

@Bean

public Foo foo() {

return new Foo(bar());

}

@Bean

public Bar bar() {

return new Bar();

}

}

Foo bean nhận được một tham chiếu đến Bar thông qua việc constructor injection. Bây giờ chúng ta hãy xem xét một ví dụ làm việc khác:

TextEditorConfig.java

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class TextEditorConfig {

@Bean

public TextEditor textEditor(){

return new TextEditor( spellChecker() );

}

@Bean

public SpellChecker spellChecker(){

return new SpellChecker( );

}

}

TextEditor.java

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

SpellChecker.java

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApp {

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(TextEditorConfig.class);

TextEditor te = ctx.getBean(TextEditor.class);

te.spellCheck();

}

}

Kết quả:

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

@Import Annotation

@Import annotation cho phép tải định nghĩa @Bean từ lớp cấu hình khác. Quan sát một lớp ConfigA:

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

Bạn có thể import khai báo Bean trong khai báo Bean khác:

@Configuration

@Import(ConfigA.class)

public class ConfigB {

@Bean

public B b() {

return new B();

}

}

Bây giờ , thay vì chỉ định cả ConfigA.class và ConfigB.class khi khởi tạo context , chỉ cần ConfigB được cung cấp:

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

// now both beans A and B will be available...

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

Lifecycle Callbacks

Annotation @Bean hỗ trợ việc chỉ định các phương thức callback khởi tạo và phá hủy tùy ý, giống như các thuộc tính init-method và destroy-method của phần tử bean trong Spring XML.

public class Foo {

public void init() {

// initialization logic

}

public void cleanup() {

// destruction logic

}

}

@Configuration

public class AppConfig {

@Bean(initMethod = "init", destroyMethod = "cleanup" )

public Foo foo() {

return new Foo();

}

}

Specifying Bean Scope

Scope mặc định là singleton, nhưng bạn có thể override nó với @Scope annotation dưới đây:

@Configuration

public class AppConfig {

@Bean

@Scope("prototype")

public Foo foo() {

return new Foo();

}

}

**Event Handling in Spring**

Bạn đã biết toàn bộ chương cốt lõi của Spring là ApplicationContext, nó quản lý vòng đời hoàn chỉnh của bean. ApplicationContext xuất bản các loại sự kiện nhất định khi load các bean.

Ví dụ, một ContextStartedEvent được xuất bản khi context được bắt đầu và ContextStoppedEvent được xuất bản khi context bị dừng.

Xử lý Sự kiện trong ApplicationContext được cung cấp thông qua ApplicationEvent và ApplicationListener interface. Do đó , nếu một bean triển khai ApplicationListener , thì mỗi khi một ApplicationEvent được xuất bản lên ApplicationContext, bean đó sẽ được thông báo.

Spring cung cấp sự kiện tiêu chuẩn sau:

1. ContextRefreshedEvent

Sự kiện này được xuất bản khi ApplicationContext được khởi tạo hoặc tái tạo. Nó cũng có thể được đánh thức bằng cách sử dụng phương thức refresh() trên ConfigurableApplicationContext interface.

1. ContextStartedEvent

Sự kiện này được xuất bản khi ApplicationContext được bắt đầu bằng cách sử dụng phương thức start() trên ConfigurableApplicationContext interface. Bạn có thể thăm dò cơ sở dữ liệu của mình hoặc bạn có thể restart bất kì ứng dụng nào đã dừng sau khi nhận được sự kiện này.

1. ContextStoppedEvent

Sự kiện này được xuất bản khi ApplicationContext được dừng bằng cách sử dụng phương thức stop() trên ConfigurableApplicationContext interface. Bạn có thể làm công việc quản lý bắt buộc sau khi nhận sự kiện này.

1. ContextClosedEvent

Sự kiện này được xuất bản khi ApplicationContext được đóng bằng cách sử dụng phương thức close() trên ConfigurableApplicationContext interface. Một context đã đóng đến hết vòng đời của nó , nó không thể được tái tạo hoặc khởi động lại.

1. RequestHandledEvent

Đây là sự kiện chỉ định cho web nói với tất cả các bean rằng một HTTP request đã được thực hiện.

Việc xử lý sự kiện của Spring là đơn luồng vì thế nếu một sự kiện được xuất bản, cho đến khi và trừ khi tất cả người nhận nhận được tin nhắn , các quá trình sẽ bị chặn và luồng sẽ không tiếp tục. Do đó , cần cẩn thận khi thiết kế ứng dụng của bạn nếu xử lý sự kiện được sử dụng.

Listening to Context Events

Để lắng nghe một sự kiện context, một bean nên triển khai ApplicationListener interface có một phương thức onApplicationEvent(). Vì thế , chúng ta viết một ví dụ để xem cách các sự kiện lan truyền và cách bạn có thể đặt code của mình thực hiện tác vụ cần thiết dựa trên các sự kiện nhất định.

HelloWorld.java

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

CStartEventHandler.java

package com.tutorialspoint;

import org.springframework.context.ApplicationListener;

import org.springframework.context.event.ContextStartedEvent;

public class CStartEventHandler

implements ApplicationListener<ContextStartedEvent>{

public void onApplicationEvent(ContextStartedEvent event) {

System.out.println("ContextStartedEvent Received");

}

}

CstopEventHandler.java

package com.tutorialspoint;

import org.springframework.context.ApplicationListener;

import org.springframework.context.event.ContextStoppedEvent;

public class CStopEventHandler

implements ApplicationListener<ContextStoppedEvent>{

public void onApplicationEvent(ContextStoppedEvent event) {

System.out.println("ContextStoppedEvent Received");

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ConfigurableApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

// Let us raise a start event.

context.start();

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

// Let us raise a stop event.

context.stop();

}

}

Beans.xml

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">

<property name = "message" value = "Hello World!"/>

</bean>

<bean id = "cStartEventHandler" class = "com.tutorialspoint.CStartEventHandler"/>

<bean id = "cStopEventHandler" class = "com.tutorialspoint.CStopEventHandler"/>

</beans>

Kết quả:

ContextStartedEvent Received

Your Message : Hello World!

ContextStoppedEvent Received

Nếu bạn thích, bạn có thể xuất bạn sự kiện tự tạo của bạn và sau đó bạn có thể nắm bắt tương tự để thực hiện bất kỳ hành động nào đối với các sự kiện tự tạo đó. Nếu bạn quan tâm đến việc viết các sự kiện tự tạo của mình, bạn có thể kiểm tra Custom Events trong Spring.

**Custom Events in Spring**

Các bước để cần thực hiện để viết và xuất bản sự kiện tự tạo của bạn:

Sau đây là hướng dẫn để viết , xuất bản và xử lý Custom Spring Event:

CustomEvent.java

package com.tutorialspoint;

import org.springframework.context.ApplicationEvent;

public class CustomEvent extends ApplicationEvent{

public CustomEvent(Object source) {

super(source);

}

public String toString(){

return "My Custom Event";

}

}

CustomEventPublisher.java

package com.tutorialspoint;

import org.springframework.context.ApplicationEventPublisher;

import org.springframework.context.ApplicationEventPublisherAware;

public class CustomEventPublisher implements ApplicationEventPublisherAware {

private ApplicationEventPublisher publisher;

public void setApplicationEventPublisher (ApplicationEventPublisher publisher) {

this.publisher = publisher;

}

public void publish() {

CustomEvent ce = new CustomEvent(this);

publisher.publishEvent(ce);

}

}

CustomEventHandler.java

package com.tutorialspoint;

import org.springframework.context.ApplicationListener;

public class CustomEventHandler implements ApplicationListener<CustomEvent> {

public void onApplicationEvent(CustomEvent event) {

System.out.println(event.toString());

}

}

MainApp.java

package com.tutorialspoint;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ConfigurableApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

CustomEventPublisher cvp =

(CustomEventPublisher) context.getBean("customEventPublisher");

cvp.publish();

cvp.publish();

}

}

Beans.xml:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "customEventHandler" class = "com.tutorialspoint.CustomEventHandler"/>

<bean id = "customEventPublisher" class = "com.tutorialspoint.CustomEventPublisher"/>

</beans>

Kết quả:

y Custom Event

y Custom Event

**AOP with Spring Framework**

Một trong những thành phần chính của Spring Framework là Lập trình hướng khía cạnh AOP framework. Lập trình hướng theo khía cạnh đòi hỏi phải chia logic chương trình thành các phần riêng biệt được gọi là mối quan tâm. Các chức năng trải dài nhiều điểm của một ứng dụng được gọi là các mối quan tâm xuyên suốt và những mối quan tâm xuyên suốt này tách biệt về mặt khái niệm với logic nghiệp vụ của ứng dụng.

Có nhiều ví dụ điển hình phổ biến về các khía cạnh như logging , auditing, declarative transactions, security, caching,…

Đơn vị quan trọng của modul trong OOP là lớp trong khi AOP , đơn vị modul là khía cạnh. Dependency Injection giúp bạn tách các đối tượng ứng dụng của mình khỏi nhau và AOP giúp bạn loại bỏ các mối quan tâm xuyên suốt khỏi các đối tượng mà chúng ảnh hưởng. AOP giống như các trigger trong ngôn ngữ lập trình như Perl, .NET , Java và các ngôn ngữ khác.

Modul Spring AOP cung cấp các interceptors để chặn một ứng dụng. Ví dụ: Khi một phương thức được thực thi, bạn có thể thêm chức năng trước và sau khi thực thi phương thức.

AOP Terminologies

Trước khi bạn làm việc với AOP , chúng ta hãy làm quen với các khai niệm và thuật ngữ AOP. Những điều khoản này không dành riêng cho Spring , chúng liên quan đến AOP.

1. Aspect

Đây là một modul có một set các API cung cấp yêu cầu xuyên suốt.

Ví dụ, một modul Logging sẽ được gọi là 1 khía cạnh AOP cho việc logging. Một ứng dụng có thể có nhiều khía cạnh dựa trên yêu cầu.

1. Join point

Điều này thể hiện một điểm trong ứng dụng của bạn nơi bạn có thể bổ sung khía cạnh AOP. Bạn cũng có thể nói , đó là vị trí thực tế trong ứng dụng nơi một hành động sẽ được thực hiện bằng cách sử dụng Spring AOP framework

1. Advice

Đây là hành động thực tế đạt được trước hoặc sau khi phương thức thực thi. Đây là một phần thực tế của code được gọi xuyên suốt khi chương trình thực thi bởi AOP.

1. Pointcut

Đây là một set của một hoặc nhiều join point nơi một advice nên được thực thi. Bạn có thể chỉ định pointcuts sử dụng expressions hoặc patterns chúng ta sẽ xem trong ví dụ AOP.

1. Introduction

Một introduction cho phép bạn thêm nhiều phương thức hoặc thuộc tính tới những lớp đang tồn tại.

1. Target object

Đối tượng đang được tư vấn bởi một hoặc nhiều khía cạnh. Đối tượng này sẽ luôn luôn là một đối tượng được hỗ trợ, còn được gọi là đối tượng được tư vấn.

1. Weaving

Weaving là tiến trình liên kết các khía cạnh với các loại ứng dụng hoặc đối tượng khác để tạo ra một đối tượng được tư vấn. Điều này có thể được thực hiện tại thời điểm biên dịch , thời gian tải hoặc trong thời gian chạy.

Types of Advice

Khía cạnh Spring có thể làm việc với 5 loại advice đề cập dưới đây:

1. before

Chạy advice trước khi phương thức thực thi

1. after

Chạy advice sau khi phương thức thực thi, bất kể kết quả của nó.

1. after-returning

Chạy advice sau khi 1 phương thức thực thi chỉ nếu phương thức hoàn thành thành công.

1. after-throwing

Chạy advice sau khi 1 phương thức thực thi chỉ nến phương thức kết thúc bởi một exception

1. around

Chạy advice trước và sau khi phương thức advised được gọi.

Custom Aspects Implementation

Spring hỗ trợ kiểu annotation @AspectJ và cách tiếp cận dựa trên schema để triển khai các khía cạnh tùy chỉnh. Hai cách tiếp cận này đã được giải thích chi tiết trong các phần sau:

1. XML Schema based

Những khía cạnh được triển khai sử dụng các lớp thông thường cũng với cấu hình dựa trên XML

1. @AspectJ based

@AspectJ đề cập đến một kiểu khai báo các khía cạnh như các lớp Java thông thường được chú thích bằng các chú thích Java 5.

**JDBC Framework**

Trong khi làm việc với database sử dụng JDBC cũ , việc viết code không cần thiết để xử lý các ngoại lệ, mở và đóng các kết nối cơ sở dữ liệu trở nên cồng kềnh…

Tuy nhiên , Spring JDBC Framework xử lý tất cả các chi tiết cấp thấp bắt đầu từ việc mở kết nối, chuẩn bị và thực thi câu lệnh SQL, xử lý exceptions , xử lý các transactions và cuối cùng là đóng kết nối.

Vì vậy , những gì bạn phải làm chỉ là định nghĩa các tham số kết nối và chỉ định câu lệnh SQL được thực thi và thực hiện công việc cần thiết cho mỗi lần lặp trong khi tìm nạp dữ liệu từ cơ sở dữ liệu.

Spring JDBC cung cấp một số cách tiếp cận và các lớp tương ứng khác nhau để giao tiếp với cơ sở dữ liệu. Tôi sẽ sử dụng cách tiếp cận cổ điện và phổ biến nhất sử dụng lớp JdbcTemplate của framework. Đây là lớp framework trung tâm quản lý tất cả các giao tiếp cơ sở dữ liệu và xử lý ngoại lệ.

JdbcTemplate Class

Lớp JDBC Template thực thi truy vấn SQL , cập nhật thành phần , gọi stores procedure, thực hiện lặp qua ResultSets và trích xuất các giá trị tham số trả về.

Nó cũng bắt các ngoại lệ JDBC và chuyển chúng sang cấu trúc Generic, nhiều thông tin hơn, được xác định trong gói org.springframework.dao.

Các thành phần của lớp JDBCTemplate là threadsafe sau khi được cấu hình. Vì vậy, bạn có thể định cấu hình một thành phần JdbcTemplate và sau đó đưa tham chiếu được chia sẻ này vào nhiều DAO một cách an toàn.

Một thực tiễn phổ biến khi sử dụng lớp JDBCTemplate là cấu hình một DataSource trong file cấu hình Spring của bạn, sau đó đưa phụ thuộc vào DataSource bean được chia sẻ đó vào các lớp DAO của bạn, và JDBCTemplate được tạo trong setter dành cho DataSource.

Configuring Data Source

Tạo một bảng database Student trong database TEST của bạn. Giả sử bạn đang làm việc với MySQL database , nếu bạn làm việc với bất kì database khác sau đó bạn có thể đổi DDL và truy vấn SQL tương ứng của bạn.

CREATE TABLE Student(

ID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ID)

);

Bây giờ chúng ta cần cung cấp DataSource cho JDBCTemplate để nó có thể tự cấu hình để có quyền truy cập database. Bạn có thể cấu hình DataSource trong file XML bằng một đoạn code như sau:

<bean id = "dataSource"

class = "org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>

<property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>

<property name = "username" value = "root"/>

<property name = "password" value = "password"/>

</bean>

Data Access Object (DAO)

DAO là viết tắt của Data Access Object, thường được sử dụng để tương tác với cơ sở duwxl iệu. Các DAO tồn tại để cung cấp một phương tiện để đọc và ghi dữ liệu tới database và chúng nên hiển thị chức năng này thông qua một interface mà phần còn lại của ứng dụng sẽ truy cập chúng.

Hỗ trợ DAO trong Spring giúp bạn dễ dàng làm việc với các công nghệ truy cập dữ liệu như JDBC , Hibernate, JPA hoặc JDO một cách nhất quán.

Executing SQL statements

Chúng ta có thực hiện CRUD (Create, Read, Update and Delete) trên các bảng cơ sở dữ liệu bằng cách sử dụng SQL và đối tượng JDBC Template.

Truy vấn cho một số nguyên:

String SQL = "select count(\*) from Student";

int rowCount = jdbcTemplateObject.queryForInt( SQL );

Truy vấn cho một số long:

String SQL = "select count(\*) from Student";

long rowCount = jdbcTemplateObject.queryForLong( SQL );

một truy vấn đơn giản sử dụng một biến ràng buộc:

String SQL = "select age from Student where id = ?";

int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});

truy vấn cho một Chuỗi:

String SQL = "select name from Student where id = ?";

String name = jdbcTemplateObject.queryForObject(SQL, new Object[]{10}, String.class);

Truy vấn và trả về một đối tượng:

String SQL = "select \* from Student where id = ?";

Student student = jdbcTemplateObject.queryForObject(

SQL, new Object[]{10}, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student();

student.setID(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

Truy vấn và trả về nhiều đối tượng:

String SQL = "select \* from Student";

List<Student> students = jdbcTemplateObject.query(

SQL, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student();

student.setID(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

Thêm một hàng tới bảng:

String SQL = "insert into Student (name, age) values (?, ?)";

jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );

cập nhật một hàng tới bảng:

String SQL = "update Student set name = ? where id = ?";

jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );

Xóa một hàng từ bảng:

String SQL = "delete Student where id = ?";

jdbcTemplateObject.update( SQL, new Object[]{20} );

Executing DDL Statements

Bạn có thể sử dụng phương thức execute(..) từ jdbcTemplate để thực thi bất kì phần tử SQL hoặc phần tử DDL. Ví dụ dưới đây sử dụng phần tử Create để tạo một bảng:

String SQL = "CREATE TABLE Student( " +

"ID INT NOT NULL AUTO\_INCREMENT, " +

"NAME VARCHAR(20) NOT NULL, " +

"AGE INT NOT NULL, " +

"PRIMARY KEY (ID));"

jdbcTemplateObject.execute( SQL );

Spring JDBC Framework Examples:

Dựa trên những chủ đề trên , chúng ta kiểm tra một vài ví dụ quan trọng sẽ giúp bạn hiểu cách sử dụng JDBC Framework trong Spring: