# Spring Framework Overview

(Version 5.0.20.RELEASE)

Spring ایجاد برنامه های سازمانی جاوا را آسان می کند. با پشتیبانی از Groovy و Kotlin به‌عنوان زبان‌های جایگزین در JVM، و با انعطاف‌پذیری برای ایجاد انواع معماری‌ها بسته به نیازهای یک برنامه، هر آنچه را که برای پذیرش زبان جاوا در یک محیط سازمانی نیاز دارید، فراهم می‌کند. از Spring Framework 5.0، Spring به JDK 8+ (Java SE 8+) نیاز دارد و از قبل از JDK 9 پشتیبانی خارج از جعبه را فراهم می کند.

Spring طیف گسترده ای از سناریوهای کاربردی را پشتیبانی می کند. در یک شرکت بزرگ، برنامه‌ها اغلب برای مدت طولانی وجود دارند و باید روی یک JDK و سرور برنامه‌ای اجرا شوند که چرخه ارتقا آن خارج از کنترل توسعه‌دهنده است. برخی دیگر ممکن است به صورت یک jar با سرور تعبیه شده اجرا شوند، احتمالاً در یک محیط ابری. با این حال، برخی دیگر ممکن است برنامه‌های مستقل باشند (مانند بارهای کاری دسته‌ای یا یکپارچه) که نیازی به سرور ندارند.

بهار منبع باز است. دارای یک جامعه بزرگ و فعال است که بازخورد مستمری را بر اساس طیف متنوعی از موارد استفاده در دنیای واقعی ارائه می دهد. این به بهار کمک کرده است تا در مدت زمان طولانی با موفقیت تکامل یابد.

## 1. What We Mean by "Spring"

اصطلاح "بهار" به معنای چیزهای متفاوت در زمینه های مختلف است. می توان از آن برای اشاره به خود پروژه Spring Framework استفاده کرد، جایی که همه چیز از آنجا شروع شد. با گذشت زمان، پروژه‌های بهار دیگری در بالای چارچوب Spring ساخته شده‌اند. اغلب، وقتی مردم می گویند "بهار"، منظور آنها کل خانواده پروژه ها است. این مستندات مرجع بر اساس تمرکز دارد: خود چارچوب بهار.

Spring Framework به ماژول ها تقسیم می شود. برنامه ها می توانند ماژول های مورد نیاز خود را انتخاب کنند. در قلب ماژول های core container، از جمله مدل پیکربندی و مکانیزم تزریق وابستگی قرار دارند. فراتر از آن، Spring Framework پشتیبانی اساسی برای معماری‌های کاربردی مختلف، از جمله پیام‌رسانی، داده‌های تراکنش و پایداری، و وب فراهم می‌کند. همچنین شامل چارچوب وب Spring MVC مبتنی بر Servlet و به طور موازی، چارچوب وب واکنشی Spring WebFlux است.

نکته ای در مورد ماژول ها: جارهای فریمورک Spring اجازه استقرار در مسیر ماژول JDK 9 ("Jigsaw") را می دهند. برای استفاده در برنامه‌های دارای قابلیت Jigsaw، شیشه‌های Spring Framework 5 دارای ورودی‌های مانیفست «Automatic-Module-Name» هستند که نام‌های ماژول در سطح زبان پایدار («spring.core»، «spring.context» و غیره) را مستقل از آرتیفکت jar تعریف می‌کنند. نام‌ها (شیشه‌ها از الگوی نام‌گذاری یکسانی با "-" به جای "." پیروی می‌کنند، به عنوان مثال، "spring-core" و "spring-context"). البته، جارهای فریمورک Spring به خوبی در مسیر کلاس در JDK 8 و 9 کار می کنند.

## 2. History of Spring and the Spring Framework

بهار در سال 2003 به عنوان پاسخی به پیچیدگی مشخصات اولیه J2EE به وجود آمد. در حالی که برخی جاوا EE و Spring را با هم رقابت می کنند، Spring در واقع مکمل جاوا EE است. مدل برنامه نویسی Spring مشخصات پلت فرم Java EE را در بر نمی گیرد. بلکه با مشخصات فردی با دقت انتخاب شده از چتر EE ادغام می شود

Servlet API (JSR 340)

WebSocket API (JSR 356)

Concurrency Utilities (JSR 236)

JSON Binding API (JSR 367)

Bean Validation (JSR 303)

JPA (JSR 338)

JMS (JSR 914)

as well as JTA/JCA setups for transaction coordination, if necessary.

چارچوب Spring همچنین از مشخصات Dependency Injection (JSR 330) و Common Annotations (JSR 250) پشتیبانی می کند، که توسعه دهندگان برنامه ممکن است به جای مکانیسم های خاص Spring ارائه شده توسط Spring Framework استفاده کنند.

از Spring Framework 5.0، Spring حداقل به سطح Java EE 7 (به عنوان مثال Servlet 3.1+، JPA 2.1+) نیاز دارد - در حالی که در عین حال یکپارچه سازی خارج از جعبه با API های جدیدتر در سطح Java EE 8 ارائه می کند. (به عنوان مثال Servlet 4.0، JSON Binding API) هنگامی که در زمان اجرا با آن مواجه می شوید. این امر اسپرینگ را کاملاً با به عنوان مثال سازگار نگه می دارد. Tomcat 8 و 9، WebSphere 9 و JBoss EAP 7.

با گذشت زمان، نقش Java EE در توسعه اپلیکیشن تکامل یافته است. در روزهای اولیه Java EE و Spring، برنامه‌ها برای استقرار در سرور برنامه ایجاد شدند. امروزه، با کمک Spring Boot، برنامه‌ها به روشی توسعه‌دهنده و سازگار با فضای ابری ایجاد می‌شوند، با ظرف Servlet تعبیه‌شده و تغییر آن بی‌اهمیت است. از Spring Framework 5، یک برنامه WebFlux حتی از Servlet API مستقیماً استفاده نمی کند و می تواند روی سرورهایی (مانند Netty) که کانتینرهای Servlet نیستند اجرا شود.

بهار به نوآوری و تکامل خود ادامه می دهد. فراتر از چارچوب Spring، پروژه های دیگری مانند Spring Boot، Spring Security، Spring Data، Spring Cloud، Spring Batch و غیره وجود دارد. مهم است که به یاد داشته باشید که هر پروژه مخزن کد منبع، ردیاب مسئله و آهنگ انتشار خود را دارد. برای لیست کامل پروژه های Spring.io/projects را ببینید.

## 3. Design Philosophy

وقتی در مورد یک چارچوب یاد می گیرید ، مهم است که نه تنها چه کاری انجام می دهد بلکه چه اصولی را دنبال می کند. در اینجا اصول راهنما چارچوب بهار آورده شده است:

انتخاب را در هر سطح ارائه دهید. بهار به شما امکان می دهد تا هر چه زودتر تصمیمات طراحی را به تعویق بیندازید. به عنوان مثال ، شما می توانید ارائه دهندگان پایداری را از طریق پیکربندی بدون تغییر کد خود تغییر دهید. در مورد بسیاری از نگرانی های زیرساخت های دیگر و ادغام با API های شخص ثالث نیز همین مسئله صادق است.

دیدگاههای متنوعی را در خود جای دهید. بهار انعطاف پذیری را در بر می گیرد و در مورد نحوه انجام کارها نظر نمی شود. این برنامه از طیف گسترده ای از نیازهای برنامه با دیدگاه های مختلف پشتیبانی می کند.

سازگاری قوی به عقب را حفظ کنید. تکامل بهار با دقت موفق شده است تا تغییرات کمی بین نسخه ها را مجبور کند. بهار از طیف وسیعی از نسخه های JDK و کتابخانه های شخص ثالث برای تسهیل نگهداری برنامه ها و کتابخانه هایی که به بهار بستگی دارند ، پشتیبانی می کند.

در مورد طراحی API مراقبت کنید. تیم بهار فکر و زمان زیادی را برای ساخت API هایی که شهودی هستند و در بسیاری از نسخه ها و سالهای زیادی حفظ می کنند ، می گذارد.

استانداردهای بالایی را برای کیفیت کد تنظیم کنید. چارچوب بهار تأکید جدی بر Javadoc معنی دار ، فعلی و دقیق دارد. این یکی از پروژه های بسیار معدودی است که می تواند ساختار کد تمیز و بدون وابستگی دایره ای بین بسته ها را ادعا کند.

## 4. Feedback and Contributions

برای سؤالات چگونه یا مشکلات تشخیص یا اشکال زدایی ، پیشنهاد می کنیم از StackOverflow استفاده کنید ، و یک صفحه سؤالی داریم که برچسب های پیشنهادی را برای استفاده در لیست قرار می دهد. اگر کاملاً مطمئن هستید که در چارچوب بهار مشکلی وجود دارد یا می خواهید یک ویژگی را پیشنهاد دهید ، لطفاً از ردیاب شماره JIRA استفاده کنید.

اگر راه حل در ذهن دارید یا یک مشکل پیشنهادی دارید ، می توانید درخواست کشش را در GitHub ارسال کنید. با این حال ، لطفاً بخاطر داشته باشید که ، برای همه اما بی اهمیت ترین مسائل ، ما انتظار داریم بلیط در ردیاب موضوع ثبت شود ، جایی که بحث ها در آن صورت می گیرد و سابقه ای را برای مرجع بعدی باقی می گذارد.

برای اطلاعات بیشتر به دستورالعمل های صفحه پروژه کمک کننده و سطح بالا مراجعه کنید.

## 5. Getting Started

اگر به تازگی با Spring شروع کرده اید، ممکن است بخواهید با ایجاد یک برنامه مبتنی بر Spring Boot استفاده از Spring Framework را شروع کنید. Spring Boot یک راه سریع (و با نظر) برای ایجاد یک برنامه کاربردی مبتنی بر Spring آماده تولید ارائه می دهد. این بر اساس چارچوب Spring است، قرارداد را به پیکربندی ترجیح می دهد، و طراحی شده است تا شما را در سریع ترین زمان ممکن راه اندازی کند.

می‌توانید از start.spring.io برای تولید یک پروژه اساسی استفاده کنید یا یکی از راهنماهای «شروع به کار» را دنبال کنید، مانند شروع به ساخت یک وب سرویس RESTful. این راهنماها علاوه بر اینکه هضم آسان‌تری دارند، بر روی کار متمرکز هستند و اکثر آنها بر اساس Spring Boot هستند. آنها همچنین پروژه های دیگری از مجموعه بهار را پوشش می دهند که ممکن است بخواهید هنگام حل یک مشکل خاص در نظر بگیرید.

# Core Technologies

این بخش از مستندات مرجع، تمام آن فناوری‌هایی را پوشش می‌دهد که کاملاً در چارچوب Spring Framework هستند.

مهمترین آنها کانتینر Spring Framework's Inversion of Control (IoC) است. بررسی کامل ظرف IoC Spring Framework با پوشش جامع فناوری های برنامه نویسی جنبه گرا (AOP) Spring دنبال می شود. Spring Framework چارچوب AOP خود را دارد که از نظر مفهومی قابل درک است و با موفقیت 80% نیازهای AOP را در برنامه نویسی سازمانی جاوا برطرف می کند.

پوشش ادغام Spring با AspectJ (در حال حاضر غنی ترین - از نظر ویژگی ها - و مطمئناً بالغ ترین پیاده سازی AOP در فضای سازمانی جاوا) نیز ارائه شده است.

## 1. The IoC container

### 1.1. Introduction to the Spring IoC container and beans

این فصل اجرای چارچوب Spring از اصل وارونگی کنترل (IoC) [1] را پوشش می دهد. IoC همچنین به عنوان تزریق وابستگی (DI) شناخته می شود. این فرآیندی است که طی آن اشیاء وابستگی های خود را تعریف می کنند، یعنی اشیایی که با آنها کار می کنند، فقط از طریق آرگومان های سازنده، آرگومان های یک متد کارخانه، یا ویژگی هایی که روی نمونه شی پس از ساخته شدن یا بازگرداندن آن از یک متد کارخانه ای تنظیم می شوند. . سپس ظرف زمانی که لوبیا را ایجاد می کند آن وابستگی ها را تزریق می کند. این فرآیند اساساً معکوس است، از این رو Inversion of Control (IoC) نامیده می شود، که خود bean با استفاده از ساخت مستقیم کلاس ها یا مکانیزمی مانند الگوی Service Locator، نمونه یا مکان وابستگی های خود را کنترل می کند.

بسته‌های org.springframework.beans و org.springframework.context مبنایی برای کانتینر IoC Spring Framework هستند. رابط BeanFactory مکانیزم پیکربندی پیشرفته ای را ارائه می دهد که قادر به مدیریت هر نوع شی است. ApplicationContext یک رابط فرعی BeanFactory است. ادغام آسان تر با ویژگی های AOP Spring را اضافه می کند. مدیریت منابع پیام (برای استفاده در بین المللی)، انتشار رویداد. و زمینه های خاص لایه برنامه مانند WebApplicationContext برای استفاده در برنامه های کاربردی وب.

به طور خلاصه، BeanFactory چارچوب پیکربندی و عملکردهای اساسی را فراهم می کند، و ApplicationContext عملکردهای ویژه سازمانی بیشتری را اضافه می کند. ApplicationContext یک ابر مجموعه کامل از BeanFactory است و به طور انحصاری در این فصل در توضیحات کانتینر IoC Spring استفاده می شود. برای اطلاعات بیشتر در مورد استفاده از BeanFactory به جای ApplicationContext، به The BeanFactory مراجعه کنید.

در Spring، اشیایی که ستون فقرات برنامه شما را تشکیل می دهند و توسط کانتینر Spring IoC مدیریت می شوند، beans نامیده می شوند. یک لوبیا شیئی است که توسط یک کانتینر Spring IoC نمونه سازی، مونتاژ و مدیریت می شود. در غیر این صورت، یک لوبیا به سادگی یکی از بسیاری از اشیاء در برنامه شما است. لوبیاها و وابستگی‌های موجود در میان آن‌ها، در فراداده‌های پیکربندی استفاده‌شده توسط یک ظرف منعکس می‌شوند.

### 1.2. Container overview

رابط org.springframework.context.ApplicationContext نشان دهنده ظرف Spring IoC است و مسئول نمونه سازی، پیکربندی و مونتاژ دانه های فوق الذکر است. ظرف با خواندن فراداده های پیکربندی دستورالعمل های خود را در مورد مواردی که باید نمونه سازی، پیکربندی و مونتاژ شود، دریافت می کند. فراداده پیکربندی در XML، حاشیه نویسی جاوا یا کد جاوا نشان داده می شود. این به شما امکان می دهد اشیایی که برنامه شما را تشکیل می دهند و وابستگی های متقابل غنی بین چنین اشیایی را بیان کنید.

چندین پیاده سازی از رابط ApplicationContext به صورت خارج از جعبه با Spring عرضه می شود. در برنامه های مستقل ایجاد یک نمونه از ClassPathXmlApplicationContext یا FileSystemXmlApplicationContext معمول است. در حالی که XML فرمت سنتی برای تعریف فراداده پیکربندی بوده است، می‌توانید با ارائه مقدار کمی از پیکربندی XML برای فعال کردن پشتیبانی از این فرمت‌های فراداده اضافی، به ظرف دستور دهید که از حاشیه‌نویسی یا کد جاوا به عنوان قالب فراداده استفاده کند.

در اکثر سناریوهای برنامه کاربردی، برای نمونه سازی یک یا چند نمونه از یک ظرف Spring IoC، به کد کاربر صریح نیازی نیست. به عنوان مثال، در یک سناریوی برنامه وب، یک هشت خط (یا بیشتر) ساده از XML توصیفگر وب boilerplate در فایل web.xml برنامه معمولاً کافی است (نمونه سازی مناسب ApplicationContext برای برنامه های کاربردی وب را ببینید). اگر از محیط توسعه مبتنی بر Spring Tool Suite Eclipse استفاده می کنید، این پیکربندی دیگ بخار را می توان به راحتی با چند کلیک ماوس یا فشار دادن کلید ایجاد کرد.

نمودار زیر نمای سطح بالایی از نحوه عملکرد Spring است. کلاس های برنامه شما با ابرداده های پیکربندی ترکیب می شوند تا پس از ایجاد و تنظیم اولیه ApplicationContext، یک سیستم یا برنامه کاملاً پیکربندی شده و قابل اجرا داشته باشید.



#### 1.2.1. Configuration metadata

همانطور که نمودار قبل نشان می دهد، محفظه Spring IoC شکلی از فراداده پیکربندی را مصرف می کند. این فراداده پیکربندی نشان می‌دهد که چگونه شما به‌عنوان یک توسعه‌دهنده برنامه به کانتینر Spring می‌گویید که اشیاء را در برنامه شما نمونه‌سازی، پیکربندی و مونتاژ کند.

فراداده پیکربندی به طور سنتی در قالب XML ساده و شهودی ارائه می‌شود، که بیشتر این فصل از آن برای انتقال مفاهیم و ویژگی‌های کلیدی کانتینر Spring IoC استفاده می‌کند.

فراداده مبتنی بر XML تنها فرم مجاز فراداده پیکربندی نیست. خود محفظه Spring IoC کاملاً از قالبی که این فراداده پیکربندی واقعاً در آن نوشته شده است جدا شده است. این روزها بسیاری از توسعه دهندگان پیکربندی مبتنی بر جاوا را برای برنامه های Spring خود انتخاب می کنند.

فراداده مبتنی بر XML تنها فرم مجاز فراداده پیکربندی نیست. خود محفظه Spring IoC کاملاً از قالبی که این فراداده پیکربندی واقعاً در آن نوشته شده است جدا شده است. این روزها بسیاری از توسعه دهندگان پیکربندی مبتنی بر جاوا را برای برنامه های Spring خود انتخاب می کنند.

برای کسب اطلاعات در مورد استفاده از سایر اشکال فراداده با ظرف Spring، نگاه کنید به:

پیکربندی مبتنی بر حاشیه نویسی: Spring 2.5 پشتیبانی از ابرداده پیکربندی مبتنی بر حاشیه نویسی را معرفی کرد.

Java bean config: با شروع Spring 3.0، بسیاری از ویژگی های ارائه شده توسط پروژه Spring JavaConfig بخشی از چارچوب اصلی Spring Framework شدند. بنابراین شما می توانید با استفاده از جاوا به جای فایل های XML، beans را برای کلاس های برنامه خود تعریف کنید. برای استفاده از این ویژگی‌های جدید، به حاشیه‌نویسی‌های @Configuration، @Bean، @Import و @DependsOn مراجعه کنید.

پیکربندی فنری شامل حداقل یک تعریف و معمولاً بیش از یک تعریف است که ظرف باید آن را مدیریت کند. فراداده پیکربندی مبتنی بر XML این دانه‌ها را به‌عنوان عناصر <bean/> در یک عنصر سطح بالای <beans/> پیکربندی می‌کند. پیکربندی جاوا معمولاً از روش‌های حاشیه‌نویسی @Bean در یک کلاس @Configuration استفاده می‌کند.

این تعاریف bean با اشیاء واقعی که برنامه شما را تشکیل می دهند مطابقت دارد. معمولاً اشیاء لایه سرویس، اشیاء دسترسی به داده (DAO)، اشیاء ارائه مانند نمونه‌های Struts Action، اشیاء زیرساخت مانند Hibernate SessionFactories، صف‌های JMS و غیره را تعریف می‌کنید. به طور معمول، یکی اشیاء دامنه ریز را در کانتینر پیکربندی نمی کند، زیرا معمولاً مسئولیت DAO ها و منطق تجاری ایجاد و بارگذاری اشیاء دامنه است. با این حال، می توانید از ادغام Spring با AspectJ برای پیکربندی اشیایی که خارج از کنترل یک ظرف IoC ایجاد شده اند استفاده کنید. استفاده از AspectJ برای وابستگی تزریق اشیاء دامنه با Spring را ببینید.

مثال زیر ساختار اصلی فراداده های پیکربندی مبتنی بر XML را نشان می دهد:

<?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

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

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

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

</bean>

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

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

</bean>

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

</beans>

خصیصه id رشته ای است که از آن برای شناسایی تعریف منحصر به فرد bean استفاده می کنید. ویژگی class نوع bean را تعریف می کند و از نام کلاس کاملاً واجد شرایط استفاده می کند. مقدار مشخصه id به اشیاء همکار اشاره دارد. XML برای ارجاع به اشیاء همکار در این مثال نشان داده نشده است. برای اطلاعات بیشتر به Dependencies مراجعه کنید.

#### 1.2.2. Instantiating a container

نمونه سازی یک کانتینر Spring IoC ساده است. مسیر یا مسیرهای ارائه‌شده به سازنده ApplicationContext در واقع رشته‌های منبعی هستند که به ظرف اجازه می‌دهند ابرداده‌های پیکربندی را از انواع منابع خارجی مانند سیستم فایل محلی، از Java CLASSPATH و غیره بارگیری کند.

ApplicationContext context = **new** ClassPathXmlApplicationContext("services.xml", "daos.xml");

پس از آشنایی با کانتینر IoC Spring، ممکن است بخواهید بیشتر در مورد انتزاع منابع Spring بدانید، همانطور که در Resources توضیح داده شده است، که مکانیزم مناسبی برای خواندن یک جریان ورودی از مکان های تعریف شده در یک نحو URI ارائه می دهد. به طور خاص، مسیرهای منبع برای ساختن زمینه های برنامه ها همانطور که در زمینه های برنامه و مسیرهای منبع توضیح داده شده است، استفاده می شود.

مثال زیر فایل پیکربندی اشیاء لایه سرویس (services.xml) را نشان می دهد:

<?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

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

*<!-- services -->*

<bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">

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

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

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

</bean>

*<!-- more bean definitions for services go here -->*

</beans>

مثال زیر فایل پیکربندی اشیاء لایه سرویس (services.xml) را نشان می دهد:

<?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

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

<bean id="accountDao"

class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">

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

</bean>

<bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">

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

</bean>

*<!-- more bean definitions for data access objects go here -->*

</beans>

در مثال قبل، لایه سرویس از کلاس PetStoreServiceImpl و دو شیء دسترسی به داده از نوع JpaAccountDao و JpaItemDao (براساس استاندارد نگاشت شیء/رابطه JPA) تشکیل شده است. عنصر نام ویژگی به نام ویژگی JavaBean و عنصر ref به نام تعریف دیگری از bean اشاره دارد. این پیوند بین عناصر id و ref وابستگی بین اشیاء همکار را بیان می کند. برای جزئیات پیکربندی وابستگی‌های یک شی، به Dependencies مراجعه کنید.

Composing XML-based configuration metadata

این می تواند مفید باشد که تعاریف bean چندین فایل XML را در بر گیرند. اغلب هر فایل پیکربندی XML فردی یک لایه یا ماژول منطقی در معماری شما را نشان می دهد.

شما می توانید از سازنده زمینه برنامه برای بارگذاری تعاریف bean از تمام این قطعات XML استفاده کنید. همانطور که در بخش قبل نشان داده شد، این سازنده چندین مکان منبع را می گیرد. از طرف دیگر، از یک یا چند مورد از عنصر <import/> برای بارگیری تعاریف bean از یک فایل یا فایل دیگر استفاده کنید. مثلا:

<beans>

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

<bean id="bean1" class="..."/>

<bean id="bean2" class="..."/>

</beans>

در مثال قبلی، تعاریف خارجی bean از سه فایل بارگذاری می‌شوند: services.xml، messageSource.xml، و themeSource.xml. همه مسیرهای مکان نسبت به فایل تعریفی هستند که وارد کردن را انجام می دهد، بنابراین services.xml باید در همان دایرکتوری یا مکان کلاس مسیر با فایلی که وارد کردن را انجام می دهد، قرار داشته باشد، در حالی که messageSource.xml و themeSource.xml باید در یک مکان منابع زیر مکان قرار داشته باشند. از فایل وارد کننده همانطور که می بینید، یک اسلش پیشرو نادیده گرفته می شود، اما با توجه به نسبی بودن این مسیرها، بهتر است که به هیچ وجه از اسلش استفاده نکنید. محتویات فایل‌هایی که وارد می‌شوند، از جمله عنصر سطح بالای <beans/>، باید طبق طرح Spring دارای تعاریف معتبر XML bean باشند.

ارجاع فایل ها در فهرست های والد با استفاده از مسیر نسبی "../" امکان پذیر است، اما توصیه نمی شود. انجام این کار باعث ایجاد وابستگی به فایلی می شود که خارج از برنامه فعلی است. به طور خاص، این مرجع برای URL های "classpath:" توصیه نمی شود (به عنوان مثال، "classpath:../services.xml")، که در آن فرآیند حل زمان اجرا "نزدیک ترین" ریشه classpath را انتخاب می کند و سپس به فهرست اصلی آن نگاه می کند. تغییرات پیکربندی Classpath ممکن است منجر به انتخاب دایرکتوری متفاوت و نادرستی شود.

همیشه می‌توانید از مکان‌های منابع کاملاً واجد شرایط به جای مسیرهای نسبی استفاده کنید: به عنوان مثال، "file:C:/config/services.xml" یا "classpath:/config/services.xml". با این حال، توجه داشته باشید که پیکربندی برنامه خود را به مکان های مطلق خاصی متصل می کنید. به طور کلی ترجیح داده می شود که برای چنین مکان های مطلق، یک جهت غیر مستقیم حفظ شود، به عنوان مثال، از طریق متغیرهای "${…​}" که در زمان اجرا با ویژگی های سیستم JVM حل می شوند.

دستورالعمل واردات یک ویژگی است که توسط خود فضای نام beans ارائه شده است. ویژگی های پیکربندی بیشتر فراتر از تعاریف ساده bean در مجموعه ای از فضاهای نام XML ارائه شده توسط Spring، به عنوان مثال، موجود است. فضای نام "context" و "util".

دستورالعمل واردات یک ویژگی است که توسط خود فضای نام beans ارائه شده است. ویژگی های پیکربندی بیشتر فراتر از تعاریف ساده bean در مجموعه ای از فضاهای نام XML ارائه شده توسط Spring، به عنوان مثال، موجود است. فضای نام "context" و "util".

The Groovy Bean Definition DSL

به عنوان مثالی دیگر برای ابرداده های پیکربندی خارجی، تعاریف bean را می توان در Spring's Groovy Bean Definition DSL، همانطور که از چارچوب Grails شناخته می شود، بیان کرد. به طور معمول، چنین پیکربندی در یک فایل ".groovy" با ساختاری به شرح زیر است:

beans {

dataSource(BasicDataSource) {

driverClassName = "org.hsqldb.jdbcDriver"

url = "jdbc:hsqldb:mem:grailsDB"

username = "sa"

password = ""

settings = [mynew:"setting"]

}

sessionFactory(SessionFactory) {

dataSource = dataSource

}

myService(MyService) {

nestedBean = { AnotherBean bean ->

dataSource = dataSource

}

}

}

این سبک پیکربندی تا حد زیادی معادل تعاریف XML Bean است و حتی از فضاهای نام پیکربندی XML Spring پشتیبانی می‌کند. همچنین اجازه می دهد تا فایل های تعریف XML bean را از طریق دستورالعمل "importBeans" وارد کنید.

#### 1.2.3. Using the container

ApplicationContext رابطی برای یک کارخانه پیشرفته است که قادر به حفظ رجیستری از دانه های مختلف و وابستگی های آنها است. با استفاده از روش T getBean(String name, Class<T> requiredType) می توانید نمونه هایی از bean های خود را بازیابی کنید.

ApplicationContext شما را قادر می سازد تا تعاریف bean را بخوانید و به صورت زیر به آنها دسترسی داشته باشید:

*// create and configure beans*

ApplicationContext context = **new** ClassPathXmlApplicationContext("services.xml", "daos.xml");

*// retrieve configured instance*

PetStoreService service = context.getBean("petStore", PetStoreService.class);

*// use configured instance*

List<String> userList = service.getUsernameList();

با پیکربندی Groovy، bootstrapping بسیار شبیه به نظر می رسد، فقط یک کلاس پیاده سازی زمینه متفاوت که Groovy-aware است (اما تعاریف XML bean را نیز درک می کند):

ApplicationContext context = **new** GenericGroovyApplicationContext("services.groovy", "daos.groovy");

منعطف ترین نوع GenericApplicationContext در ترکیب با نمایندگان خواننده است، به عنوان مثال. با XmlBeanDefinitionReader برای فایل های XML:

GenericApplicationContext context = **new** GenericApplicationContext();

**new** XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml");

context.refresh();

یا با GroovyBeanDefinitionReader برای فایل های Groovy:

GenericApplicationContext context = **new** GenericApplicationContext();

**new** GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy", "daos.groovy");

context.refresh();

چنین نمایندگان خواننده را می توان در همان ApplicationContext با هم ترکیب کرد و در صورت تمایل، تعاریف bean را از منابع پیکربندی متنوع خواند.

سپس می توانید از getBean برای بازیابی نمونه هایی از لوبیاهای خود استفاده کنید. رابط ApplicationContext چند روش دیگر برای بازیابی لوبیا دارد، اما در حالت ایده آل کد برنامه شما هرگز نباید از آنها استفاده کند. در واقع، کد برنامه شما نباید به هیچ وجه به متد ()getBean فراخوانی کند، و بنابراین اصلاً وابستگی به API های Spring نداشته باشد. به عنوان مثال، ادغام Spring با چارچوب‌های وب، تزریق وابستگی را برای اجزای مختلف چارچوب وب مانند کنترل‌کننده‌ها و دانه‌های مدیریت‌شده با JSF فراهم می‌کند و به شما این امکان را می‌دهد که وابستگی به یک Bean خاص را از طریق ابرداده (مثلاً یک حاشیه‌نویسی سیم‌کشی خودکار) اعلام کنید.

## 1.3. Bean overview

ظرف Spring IoC یک یا چند لوبیا را مدیریت می کند. این دانه‌ها با ابرداده‌های پیکربندی که به کانتینر ارائه می‌دهید، ایجاد می‌شوند، به عنوان مثال، در قالب تعاریف XML <bean/>.

در خود ظرف، این تعاریف bean به عنوان اشیاء BeanDefinition نشان داده می‌شوند که شامل (در میان اطلاعات دیگر) فوق داده‌های زیر است:

نام کلاس واجد شرایط بسته: معمولاً کلاس پیاده سازی واقعی bean تعریف می شود.

عناصر پیکربندی رفتاری Bean، که بیان می‌کند که bean چگونه باید در کانتینر رفتار کند (محدوده، تماس‌های چرخه حیات، و غیره).

ارجاع به دانه های دیگری که برای انجام کار لوبیا مورد نیاز است. به این مراجع همکار یا وابستگی نیز می گویند.

سایر تنظیمات پیکربندی برای تنظیم در شی جدید ایجاد شده، به عنوان مثال، تعداد اتصالات برای استفاده در یک Bean که یک مخزن اتصال را مدیریت می کند، یا محدودیت اندازه استخر.

این ابرداده به مجموعه ای از ویژگی ها ترجمه می شود که هر تعریف bean را تشکیل می دهد.

| Table 1. The bean definition | |
| --- | --- |
| **Property** | **Explained in…​** |
| class | [Instantiating beans](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-class) |
| name | [Naming beans](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-beanname) |
| scope | [Bean scopes](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-scopes) |
| constructor arguments | [Dependency Injection](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-collaborators) |
| properties | [Dependency Injection](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-collaborators) |
| autowiring mode | [Autowiring collaborators](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-autowire) |
| lazy-initialization mode | [Lazy-initialized beans](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-lazy-init) |
| initialization method | [Initialization callbacks](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) |
| destruction method | [Destruction callbacks](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean) |
|  |  |

علاوه بر تعاریف bean که حاوی اطلاعاتی در مورد نحوه ایجاد یک bean خاص است، پیاده سازی های ApplicationContext همچنین اجازه ثبت اشیاء موجود را که خارج از کانتینر توسط کاربران ایجاد می شوند را می دهد. این کار با دسترسی به BeanFactory ApplicationContext از طریق متد getBeanFactory() انجام می شود که اجرای BeanFactory DefaultListableBeanFactory را برمی گرداند. DefaultListableBeanFactory این ثبت را از طریق متدهای registerSingleton(..) و registerBeanDefinition(..) پشتیبانی می کند. با این حال، برنامه‌های کاربردی معمولی تنها با لوبیاهایی که از طریق تعاریف متادیتا bean تعریف شده‌اند، کار می‌کنند.

فراداده‌های Bean و نمونه‌های تک‌تنه ارائه‌شده به‌صورت دستی باید در اسرع وقت ثبت شوند تا ظرف به درستی درباره آنها در طول سیم‌کشی خودکار و سایر مراحل درون‌نگاری استدلال کند. در حالی که نادیده گرفتن ابرداده‌های موجود و نمونه‌های singleton موجود تا حدی پشتیبانی می‌شود، ثبت bean جدید در زمان اجرا (همزمان با دسترسی زنده به کارخانه) به طور رسمی پشتیبانی نمی‌شود و ممکن است منجر به استثناهای دسترسی همزمان و/یا حالت ناسازگار در ظرف bean شود. .

### 1.3.1. Naming beans

هر لوبیا یک یا چند شناسه دارد. این شناسه ها باید در ظرفی که لوبیا را میزبانی می کند منحصر به فرد باشد. یک لوبیا معمولاً فقط یک شناسه دارد، اما اگر به بیش از یک شناسه نیاز داشته باشد، موارد اضافی را می توان نام مستعار در نظر گرفت.

در فراداده پیکربندی مبتنی بر XML، از ویژگی‌های id و/یا نام برای تعیین شناسه(های) bean استفاده می‌کنید. ویژگی id به شما امکان می دهد دقیقا یک شناسه را مشخص کنید. معمولاً این نام‌ها حروف عددی هستند ("myBean"، "fooService"، و غیره)، اما ممکن است شامل کاراکترهای خاصی نیز باشند. اگر می‌خواهید نام‌های مستعار دیگری را به bean معرفی کنید، می‌توانید آن‌ها را در ویژگی name که با کاما (،)، نقطه ویرگول (;) یا فاصله سفید از هم جدا شده‌اند، مشخص کنید. به عنوان یک یادداشت تاریخی، در نسخه های قبل از Spring 3.1، ویژگی id به عنوان یک نوع xsd:ID تعریف شده بود که کاراکترهای ممکن را محدود می کرد. از 3.1، به عنوان یک نوع xsd:string تعریف شده است. توجه داشته باشید که منحصر به فرد بودن شناسه bean هنوز توسط کانتینر اعمال می شود، البته دیگر توسط تجزیه کننده های XML اعمال نمی شود.

شما نیازی به ارائه نام یا شناسه برای لوبیا ندارید. اگر هیچ نام یا شناسه ای به صراحت ارائه نشده باشد، ظرف یک نام منحصر به فرد برای آن لوبیا ایجاد می کند. با این حال، اگر می‌خواهید به آن bean با نام اشاره کنید، از طریق استفاده از عنصر ref یا جستجوی سبک Service Locator، باید یک نام ارائه کنید. انگیزه های عدم ارائه نام مربوط به استفاده از ایننر لوبیا و همکاران سیم کشی خودکار است.

Bean Naming Conventions

قرارداد استفاده از قرارداد استاندارد جاوا برای مثال نام فیلدها هنگام نامگذاری دانه ها است. یعنی نام لوبیا با حروف کوچک شروع می شود و از آن به بعد شتری است. نمونه هایی از این نام ها (بدون نقل قول) 'accountManager'، 'accountService'، 'userDao'، 'loginController' و غیره خواهد بود.

نام گذاری دانه ها به طور مداوم پیکربندی شما را برای خواندن و درک آسان تر می کند، و اگر از Spring AOP استفاده می کنید، هنگام اعمال توصیه به مجموعه ای از دانه های مرتبط با نام کمک زیادی می کند.

با اسکن کامپوننت در مسیر کلاس، Spring نام‌های bean را برای مؤلفه‌های بدون نام تولید می‌کند، از قوانین بالا پیروی می‌کند: اساساً، نام کلاس ساده را می‌گیرد و کاراکتر اولیه آن را به حروف کوچک تبدیل می‌کند. با این حال، در حالت خاص (غیر معمول) که بیش از یک کاراکتر وجود دارد و هر دو کاراکتر اول و دوم بزرگ هستند، پوشش اصلی حفظ می‌شود. اینها همان قوانینی هستند که توسط java.beans.Introspector.decapitalize (که Spring در اینجا از آن استفاده می کند) تعریف شده است.

Aliasing a bean outside the bean definition

در خود تعریف bean، با استفاده از ترکیبی از حداکثر یک نام مشخص شده توسط ویژگی id، و هر تعداد نام دیگر در ویژگی name، می توانید بیش از یک نام برای bean ارائه دهید. این نام‌ها می‌توانند نام مستعار معادل همان bean باشند، و برای برخی موقعیت‌ها مفید هستند، مانند اجازه دادن به هر جزء در یک برنامه کاربردی برای ارجاع به یک وابستگی مشترک با استفاده از نام bean که مختص خود آن جزء است.

با این حال، مشخص کردن همه نام‌های مستعار که در آن bean واقعاً تعریف شده است، همیشه کافی نیست. گاهی اوقات توصیه می شود نام مستعار برای یک لوبیا معرفی شود که در جای دیگری تعریف شده است. معمولاً در سیستم‌های بزرگ که پیکربندی بین هر زیرسیستم تقسیم می‌شود، هر زیرسیستم مجموعه‌ای از تعاریف شی خاص خود را دارد. در فراداده پیکربندی مبتنی بر XML، می‌توانید از عنصر <alias/> برای انجام این کار استفاده کنید.

<alias name="fromName" alias="toName"/>

در این مورد، یک لوبیا (در همان ظرف) با نام fromName نیز ممکن است پس از استفاده از این تعریف مستعار، به عنوان toName نامیده شود.

به عنوان مثال، فراداده پیکربندی برای زیرسیستم A ممکن است به یک DataSource با نام subsystemA-dataSource اشاره کند. فراداده پیکربندی برای زیرسیستم B ممکن است به یک DataSource با نام subsystemB-dataSource اشاره کند. هنگام نوشتن برنامه اصلی که از هر دو زیرسیستم استفاده می کند، برنامه اصلی به DataSource با نام myApp-dataSource اشاره می کند. برای اینکه هر سه نام به یک شی اشاره داشته باشند، می توانید تعاریف نام مستعار زیر را به فراداده پیکربندی اضافه کنید:

<alias name="myApp-dataSource" alias="subsystemA-dataSource"/>

<alias name="myApp-dataSource" alias="subsystemB-dataSource"/>

Java-configuration

اگر از پیکربندی جاوا استفاده می‌کنید، می‌توان از حاشیه‌نویسی @Bean برای ارائه نام‌های مستعار استفاده کرد، برای جزئیات بیشتر به استفاده از حاشیه‌نویسی @Bean مراجعه کنید.

### 1.3.2. Instantiating beans

تعریف لوبیا در اصل دستور العمل ایجاد یک یا چند شی است. کانتینر در صورت درخواست از دستور العمل یک لوبیای نامگذاری شده نگاه می کند و از ابرداده پیکربندی محصور شده توسط آن تعریف لوبیا برای ایجاد (یا به دست آوردن) یک شیء واقعی استفاده می کند.

اگر از ابرداده پیکربندی مبتنی بر XML استفاده می کنید ، نوع (یا کلاس) شیء را که باید در ویژگی کلاس عنصر <bean/> فوری شود ، مشخص می کنید. این ویژگی کلاس ، که در داخل یک ویژگی کلاس در نمونه Beandefinition است ، معمولاً اجباری است. (برای استثنائات ، با استفاده از یک روش کارخانه نمونه و وراثت تعریف لوبیا به فوری مراجعه کنید.) شما از یکی از دو روش از ویژگی کلاس استفاده می کنید:

به طور معمول ، برای مشخص کردن کلاس لوبیا در موردی که خود ظرف به طور مستقیم لوبیا را با فراخوانی سازنده خود به طور انعکاس ، تا حدودی معادل کد جاوا با استفاده از اپراتور جدید ایجاد می کند ، ایجاد می شود.

برای مشخص کردن کلاس واقعی حاوی روش کارخانه استاتیک که برای ایجاد جسم مورد استفاده قرار می گیرد ، در مورد کمتر متداول که ظرف یک روش کارخانه استاتیک را در یک کلاس فراخوانی می کند تا لوبیا ایجاد شود. نوع شیء که از فراخوانی روش کارخانه استاتیک برگشته است ممکن است همان کلاس یا کلاس دیگری باشد.

Inner class names

تعریف لوبیا در اصل دستور العمل ایجاد یک یا چند شی است. کانتینر در صورت درخواست از دستور العمل یک لوبیای نامگذاری شده نگاه می کند و از ابرداده پیکربندی محصور شده توسط آن تعریف لوبیا برای ایجاد (یا به دست آوردن) یک شیء واقعی استفاده می کند.

اگر از ابرداده پیکربندی مبتنی بر XML استفاده می کنید ، نوع (یا کلاس) شیء را که باید در ویژگی کلاس عنصر <bean/> فوری شود ، مشخص می کنید. این ویژگی کلاس ، که در داخل یک ویژگی کلاس در نمونه Beandefinition است ، معمولاً اجباری است. (برای استثنائات ، با استفاده از یک روش کارخانه نمونه و وراثت تعریف لوبیا به فوری مراجعه کنید.) شما از یکی از دو روش از ویژگی کلاس استفاده می کنید:

به طور معمول ، برای مشخص کردن کلاس لوبیا در موردی که خود ظرف به طور مستقیم لوبیا را با فراخوانی سازنده خود به طور انعکاس ، تا حدودی معادل کد جاوا با استفاده از اپراتور جدید ایجاد می کند ، ایجاد می شود.

برای مشخص کردن کلاس واقعی حاوی روش کارخانه استاتیک که برای ایجاد جسم مورد استفاده قرار می گیرد ، در مورد کمتر متداول که ظرف یک روش کارخانه استاتیک را در یک کلاس فراخوانی می کند تا لوبیا ایجاد شود. نوع شیء که از فراخوانی روش کارخانه استاتیک برگشته است ممکن است همان کلاس یا کلاس دیگری باشد.

Instantiation with a constructor

وقتی یک bean با رویکرد سازنده ایجاد می‌کنید، همه کلاس‌های عادی توسط Spring قابل استفاده و سازگار هستند. یعنی کلاسی که توسعه می‌یابد نیازی به پیاده‌سازی رابط‌های خاص یا کدگذاری به روش خاصی ندارد. صرفاً مشخص کردن کلاس bean کافی است. با این حال، بسته به نوع IoC که برای آن Bean خاص استفاده می‌کنید، ممکن است به یک سازنده پیش‌فرض (خالی) نیاز داشته باشید.

کانتینر Spring IoC می تواند تقریباً هر کلاسی را که می خواهید مدیریت کند، مدیریت کند. این به مدیریت جاوابین واقعی محدود نمی شود. اکثر کاربران Spring JavaBeans واقعی را تنها با سازنده پیش‌فرض (بدون آرگومان) و تنظیم‌کننده‌ها و دریافت‌کننده‌های مناسب که بر اساس ویژگی‌های موجود در ظرف مدل‌سازی شده‌اند، ترجیح می‌دهند. همچنین می‌توانید کلاس‌های عجیب‌تری به سبک غیر لوبیا در ظرف خود داشته باشید. به عنوان مثال، اگر شما نیاز به استفاده از یک استخر اتصال قدیمی دارید که مطلقاً با مشخصات JavaBean مطابقت ندارد، Spring می‌تواند آن را نیز مدیریت کند.

با فراداده پیکربندی مبتنی بر XML می توانید کلاس bean خود را به صورت زیر مشخص کنید:

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

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>

برای جزئیات در مورد مکانیسم ارائه آرگومان‌ها به سازنده (در صورت نیاز) و تنظیم ویژگی‌های نمونه شی پس از ساخته شدن شی، به تزریق وابستگی‌ها مراجعه کنید.

Instantiation with a static factory method

هنگام تعریف bean که با روش کارخانه ایستا ایجاد می کنید، از ویژگی class برای تعیین کلاس حاوی متد static factory و از ویژگی با نام factory-method برای تعیین نام خود متد کارخانه استفاده می کنید. شما باید بتوانید این متد را فراخوانی کنید (با آرگومان های اختیاری همانطور که در ادامه توضیح داده شد) و یک شی زنده را برگردانید، که متعاقباً طوری رفتار می شود که گویی از طریق سازنده ایجاد شده است. یکی از کاربردهای چنین تعریفی، فراخوانی کارخانه‌های استاتیک در کدهای قدیمی است.

تعریف bean زیر مشخص می کند که bean با فراخوانی یک روش کارخانه ایجاد می شود. تعریف نوع (کلاس) شیء برگشتی را مشخص نمی کند، فقط کلاس حاوی متد کارخانه را مشخص می کند. در این مثال، متد ()createInstance باید یک متد ثابت باشد.

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

**public** **class** **ClientService** {

**private** **static** ClientService clientService = **new** ClientService();

**private** ClientService() {}

**public** **static** ClientService createInstance() {

**return** clientService;

}

}

برای جزئیات بیشتر درباره مکانیسم ارائه آرگومان‌های (اختیاری) به روش کارخانه و تنظیم ویژگی‌های نمونه شی پس از بازگرداندن شی از کارخانه، به جزئیات وابستگی‌ها و پیکربندی مراجعه کنید.

Instantiation using an instance factory method

مشابه نمونه سازی از طریق روش کارخانه ایستا، نمونه سازی با روش کارخانه نمونه از یک روش غیر استاتیک از یک دانه موجود از ظرف برای ایجاد یک دانه جدید فراخوانی می کند. برای استفاده از این مکانیسم، صفت کلاس را خالی بگذارید، و در ویژگی factory-bean، نام یک bean را در ظرف فعلی (یا والد/اجداد) که حاوی متد نمونه‌ای است که قرار است برای ایجاد شیء فراخوانی شود، مشخص کنید. نام خود روش کارخانه را با ویژگی factory-method تنظیم کنید.

*<!-- the factory bean, which contains a method called createInstance() -->*

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

*<!-- inject any dependencies required by this locator bean -->*

</bean>

*<!-- the bean to be created via the factory bean -->*

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

**public** **class** **DefaultServiceLocator** {

**private** **static** ClientService clientService = **new** ClientServiceImpl();

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

}

همانطور که در اینجا نشان داده شده است، یک کلاس کارخانه می تواند بیش از یک روش کارخانه را نیز در خود جای دهد:

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

*<!-- inject any dependencies required by this locator bean -->*

</bean>

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

<bean id="accountService"

factory-bean="serviceLocator"

factory-method="createAccountServiceInstance"/>

**public** **class** **DefaultServiceLocator** {

**private** **static** ClientService clientService = **new** ClientServiceImpl();

**private** **static** AccountService accountService = **new** AccountServiceImpl();

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

**public** AccountService createAccountServiceInstance() {

**return** accountService;

}

}

این رویکرد نشان می دهد که خود کارخانه لوبیا را می توان از طریق تزریق وابستگی (DI) مدیریت و پیکربندی کرد. به جزئیات وابستگی ها و پیکربندی ها مراجعه کنید.

در مستندات Spring، Factory Bean به Bean اطلاق می‌شود که در ظرف Spring پیکربندی شده است که از طریق یک نمونه یا روش کارخانه ایستا، اشیاء را ایجاد می‌کند. در مقابل، FactoryBean (به حروف بزرگ توجه کنید) به FactoryBean مخصوص Spring اشاره دارد.

## 1.4. Dependencies

یک برنامه کاربردی معمولی سازمانی از یک شی (یا در اصطلاح بهار) تشکیل نشده است. حتی ساده ترین برنامه دارای چند شی است که با هم کار می کنند تا آنچه را که کاربر نهایی به عنوان یک برنامه منسجم می بیند ارائه دهد. این بخش بعدی توضیح می‌دهد که چگونه از تعریف تعدادی از تعاریف bean که به تنهایی می‌مانند، به یک کاربرد کاملاً تحقق‌یافته که در آن اشیاء برای رسیدن به یک هدف با یکدیگر همکاری می‌کنند، می‌روید.

### 1.4.1. Dependency Injection

تزریق وابستگی (DI) فرآیندی است که طی آن اشیاء وابستگی‌های خود، یعنی سایر اشیاء را که با آنها کار می‌کنند، تنها از طریق آرگومان‌های سازنده، آرگومان‌های یک متد کارخانه یا ویژگی‌هایی که بر روی نمونه شی پس از ساخته شدن یا برگرداندن آن تنظیم می‌شوند، تعریف می‌کنند. از روش کارخانه ای سپس ظرف زمانی که لوبیا را ایجاد می کند آن وابستگی ها را تزریق می کند. این فرآیند اساساً معکوس است، از این رو Inversion of Control (IoC) نامیده می شود، که خود bean نمونه یا مکان وابستگی های خود را به تنهایی با استفاده از ساخت مستقیم کلاس ها یا الگوی Service Locator کنترل می کند.

کد با اصل DI تمیزتر است و جداسازی زمانی موثرتر است که اشیا با وابستگی های خود ارائه شوند. شیء وابستگی های خود را جستجو نمی کند و مکان یا کلاس وابستگی ها را نمی داند. به این ترتیب، تست کلاس‌های شما آسان‌تر می‌شود، به‌ویژه زمانی که وابستگی‌ها بر روی رابط‌ها یا کلاس‌های پایه انتزاعی هستند، که اجازه می‌دهد از پیاده‌سازی‌های خرد یا ساختگی در تست‌های واحد استفاده شود.

DI در دو نوع اصلی وجود دارد، تزریق وابستگی مبتنی بر سازنده و تزریق وابستگی مبتنی بر Setter.

#### Constructor-based dependency injection

DI مبتنی بر سازنده با فراخوانی یک سازنده با تعدادی آرگومان که هر کدام یک وابستگی را نشان می‌دهند، انجام می‌شود. فراخوانی یک روش کارخانه ایستا با آرگومان های خاص برای ساخت bean تقریباً معادل است، و این بحث آرگومان ها را برای سازنده و روش کارخانه ایستا به طور مشابه مورد بررسی قرار می دهد. مثال زیر کلاسی را نشان می‌دهد که فقط با تزریق سازنده می‌تواند وابستگی تزریق شود. توجه داشته باشید که هیچ چیز خاصی در مورد این کلاس وجود ندارد، این یک POJO است که هیچ وابستگی به رابط های خاص کانتینر، کلاس های پایه یا حاشیه نویسی ندارد.

**public** **class** **SimpleMovieLister** {

*// the SimpleMovieLister has a dependency on a MovieFinder*

**private** MovieFinder movieFinder;

*// a constructor so that the Spring container can inject a MovieFinder*

**public** SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// business logic that actually uses the injected MovieFinder is omitted...*

}

#### Constructor argument resolution

تطبیق وضوح آرگومان سازنده با استفاده از نوع آرگومان رخ می دهد. اگر هیچ ابهامی بالقوه در آرگومان‌های سازنده تعریف bean وجود نداشته باشد، ترتیبی که آرگومان‌های سازنده در تعریف bean تعریف می‌شوند، ترتیبی است که در آن آرگومان‌ها به سازنده مناسب زمانی که bean در حال نمونه‌سازی است، ارائه می‌شود. کلاس زیر را در نظر بگیرید:

**package** x.y;

**public** **class** **Foo** {

**public** Foo(Bar bar, Baz baz) {

*// ...*

}

}

هیچ ابهامی بالقوه وجود ندارد، با این فرض که کلاس های Bar و Baz با وراثت به هم مرتبط نیستند. بنابراین، پیکربندی زیر به خوبی کار می‌کند، و نیازی نیست که شاخص‌ها و/یا انواع آرگومان سازنده را به طور صریح در عنصر <constructor-arg/> مشخص کنید.

<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>

هنگامی که به bean دیگری ارجاع داده می شود، نوع آن مشخص می شود و تطبیق می تواند رخ دهد (همانطور که در مثال قبل رخ داد). وقتی از یک نوع ساده مانند <value>true</value> استفاده می‌شود، Spring نمی‌تواند نوع مقدار را تعیین کند و بنابراین نمی‌تواند بدون کمک با نوع آن مطابقت داشته باشد. کلاس زیر را در نظر بگیرید:

**package** examples;

**public** **class** **ExampleBean** {

*// Number of years to calculate the Ultimate Answer*

**private** **int** years;

*// The Answer to Life, the Universe, and Everything*

**private** String ultimateAnswer;

**public** ExampleBean(**int** years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

Constructor argument type matching

در سناریوی قبلی، اگر شما به صراحت نوع آرگومان سازنده را با استفاده از ویژگی type مشخص کنید، کانتینر می تواند از تطبیق نوع با انواع ساده استفاده کند. مثلا:

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

<constructor-arg type="int" value="7500000"/>

<constructor-arg type="java.lang.String" value="42"/>

</bean>

Constructor argument index

از ویژگی index برای مشخص کردن صریح شاخص آرگومان های سازنده استفاده کنید. مثلا:

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

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

علاوه بر حل ابهام چندین مقدار ساده، مشخص کردن یک شاخص ابهام را در جایی که سازنده دو آرگومان از یک نوع داشته باشد برطرف می کند. توجه داشته باشید که شاخص بر اساس 0 است.

Constructor argument name

همچنین می‌توانید از نام پارامتر سازنده برای ابهام‌زدایی مقدار استفاده کنید:

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

<constructor-arg name="years" value="7500000"/>

<constructor-arg name="ultimateAnswer" value="42"/>

</bean>

به خاطر داشته باشید که برای انجام این کار خارج از جعبه، کد شما باید با فعال بودن پرچم اشکال زدایی کامپایل شود تا Spring بتواند نام پارامتر را از سازنده جستجو کند. اگر نمی‌توانید کد خود را با پرچم اشکال‌زدایی کامپایل کنید (یا نمی‌خواهید)، می‌توانید از حاشیه‌نویسی @ConstructorProperties JDK برای نام‌گذاری صریح آرگومان‌های سازنده خود استفاده کنید. سپس کلاس نمونه باید به صورت زیر باشد:

**package** examples;

**public** **class** **ExampleBean** {

*// Fields omitted*

@ConstructorProperties({"years", "ultimateAnswer"})

**public** ExampleBean(**int** years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

##### **Setter-based dependency injection**

DI مبتنی بر تنظیم توسط ظرفی که متدهای تنظیم کننده را روی دانه های شما فراخوانی می کند، پس از فراخوانی یک سازنده بدون آرگومان یا روش کارخانه ایستا بدون آرگومان برای نمونه سازی bean شما انجام می شود.

مثال زیر کلاسی را نشان می‌دهد که فقط با استفاده از تزریق ستتر خالص می‌توان به آن وابستگی تزریق کرد. این کلاس جاوای معمولی است. این یک POJO است که هیچ وابستگی به رابط های خاص کانتینر، کلاس های پایه یا حاشیه نویسی ندارد.

**public** **class** **SimpleMovieLister** {

*// the SimpleMovieLister has a dependency on the MovieFinder*

**private** MovieFinder movieFinder;

*// a setter method so that the Spring container can inject a MovieFinder*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// business logic that actually uses the injected MovieFinder is omitted...*

}

ApplicationContext از DI مبتنی بر سازنده و مبتنی بر تنظیم کننده برای دانه هایی که مدیریت می کند پشتیبانی می کند. همچنین پس از تزریق برخی وابستگی ها از طریق رویکرد سازنده، از DI مبتنی بر تنظیم کننده پشتیبانی می کند. شما وابستگی ها را در قالب BeanDefinition پیکربندی می کنید، که در ارتباط با نمونه های PropertyEditor برای تبدیل ویژگی ها از یک فرمت به فرمت دیگر استفاده می کنید. با این حال، اکثر کاربران Spring مستقیماً با این کلاس‌ها کار نمی‌کنند (یعنی به صورت برنامه‌نویسی) بلکه با تعاریف XML Bean، اجزای حاشیه‌نویسی شده (به عنوان مثال، کلاس‌های حاشیه‌نویسی شده با @Component، @Controller، و غیره) یا روش‌های @Bean در جاوا کار می‌کنند. configuration@کلاس . سپس این منابع به صورت داخلی به نمونه هایی از BeanDefinition تبدیل می شوند و برای بارگیری کل نمونه کانتینر Spring IoC استفاده می شوند.

Constructor-based or setter-based DI?

از آنجایی که می‌توانید DI مبتنی بر سازنده و تنظیم‌کننده را با هم ترکیب کنید، استفاده از سازنده‌ها برای وابستگی‌های اجباری و روش‌های تنظیم‌کننده یا روش‌های پیکربندی برای وابستگی‌های اختیاری یک قانون خوب است. توجه داشته باشید که استفاده از حاشیه‌نویسی @Required در یک متد تنظیم کننده می‌تواند برای تبدیل ویژگی به یک وابستگی ضروری استفاده شود.

تیم Spring عموماً از تزریق سازنده حمایت می کند زیرا به فرد امکان می دهد اجزای برنامه را به عنوان اشیاء تغییرناپذیر پیاده سازی کند و اطمینان حاصل کند که وابستگی های مورد نیاز خالی نیستند. علاوه بر این، مؤلفه های تزریق شده توسط سازنده همیشه در یک حالت کاملاً اولیه به کد مشتری (در حال فراخوان) بازگردانده می شوند. به عنوان یک نکته جانبی، تعداد زیادی از آرگومان‌های سازنده بوی کد بدی دارند، به این معنی که کلاس احتمالاً مسئولیت‌های زیادی دارد و باید برای پرداختن به تفکیک صحیح نگرانی‌ها مجدداً اصلاح شود.

تزریق ستر در درجه اول باید فقط برای وابستگی های اختیاری استفاده شود که بتوان مقادیر پیش فرض معقولی را در کلاس به آنها اختصاص داد. در غیر این صورت، در هر جایی که کد از وابستگی استفاده می کند، باید بررسی های غیر تهی انجام شود. یکی از مزایای تزریق تنظیم کننده این است که روش های تنظیم کننده، اشیاء آن کلاس را برای پیکربندی مجدد یا تزریق مجدد بعداً آماده می کنند. بنابراین مدیریت از طریق JMX MBeans یک مورد استفاده قانع کننده برای تزریق ستر است.

از سبک DI استفاده کنید که برای یک کلاس خاص بیشترین معنا را دارد. گاهی اوقات، هنگام برخورد با کلاس های شخص ثالث که منبع آنها را ندارید، انتخاب برای شما انجام می شود. به عنوان مثال، اگر یک کلاس شخص ثالث هیچ روش تنظیم کننده را نشان ندهد، تزریق سازنده ممکن است تنها شکل موجود DI باشد.

##### **Dependency resolution process**

container تفکیک وابستگی لوبیا را به صورت زیر انجام می دهد:

ApplicationContext با فراداده های پیکربندی که همه دانه ها را توصیف می کند ایجاد و مقداردهی اولیه می شود. فراداده پیکربندی را می توان از طریق XML، کد جاوا یا حاشیه نویسی مشخص کرد.

برای هر bean، وابستگی‌های آن در قالب ویژگی‌ها، آرگومان‌های سازنده یا آرگومان‌هایی به روش static-factory بیان می‌شود، اگر از آن به‌جای یک سازنده معمولی استفاده می‌کنید. این وابستگی ها در زمانی که لوبیا واقعاً ایجاد می شود به Bean ارائه می شود.

هر ویژگی یا آرگومان سازنده یک تعریف واقعی از مقداری است که باید تنظیم شود، یا ارجاع به bean دیگری در ظرف است.

هر ویژگی یا آرگومان سازنده که یک مقدار است از قالب مشخص شده خود به نوع واقعی آن ویژگی یا آرگومان سازنده تبدیل می شود. به‌طور پیش‌فرض Spring می‌تواند یک مقدار ارائه‌شده در قالب رشته را به همه انواع داخلی مانند int، long، string، boolean و غیره تبدیل کند.

کانتینر Spring پیکربندی هر دانه را به هنگام ایجاد ظرف تأیید می کند. با این حال، خود خواص لوبیا تا زمانی که لوبیا واقعاً ایجاد نشود تنظیم نمی شود. لوبیاهایی که به صورت singleton هستند و به‌صورت پیش‌فرض تنظیم شده‌اند، هنگام ایجاد ظرف ایجاد می‌شوند. محدوده ها در محدوده های Bean تعریف شده اند. در غیر این صورت، لوبیا فقط در صورت درخواست ایجاد می شود. ایجاد یک bean به طور بالقوه باعث می شود که یک نمودار از bean ایجاد شود، زیرا وابستگی های bean و وابستگی های آن (و غیره) ایجاد و تخصیص داده می شوند. توجه داشته باشید که عدم تطابق وضوح بین آن وابستگی‌ها ممکن است دیر ظاهر شود، یعنی در اولین ایجاد bean تحت تأثیر.

Circular dependencies

اگر عمدتاً از تزریق سازنده استفاده می کنید، ممکن است یک سناریوی وابستگی دایره ای حل نشدنی ایجاد کنید.

به عنوان مثال: کلاس A به یک نمونه از کلاس B از طریق تزریق سازنده نیاز دارد، و کلاس B به نمونه ای از کلاس A از طریق تزریق سازنده نیاز دارد. اگر دانه‌ها را برای کلاس‌های A و B پیکربندی کنید تا به یکدیگر تزریق شوند، ظرف Spring IoC این مرجع دایره‌ای را در زمان اجرا شناسایی می‌کند و BeanCurrentlyInCreationException را پرتاب می‌کند.

یک راه حل ممکن این است که کد منبع برخی از کلاس ها را ویرایش کنید تا توسط تنظیم کننده ها به جای سازنده ها پیکربندی شوند. روش دیگر، از تزریق سازنده خودداری کنید و فقط از تزریق ستر استفاده کنید. به عبارت دیگر، اگرچه توصیه نمی شود، اما می توانید وابستگی های دایره ای را با تزریق ستر پیکربندی کنید.

برخلاف حالت معمولی (بدون وابستگی دایره‌ای)، وابستگی دایره‌ای بین لوبیا A و لوبیا B باعث می‌شود یکی از لوبیاها قبل از اینکه کاملاً مقداردهی شود به دیگری تزریق شود (یک سناریوی کلاسیک مرغ/تخم مرغ).

به طور کلی می توانید به بهار اعتماد کنید تا کار درست را انجام دهد. مشکلات پیکربندی، مانند ارجاع به دانه‌های موجود و وابستگی‌های دایره‌ای را در زمان بارگیری ظرف شناسایی می‌کند. اسپرینگ ویژگی ها را تنظیم می کند و وابستگی ها را تا جایی که ممکن است دیرتر برطرف می کند، زمانی که لوبیا واقعاً ایجاد می شود. این بدان معنی است که یک کانتینر Spring که به درستی بارگیری شده است، می‌تواند بعداً در صورت درخواست یک شی، در صورت وجود مشکل در ایجاد آن شی یا یکی از وابستگی‌های آن، استثنا ایجاد کند. به عنوان مثال، bean یک استثنا را در نتیجه یک ویژگی مفقود یا نامعتبر پرتاب می کند. این مشاهده بالقوه تأخیر در برخی از مسائل پیکربندی به همین دلیل است که پیاده سازی های ApplicationContext به طور پیش فرض از پیش نمونه های singleton bean می کنند. با صرف مقداری زمان و حافظه اولیه برای ایجاد این دانه‌ها قبل از نیاز واقعی، مشکلات پیکربندی را زمانی که ApplicationContext ایجاد می‌شود، کشف می‌کنید، نه بعداً. همچنان می‌توانید این رفتار پیش‌فرض را نادیده بگیرید تا لوبیاهای سینگلتون به جای اینکه از قبل آماده شوند، تنبلی اولیه شوند.

اگر هیچ وابستگی دایره‌ای وجود نداشته باشد، هنگامی که یک یا چند دانه همکار به یک دانه وابسته تزریق می‌شود، هر یک از دانه‌های همکار قبل از تزریق به دانه وابسته کاملاً پیکربندی می‌شود. این بدان معناست که اگر bean A وابستگی به bean B داشته باشد، محفظه Spring IoC قبل از فراخوانی روش تنظیم کننده در Bean A، به طور کامل bean B را پیکربندی می کند. وابستگی ها تنظیم می شوند و روش های چرخه حیات مربوطه (مانند روش init پیکربندی شده یا روش InitializingBean callback) فراخوانی می شوند.

##### Examples of dependency injection

مثال زیر از فراداده های پیکربندی مبتنی بر XML برای DI مبتنی بر تنظیم کننده استفاده می کند. بخش کوچکی از فایل پیکربندی Spring XML برخی از تعاریف bean را مشخص می کند:

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

*<!-- setter injection using the nested ref element -->*

<property name="beanOne">

<ref bean="anotherExampleBean"/>

</property>

*<!-- setter injection using the neater ref attribute -->*

<property name="beanTwo" ref="yetAnotherBean"/>

<property name="integerProperty" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** **ExampleBean** {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** **void** setBeanOne(AnotherBean beanOne) {

this.beanOne = beanOne;

}

**public** **void** setBeanTwo(YetAnotherBean beanTwo) {

this.beanTwo = beanTwo;

}

**public** **void** setIntegerProperty(**int** i) {

this.i = i;

}

}

در مثال قبل، تنظیم کننده ها با ویژگی های مشخص شده در فایل XML مطابقت دارند. مثال زیر از DI مبتنی بر سازنده استفاده می کند:

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

*<!-- constructor injection using the nested ref element -->*

<constructor-arg>

<ref bean="anotherExampleBean"/>

</constructor-arg>

*<!-- constructor injection using the neater ref attribute -->*

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg type="int" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** **ExampleBean** {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** ExampleBean(

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

this.beanOne = anotherBean;

this.beanTwo = yetAnotherBean;

this.i = i;

}

}

آرگومان های سازنده مشخص شده در تعریف bean به عنوان آرگومان های سازنده ExampleBean استفاده خواهند شد.

حال یک نوع از این مثال را در نظر بگیرید، جایی که به جای استفاده از سازنده، به Spring گفته می شود که یک متد کارخانه ای استاتیک را فراخوانی کند تا نمونه ای از شی را برگرداند:

<bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance">

<constructor-arg ref="anotherExampleBean"/>

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** **ExampleBean** {

*// a private constructor*

**private** ExampleBean(...) {

...

}

*// a static factory method; the arguments to this method can be*

*// considered the dependencies of the bean that is returned,*

*// regardless of how those arguments are actually used.*

**public** **static** ExampleBean createInstance (

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

ExampleBean eb = **new** ExampleBean (...);

*// some other operations...*

**return** eb;

}

}

آرگومان‌های روش کارخانه ایستا از طریق عناصر <constructor-arg/> ارائه می‌شوند، دقیقاً مانند اینکه یک سازنده واقعاً استفاده شده باشد. نوع کلاسی که با روش کارخانه برگردانده می شود، لازم نیست از همان نوع کلاسی باشد که متد کارخانه ایستا را در بر می گیرد، اگرچه در این مثال چنین است. یک روش نمونه (غیر ایستا) کارخانه به شکلی اساساً یکسان استفاده می شود (به غیر از استفاده از ویژگی Factory-bean به جای ویژگی کلاس)، بنابراین جزئیات در اینجا مورد بحث قرار نمی گیرند.

### 1.4.2. Dependencies and configuration in detail

همانطور که در بخش قبل ذکر شد، می‌توانید ویژگی‌های bean و آرگومان‌های سازنده را به‌عنوان ارجاع به سایر bean‌های مدیریت‌شده (همکاران)، یا به‌عنوان مقادیر تعریف‌شده درون خطی تعریف کنید. ابرداده پیکربندی مبتنی بر XML Spring از انواع عناصر فرعی در عناصر <property/> و <constructor-arg/> برای این منظور پشتیبانی می کند.

#### Straight values (primitives, Strings, and so on)

ویژگی مقدار عنصر <property/> یک ویژگی یا آرگومان سازنده را به عنوان نمایش رشته قابل خواندن توسط انسان مشخص می کند. سرویس تبدیل Spring برای تبدیل این مقادیر از یک رشته به نوع واقعی ویژگی یا آرگومان استفاده می شود.

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

*<!-- results in a setDriverClassName(String) call -->*

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

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

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

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

</bean>

مثال زیر از فضای نام p برای پیکربندی XML حتی مختصرتر استفاده می کند.

<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

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

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:driverClassName="com.mysql.jdbc.Driver"

p:url="jdbc:mysql://localhost:3306/mydb"

p:username="root"

p:password="masterkaoli"/>

</beans>

XML قبلی مختصرتر است. با این حال، اشتباهات تایپی در زمان اجرا کشف می شوند تا زمان طراحی، مگر اینکه از یک IDE مانند IntelliJ IDEA یا Spring Tool Suite (STS) استفاده کنید که از تکمیل خودکار ویژگی ها هنگام ایجاد تعاریف bean پشتیبانی می کند. چنین کمک های IDE به شدت توصیه می شود.

همچنین می توانید یک نمونه java.util.Properties را به صورت زیر پیکربندی کنید:

<bean id="mappings"

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

*<!-- typed as a java.util.Properties -->*

<property name="properties">

<value>

jdbc.driver.className=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mydb

</value>

</property>

</bean>

ظرف Spring متن داخل عنصر <value/> را با استفاده از مکانیسم JavaBeans PropertyEditor به یک نمونه java.util.Properties تبدیل می کند. این یک میانبر خوب است و یکی از معدود جاهایی است که تیم Spring استفاده از عنصر تودرتوی <value/> را بر سبک ویژگی ارزش ترجیح می دهد.

The idref element

عنصر idref به سادگی یک روش ضد خطا برای ارسال id (مقدار رشته - نه یک مرجع) یک bean دیگر در ظرف به عنصر <constructor-arg/> یا <property/> است.

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

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

<property name="targetName">

<idref bean="theTargetBean"/>

</property>

</bean>

قطعه تعریف bean بالا دقیقاً معادل قطعه زیر (در زمان اجرا) است:

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

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

<property name="targetName" value="theTargetBean"/>

</bean>

شکل اول بر دومی ارجحیت دارد، زیرا استفاده از تگ idref به کانتینر اجازه می دهد تا در زمان استقرار اعتبار سنجی کند که bean ارجاع شده با نام واقعاً وجود دارد. در تغییر دوم، هیچ اعتبارسنجی روی مقداری که به ویژگی targetName از کلاینت bean ارسال می‌شود، انجام نمی‌شود. اشتباهات تایپی تنها زمانی کشف می شوند (با به احتمال زیاد نتایج کشنده) که کلاینت bean واقعاً نمونه سازی شود. اگر کلاینت bean یک نمونه اولیه باشد، این اشتباه تایپی و استثنای ناشی از آن ممکن است مدت‌ها پس از استقرار کانتینر کشف شود.

ویژگی محلی در عنصر idref دیگر در 4.0 beans xsd پشتیبانی نمی‌شود، زیرا دیگر مقداری نسبت به یک مرجع bean معمولی ارائه نمی‌کند. به سادگی هنگام ارتقاء به طرح 4.0، مراجع محلی idref موجود خود را به idref bean تغییر دهید.

یک مکان رایج (حداقل در نسخه‌های زودتر از Spring 2.0) که عنصر <idref/> ارزش می‌آورد، در پیکربندی رهگیرهای AOP در تعریف Bean ProxyFactoryBean است. استفاده از عناصر <idref/> زمانی که نام رهگیر را مشخص می‌کنید، از املای اشتباه شناسه رهگیر جلوگیری می‌کند.

عنصر ref عنصر نهایی درون عنصر تعریف <constructor-arg/> یا <property/> است. در اینجا مقدار خاصیت مشخص شده یک bean را به عنوان مرجعی به bean دیگری (یک همکار) که توسط ظرف مدیریت می شود تنظیم می کنید. bean ارجاع شده یک وابستگی به bean است که ویژگی آن تنظیم می شود و در صورت نیاز قبل از تنظیم ویژگی مقداردهی اولیه می شود. (اگر همکار یک لوبیای تکی باشد، ممکن است قبلاً توسط ظرف مقداردهی اولیه شده باشد.) همه ارجاعات در نهایت ارجاع به شی دیگری هستند. محدوده و اعتبارسنجی به این بستگی دارد که آیا شما شناسه/نام شی دیگر را از طریق صفات bean، local یا والد مشخص کنید.

تعیین bean هدف از طریق ویژگی bean تگ <ref/> عمومی ترین شکل است و اجازه می دهد تا یک مرجع به هر bean در همان ظرف یا ظرف اصلی ایجاد شود، صرف نظر از اینکه در همان فایل XML باشد یا خیر. مقدار مشخصه bean ممکن است با ویژگی id bean هدف یا به عنوان یکی از مقادیر در ویژگی name bean هدف یکسان باشد.

<ref bean="someBean"/>

مشخص کردن bean هدف از طریق ویژگی والد، ارجاعی به bean ایجاد می‌کند که در ظرف والد ظرف فعلی است. مقدار خصیصه والد ممکن است با ویژگی id دانه هدف، یا یکی از مقادیر در ویژگی نام دانه هدف یکسان باشد و دانه هدف باید در ظرف والد موجود باشد. شما از این نوع مرجع bean عمدتاً زمانی استفاده می‌کنید که سلسله مراتبی از کانتینرها دارید و می‌خواهید یک bean موجود را در یک ظرف والد با یک پروکسی بپیچید که همان نام باقلا اصلی است.

*<!-- in the parent context -->*

<bean id="accountService" class="com.foo.SimpleAccountService">

*<!-- insert dependencies as required as here -->*

</bean>

*<!-- in the child (descendant) context -->*

<bean id="accountService" <!-- bean name is the same as the parent bean -->

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target">

<ref parent="accountService"/> *<!-- notice how we refer to the parent bean -->*

</property>

*<!-- insert other configuration and dependencies as required here -->*

</bean>

#### Inner beans

یک عنصر <bean/> در داخل عناصر <property/> یا <constructor-arg/> یک به اصطلاح inner bean را تعریف می کند.

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

*<!-- instead of using a reference to a target bean, simply define the target bean inline -->*

<property name="target">

<bean class="com.example.Person"> *<!-- this is the inner bean -->*

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

#### Collections

در عناصر <list/>، <set/>، <map/> و <props/>، به ترتیب ویژگی‌ها و آرگومان‌های فهرست، مجموعه، نقشه و ویژگی‌های مجموعه جاوا را تنظیم می‌کنید.

<bean id="moreComplexObject" class="example.ComplexObject">

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

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.org</prop>

<prop key="support">support@example.org</prop>

<prop key="development">development@example.org</prop>

</props>

</property>

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

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource" />

</list>

</property>

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

<property name="someMap">

<map>

<entry key="an entry" value="just some string"/>

<entry key ="a ref" value-ref="myDataSource"/>

</map>

</property>

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

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>