*Бин - в Spring это ссылка на любой компонент, управляемый контейнером (по сути экземпляр объекта)*

|  |  |  |
| --- | --- | --- |
| Стиль Xml | Стиль annotation | Примечание |
|  | <context:annotation-config/>  <context:component-scan base-package="**КлассРеализации**">  <context:exclude-filter type="assignable" expression="**КлассРеализации**"/> </context:component-scan>  *дескриптор* annotation-config *сообщает Spring о необходимости просканировать кодовую базу на предмет требований зависимостей*  *дескриптор* component-scan *заставляет Spring сканировать код для поиска внедряемых бинов в рамках указанного пакета (и всех его подпакетов)* | **Выбор варианта конфигурации**  *Для поддержки аннотаций в Spring необходимо поместить в XML-файл конфигурации 2 дескриптора* |
| <bean id="**ИдБина**" name="**Псевдоним1** **Псевдоним2**" class="**КлассРеализации**"/>  *Атрибуты id, name не обязательны*  *Каждый бин должен иметь, по крайней мере, одно имя, которое является уникальным в контексте, содержащим бин.*  *Соглашения: используется имя указаное в атрибуте* id*, иначе первое имя перечсленое в* name*, иначе имя класса реализации*  *В общем случае нужно всегда бину давать имя через* id *и при необходимости ассоциировать его с другими именами* | @Service("Имя\_Бина")  Public class ИмяКласса …..  *В* app-context-annotation.xml *добавить*  <**context:component-scan**  *Дескриптор заставляет Spring сканировать код для поиска внедряемых бинов в рамках указанного пакета (и всех его подпакетов), можно определить множество пакетов через [, ; пробел].* | **Объявление и наименование бинов, псевдонимы**  *Аннотация @Service(имя бина) используется для указания на то, что этот бин предоставляет службы, которые могут понадобиться другим службам* |
| <bean id="**ИдБина**" class="**КлассРеализации**">  <property name="**имяСвойства**">  <ref bean="**Ид или псевдоним бина**"/>  </property> </bean>  *Альтернативный вариант*  <bean id=" ИдБина" class="**КлассРеализации**"  p:messageProvider-ref="**Ид или ПсевдонимБина**"/>  *Прим.: атрибут local в дескрипторе ref устарел, и более не пользуется* | *Добавить к методу установки*  *@*Autowired  *Или*  *@*Resource(name="**ИмяВнедряемогоБина**")  *В* app-context-annotation.xml *добавить* <context:annotation-config/>  *Данный дескриптор сообщает Spring о необходимости просканировать кодовую базу на предмет требований зависимостей Во время иницализаци ApplicationContext Spring обнаруживает аннотации @Autowired для классов аннотируемых с помощью @Service и внедряет эту зависимость, т.е. вызывает set-метод, где в качестве параметра передаётся экземпляр внедряемого объекта. Т.к. таких м.б. несколько, лучше всегда использовать имя конкретного бина в аннотации @Resource* | **Внедрение бинов через метод установки (Setter**  **DI)**  *Resource - для реализации более точных требований DI* |
| <**bean id="ИдБина" class="КлассРеализации"**>  <**constructor-arg type="int" index="0" value="3"**/>   *<!-- <constructor-arg type="java.lang.String" index="0" value="ЗначениеАргумента"/>-->*  *<!-- <constructor-arg>  <value>ЗначениеАргумента ПростогоТипа</value>  <ref>имяБина</>*  *</constructor-arg>-->* </**bean**>  *<!-- Тоже самое, пример внедрения через конструктор используя пространство имён "c"-->* <**bean id=" ИдБина" class=" КлассРеализации" c:аргумент="Конфигурируемое сообщение"**/> | *Добавить к одному из конструкторов аннотацию @Autowired и определить значение аргумента в аннотации Value*  *@*Autowired  **public** ConfigurableMessageProvider(@Value(**"Конфигурируемое сообщение"**) String **message**) {…}  *Для того чтобы избавиться от аннотации Value в целевом бине нужно объявить имя бина в контексте и его идентификатор должен совпадать с именем аргумента, в нашем случае message*  <**bean id="message" class="java.lang.String" c:\_0="Конфигурируемое сообщение"** /> | **Внедрение бинов через конструктор (Constructor DI)**  *Если несколько аргументов или несколько конструкторов, то каждому дескриптору constructor-arg нужно предоставить атрибут* ***index****. Если у класса несколько конструкторов, то нужно указать ещё и тип аргумента, для того чтобы создать объект (экземпляр класса-бин) вызвав нужный конструктор.*  *Дескриптор <****value****> используется для простых типов, для внедрения объектов попрежнему нужно использовать <ref>* |
| <**bean id="injectSimple" class="КлассРеализации"**>  <**property name="name"**>  <**value**>John Smith</**value**>  </**property**>  <**property name="height"**>  <**value**>1.78</**value**>  </**property**> | @Value(**"John Smith"**)  **private** String **name**;  @Value(**"35"**)  **private int age**; | **Параметры внедрения (простые значения)** |
| *<!-- Конфигурационный бин, содержащий значения, предназначенные для внедрения -->*  <**bean id="injectSimpleConfig" class="xml.InjectSimpleConfig"**/>  *<!-- Внедрение свойств, значения которые берутся из конфигурационого бина -->* <**bean id="injectSimpleSpel" class="xml.InjectSimpleSpel"**>  <**property name="name" value="#{injectSimpleConfig.name}"**/>  <**property name="age" value="#{injectSimpleConfig.age + 1}"**/>  <**property name="height" value="#{injectSimpleConfig.height}"**/>  <**property name="programmer" value="#{injectSimpleConfig.programmer}"**/>  <**property name="ageInSeconds" value="#{injectSimpleConfig.ageInSeconds}"**/> </**bean**> | 1. *Пометить конфигурационный класс аннотицией @Component*   *Вместо @Service. Обе аннотации дают одинаковый эффект, но т.к. класс хранит только конфигурацию и не предоставляет бизнес-службу другим уровням приложения, использование @Component имеет больший смысл*  @Component(**"injectSimpleConfig"**) **public class** InjectSimpleConfig {   1. *Далее, для внедрения достаточно использования аннотациии @Value.*   @Service(**"injectSimpleSpel"**) **public class** InjectSimpleSpel {   @Value(**"#{injectSimpleConfig.name}"**)  **private** String **name**; | **Параметры внедрения (SpEL - Spring Expression Language)**  *Суть в том, что параметры конфигурации выносятся в конфигурационный Java-классе (injectSimpleConfig) и далее внедряются в бины с использованием специального языка - SpEL*  *Такой подход позволяет динамически вычислять параметры и затем применять в ApplicationContext* |
| *Чтобы вложить один контекст в другой необходимо вызвать метод setParent в дочернем контексте. Это позволяет разделять конфигурацию на отдельные файлы. Удобно для крупных проектов, содержащих множество бинов.*  GenericXmlApplicationContext parent = **new** GenericXmlApplicationContext();  parent.load(**"classpath:META-INF/spring/parent.xml"**);  parent.refresh();    GenericXmlApplicationContext child = **new** GenericXmlApplicationContext();  child.load(**"classpath:META-INF/spring/app-context-xml.xml"**); *// вложение контекста* child.**setParent**(parent);  child.refresh();  <**bean id="target3" class=" КлассРеализации"**>  <**property name="val"**>  <**ref parent="injectBean"**/>  </**property**> |  | **Иерархия ApplicationContext**  *в дескрипторе ref можно ссылаться на бины как в дочернем так и в родительском контекстах. это позволяет перемещать бины между конфигурационными файлами по мере роста приложения*  *Если 2 обинаковых бина определён ав обоих контекстах, то используется собственый контекст.*  *для внедрения родительского бина требуется явное указание атрибута parent* |
| <**bean id="injectCollection" class="org.billing.jlab.spring.ch4.xml.CollectionInjection"**>  <**property name="map"**>  <**map**>  <**entry key="someValue"**>  <**value**>Hello World!</**value**>  </**entry**>  <**entry key="someBean"**>  <**ref bean="oracle"**/>  </**entry**>  </**map**>  </**property**>  <**property name="props"**>  <**props**>  <**prop key="firstName"**>Clarence</**prop**>  <**prop key="secondName"**>Ho</**prop**>  </**props**>  </**property**>  <**property name="set"**>  <**set**>  <**value**>Hello World!</**value**>  <**ref bean="oracle"**/>  </**set**>  </**property**>  <**property name="list"**>  <**list**>  <**value**>Hello World!</**value**>  <**ref bean="oracle"**/>  </**list**>  </**property**> </**bean**> | *<!-- Пример внедрения коллекций -->* <**util:map id="map" map-class="java.util.HashMap"**>  <**entry key="someValue" value="Hello World!"**/>  <**entry key="someBean" value-ref="oracle"**/> </**util:map**> <**util:properties id="props"**>  <**prop key="firstName"**>Clarence</**prop**>  <**prop key="secondName"**>Ho</**prop**> </**util:properties**> <**util:set id="set"**>  <**value**>Hello World!</**value**>  <**ref bean="oracle"**/> </**util:set**> <**util:list id="list"**>  <**value**>Hello World!</**value**>  <**ref bean="oracle"**/> </**util:list**>  *В java-классе используются обычные аннотации*  @Resource(name=**"map"**) **private** Map<String, Object> **map**;  @Resource(name=**"props"**)  **private** Properties **props**;  @Resource(name=**"set"**)  **private** Set **set**;  @Resource(name=**"list"**)  **private** List **list**; | **Внедрение коллекций**  *Для доступа к коллекциям бинам, а не только к отдельным бинам или значениям*  *Чтобы внедрить коллекцию необходимо выбрать дескриптор list, map, set, props для предоставления экзамляров List, Map, Set, Properties соответственно Для list, map, set можно использовать любой дескриптор при внедрении в свойство и даже дескриптор другой коллекции Для props позволяет передать только String* |
|  |  | **Внедрение через метод** |
|  |  | **Замена метода** |
| *<!-- Конфигурация необиночного бина  область действия бина на уровне прототипа заставляет Spring создавать новый экзампляр бина каждый раз, когда он запрашивается приложением -->* <**bean id="nonSingleton" class="java.lang.String" scope="prototype"**>  *// получаем экзампляры объектов с использованием одного и того же имени бина* String s1 = (String) ctx.getBean(**"nonSingleton"**); String s2 = (String) ctx.getBean(**"nonSingleton"**);  System.***out***.println(**"Экзампляры разные!!! "** + (s1 ==s2)) | *Примечание: Не путать одиночный объект - имеющий единственный экзампляр в приложении и шаблон проектирования Singleton Singleton увеличивает степень связанности, т.е. код приложения должен всегда явно знать о классе Singleton, чтобы получить его экземпляр и это затрудняет замену реализаций произвольным образом*  *По умолчанию Spring просто НЕ создает новый экзампляр бина каждый раз, когда он запрашивается приложением, а даёт ссылку на уже существующий, хотя сам класс может не поддерживать ШП Singleton, т.е. если его явно создать в приложении с помощью new, то он создастся как новый.* | **Режим создания бинов**  *По дефолту все бины в Spring являются одиночными (Singleton)*  *Главное преимущество связано с уменьшением потребления памяти благодаря одиночным экзамплярам* |
| *Требуется, когда зависимости внедряются без уведомдения об этом Spring, например в конструкторе beanA получаем экземпляр beanB путем явного вызова* ctx.getBean(**"beanB"**)  *Если контекст Spring об внедрении этой зависимости ничего не знает, то бины создадуться и сконфигурируются в произвольном порядке. Очевидно, что если beanA создасться и сконфируриться до создания его зависимостей , т.е. beanB, то могут возникнуть проблемы.*  <**bean id="beanA" class="КлассРеализации" depends-on=”beanB”**> | | **Разрешение зависимостей**  *При проектировании такого подхода лучше избегать, но не всегда получается, при интеграции с унаследованным кодом* |
|  |  | **Автосвязывание** |
|  |  | **Наследование бинов** |
|  |  |  |
|  |  |  |