# 1. Core Technologies

## 1.1 Introduction to the Spring IoC Container and Beans

Dependency Injection это специализированная форма IoC, при которой объекты определяют свои зависимости только через аргументы конструктора, аргументы фабричного метода или свойства, которые установлены в экземпляре объекта после того как он был сконструирован и возвращен от фабричного метода.

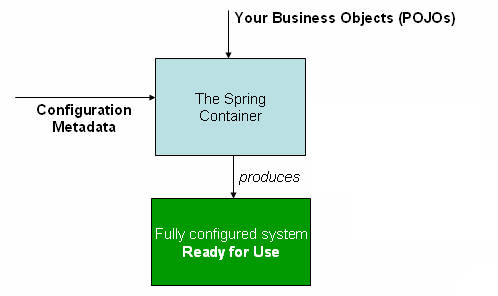
По сути своей этот механизм является обратным, отсюда и название.

BeanFactory интерфейс обеспечивает конфигурационный механизм, способный управлять любым типом объекта. ApplicationContext является наследником BeanFactory. Он добавляет:

* Легкую интеграцию с Spring AOP
* Message resource handling для интернационализации
* Публикации ивентов
* Специфичный контекст на уровне слоя приложения такие как WebApplicationContext для использования в вэб приложениях.

Объекты в Spring, которые формируют скелет приложения и управляются Spring IoC контейнером называются бинами.

## 1.2 Container Overview



### 1.2.1 Configuration metadata

Конфигурационные метаданные могут собираться исходя из разных условий:

1. Из конфигурационного xml файла
2. Из Java аннотаций (на сегодня всё более используемый)
3. Непосредственно из кода Java (@Bean, @DependsOn, @Configuration)

### 1.2.2 Instantiating a Container

Конфигурационные метаданные могут собираться исходя из разных условий:

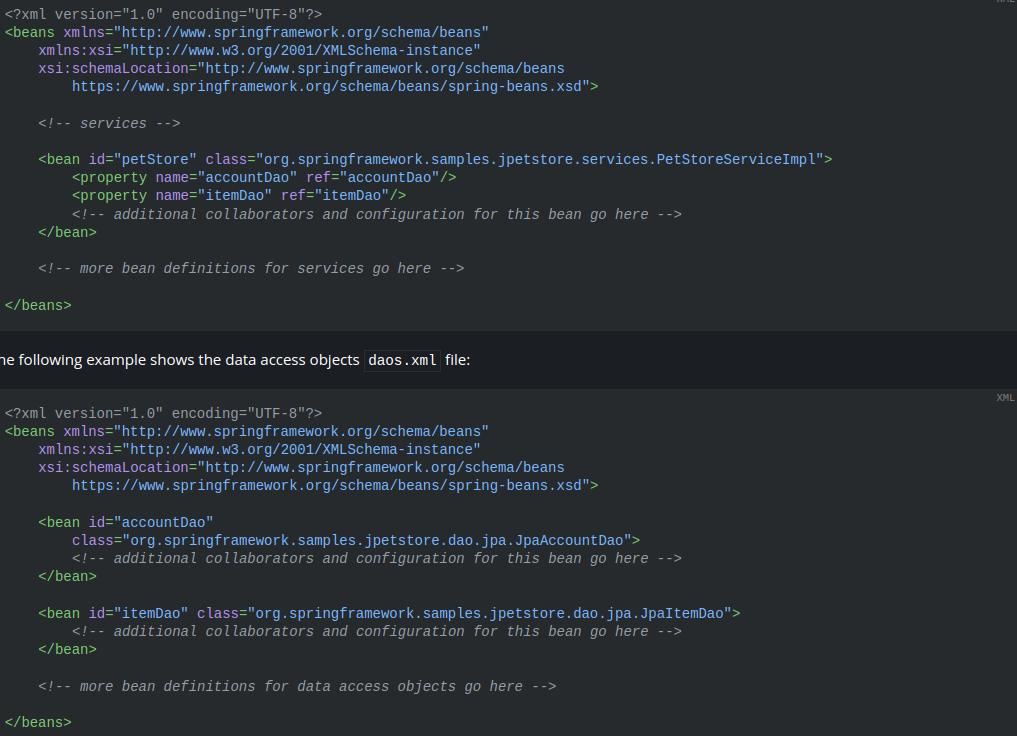
По сути контейнер спринг и его контекст собирается исходя из метаданных, которые могут быть описаны в разных форматах.

Чаще всего используют xml, Java-based форматы. Но возможны несколько других, например Groovy DSL формат, в котором бины и их зависимости описываются языком groovy.

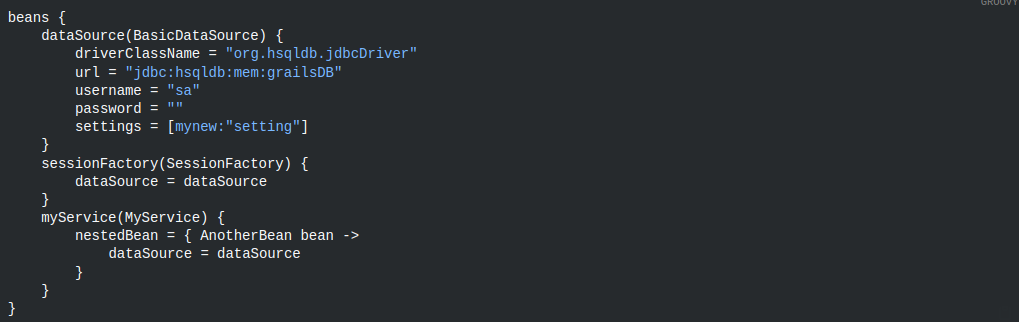
Контекст может быть сформирован только из одного формата данных, либо можно использовать контекст, который имеет возможность загружать метаданные из всех форматов метаданных

XML формат:

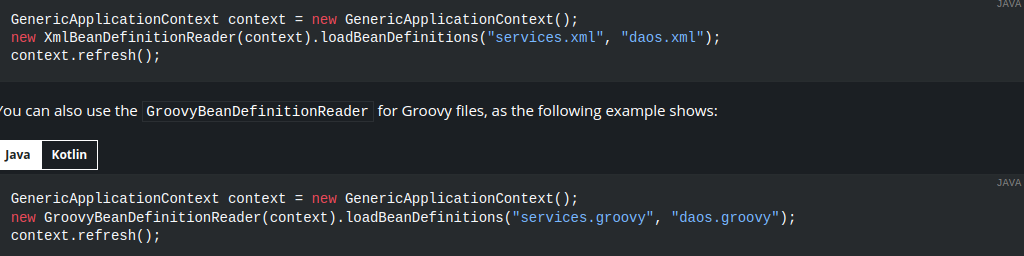




Groovy Bean Definition DSL формат:



 Использование комбинированного контекста:



Можно использовать метод getBean, который присутствует в интерфейсе ApplicationContext. При помощи него можно получить бин из конгтейнера. Но в коде приложения вообще не должно быть использование Spring Api. Напротив, ваши объекты должны работать самостоятельно и использовать зависимости, которые описаны в конфигурационных метаданных для автоматического их использования.

## 1.3 Bean Overview

Спринг IoC контейнер управляет одним или более бинами. Эти бины создаются из конфигурационных метаданных, которые вы поставляете в контейнер, например, через xml конфигурацию.

Внутри самого контейнера эти бины представлены как BeanDefinition объектами, которые содержат (среди прочей информации) следующие метаданные:

1. Наименование класса с его пакетом.
2. Конфигурационные элементы поведения бины, которые определяют как бин должен себя вести в контейнере (область видимости, коллбэки жизненного цикла и т. д.)
3. Ссылки на другие бины, которые нужны, чтобы этот бин работал. Такие бины так же называются соавторами или зависимостями.
4. Другие конфигурационные настройки, которые необходимо установить во вновь созданном объекте.

### 1.3.1 Naming Beans

Каждый бин имеет свой один или более идентификаторов. Эти идентификаторы должны быть уникальны в пределах контейнера, который их содержит. Обычно у бина только один идентификатор.

Бины обычно именуются с маленькой буквы в кэмел кейсе и отражают название класса (applicationService, acceptDao и т. д.).

Можно создавать псевдонимы для бинов, т. е. условно его второе имя для использования в других сервисах. Например, для xml конфигурации можно использовать как:



### 1.3.2 Instantiating Beans

По сути своей bean definition это рецепт для создания одного или более объектов. Контейнер смотрит на рецепт для запрашиваемого бина и использует конфигурационные метаданные инкапсулированные bean definition чтобы создать или получить актуальный объект.

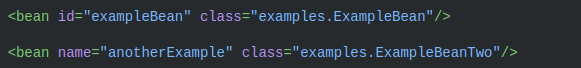
Если использовать xml конфигурацию, обычно определяется класс в атрибуте class <bean/> элемента. Этот атрибут Class (который внутренне является свойством Class в Bean Definition) обычно обязательный. Можно использовать Class по одному из двух способов:

* Обычно чтобы указать класс бина для его создания, в случае, если контейнер сам непосредственно создает bean компонент, рефлексивно вызывая его конструктор, что в некторой степени эквивалентно new оператору.
* Чтобы указать фактический класс, содержащий статический фабричный метод, который вызывается для создания объекта, в менее распространенном случае, когда контейнер вызовет статический фабричный метод класса для создания компонента.

#### 1.3.2.1 Instantiation with a Constructor

Когда вы создаете bean-компонент с помощью конструктора, все обычные классы могут использоваться и совместимы с Spring. То есть разрабатываемый класс не должен реализовывать какие-либо конкретные интерфейсы или быть закодирован определенным образом. Достаточно просто указать класс компонента. Однако в зависимости от того, какой тип IoC вы используете для этого конкретного компонента, вам может потребоваться конструктор по умолчанию (пустой).

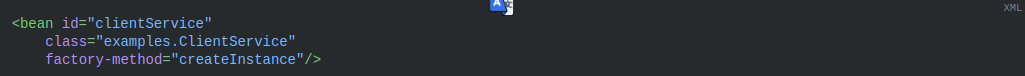
Пример XML:



#### 1.3.2.2 Instantiating with a Static Factory Method

При определении bean-компонента, который вы создаете с помощью статического фабричного метода, используйте атрибут class, чтобы указать класс, который содержит статический фабричный метод, и атрибут с именем Factory-method, чтобы указать имя самого фабричного метода. У вас должна быть возможность вызвать этот метод (с необязательными аргументами, как описано ниже) и вернуть живой объект, который впоследствии обрабатывается так, как если бы он был создан с помощью конструктора. Одним из применений такого определения компонента является вызов статических фабрик в устаревшем коде.

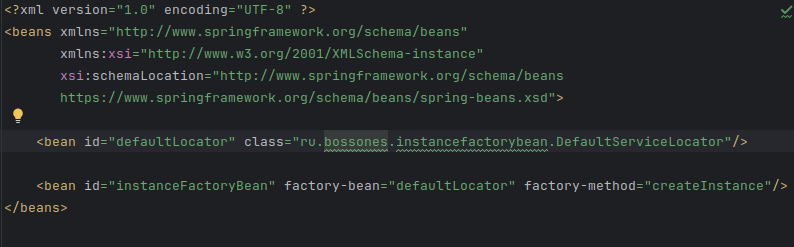
Пример XML:

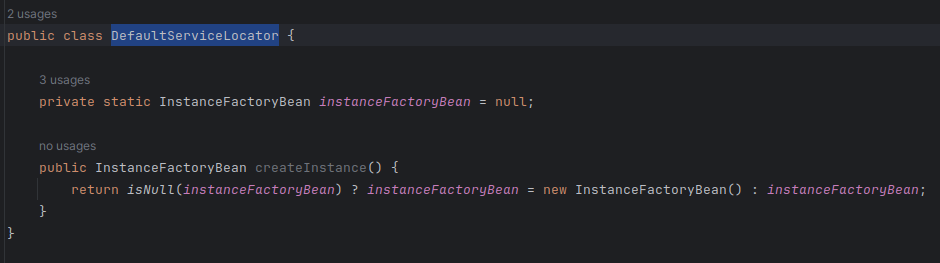


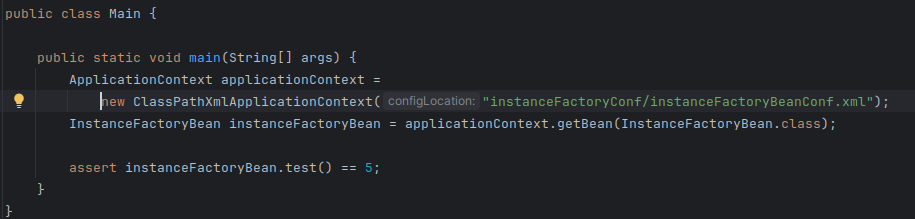
#### 1.3.2.3 Instantiation by Using an Instance Factory Method

Создание экземпляра через данный фабричный метод похож на статический фабричный метод, но здесь применяется нестатический метод экземпляра класса. Чтобы использовать данные механизм, не нужно задавать атрибут class в xml bean definition и в factory-bean атрибуте определяется имя бина в (текущем или родительском) контейнере который содержит инстансный метод который будет вызван чтобы создать объект.

Примеры XML:







## 1.4 Dependencies

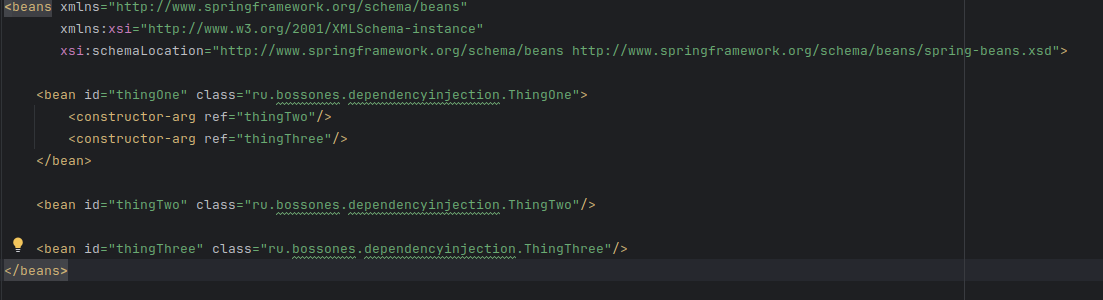
### 1.4.1 Dependency Injection

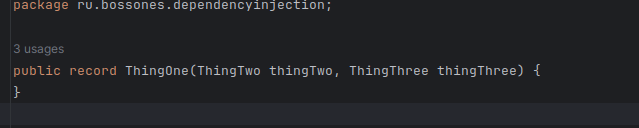
Dependency Injection это процесс, посредством которого объекты определяют их зависимости только через аргументы конструктора, аргументы фабричного метода или свойствами, которые устанаваливаются экземпляру объекта после того как он сконструирован или возвращен из фабричного метода. Затем контейнер внедряет эти зависимости при создании бина. Этот процесс по сути является инверсивным (отсюда и название — Inversion of Control) самому компоненту, управляющему созданием своих зависимостей самостоятельно, используя прямое создание классов или шаблон Service Locator.

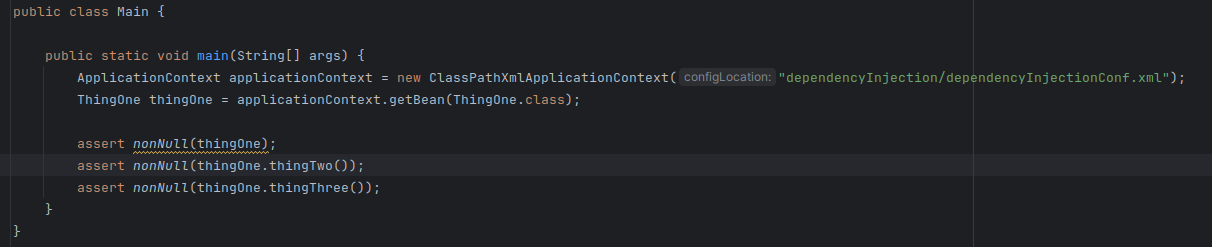
#### 1.4.1.1 Constructor Argument Resolution

Разрешение аргументов конструктора имеет место в использовании типов аргументов. Если нет потенциальной двусмысленности в аргументах конструктора bean definition, порядок аргументов конструктора, который определен в bean definition, является порядком в котором эти аргументы поставляются соответствующему конструктору при создании экземпляра компонента.

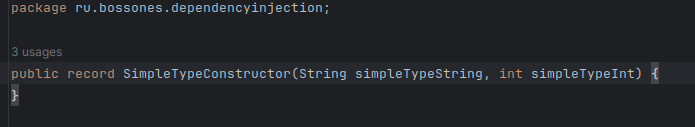
Например в XML Bean Definition:

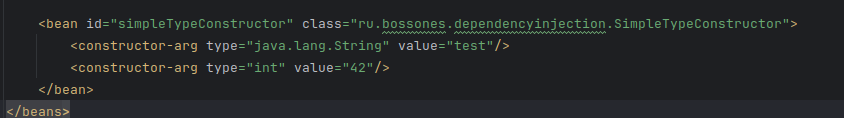




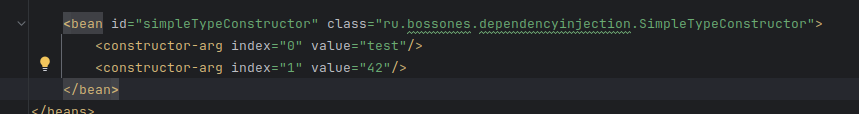


Когда идет ссылка на другой бин, то тип его известен и совпадение может произойти, как в кейсе выше. Но когда применяются примитивные типы в конструкторе, то Spring может не понять какой тип нужно использовать. Поэтому нужно ему помочь в этом.





Так же можно задавать индекс аргументов конструктора в bean definition.

 При использовании одинаковых примитивных типов в конструкторе, использование индекса в bean definition помогает разрешить двусмысленность.

Так же можно использовать имя аргументов в конструкторе, чтобы разрешать двусмысленность.

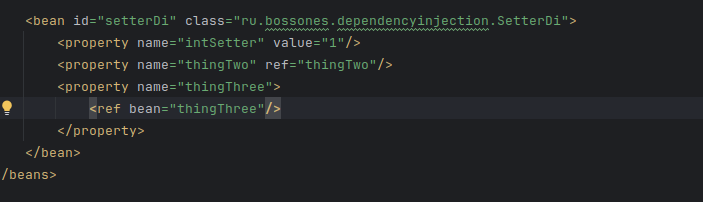
#### 1.4.1.2 Setter-based DI

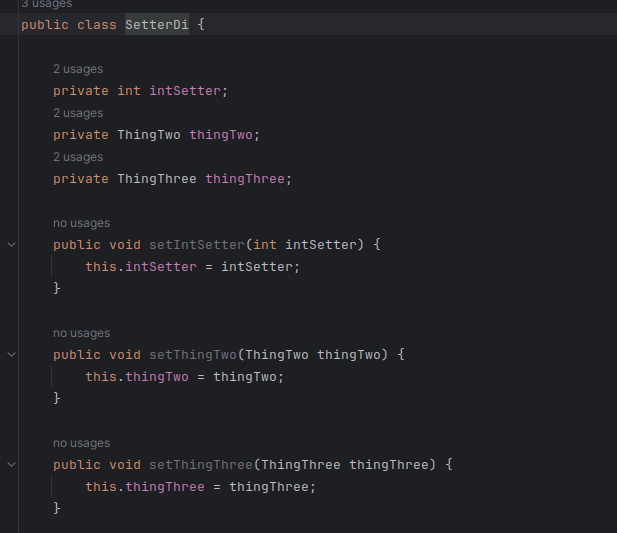
DI базированный на сет методах выполняется контейнером вызывая сеттер методы на бине после выполнения no-args конструктора или no-args статическтого фабричного метода для создания экземпляра вашего бина.

Может использоваться в том случае, если зависимости опицональны, то есть равны null.

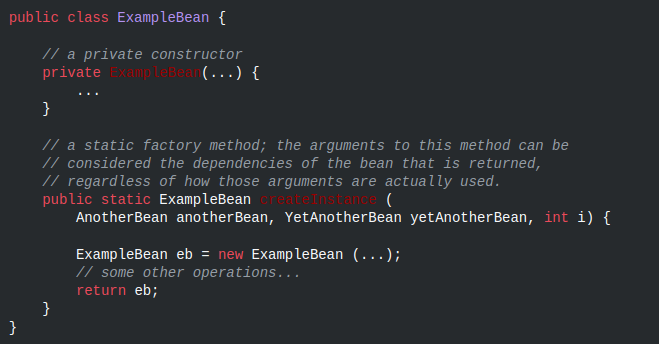
Так же такой метод DI может быть полезен при разрешении циклических зависимостей, т. к. сеттер DI используется уже после того, как бин текущий и бин зависимостей будут созданы. Соответственно зависимость сможет ввнедриться в нужный бин через сеттер метод.

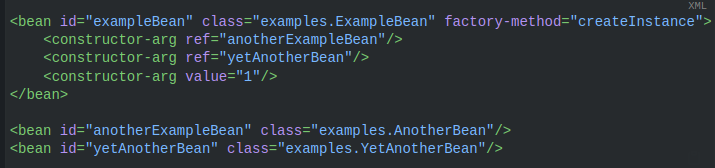
Пример XML:





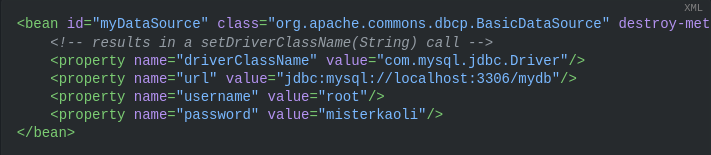
Аналогично Constructor DI можно сделать фабричный метод с несколькими аргументами, которые будут заинжекчены в конструктор в том же порядке:

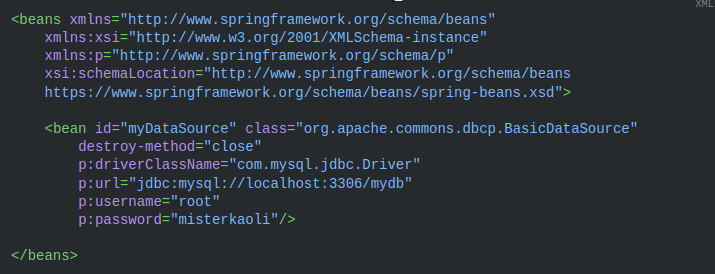




### 1.4.2 Dependencies and Configuration in details.

Для установки значений через сеттеры можно использовать p-namespace, который используется как property элемент. К примеру:





Объявление получается очень кратким, но опечатки обнаруживаются только в рантайме, а не во время разработки. Поэтому лучше использовать IDE.

Так же дополнительно можно конфигурировать java.util.Properties которые всегда можно взять в приложении.



#### 1.4.2.1 Idref элемент.

Данный элемент позволяет очень легко избежать ошибок при передаче идентификатора бина как зависимость для другого через сеттер или через конструктор.

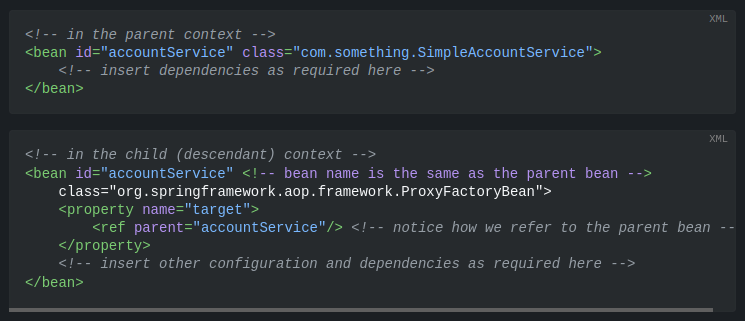
У меня не работет, хз почему(((

#### 1.4.2.2 References to Other Beans (Collaborators)

Общей формой для ссылки на другой бин является использование элемента <ref bean=<<someBean>>> где указывается либо идентификатор бина либо его другое любое наименование в этом контейнере или в любом родительском контейнере.

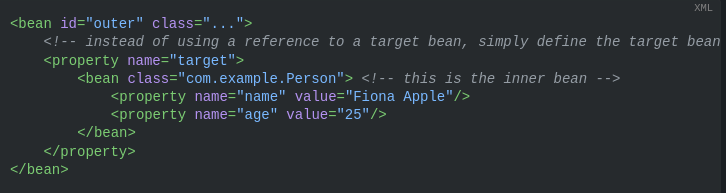
Если определять через parent атрибут, то он создает ссылку на бин, который находится в родительском контейнере текущего контейнера. Значение этого атрибута должно быть идентификатором бина или любым именем этого бина в родительском контейнере текущего контейнера. Необходимо использовать такую ссылку на бин, если у присутствует иерархия контейнеров и необходимо обернуть существующий бин в родительском контейнере с прокси, который имеет такое же имя как родительский бин.

Пример:



#### 1.4.2.3 Внутренние бины

Используя элемент <bean внутри <property/> или <constructor-arg/> элементах можно определить внутренний бин как показано на примере:

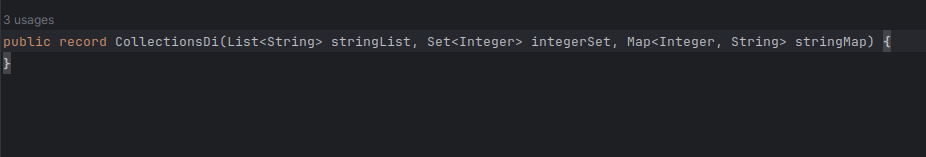


Внутреннее определение бина не требует идентификатора или имени. Если он использован, то контейнер не использует значение как идентификатор. Контейнер так же игнорирует scope при создании, поскольку внутренние бины всегда анонимны и создаются вместе с внешним бином. Нет возможности получить доступ к внутреннему бину независимо от внешнего или инжектировать его в другие бины.

#### 1.4.2.4 Collections

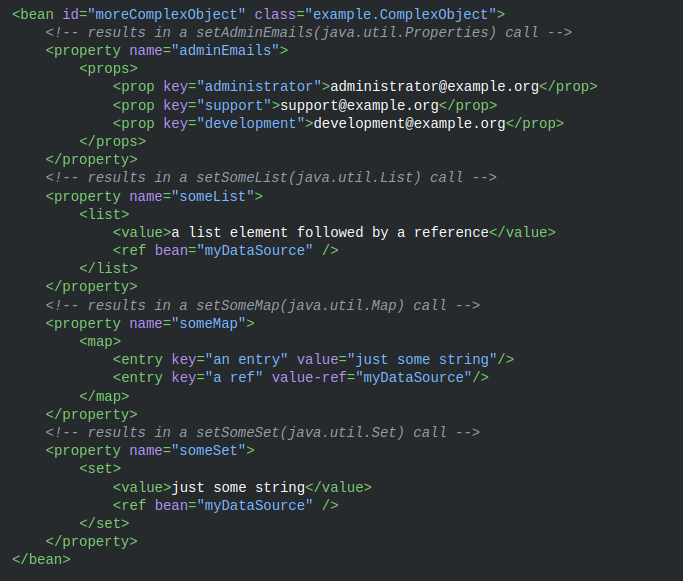
Так же можно задавать элементы коллекций для использования их как аргументы конструктора или как setter DI.

К примеру:



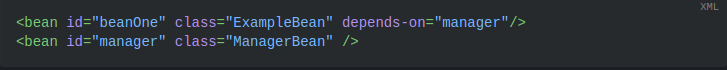


Вместо обычных примитивных значений это могут быть списки бинов или сеты бинов или мапы бинов. Одним словом, практически всё что угодно.

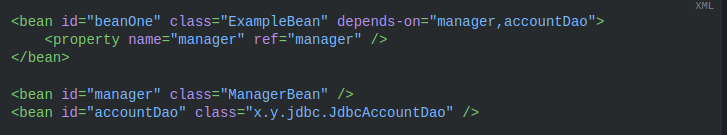


### 1.4.3 Using depends-on

Атрибут depends-on позволяет явно обозначить, что один или более бинов должны быть инициализированы до того, как бин, использующий этот элемент, будет инициализирован.



Для того, чтобы перечислить несколько бинов они перечисляются через запятые, пробелы и точки с запятой:



Атрибут depends-on может задавать как зависимость от времени инициализации, так и, в случае только бинов-одиночек, соответствующую зависимость от времени уничтожения. Зависимые бины, которые определяют depends-on отношения с данным бином, уничтожаются в первую очередь, вплоть до уничтожения самого данного бина. Таким образом, depend-on может также контролировать порядок завершения.

### 1.4.4 Lazy-initialized beans

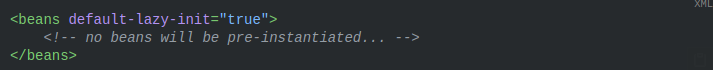
По умолчанию реализации ApplicationContext жадно создают и конфигурируют все синглтон бины как часть процесса инициализации. Как правило, такая предварительная реализация желательна, поскольку ошибки в конфигурации или окружающей среде обнаруживаются немедленно, а не часами или днями позже. Если же поведение не определено, можно самостоятельно предотвращать пре-инициализацию синглтон-бинов, если указать, что они lazy-initialized. Это свойство бина говорит контейнеру, чтобы создать экземпляр бина когда он впервые потребуется, а не при запуске.

В xml bean definition это свойство регулируется атрибутом lazy-init внутри <bean/> элемента:

 В примере выше бин lazy предварительно не создается, когда запускается ApplicationContext, в то время как бин not.lazy создается предварительно без всяких задержек.

Однако, если бин с отложенной инициализацией является зависимостью синглтон-бина, который не является lazy-бином, то ApplicationContext создает lazy-бин при запуске, поскольку он должен удовлетворять зависимостям бина-одиночки. Lazy-бин внедряется в синглтон-бин в другом месте, где неот отложенной инициализации.

Можно так же управлять lazy-инициализацией на уровне контейнера, используя атрибут default-lazy-init в элементе <beans/>



### 1.4.5 Autowiring Collaborators

Spring контейнер может автоматически связывать отношения между взаимодействующими бинами. Можно дать спрингу автоматически разрешить установление связни для бинов посредством инспектирования контента в ApplicationContext. Автосвязывание имеет следующие преимущества:

1. Существенно может уменьшать необходимость определять свойства (properties) или аргументы конструктора.

2. Автосвязывание может обновлять конфигурации по мере развития объектов. Например, если необходимо добавить зависимость в класс, то эта зависимость может быть установлена автоматически без необходимости модифицироовать конфигурацию. Таки образом, автосвязывание может быть особенно полезно при разработке, не исключая возможности переключения на явное связывание, когда база кода станет более стабильной.

Для xml bean definition можно применить атрибут autowire в <bean/> элементе. Имеется 4 режима автосвязывания.

|  |  |
| --- | --- |
| Режим | Разъяснения |
| no | Дефолт. Без автосвязывания. Ссылки на бины должны быть определены через ref элементы. Изменение дефолтных настроек не рекомендуется для больших приложений, т. к. определение связности зависимостей явно даёт больший контроль и четкость. |
| byName | Автосвязывание по имени свойства. Спринг будет смотреть на бин с таким же именем как у свойства, которое должно быть связано. Например, если у bean definition установлена автосвязность по имени и содержит свойство master (так, что у него есть сеттер setMaster()), спринг найдет bean definition с именем master и использует его для установления свойства. |
| byType | Позволяет свойству быть связанным, если один и только один бин с типом как у этого свойства существует в контейнере. Если существует более одного бина, будет выброшено исключение, которое говорит, что нельзя использовать автосвязывание по типу для этого бина. Если нет ни одного подходящего бина, то свойство не устанавливается. |
| constructor | Аналогично режиму byType но применяется для аргументов конструктора. Если нет ни одного подходящего бина, выбрасывается ошибка. |

Ограничения и недостатки автосвязывания:

1. Явно установленные зависимости в property и constructor-arg всегда переопределяют автосвязывание.

2. Автосвязывание менее понимаемо, в отличии от явного связывания. Хотя спринг с осторожностью пытается избежать угадывание в случае двусмысленности, которое может привести к непредвиденным результатам. Отношения между управляемыми объектами Спринга больше не документируется явно.

3. Информация о подключении может быть не доступна для инструментов, которые могут генерировать документацию из контейнера спринга.

Исключение бина из автосвязывания.

Можно попробовать исключать бины для автосвязывания. Для xml фоормата устанавливается атрибут autowire-candidate=false для элемента <bean/>. Контейнер сделает этот особенный bean definition не доступным для автосвязывания (включая автосвязывание через аннотацию @Autowiring).

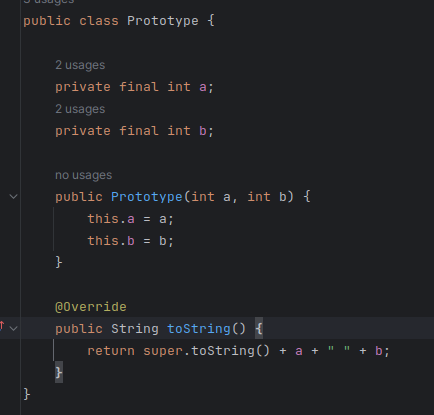
### 1.4.6 Method injection

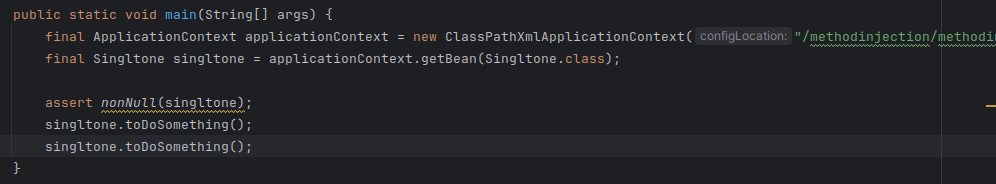
В большинстве сценариев подавляющее количество бинов являются синглтонами. Когда синглотн бин взаимодействет с другим синглтон бином или не синглтон бин взаимодействет с другим не синглтон бином, вы обычно управляете зависимостями путем определения одного бина как свойство другого бина. Проблемы возникают, когда жизненные циклы бинов разные. Прдположим, что синглтон бин А требует использовать не синглтон (prototype) бин Б, возможно в каждом вызове метода в бине А. Контейнер создаст синглтон бин А только один раз и, таким образом, получит воззможность установить свойства. Контейнер не может дать бину А новый экземпляр бина Б каждый раз, когда он нужен.

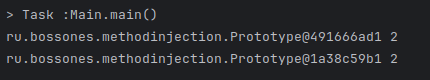
Решение заключается в том, чтобы отказаться от инверсии управления. Можно сделать бин осведомленным о контейнере благодаря реализации ApplicationContextAware интерфйеса, вызывая создание бина Б через контейнер каждый раз, когда он необходим.

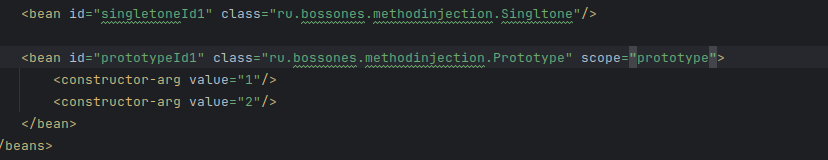
Пример:







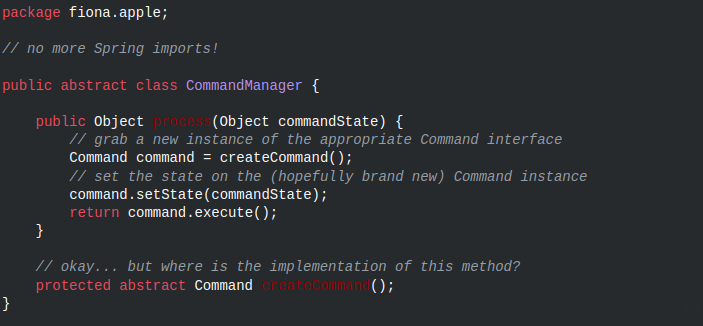




#### 1.4.6.1 Lookup Method Injection

Внедрение зависимости через метод поиска - это способность контейнера переопределять методы управляемых контейнером бинов и возвращать результат поиска для другого именованного бина в контейнере. Поиск обычно предусматривает бин-прототип, как в сценарии, описанном в предыдущем разделе. Фреймворк Spring реализует это внедрение зависимости через метод, используя генерацию байт-кода из библиотеки CGLIB для динамической генерации подкласса, который переопределяет метод.

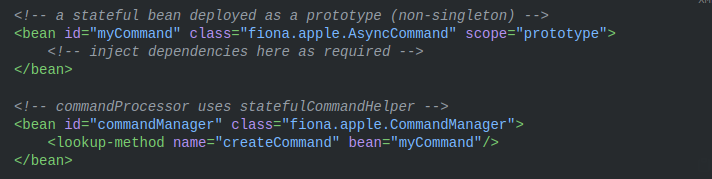
* Чтобы это динамическое построение подклассов работало, класс, который контейнер бина Spring разделяет на подклассы, не может быть final, и переопределяемый метод тоже не может быть final.
* Модульное тестирование (юнит-тестирование) класса, в котором присутствует abstract метод, требует, чтобы вы самостоятельно создали подкласс класса и обеспечили реализацию функции-заглушки abstract метода.
* Также необходимы конкретные методы для сканирования компонентов, для чего требуются конкретные классы.
* Еще одним ключевым ограничением является то, что методы поиска не работают с фабричными методами и, в частности, с методами @Bean в конфигурационных классах, поскольку в этом случае контейнер не отвечает за создание экземпляра и поэтому не может создать генерируемый во время выполнения подкласс на лету.



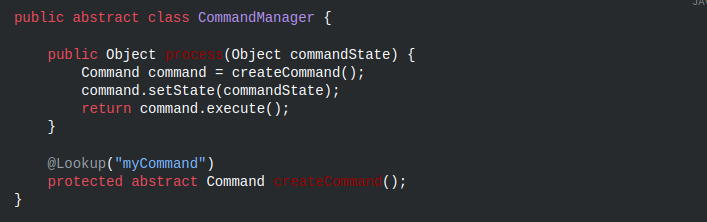
Метод поиска должен соответствовать следующей сигнатуре:



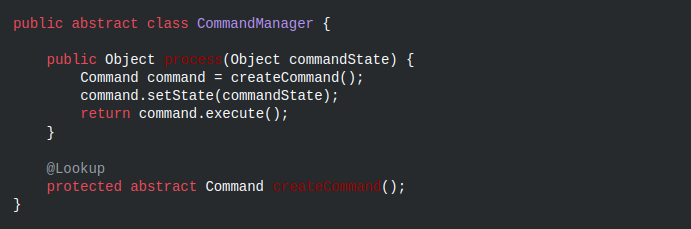
В xml bean definition будет так:



Можно также использовать аннотации Java с указанием конкретного бина:



А можно и без указания, положившись на тип бина, который должен быть подставлен в возвращаемый результат метода.



## 1.5 Bean Scopes

Для бинов так же можно определять и область их действия, границ. Это очень удобная, мощная и гибкая штуковина, которая позволяет над определенным bean definition проставить его область действия (scopes). Можно не заботиться об области действия внутри Java кода. Спринг поддерживет 6 типов области существования, 4 из которых существуют только, если используется web ApplicationContext.

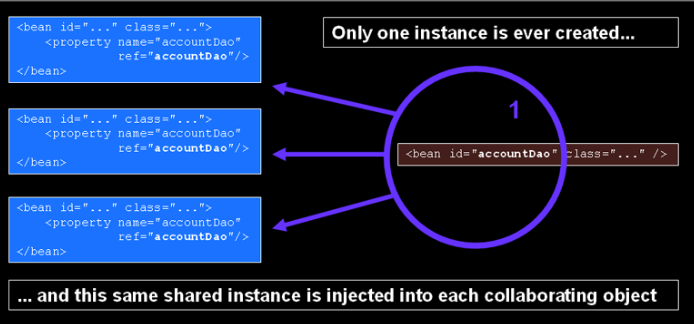
Bean scopes:

|  |  |
| --- | --- |
| Singleton | Дефолтный скоп для бина, который создает единственный жкземпляр класса для всего IoC контейнера. |
| Prototype | Скоуп для бина, который значит, что у данного класса возможно более одного экземпляра (неограниченно) |
| Request | Применяет область видимости определения единственного бина для жизненного цикла единственного HTTP-запроса. Таким образом, для каждого нового HTTP-запроса создается новый экземпляр бина на основе единственного определения бина. Действует только в контексте ориентированного на использование в веб-среде (web-aware) ApplicationContext фреймворка Spring. |
| Session | Применяет область видимости единственного определения бина для жизненного цикла HTTP Session. Действует только в контексте ориентированного на использование в веб-среде (web-aware) ApplicationContext фреймворка Spring. |
| Application | Применяет область видимости единственного определения бина для жизненного цикла ServletContext. Действует только в контексте ориентированного на использование в веб-среде (web-aware) ApplicationContext фреймворка Spring. |
| Websocket | Применяет область видимости единственного определения бина для жизненного цикла WebSocket. Действует только в контексте ориентированного на использование в веб-среде (web-aware) ApplicationContext фреймворка Spring. |

На сегодняшний день в Spring 3.0 доступна потоковая область видимости, но она не представлена по умолчанию. Подробнее в SimpleThreadScope.

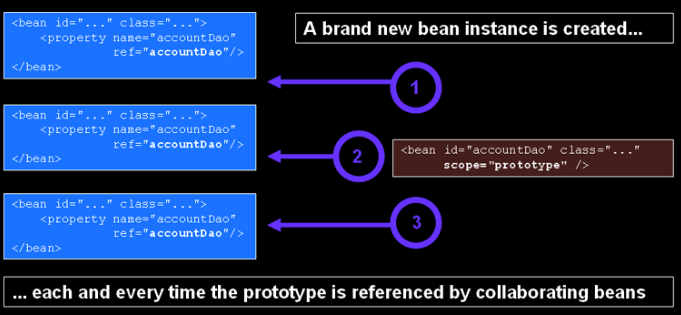
### 1.5.1 Singleton Scope

Когда определяется singleton бин, контейнер спринга создает только один экземпляр объекта. Он записывается в кэш таких синглтон бинов, а все запросы и ссылки на этот бин возвращает кэшированный объект.



### 1.5.2 Prototype Scope

При таком скопуе бина создается новый экземпляр каждый раз, когда он запрашивается. Таким образом, новые экземпляры будет заинжекчены во все бины, которые требуют prototype бин. Также при методе getBean() будет создаваться новый экземпляр объекта. Как правило, такой скоуп необходимо использовать для всех бинов, у которых есть внутреннее состояние, а singleton скоуп — для тех бинов, у которых состояния отсутствует.



В отличие от других бинов, спринг не управляет полным жизненным циклом бинов со скоупом prototype. Контейнер создает, конфигурирует и иным образом компонует объект-прототип и передает его клиенту без дальнейших записей об этом экземпляре прототипа. Таким образом, хотя методы обратного вызова жизненного цикла инциализации вызываются для всех объектов независимо от области видимости, в случае прототипов, сконфигурированные обратные вызовы жизненного цикла разрушения не вызываются. Клиентский код должен подчищать объекты, входящие в область видимости prototype, и высвободить ценные ресурсы, которые потребляют бины-прототипы. Чтобы заставить контейнер высвободить ресурсы, потребляемыми бинами-прототипами, попробуйте использовать специальный постпроцессор бинов, содержащий ссылку на бины, которые необходимо подчистить.

### 1.5.3 Request, Session, Application, WebSocket scopes.

Данный типы областей видимости доступны только при использовании реализации web-ориентированного ApplicationContext (например XmlWebApplicationContext). Если использовать их с обычным Spring IoC контейнером, таком как ClassPathXmlApplicationContext, то будет выброшено IllegalStateException, которое скажет о том, что не знает такие скоупы.

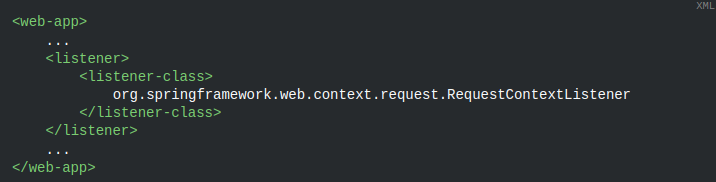
#### 1.5.3.1 Начальная веб конфигурация

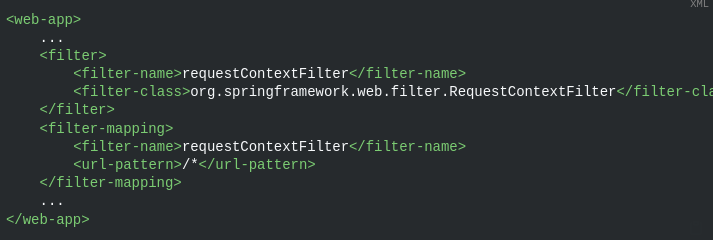
Чтобы использовать эти скоупы необходимо сделать несколько минорных начальных конфигураций.

То, каким образом вы выполните эту начальную настройку, зависит от вашей конкретной среды сервлетов.

Если вы обращаетесь к бинам, находящимся в области видимости, в Spring Web MVC, то есть в рамках запроса, который обрабатывается Spring DispatcherServlet, никакой специальной настройки не требуется. DispatcherServlet уже раскрывает все соответствующие состояния.

Если вы используете веб-контейнер Servlet 2.5, в котором запросы обрабатываются вне Spring DispatcherServlet (например, при использовании JSF или Struts), вам необходимо зарегистрировать org.springframework.web.context.request.RequestContextListener ServletRequestListener. Для Servlet 3.0+ это можно сделать программно, используя интерфейс WebApplicationInitializer. В качестве альтернативы или для более старых версий контейнеров добавьте следующее объявление в файл web web.xml вашего веб-приложения:

 Существует альтернатива, если есть проблемы для установки этого листенера: RequestContextFilter.



DispatcherServlet, RequestContextListener и RequestContextFilter делают одно и то же, а именно связывают объект HTTP-запроса с Thread, который обслуживает этот запрос. Это позволяет получить доступ к бинам, входящим в область видимости request или session, далее по цепочке вызовов.

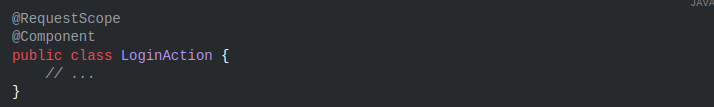
#### 1.5.3.2 Request Scope

Пускай будет объявлен следующий bean definition:



Сприн контейнер создает новый экземпляр бина LoginAction используя bean definition loginAction для каждого и всех HTTP запросов. Можно изменить внутреннее состояние для экземпляра который был создан так как угодно, потому что остальные экземпляры, созданные из этого же loginAction bean definition не увидят изменений в состояний. Они являются индивидуальными для каждого индивидуального запроса. Когда процесс запроса пройдет, бин, ограниченный запросом отбрасывается.

Можно использовать Java аннотации для этого:



#### 1.5.3.3 Session scope

Xml bean definition:



Такой бин создается каждый раз при новой http сессии. Можно менять внутреннее состояние как угодно, т. к. сессионые бины созданные из этого же bean definition не увидят изменения в состоянии других сессий.

Когда оканчивается HTTP сессия, то и сессионые бины отбрасываются.

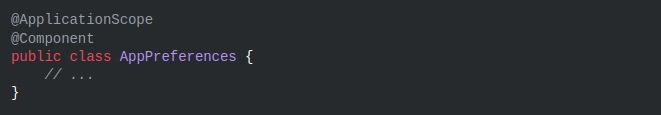
Java аннотации:

#### 1.5.3.4 Application Scope

Xml bean definiton:

 Контейнер Spring создает новый экземпляр бина AppPreferences, используя определение бина appPreferences единожды для всего веб-приложения. То есть бин appPreferences находится в области видимости на уровне ServletContext и хранится как обычный атрибут ServletContext. Он в некоторой степени похож на бин-одиночку Spring, но отличается от него двумя важными особенностями: Он является объектом-одиночкой для каждого ServletContext, а не для Spring ApplicationContext (которых может быть несколько в любом конкретном веб-приложении), и он фактически открывается, поэтому и виден как атрибут ServletContext.

Java аннотации:



#### 1.5.3.5 WebSocket Scope

Область видимости WebSocket связана с жизненным циклом сессии WebSocket и применяется к приложениям STOMP over WebSocket.

### 1.5.4 Scoped Beans as Dependencies

IoC-контейнер Spring управляет не только созданием экземпляров ваших объектов (бинов), но и связыванием взаимодействующих объектов (или зависимостей). Если вам нужно внедрить (например) бин, находящийся в зоне видимости HTTP-request в другой бин с более долговременной областью видимости, вы можете внедрить АОП-прокси вместо бина, находящегося в области видимости. То есть вам нужно внедрить прокси-объект, который раскрывает тот же публичный интерфейс, что и объект, находящийся в области видимости, но который также может извлекать реальный целевой объект из соответствующей области видимости (например, HTTP-request) и делегировать вызовы методов на реальный объект.

Вы также можете использовать <aop:scoped-proxy/> между бинами, которые определены как singleton, при этом ссылка проходит через промежуточный прокси, который является сериализуемым и поэтому способен повторно получить целевой бин-одиночку при десериализации.

При объявлении <aop:scoped-proxy/> для бина области видимсоти prototype , каждый вызов метода на общем прокси приводит к созданию нового целевого экземпляра, которому затем передается вызов.

Кроме того, прокси, находящиеся в области видимости, - не единственный способ получить доступ к бинам из более коротких областей видимости безопасным для жизненного цикла способом. Также можно объявить точку внедрения (то есть аргумент конструктора или сеттера или автоматически связанное поле) как ObjectFactory<MyTargetBean>, что позволит вызову getObject() получать текущий экземпляр по требованию каждый раз, когда это необходимо - без необходимости удерживать экземпляр или хранить его отдельно.

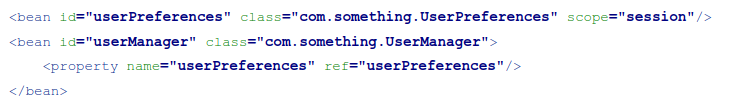
В качестве расширенного способа можно объявить ObjectProvider<MyTargetBean>, который обеспечивает несколько дополнительных вариантов доступа, включая getIfAvailable и getIfUnique.

Вариант по стандарту JSR-330 называется Provider и используется с объявлением Provider<MyTargetBean> и соответствующим вызовом get() для каждой попытки получения.

Конфигурация в следующем примере состоит всего из одной строки, но важно понимать "почему" и "как" так получается:



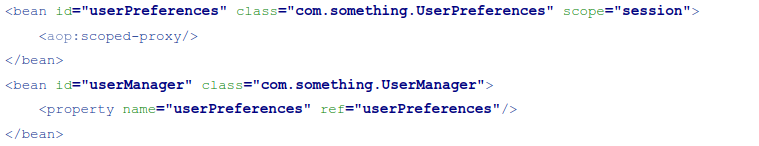
Чтобы создать такой прокси, вам нужно вставить дочерний элемент <aop:scoped-proxy/> в определение бина, находящегося в области видимости. Почему определения бинов, находящихся в области видимости на уровнях request, session или в специальной области видимости, требуют элемент <aop:scoped-proxy/>? Рассмотрите следующее определение бина-одиночки и сравните его с тем, что вам нужно определить для вышеупомянутых областей видимости (обратите внимание, что следующее определение бина userPreferences в его нынешнем виде является неполным):



В предыдущем примере бин-одиночка (userManager) внедрен c помощью ссылки на бин (userPreferences), находящийся в зоне видимости HTTP Session. Важным моментом здесь является то, что бин userManager является объектом-одиночкой: его экземпляр создается ровно один раз для каждого контейнера, а его зависимости (в данном случае только одна, бин userPreferences) также внедряются только один раз. Это означает, что бин userManager работает только с тем же самым объектом userPreferences (то есть с тем, с которым он был первоначально внедрен).

Это не та форма поведения, которая подойдет при внедрении бина с более коротким жизненным циклом, находящегося в области видимости, в бин с более длинным жизненным циклом, находящийся в области видимости (например, внедрение взаимодействующего бина, находящегося в области видимости HTTP Session, в качестве зависимости в бин-одиночку). Скорее, требуется один объект userManager, а на время существования HTTP Session - объект userPreferences, специфичный для данной HTTP Session. Таким образом, контейнер создает объект, который открывает точно такой же публичный интерфейс, как и класс UserPreferences (в идеале объект, который является экземпляром UserPreferences), который может получить реальный объект UserPreferences из механизма определения (HTTP запрос, Session и так далее). Контейнер внедряет этот прокси-объект в бин userManager, который не знает, что эта ссылка UserPreferences является прокси. В этом примере, когда экземпляр UserManager инициирует метод на объекте UserPreferences с внедренной зависимостью он фактически вызывает метод на прокси. Затем прокси выполняет выборку реального объекта UserPreferences из (в данном случае) HTTP Session и делегирует вызов метода на полученный реальный объект UserPreferences.

Таким образом, необходима следующая (правильная и полная) конфигурация при внедрении бинов, находящихся в области видимости request и session, в взаимодействующие объекты, как показано в следующем примере:



## 1.6 Customizing the Nature of a Bean

### 1.6.1 Lifecycle Callbacks

Чтобы взаимодействовать с управлением жизненного цикла бинов контейнера, необходимо имплементировать Spring InitializingBean и DisposableBean интерфейсы. Контейнер вызывает afterPropertiesSet() для фомирования и destroy() чтобы позволить компоненту выполнить определенные действия при инициализации и уничтожении ваших компонентов.

JSR-250 @PostConstruct и @PreDestroy аннотации считаются лучшими в использовании для получения обратных вызовов жизненного цикла в современном Spring приложении. Использование этих аннотаций значит, что ваши бины не зависят от Spring специфичных интерфейсов.

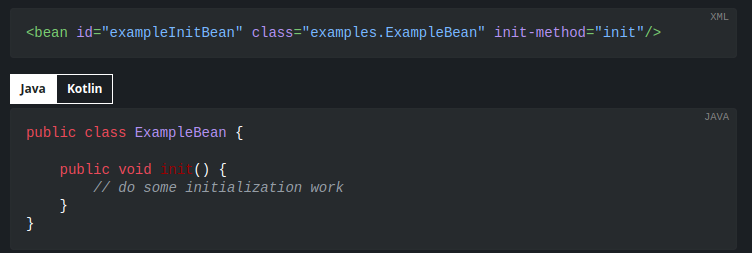
Внутренне, Spring использует BeanPostProcessor реализации чтобы обрабатывать любой колбэк интерфейсов, который он может найти и вызывать соответствующие методы. Если необходимо кастомизировать фичи или другое поведение жизненного цикла, которые спринг не предлагает по умолчанию, можно использовать собственную реализацию BeanPostProcessor.

#### 1.6.1.1 Initialization Callbacks

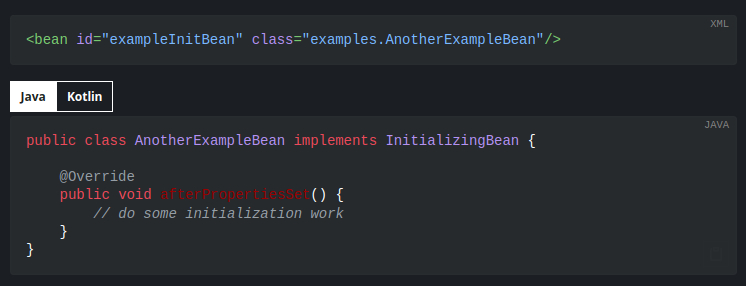
InitializingBean интерфейся дает бину сделать работу по инициализации после того, как контейнер установит все необходимые свойства бина. Этот интерфейс имеет единственный метод:



Рекомендуется использовать @PostConstruct аннотацию вместо этого интерфейса, чтобы избежать имплементации от Spring специфичных интерфейсов. Так же можно применять POJO метод инициализации. В случае XML конфигурационных метаданных, можно использовать атрибут init-methid для определения имени метода, который не должен содержать какие либо аргументы и должен возвращать void:



Тоже самое с использованием интерфейсов:



#### 1.6.1.2 Destruction callbacks

Реализация Disposable Bean позволяет бину выполнить метод, который будет выполняться тогда, когда контейнер, содержащий бин, будет уничтожен.



Отдельно, как и в случае с init-method можно в xml bean definition прописать методо POJO, который будет вызываться при уничтожении контейнера:



#### 1.6.1.3 Combining Lifecycle Mecganisms

С версии Spring 2.5 существуют 3 способа задать колбэки жизненного цикла:

1. InitializingBean and DisposableBean

2. Custom init and destroy methods.

3. @PostConstruct and @PreDestroy annotations.

Если же использовать все вместе, то у них есть определенный порядок выполнения:

1. Методы с аннотацией @PostConstruct

2. Метод afterPropertiesSet() из InitializingBean.

3. Кастомный init method.

При уничтожении контейнера порядок такой:

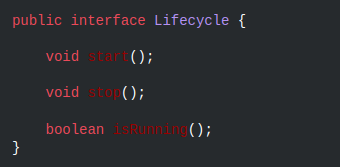
1. Методы с аннотацией @PreDestroy.

2. destroy() метод из DisposableBean интерфейса.

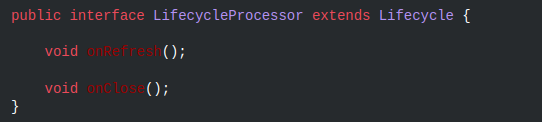
3. Кастомный destroy() метод.

### 1.6.2 Startup and Shutdown Callbacks

Существует интерфейс Lifecycle, который содержит основные методы для любого объекта, который будет имплементировать его.



Любой управляемый Springом объект может имплементировать этот интерфейс. Тогда, когда ApplicationContext получиит сигнал старта или сигнал остановки, но он вызовет те имплементированные методы интерфейса Lifecycle. Он это сделает путем делегирования на LifecycleProcessor интерфейс:



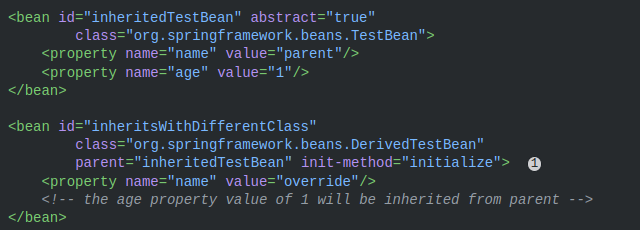
Заметьте, что привычный интерфейс org.springframework.context.Lifecycle является обычным контрактом для явных уведомлений о запуске и остановке и не подразумевает автоматического запуска при обновлении контекста. Для более тонкого контроля над автоматическим запуском конкретного бина (включая фазы запуска), рассмотрите возможность реализации org.springframework.context.SmartLifecycle.

Также обратите внимание, что уведомления об остановке не гарантированно приходят до уничтожения. При обычном завершении работы все бины Lifecycle сначала получают уведомление об остановке, прежде чем распространятся общие обратные вызовы уничтожения. Однако в случае горячего обновления (hot refresh) в течение времени существования контекста или при остановленных попытках обновления вызываются только методы уничтожения.

## 1.7 Bean Definition Inheritance

Bean definition может содержать большой набор данных. Дочерние bean definition наследуют конфигурационные метаданные из родительского. Дочерний bean definition может переопределить некоторые значения или добавить другие какие требуются. Используя родительские и дочерние bean definition может сильно сэкономить времени на печатение текста.

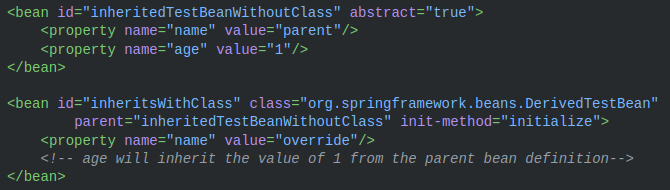
В xml bean definition можно использовать parent атрибут, чтобы указать родительский бин как значение этого атрибута.



Таким образом, в данном случае, приведенном выше, через Setter DI в экземпляр класса DerivedTestBean будут установлены зависимости age = 1. При этом name будет переопределен. Дочерний класс компонента должен быть совместим с родительским (то есть он должен принимать значения свойств родительского компонента).

Остальные настройки всегда берутся из дочернего определения: depends-on, autowiring, проверка зависимостей, синглтон и отложенная инициализация.

Для наглядности родительский бин можно пометить словом abstract. Это озночает, что на этот бин нельзя будет ссылаться, он не может быть получен при вызове getBean() метода. В таких случаях будет выброшена ошибка. Abstract bean definition будет в итоге выступать в качестве шаблона, который будут использовать дочернии бины при наследовании bean definition. + этот бин не будет создан как синглтон, он будет проигнорирован контейнером.



## 1.8 Container Extension Points

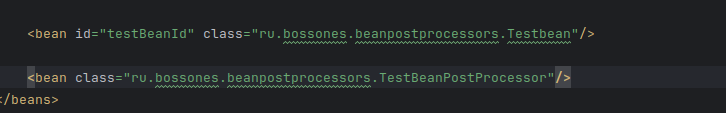
### 1.8.1 BeanPostProcessor

BeanPostProcessor интерфейс определяет методы обратного вызова, которые необходимо имплементировать, чтобы обеспечить собственную логику создания экзмепляра класса, логику рпзрешения зависимостей и т. д. Если необходимо реализовать какую-нибудь кастомную логику после того как Spring Container завершил создание экземпляров, конфигурирование и инициализирование бинов, можно подключить один или более BeanPostProcessor реализаций.

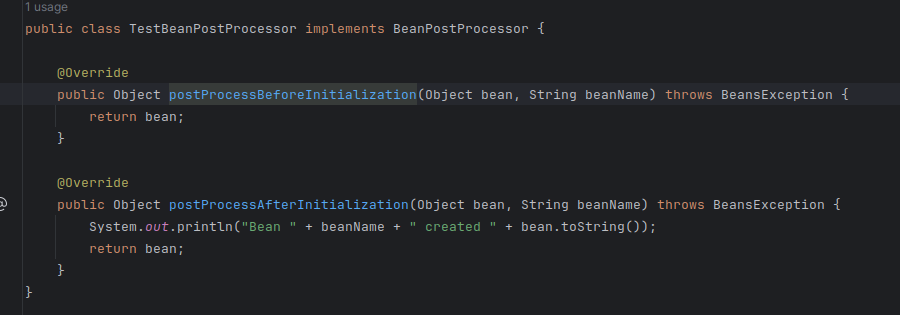
Так же можно контроилровать порядок этих реализаций, устанавливая order свойство. Но для этого необходимо ипмплементировать собственный Ordered интерфейс.

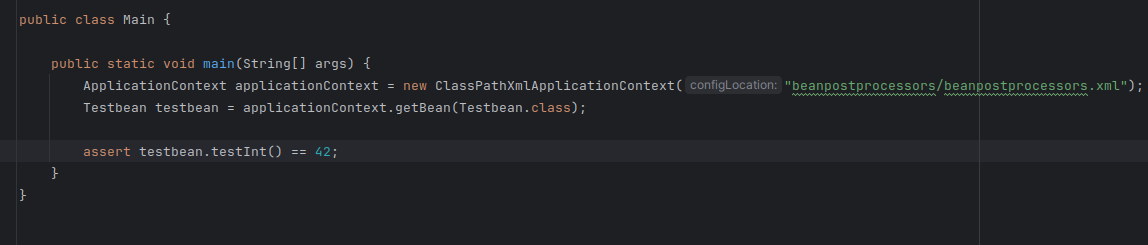
Этот интерфейс состоит из двух методов обратного вызова. После того, как каждый бин инстанс был создан контейнером, пост процессор получает вызовы от контейнера до того, как будут выполнены методы инициализации, такие как afterPropertiesSet или init мметодов. Пост процессор может выполнить любые действия с экзмепляром бина, включая полное игнорирование коллбэка. Обычно пост процессор проверяет коллбэки или может обернуть бин в прокси.

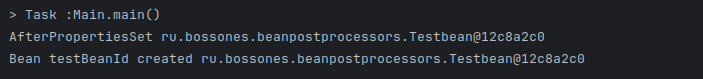
Пример:











### 1.8.2 BeanFactoryPostProcessor

У данного типа постпроцессора схожая семантика с BeanPostProcessor, с одним главным отличием: данный построцессор манипулирует с конифгурационными метаданными бина. Таким образом, спринг контейнер дает BeanFactoryPostProcessor экземплярам прочитать конфигурационные метаданные и потенциально изменить их до того, как контейнер создаст экземпляры каких либо бинов, кроме BeanFactoryPostProcessor бинов.

По аналогии с BeanPostProcessor можно устанавливать порядок с которым эти постпроцессоры будут запускаться , но необходимо имплементировать Ordered интерфейс.

### 1.8.3 FactoryBean

Можно имплементировать данный интерфейс для того, чтобы расширить логику создания экземпляра в коде. Если есть сложный код для инициализации, то лучше его описать в Java, что потенциально может уменьшить количество XML. Можно создать собственный FactoryBean, написать сложный инициализационный код внутри класса и передать его в контейнер.

У FactoryBean<T> есть три метода:

getObject() - возвращается инстанс объекта, который создается этой фабрикой.

IsSingleton() - возвращает true или false. Default — true.

Class<?> getObjectType() - возвращает тип объекта или null, если типр не известен.

FactoryBean используется в большом количестве Spring фреймворка. Более 50 имплементации использовано в спринге.

## 1.9 Annotation-based Contatiner Configuration

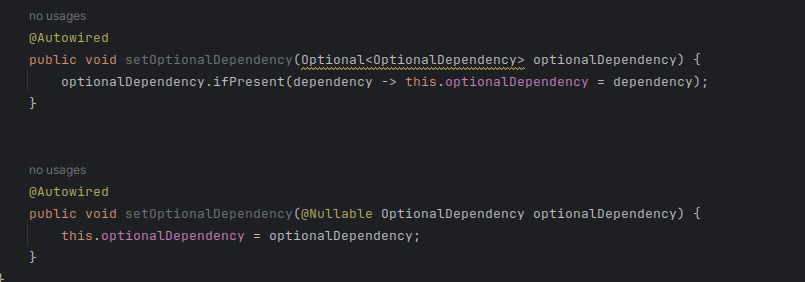
### 1.9.1 @Autowired

* Такой тип аннотации можно применять на:
* Конструкторах (не обязательно, если один конструктор)
* Сеттерах
* Кастомном методе
* На полях бина, миксуя с конструкторами
* Над полями, где объявлены массивы классов

У этой аннотации имеется атрибут required, который можно задать как false. В этом случае, если не найдено ни одного бина для того, чтобы его заинжектить, в соответствующее поле класса или аргумент метода или конструктора будет проставлен null. Если же @Autowired(required=false) расположен над методом и не найдено никаких бинов для того, чтобы передать его в этот метод, то метод не вызывается вообще.

Только один конструктор любого данного класса bean-компонента может объявить @Autowired с обязательным атрибутом, установленным в true, что указывает на то, что конструктор выполняет автоматическое связывание при использовании в качестве bean-компонента Spring. Как следствие, если для обязательного атрибута оставить значение по умолчанию true, только один конструктор может быть помечен @Autowired. Если аннотацию объявляют несколько конструкторов, все они должны будут объявить require=false, чтобы их можно было рассматривать как кандидатов на автоматическое связывание (аналог autowire=constructor в XML). Будет выбран конструктор с наибольшим количеством зависимостей, которые могут быть удовлетворены путем сопоставления bean-компонентов в контейнере Spring. Если ни один из кандидатов не может быть удовлетворен, то будет использоваться основной конструктор/конструктор по умолчанию (если он присутствует). Аналогично, если класс объявляет несколько конструкторов, но ни один из них не имеет аннотации @Autowired, то будет использоваться основной конструктор/конструктор по умолчанию (если он присутствует). Если класс вначале объявляет только один конструктор, он всегда будет использоваться.даже если без аннотаций. Обратите внимание, что аннотированный конструктор не обязательно должен быть общедоступным.

Для необязательных бинов, кторые должны быть автосвязаны, можно использовать Optional класс, либо использовать аннотацию @Nullable от spring’а:



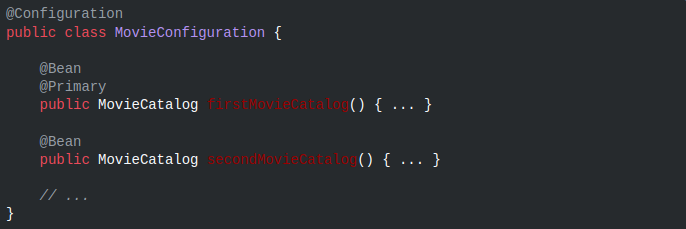
Можно использовать @Autowired для уже известных интерфейсов Spring API:

BeanFactory, ApplicationContext, Environment, ResourceLoader, ApplicationEventPublisher, and MessageSource

### 1.9.2 @Primary

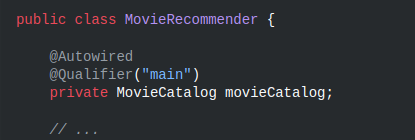
Т.к. автосвязывание может ссылаться на множество кандидатов, часто необходимо иметь больше контроля над процессом выбора кандидатов для автосвязывания. Один из способов — это использование @Primary аннотации.

Эта аннотация указывается над определенным бинов, который должен иметь приоритет над остальными бинами данного класса.



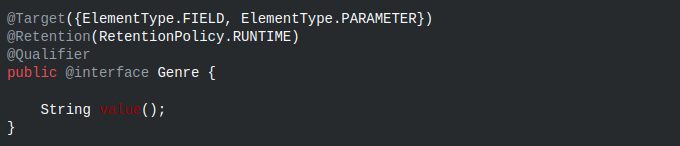
### 1.9.4 @Qualifier

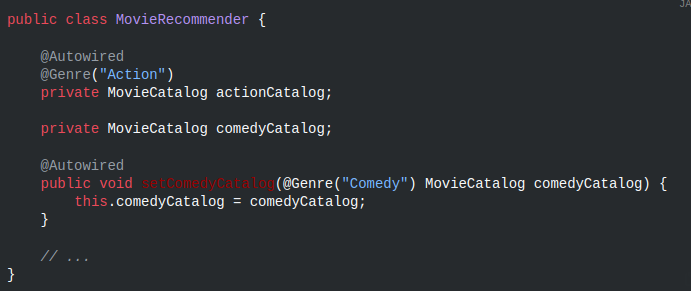
@Primary — это эффективный вид для автосвязывания типов с несколькими экземплярами, где один кандидат является более предпочтительным. Когда нужно больше контроля над процессов выбора, то можно использовать @Qualifier аннотацию. Благодаря этому можно выбрать тот экземпляр, который нужен по логику данному бину.



Данную аннотацию можно применить к полям, Setter DU, много-аргументным методам.

При этом можно делать свои кастомные Qualifier аннотации:





Таким образом, кастомные аннотации позволяют немного конкретизировать область их применения. Как на примере, в каталог фильмо инжектится только те зависимости, которые имеют «Жанр комедия» или «Жанр экшн».

### 1.9.5 Injection with @Resource

Используя JSR-250 было добавлено использование @Resource аннотации из Jakarta EE для того, чтобы использовать ее как просмотр кандидатов на автосвязывание. Она может применяться над полями или над сеттер методами.

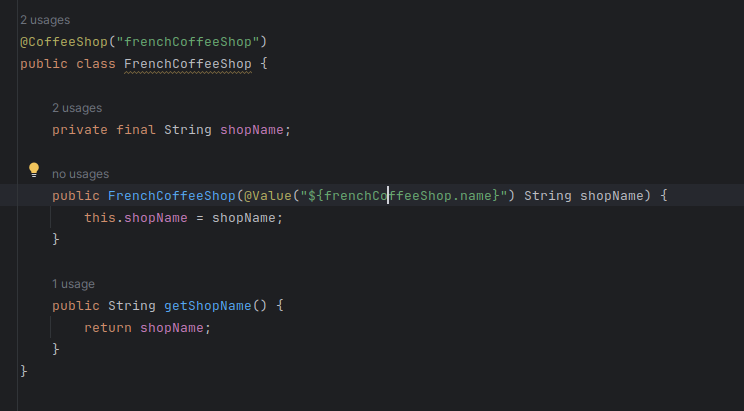
Это аннотация принимает атрибут «name», Spring воспринимает этот атрибут по умолчанию как имя бина, который необходимо заинжектить.

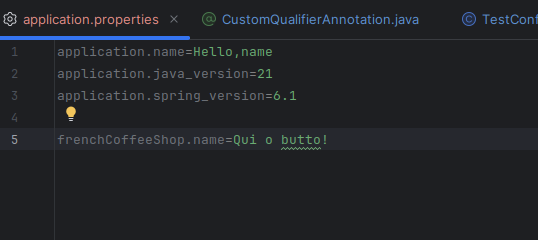
Если не определен никакой атрибут, то инжекция будет происходить по имени бина класс. Т.е. если класс называется TestBean, то сначала будет произведен поиск бина класса TestBean с именем бина «testBean», если такой не найден, то будет использован наиболее подходящий бин типа TestBean.

### 1.9.6 @Value

Данная аннотация обычно используется для инжектирования внешних проперти.

К примеру:

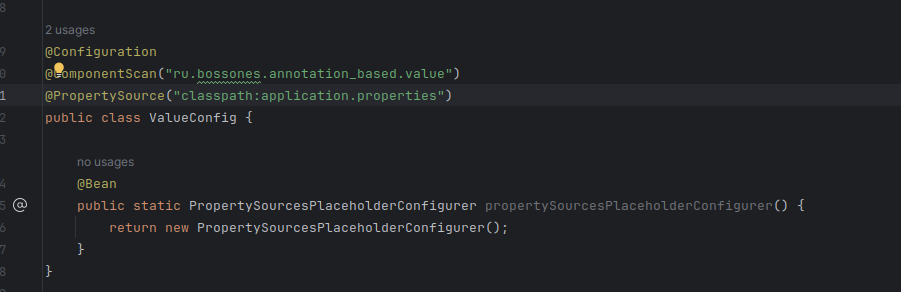




Для того, чтобы файл с пропертями был видел контексту спринга, необходимо явно обозначить его в конфигурационном файле:

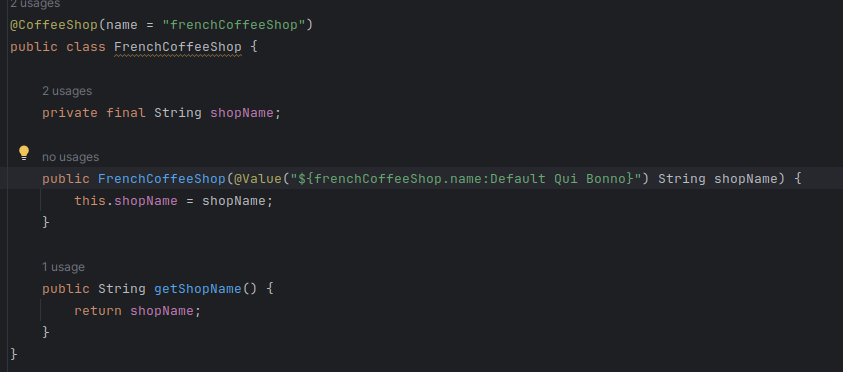
 По умолчанию используется мягкий преобразователь значений проперти в значение, и если нет какого то проперти, то он заинжектит в значение именно запись «${property,name}»

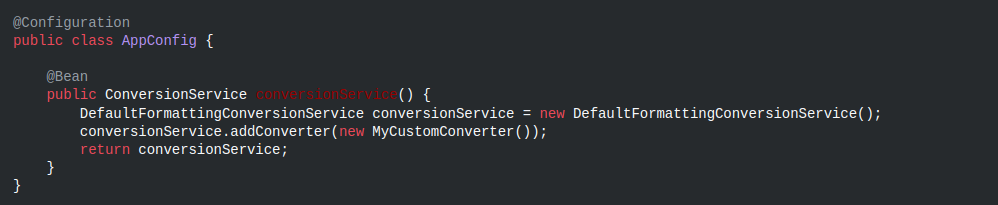
Если необходим более явный контроль над несуществующими свойствами, необходимо использовать конфигуратор ProperySourcesPlaceholderConfiguirer бин, как показано ниже. При этом, когда используется подход с Java аннотациями, то метод бина должен быть помечен как static:



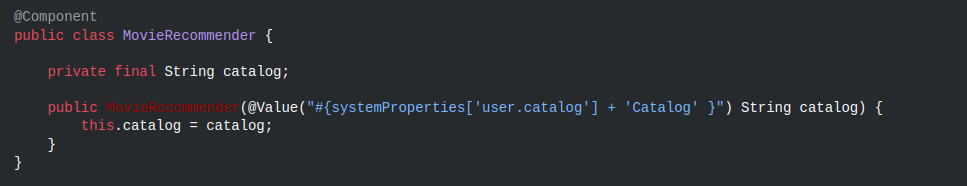
В таком случае, если нет проперти по указанному пути, Spring выбросит исключение и приложение не будет запущено.

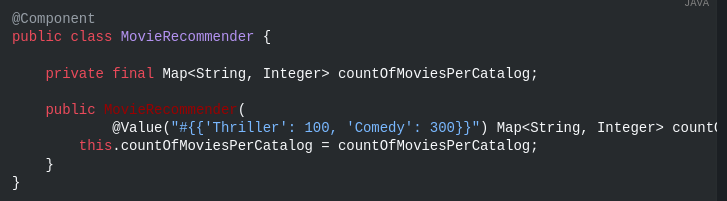
При этом, можно передавать дефолтные значения, которые будут применены, если в проперти файлах отсутствует значение.

 Внутри спринга через BeanPostProcessor происходит использование сервиса Conversionservice, который позволяет конвертировать значения в пропертях в необходимый тип. Если необходимо создать свою кастомную поддержку конверсии для собственного типа, то можно обеспечить собственный ConversionService бин инстанс примерно так:



Внутри @Value можно применять SpEL выражения для динамического вычисления, которое будет вычеслено в рантайме:

 SpEL позволяет использовать даже более сложные типы данных.

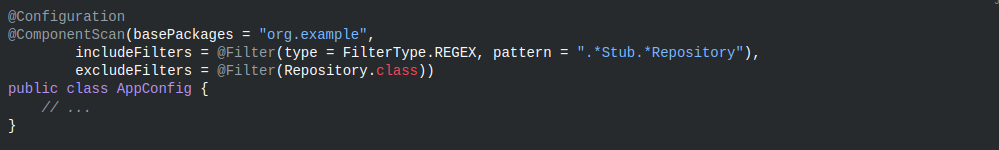


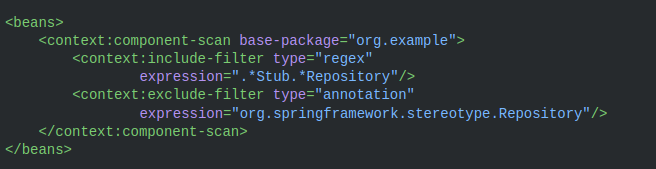
## 1.10 Classpath Scanning and Managed Components

Для автосвязывания можно расширить поведения, используя фильтры на аннотации @ComponentScan() для того, чтобы заинжектить необходимые бины в определенном пакете или пакетах. Необходимо в аннотацию добавить атрибут includeFilters или excludeFilters. Каждый фильтр требует 2 атрибута для поиска: type и expression. Существуют несколько типов фильтров:

|  |  |  |
| --- | --- | --- |
| annotation (by default) | org.example.SomeAnnotation | Должна быть представлена аннотация или мета-аннотация над типом целевых компонентов. |
| assignable | org.exapmle.SomeClass | Класс или интерфейс для поиска |
| aspectj | org.example..\*Service+ | Стиль выражения AspectJ для того, чтобы сматчить необходимые компоненты |
| regex | org\.example\.Default.\* | Используется Regex для поиска целевых компонентов по имени класса. |
| custom | org.example.MyTypeFilter | Кастомная реализация TypeFilter интерфейса. |

К примеру, вот запрос на поиск зависимостей, которые не будут аннотированы аннотацией @Repository и вместо этого, у них будет наименование Stub Repository.





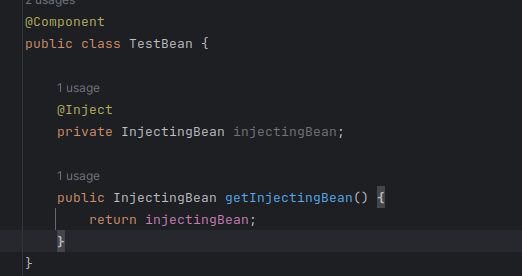
Методы, которые помечены @Bean аннотациями внутри компонента Spring (@Component) работают немного по-другому, чем те же методы с аннотацией @Bean в конфигурации (@Configuration). Разница состоит в том, что классы компонентов не дополнены CGLIB для перехвата вызова методов и полей.

Можно также объявить методы @Bean и сделать их статическими, позволяя создать инстансы без создания содержащего их класс конфигурации.

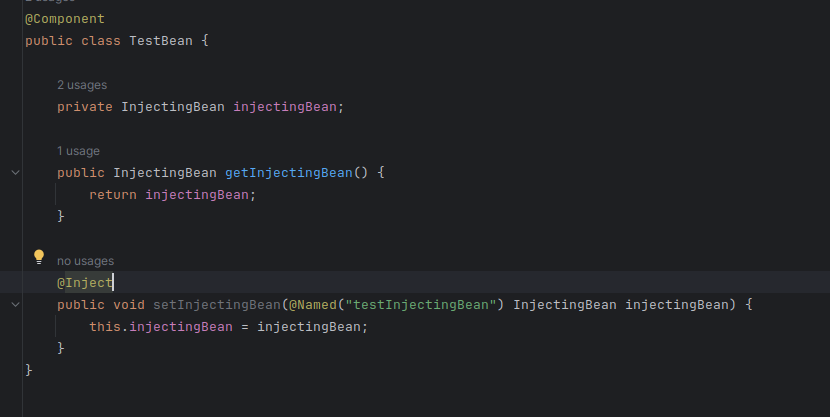
## 1.11 Using JSR 330 Standard Annotations

Спринг позволяет использовать стандартные аннотации для JSR-330. Они сканируются таким же образом как и спринг аннотации.

Для автосвязывания можно использовать аннотацию @Inject, которая так же как и @Autowired применяется на полях, методах и используются в Provider, чтобы получить бины с более ограниченной областью видимости.



Для того, чтобы получить необходимый бин по имени как @Qualifier, можно использовать аннотацию @Named:



Так же как и @Autowired, можно использовать инжектирование вместе с аннотациями @Nullable или Optional:

 Вместо использования аннотации @Component, можно использовать аннотацию как @Named или @ManagedBean:



Но в отличии от @Component аннотация @Named и @ManagedBean не являются составными. Поэтому, для своих кастомных аннотаций необходимо использовать спринговские аннотации.

### 1.11.1 Limitations of JSR-330 Standard Annotations

|  |  |  |
| --- | --- | --- |
| @Autowired | @Inject | @Inject не имеет required атрибута. Вместо этого может быть использован вместе с Optional. |
| @Component | @Named/@ManagedBean | JSR-330 не использует составную модель. Можно использовать только в одном виде для наименования компонента. ManagedBean является deprecated и будет удален. |
| @Scope(«singleton») | @Singleton | Область действия по умолчанию JSR-330 похожа на прототип Spring. Однако, чтобы обеспечить соответствие общим настройкам Spring по умолчанию, компонент JSR-330, объявленный в контейнере Spring, по умолчанию является одноэлементным. Чтобы использовать область действия, отличную от Singleton, вам следует использовать аннотацию Spring @Scope. jakarta.inject также предоставляет аннотацию jakarta.inject.Scope: однако она предназначена только для создания пользовательских аннотаций. |
| @Qualifier | @Qualifier/@Named | У jakarta.inject.Qualifier используется только как мета аннотация для создания кастомных квалификаторов. Вместо этого можно использовать @Named |
| @Value | - | Нет такого |
| @Lazy | - | Нет такого |
| ObjectProvider | Provider | Jakarta.inject.Provider является прямой альтернативой для ObjectFactory, только с более коротким get() методом. Так же может применяться с комбинацией спринговой аннотации @Autowired, либо без аннотаций с конструктором и сеттер методами. |

## 1.12 Java-based Container Configuration

Когда методы @Bean объявляются в классах, не аннотированных @Configuration, их называют обрабатываемыми в "облегченном" режиме. Методы бинов, объявленные в аннотации @Component или даже в обычном классе, считаются "облегченными", с другой основной целью объемлющего класса, а метод, помеченный аннотацией @Bean, является там своего рода бонусом. Например, служебные компоненты могут предоставлять контейнеру представления управления через дополнительный метод, аннотированный @Bean, для каждого класса соответствующего компонента. В таких сценариях методы, аннотированные @Bean, являются механизмом фабричных методов общего назначения.

В отличие от полной @Configuration, облеченные методы @Bean не могут объявлять межбиновые зависимости. Вместо этого они работают с внутренним состоянием содержащего их компонента и, по необходимости, с аргументами, которые они могут объявить. Поэтому такой метод, помеченный аннотацией @Bean, не должен вызывать другие методы, помеченные аннотацией @Bean. Каждый такой метод в буквальном смысле является лишь фабричным методом для конкретной ссылки на бин, без какой-либо специальной семантики во время выполнения. Положительным побочным эффектом здесь является то, что отсутствует необходимость применять подклассификацию CGLIB во время выполнения, поэтому нет никаких ограничений в плане проектирования классов (таким образом, объемлющий класс может быть final и так далее).

В обычных сценариях методы, помеченные аннотацией @Bean, должны быть объявлены в классах, помеченных аннотацией @Configuration, что обеспечит постоянное использование "полного" режима и перенаправление перекрестных ссылок на управление жизненным циклом контейнера. Это предотвращает случайный вызов одного и того же метода, помеченного аннотацией @Bean, через обычный вызов Java, что помогает уменьшить количество трудно находимых ошибок, которые трудно отследить при работе в "облегченном" режиме

## 1.13 Environment Abstraction

Профиль это именованная, логическая группа определений бинов, которая регистрируется контейнером, только когда профиль активен. Роль объекта Environment по отношению к профилям заключается в определении того, какие профили активны в данный момент, а какие должны быть активны по умолчанию.

Проперти играют важную роль практически по всех приложениях и могут быть расположены в различных источниках: файлы, JVM системные свойства, переменные окружения, JNDI, параметры контекста сервлета, ad-hoс свойства объектов, мапы, и т. д.

### 1.13.1 Bean definition Profiles

Профили это механизм в ядре контейнера, который позволяет регистрировать различные бины в различных средах. Само слово «среда» может означать различные вещи для разных пользователях, и эта особенность может помочь в различных кейсах:

* Работа с in-memory источниками данными в разработке вместо поиска того же источника данных из JNDI при тестировании или на проде.
* Регистрация система мониторинга только при разворачивании приложения в среде производительности
* Регистрация кастомных реализаций бина для потребителей.

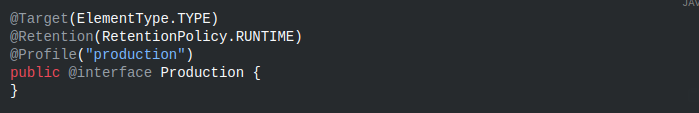
Аннотация @Profile позволяет обозначить, что компонент может быть доступен для регистрации когда один или несколько разных профилей активны. Используя их можно, например, сделать различные бины для источника данных при разработке и при разворачивании на прод среду.

 Профили могут содержать простые наименование профилей, например «production» или выражение из профилей. Выражение позволяет выполнить более сложную логику для профилей, например, «producation & us-east». Следующие операторы поддерживаются в проифльных выражениях:

* !: A logical NOT of the profile
* &: A logical AND of the profiles
* |: A logical OR of the profiles

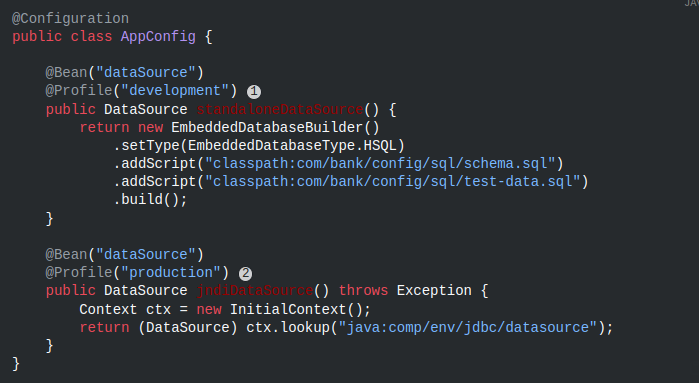
Нельзя миксовать логичское И и логическое ИЛИ без скобок (producaiton & us-east | eu-central). Должно быть так: «production & (us-east | eu-central)»

@Profile можно использовать как мета-аннотацию для целей создания своих кастомных аннотаций.

 Если класс @Configuration помечен определенным профилем, то все методы определения бины и @Import аннотации, ассоциированные с данном классом, пропускаются, если один из указанных профилей не является активным.

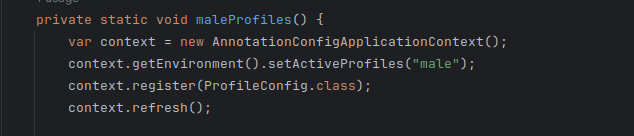
Если класс аннотирован как @Profile({«p1», «p2»}), то этот класс не будет зарегестрирован, пока не активен какой нибудь из профилей p1 или p2.

Так же аннотация @Profile может быть проставлена над методами бина:

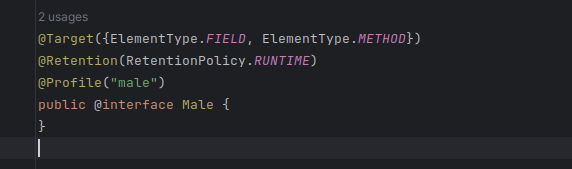


### 1.13.2 Activating Profiles

Активацию профиля можно сделать несколькими способами, один из них это явно сказать контексту, что необходимо установить активный профиль:

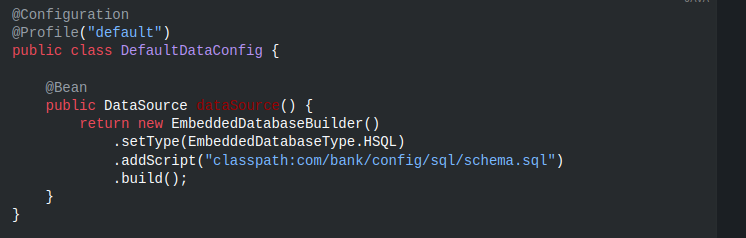






### 1.13.3 Default profile

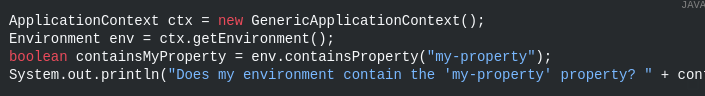
Профиль по умолчанию представляет собой профиль, который включается, когда не активны какие либо другие профилли.



Наименование дефолтного профиля можно изменить. Используя setDeafultProfiles() на Environment объекте. Или декларативно, использую свойство spring.profiles.default

### 1.13.4 PropertySource Abstraction

Класс Environment обеспечивает операции поиска над настраиваемой иерархией источников свойств.

 Поиск пропертей является иерархичным. По умолчанию, системные проперти имеют приоритет над переменными окружения. Так, если my-property установлен в обоих местах и если постараться вызвать это свойство, то системное проперти выйграет и будет возвращено. Значение проперти не мержутся но полностью перезатираются в порядке приоритета.

В случае общего StandardServlerEnvironment, полная иерархия представлена порядком:

1. ServletConfig параметры.
2. ServletContext параметры (web.xml)
3. JNDI переменные (java:comp/env/)
4. JVM проперти (-D аргументы в строке)
5. JVM системные проперти (переменные окружения ОС)

### 1.13.5 Using @PropertySource

Аннотация @PropertySource обеспечивает удобный и декларативный механизм для добавления источника пропертей в спринг.



Внутри @PropertySource можно использовать так же ${} плейсхолдеры, если уже в спринге определены другие источники пропертей.



## 1.14 Additional Capabilities of ApplicationContext

### 1.14.1 Standard and Custom Events

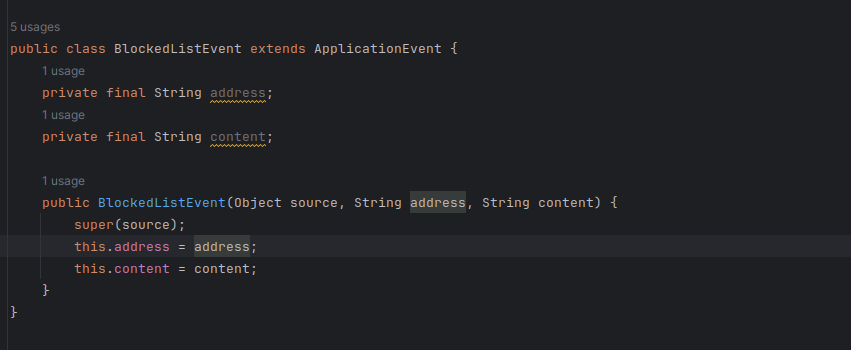
Перехватывание ивентов в ApplicationContext обеспечивается через класс ApplicationEvent и ApplicationListener интерфейс. Если бин имплементирует ApplicationListener интерфейс он деплоится в контекст, каждый раз когда ApplicationEvent публикуется в контекст, этот компонент уведомляется.

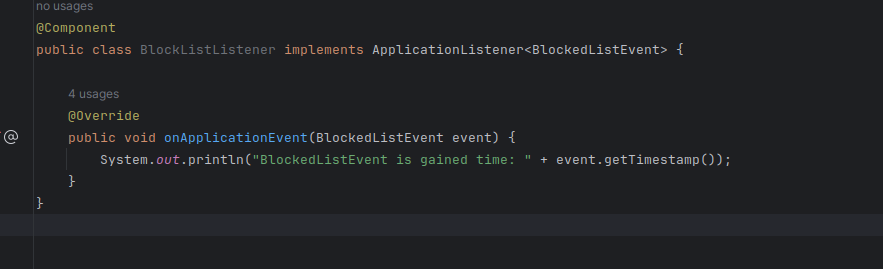
В Spring 4.2 инфраструктура ивентов сильно улучшена и предлагает модель аннотаций как возможность публикации ивентов.

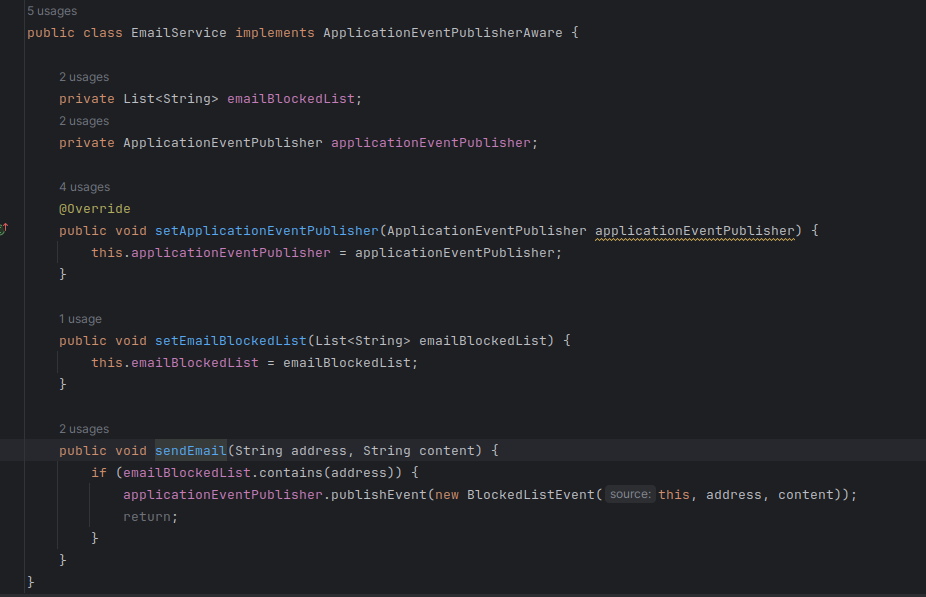
Типы стандартных ивентов:

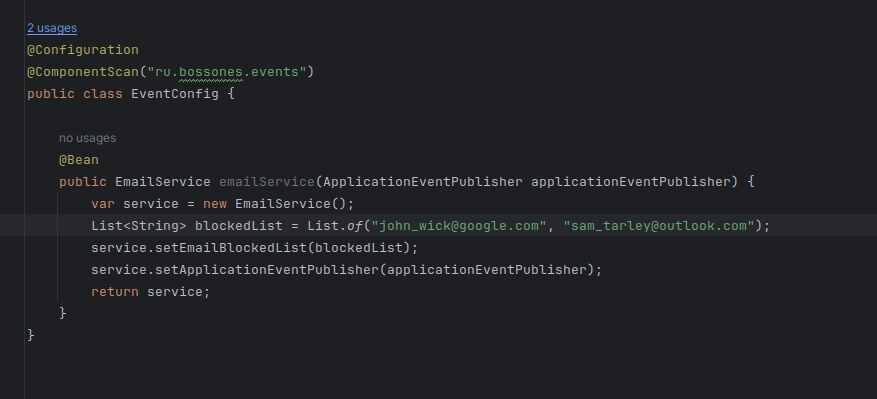
|  |  |
| --- | --- |
| ContextRefreshedEvent | Публикуется, когда контекст инициализируется или обновляется. Инициализируется это значит, что все бины загружены, поспроцессоры зарегестрированы и активированы, синглтоны пре-инстанциированы, и контекст готов к использованию.  Пока контекст не закрыт, обновление может быть вызвано множество раз. |
| ContextStartedEvent | Публиикуется, когда контекст стартанул путем использования start() метода. Стартанул это значит, что все Lifecycle бины получили сигнал старта. Обычно, сигнал испорльзуется для рестарта бина после его явной остановки, но может так же быть использован для старта компонентов, которые не были конфигурированы для автостарта. |
| ContextStoppedEvent | Публикуется, когда контекст останавливается при использовании stop() метода. Останавливается означает, что все Lifecycle бины получили явный сигнал об остановке. Остановленный контекст может быть заново запущен вызовом start() метода. |
| ContextClosedEvent | Публикуется, когда контекст закрывается при использовании close() метода или через JVM shutdown hook. Закрывается — это означает, что все синглтоны уничтожены. Когда контекст закрыт, то он не может быть обновлен или заново запущен |
| RequestHandledEvent | Веб специфичный ивент, который говорит всем бинам, что HTTP запрос был обслужен. |
| ServletRequestHandledEvent | Подкласс RequestHandledEvent с добавлением servlet специфичной логики. |

Можно так же создавать и публиковать свои собственные кастомные ивенты.









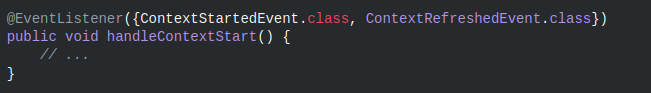
Можно заводить сколько угодно слушателей по желанию, но по умолчанию они все выполняются синхронно. Т.е. следующий ивент не обработается, пока предыдущая обработка не завершится.

### 1.14.2 Annotation-based event listeners

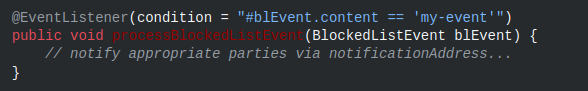
Так же события можно перехватывать при помощи аннотации @EventListener. Внутри этой аннотации указываются необходимые ивенты для обработки.

Аннотация @EventListener указываются над методами, которые обрабатывают данные события.

Аннотация @EventListener позволяет принимать несколько событий, которые будут обработаны методом:



Так же можно добавить дополнительную логику фильтрации в рантайме, используя SpEL, например:

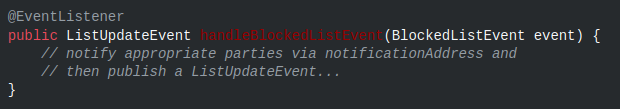


Таким образом, обработка события будет осуществлена, если у ивента BlockedListEvent поле content содержит строку „my-event“.

Каждое выражение SpEL оценивается в соответствии с выделенным контекстом. В таблице перечислены жлементы, доступные контексту, для использовании в SpEL:

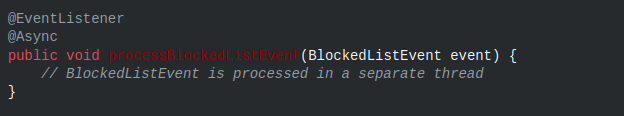
|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Локация | Описание | Пример |
| Event | Root object | Актуальный ApplicationEvent | #root.event or event |
| Arguments array | Root object | Аргументы как массив объектов, использующиеся при вызове метода | #root.args или args, args[0] для доступа к первому элементу и т. д. |
| Argument name | Evaluation context | Наименование одного ииз аргументов метода. Если по некоторым причинам наименование не доступно, индивидуальные аргументы доступны при использовании #a<#arg> синтаксиса, где <#arg> является индексом аргумента. | #blEvent или #a0 (вы также можете использовать обозначение параметра #p0 или #p<#arg> в качестве псевдонима) |

Если необходимо опубликовать событие как результат обработки другого события, можно изменить сигнатуру метода и указать возвращаемый ивент, который должен быть опубликован:



### 1.14.3 Asynchronous Listeners

Если необходимо, чтобы какие то ивенты обрабатывались асинзронно, то можно использовать стандартную поддрежку асинхронных операции с использованием @Async аннотации:



Но есть ограничения при их использовании:

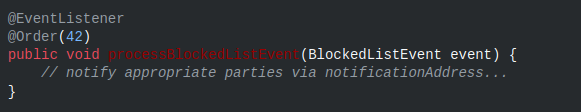
1. Если асинхронный слушатель выбрасывает исключение, то оно не передается публикатору.

2. Методы, реализующие асинхронную обработку событий, не могут публиковать события по окончании обработки. Для этого необходимо явно инжектировать ApplicationEventPublisher, чтобы опубликовать событие вручную.

3. ThreadLocals и контекст логирования не распространяется по умолчанию на обработку таких событий.

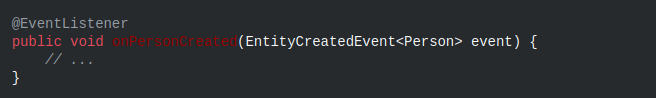
### 1.14.4 Ordering Listeners

Если необходимо опубликовать событие, которое должно быть обработано до какого-то другого, то необходимо указывать порядок:



### 1.14.5 Generic events

Можно так же использовать дженерики для определения структуры события. Т.е. можно получать события одного и того же класса, но с разным типом дженериков:



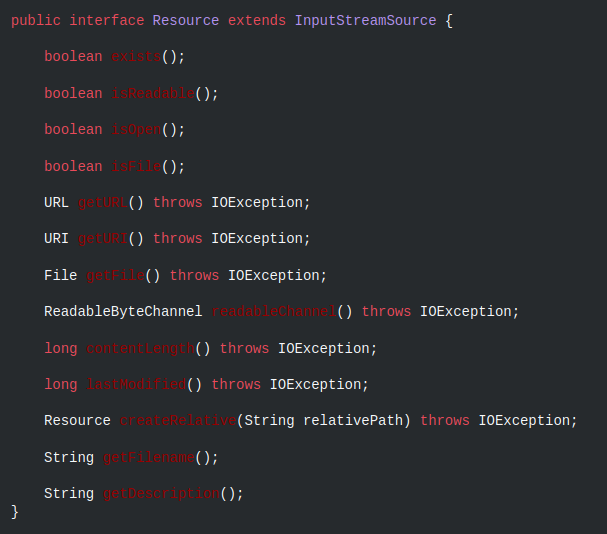
# 2 Resources

## 2.1 Introduction

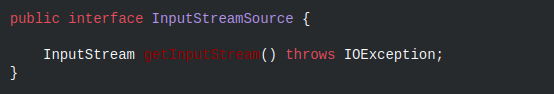
Стандартне классы и обработчики Java java.net.URL, к сожалению, являются не очень адеквтаными для доступа к низкоуровневым ресурсам. Например, нет стандартизованной реализации URL, если необходимо получить доступ к ресурсу из classpath или относительно ServletContext.

## 2.2 Resource Interface

Спринговый интерфейс Resource расположен в org.springframework.core.io пакете и является более функциональным интерфейсом для абстракции доступа к низкоуровневым ресурсам.



Данный интерфейс импелемнтирует InputStreamSource интефрейс, который возвращает InputStream:



Некоторые из самых важных методов это:

* getInputStream(): обнаруживет и открывает ресурс, который возвращает InputStream для чтения из ресурса. Полагается, что каждое выполнение этого метода возвращает новый InputStream. Ответственность на закрытии лежит на вызывающем методе.
* Exists(): возвращает boolean значение, которое обозначает, что ресурс действительно существует в физической форме.
* IsOpen(): возвращает boolean значение, которое говорит о том, что если значение true, то InputStream не может быть прочитана несколько раз и должен быть прочитан, а затем закрыт для избежания утечек. Возвращает false для всех обычных реализаций ресурсов, за исключением InputStreamResource.
* getDescription(): возвращает описание этого ресурса. Используется для вывода ошибок, когда осуществляется работа с ресурсом. Часто это полное имя файла или фактический URL адрес ресурса.

Т.к. Resource интерфейс очень часто используется в спринге и самим спрингом, то это на самом деле очень удобный путь для использвования как основной утилитарный класс для использования в собственном коде для доступа к ресурсам, даже если код ничего не знает об остальных частей спринга.

## 2.3 Built-in Resource Implementations

Спринг включает в себя несколько встроенных реализаций Resource:

* UrlResource
* ClassPathResource
* FileSystemResource
* PathResource
* ServletContextResource
* InputStreamResource
* ByteArrayResource

### 2.3.1 UrlResource

Данный класс является оберткой над URL и может быть использован для доступа к любому объекту с URL, такие как файлы, HTTP, FTP и другие. Все URL имеют стандартизированное строкове представление.

### 2.3.2 ClassPathResource

Данный класс представляет ресурс, который должен быть поолучен из classpath.

Эта реализация ресурса поддерживает разрешение как java.io.File, если ресурс пути к классу находится в файловой системе, но не для ресурсов пути к классам, которые находятся в jar и не были расширены (механизмом сервлетов или какой-либо другой средой) до файловая система. Чтобы решить эту проблему, различные реализации ресурсов всегда поддерживают разрешение в виде java.net.URL.

### 2.3.3 FileSystemResource

Жто реализация Resource для java.io.File обработчикоа. Так же поддерживает java.nio.file.Path. Можно использовать PathResource для использования чистого java.nio.file.Path.

### 2.3.4 PathResource

Это реализация ресурса для дескрипторов java.nio.file.Path, выполняющая все операции и преобразования через API Path. Он поддерживает разрешение как файла и URL-адреса, а также реализует расширенный интерфейс WritableResource. PathResource фактически является чистой альтернативой FileSystemResource на основе java.nio.path.Path с другим поведением createRelative.

### 2.3.5 ServletContextResource

Это реализация ресурса для ресурсов ServletContext, которая интерпретирует относительные пути в корневом каталоге соответствующего веб-приложения. Он всегда поддерживает доступ к потоку и доступ по URL-адресу, но разрешает доступ к java.io.File только тогда, когда архив веб-приложения расширен и ресурс физически находится в файловой системе. Независимо от того, расширен ли он и находится ли он в файловой системе или доступен ли он непосредственно из JAR или где-то еще, например, из базы данных (что вполне возможно), на самом деле зависит от контейнера сервлетов.

### 2.3.6 InputStreamResource

InputStreamResource — это реализация ресурса для данного InputStream. Его следует использовать только в том случае, если конкретная реализация Ресурса не применима. В частности, по возможности отдавайте предпочтение ByteArrayResource или любой из реализаций файловых ресурсов. В отличие от других реализаций ресурса, это дескриптор уже открытого ресурса. Поэтому он возвращает true из isOpen(). Не используйте его, если вам нужно где-то сохранить дескриптор ресурса или если вам нужно прочитать поток несколько раз.

### 2.3.7 ByteArrayResource

Это реализация ресурса для данного массива байтов. Он создает ByteArrayInputStream для данного массива байтов. Это полезно для загрузки содержимого из любого заданного массива байтов без необходимости прибегать к одноразовому InputStreamResource.

### 2.4 ResourceLoader Interface

# 3. Validation, Data Binding and Type Conversion

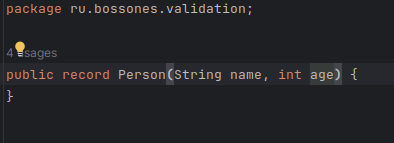
## 3.1 Validation by Using Spring’s Validator interface

Интерфейс Validator предлагается для использования его в валидации. Он работает с использованием Errors объекта, поэтому валидаторы могут сообщать ошибки в Errors объект.

Для создания собственного валидатора необходимо переопределить 2 метода у интерфейса Validator:

* supports(Class): Проверка на то, что Validator может валидировать экземпляры предоставленного класса.
* validate(Object, Errors): Валидирует объект, и в случае ошибок, регестрирует их в объект Errors.

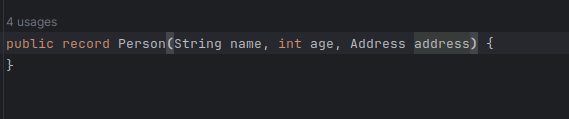
Специально существует вспомогательный класс ValidationUtils, который позволяет упростить валидацию и передачу ошибок в объект Errors.

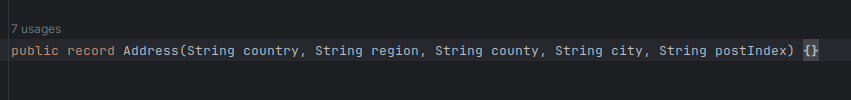




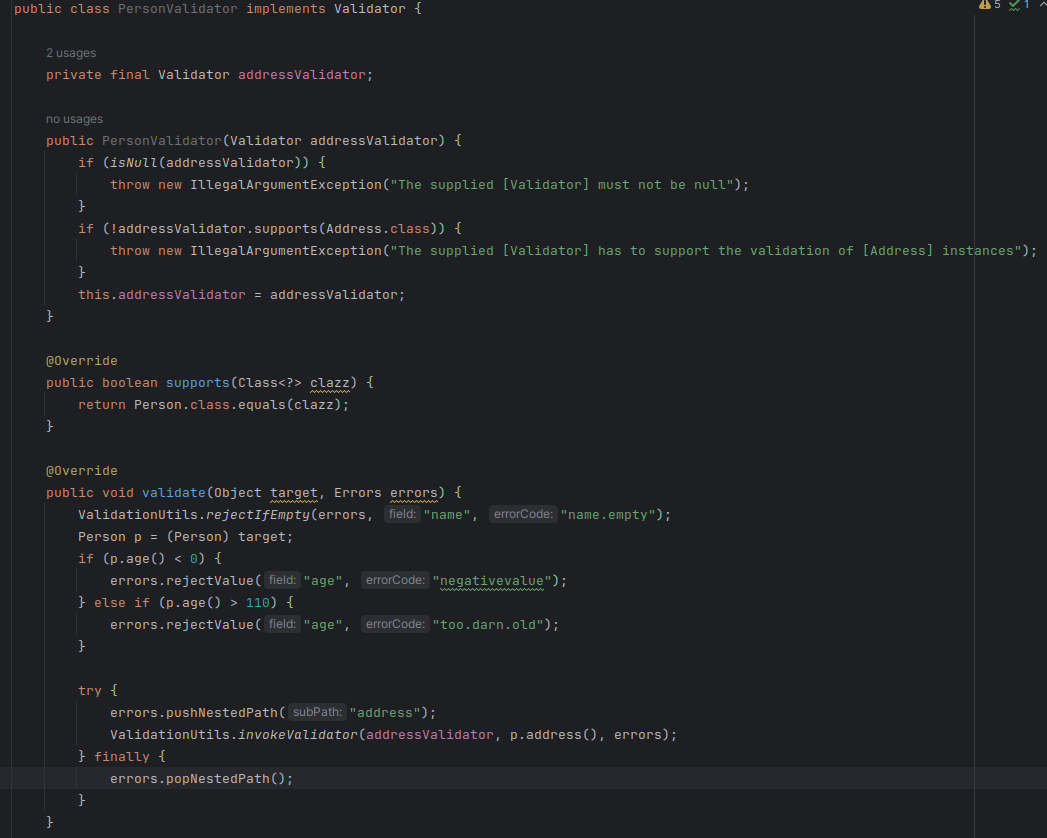
ValidationUtils.rejectIfEmpty() позволяет отклонить name проперти если оно null или является пустой строкой.

Если, например, существует класс, который в себе содержит большое количество других классов, которые так же необходимо провалидировать в рамках объемлющего класса, то можно переиспользовать валидаторы других классов. Например, если у Заказчика есть сложная композиционная структура как Адрес (с несколькими параметрами), то можно отдельно написать валидатор для Адреса и переиспользовать этот валидатор в Заказчике.









## 3.2 Data Binding

Data binding очень юзабельна для связывания данных, введенные пользователем в целевой объект где пользовательские данные представлены мапой с путями пропертей как ключи, следуя JavaBeans convention. DataBinder это основной класс, который поддерживает это и обеспечивает 2 пути для связывания данных, введенных пользователем:

* Constructor binding — связывает пользовательский ввод в публичный конструктор, осуществляя поиск значений аргументов конструктора во входных данных пользователя.
* Property binding — связывает пользовательский ввод к сеттерам, осуществляя поиск совпадений из пользовательского ввода к пропертям структуры целевого объекта.

Можно применять оба пути или только один

### 3.2.1 Constructor Binding

Для того, чтобы использовать данную связь:

1. Создать DataBinder с null как целевым объектом.

2. Установить targetType для целевого класса.

3. Вызывать конструктор.

Целевой класс должен иметь один публичный конструктор или один непубличный конструктор с аргументами. Если существуют несколько конструкторов, то дефолтный конструктор если представлен будет использоваться.

По дефолту наименования параметров конструктора используютсся для поиска значений аргументов, но можно самостоятельно конфигурировать через NameResolver.

### 3.2.2 Property Binding with BeanWrapper

JavaBean это класс с дефолтным безаргументынм конструктором, который соответствует соглашению об именах, согласно которому (например) свойство с именем bingoMadness будет иметь метод установки setBingoMadness() и метод получения getBingoMadness().

Здесь один достаточно вадный класс — это BeanWrapper интерфйес и его соответствующая реализация — BeanWrapperImpl.

#### 3.2.2.1 Setting and Getting Basic and Nested Properties

Установка и получение пропертей осушествляется через setPropertyValue и getPropertyValue перегруженными вариантами метода BeanWrapper:

|  |  |
| --- | --- |
| Expression | Explanation |
| name | Указывает имя свойства, соответствующее методам getName() или isName() и setName(..). |
| account.name | Указывает на вложенное проперти name внутри проперти account который соответствует методам getAccount().setName() или getAccount().getname() |
| account[2] | Укащывает на третий элемент в проперти в индексированном account. Индексированными могут быть array, list или другие натуральными отсортированными коллекциями |
| account[COMPANYNAME] | Указыват на значение мазы индекс которого называется COMPANYNAME ключ у account Map. |

Пример использования:



### 3.2.3 PropertyEditor’s

Спринг используеь концепт PropertyEditor для влияния конверсии между Object и String. Он может быть удобным для представления пропертей в различном виде. Например, Date может быть представлен в человеческом читаемом виде (String: „2007-14-09“) и мы можем продолжать конвертировать humanreadable форму обратно в оригинальную дату или даже лучше, конвертировать любую дату введенную в человеческом формате обратно в Date объекты. Это поведение может быть получено путем регестрирования кастомного редактора PropertyEditor.

Spring имеет множество встроенных PropertyEditor реализаций для облегчения жизни. Все они расположены в пакете org.springframework,beans.properyeditors.