**Spring Core** представляет собой просто контейнер внедрения зависимостей, с несколькими удобными слоями (например: доступ к базе данных, прокси, аспектно-ориентированное программирование, RPC, веб-инфраструктура MVC). включает в себя Beans, Core, Context и SpEL (expression language).

На практике DI осуществляется путем передачи параметров конструктора или с помощью setter-ов.

**Преимущества DI**

* Уменьшенная связь между частями приложения
* Улучшенное тестирование
* Улучшенная архитектура приложения
* Уменьшает шаблонный код
* Стандартизирует разработку приложения

Ключевым элементом Spring Framework является Spring Container. Container **создаёт объекты**, **связывает их вместе**, **настраивает** и **управляет ими** от создания до момента уничтожения.

Конфигурировать Spring Container можно 3 способами:

* XML
* Аннотации Java
* Java код

В Spring имеется 2 вида контейнеров:

1. Spring BeanFactory Container
2. Spring ApplicationContext Container

**Жизненный цикл контейнера**

1. Контейнер создается при запуске приложения
2. Контейнер считывает конфигурационные данные с помощью класса BeanDefinitionReader.
3. Из конфигурационных данных создается описание бинов, и кладутся в мапу <id бина, декларация (какой класс, инит метод, свойства и т.д.)>
4. BeanFactoryPostProcessors обрабатывают описание бина
5. BeanFactory создает бины используя их definition.
6. Бины инициализируются — значения свойств и зависимости внедряются в бин
7. BeanPostProcessor дополнительно обрабатывает бины перед тем, как положить их в контейнер методом **postProcessBeforeInitialization.**
8. Вызывается init метод.
9. BeanPostProcessor еще раз выполняет настройку бинов уже методом **postProcessAfterInitialization.**
10. Приложение запущено и работает
11. Инициализируется закрытие приложения
12. Контейнер закрывается
13. Вызываются callback methods

**Bean –** это объект, который создается и контролируется спрингом, и находятся внутри его DI-контейнера.

**POJO (Plain Old Java Object)** - «старый добрый Java-объект», простой Java-объект, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов сверх тех, которые нужны для бизнес-модели. Обычно имеется ввиду объект без ссылок на какие-либо фреймворки. Такой Класс может быть использован любой джава программой, поскольку он не привязан к какому-либо фреймворку. POJO не имеет никаких соглашений об именовании свойств и методов

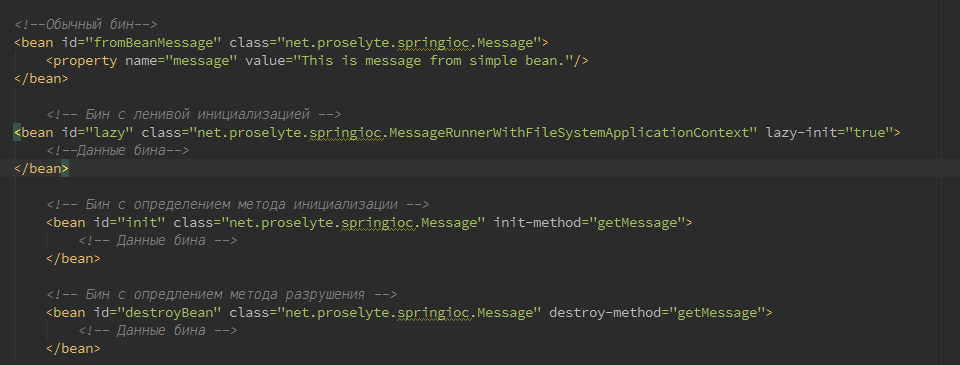
Изначально в Java bean – тот же POJO класс, но написанный по определенным правилам. Используются для объединения нескольких объектов в один. Спецификация [Sun Microsystems](https://ru.wikipedia.org/wiki/Sun_Microsystems) определяет JavaBeans как повторно используемые программные компоненты, которыми можно управлять, используя графические конструкторы и средства [IDE](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8).

Bean должен соответствовать следующим правилам:

* Класс должен иметь конструктор без параметров, с модификатором доступа public. Такой конструктор позволяет инструментам создать объект без дополнительных сложностей с параметрами.
* Свойства класса приватные, должны быть доступны через get, set и другие методы (так называемые методы доступа), которые должны подчиняться стандартному соглашению об именах. Это легко позволяет инструментам автоматически определять и обновлять содержание bean’ов. Многие инструменты даже имеют специализированные редакторы для различных типов свойств.
* Класс должен быть сериализуем. Это даёт возможность надёжно сохранять, хранить и восстанавливать состояние bean независимым от платформы и виртуальной машины способом.
* Класс должен иметь переопределенные методы equals(), hashCode() и toString().

Определения бина в Spring имеет следующие атрибуты:

* **class** – обязательный атрибут, указывает конкретный класс Java-приложения, который будет использоваться для создания бина;
* **name** – уникальный идентификатор бина. В случае конфигурации с помощью xml-файла, вы можете использовать свойство “id” и/или “name” для идентификации бина; В **name** можно указать **несколько значений** отделив их запятой, точкой с запятой или пробелом.
* **scope** – область видимости создаваемых объектов.
* **constructor**-**arg –** определяет конструктор, использующийся для создания бина.
* **properties –** определяет свойства для создания бина.
* **initialization method -** здесь определяется метод инициализации бина.
* **destruction method -** Метод уничтожения бина, который будет использоваться при уничтожении контейнера, содержащего бин.
* **autowiring mode -** Определяет режим автоматического связывания при внедрении зависимости.
* **lazy-initialization mode -** Режим ленивой инициализации даёт контейнеру IoC команду создавать экземпляр бина при первом запросе, а не при запуске приложения.



**Bean Scopes**

В Spring Framework имеются пять возможных значений свойства scope:

* **singleton –** создается только один экземпляр бина. Этот тип используется по умолчанию. Следует осторожно использовать публичные свойства класса, т. к. они не будут потокобезопасными.  
  Этот экземпляр помещается в кэш синглтонов, и все последующие запросы бина с таким именем будут возвращать объект из кэша. Синглтоны создаются при поднятии контекста.
* **Prototype –** создает и возвращает новый экземпляр бина на каждый запрос. Создаются в тот момент когда они нужны. Они не хранятся в контейнере, и метод **destroy** для них **не вызывается**, так как спринг не управляет жизненным циклом прототипов, он передает их клиенту и дальше не парится. Обычно используются когда бин может менять свое состояние.
* **Request -** Создаётся один экземпляр бина на каждый HTTP запрос. Касается исключительно ApplicationContext.
* **Session** - Создаётся один экземпляр бина на каждую HTTP сессию. Касается исключительно ApplicationContext.
* **global-session -** Создаётся один экземпляр бина на каждую глобальную HTTP сессию. Касается исключительно ApplicationContext.

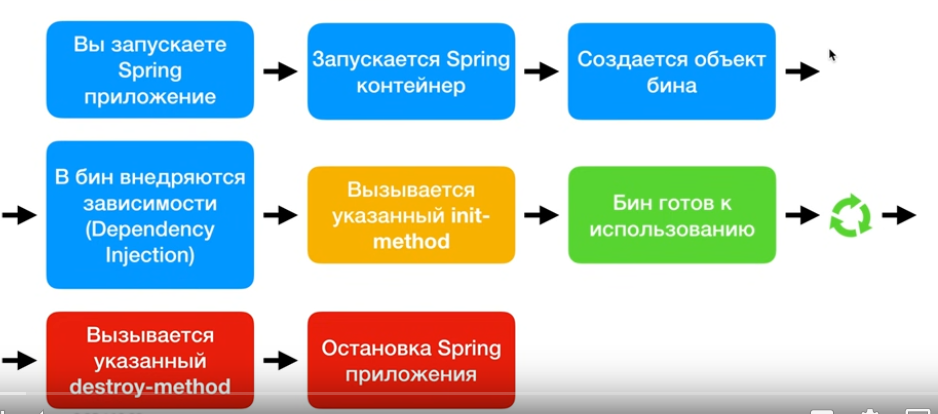
Если бин со скоупом Singlteton ссылается на prototype, то prototype не будет обновляться внутри синглтона, ибо в этом нет нужды. Если все-таки нужно при каждом обращении к prototype работать с новым бином есть 2 пути:

1. прописать **Scope(“prototype”, proxyMode = ScopedProxyMode.TARGET\_CLASS)**. Тогда любое обращение к prototype бину будет возвращать новый объект.
2. Если мы хотим не всегда возвращать новый экземпляр prototype, а только в определенном singleton бине: Можно написать абстрактный метод внутри синглтона для получения прототайпа, и сделать сам класс абстрактным. А в конфигурации бина с помощью java кода возвращать имплементацию, в которой метод getPrototypeObject будет возвращать нужный бин.



В Spring имеется интерфейс **Scope,** с помощью которого можно даже реализовывать свой кастомный скоуп. В BeanFactoryPostProcessor его можно зарегистрировать с помощью метода **beanFactory.registerScope(“name”, new ScopeObj())**.

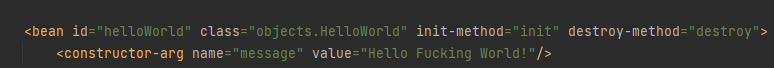
**Жизненный цикл бина**



В первую очередь запускается спринг-контейнер. После этого, он по необходимости и в соответствии с запросами создает экземпляры бинов и внедряет необходимые зависимости. И наконец, бины, связанные с контейнером, уничтожаются, когда контейнер завершает свою работу.

Если мы хотим выполнять какой-то код во время инстанцирования бина или при его уничтожении, мы можем вынести его в специальные **init()** и **destroy()** методы.

**Init метод** нужен на случай, если нам нужно использовать объекты, внедряемые спрингом во время инициализации объекта. В конструкторе, как правило эти объекты будут еще null.





Либо можно заимплементить 2 интерфейса: **InitializingBean** и **DisposableBean** а затем переопределить методы **afterPropertiesSet**() и **destroy**(). Первый метод вызывается после создания бина и внедрения зависимостей, второй при завершении работы контейнера.

Можно также воспользоваться аннотациями **@PostConstruct** и **@PreDestroy**.

Если использовать все эти способы одновременно, то сначала вызывается @PostConstruct, затем afterPropertiesSet, и наконец init-метод.

**BeanPostProcessor**

Позволяет настраивать бины до того, как они попадают в контейнер. Для prototype вызывается каждый раз при запросе.

Как правило на этом этапе обрабатываются аннотации такие как @Autowied и т.п.

Имеет 2 метода:

– **postProcessBeforeInitialization(Object bean, String beanName)** вызывается до init метода.

– **postProcessAfterInitialization(Object bean, String beanName)** вызывается после init метода. Второй вызов необходим, если мы хотим например навесить какие-нибудь аннотации, которые изменят код нашего класса, либо вообще обернут его в обертку (Proxy). Например какие-нибудь @Transactional либо наши самописные аннотации. Мы хотим быть уверены, что инит метод и первый вызов BeanPostPocessor будут работать с оригинальным объектом. В спринг встроены CommonAnnotationPostProcessor или AutowiredAnnotationBeanPostProcessor.

В Spring имеются уже реализованные BeanPostProcessors:

* **AutowiredAnnotationBeanPostProcessor –** обрабатывает аннотации @Autowired, @Value, @Inject и инджектит нужные зависимости.
* **InitDestroyAnnotationBeanPostProcessor –** вызывает аннотированные init и destroy методы.

**BeanFactory**

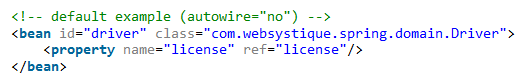
Самый простой контейнер, основан на интерфейсе *org.springframework.beans.factory.BeanFactory*.

Наиболее часто используемая реализация интерфейса BeanFactory – **XmlBeanFactory.** Он получает метаданные из конфигурационного XML файла. Может работать только с двумя типами скоупов: Prototype и Singleton. Бины создаются только по запросу.

Позволяет автоматически внедрять нужные зависимости в бин. Для этого нужно указать атрибут **autowire** в определении бина, и указать тип (byName, byType, constructor, no):

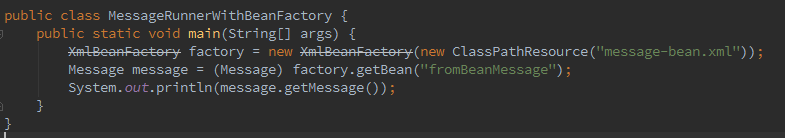


При указании типа «no» нужно явно указывать зависимости через атрибут **ref**.



Не поддерживает конфигурацию на основе аннотаций.

Для использования BeanPostprocessor и BeanFactoryPostprocessor их нужно зарегистрировать.



**BeanFactoryPostProcessor**

Он позволяет настраивать bean-definitions до того, как создаются бины. Например позволяет подставлять переменные среды (${password} заменяется на настоящий пароль). Либо можно вообще заменить класс бина на другой (того же интерфейса).

Позволяет еще подкрутить и BeanFactory.

Этот интерфейс имеет единственный метод **postProcessBeanFactory(ConfigurableListableBeanFactory var1)**

Spring поставляет несколько полезных реализаций BeanFactoryPostProcessor, например, читающий property-файлы и получающий из них свойства бинов.

**PropertyResourceConfigurer** это интерфейс, его реализации: **PropertyOverrideConfigurer** – читает файл свойств, и находит в нем строки вида "beanName.property=value" и вставляет эти значения в beanDefinitions.

**PropertyPlaceholderConfigurer –** заменяет **“${…}”** на соответствующие значения из property файлов.

**ApplicationContext**

Обертка над bean factory, предоставляющая некоторые дополнительные возможности, например AOP, транзакции, безопасность, i18n, публикация и прослушивание событий, и т.п.

Автоматически регистрирует BeanFactoryPostProcessor и BeanPostProcessor.

В отличи от BeanFactory создает бин сразу **при запуске приложения**.

Чаще всего используются следующие реализации ApplicationContext:

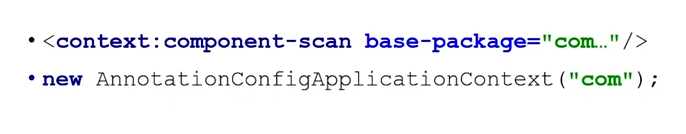
* **FileSystemXmlApplicationContext –** загружает данные о бинах из XML файла. В конструкторе необходимо указать полный адрес конфигурационного файла.
* **ClassPathXmlApplicationContext –** в конструкторе необходимо относительный адрес конфигурационного файла(CLASSPATH).
* **WebXmlApplicationContext –** эта реализация получает необходимую информацию из веб-приложения.
* **AnnotationConfigApplicationContext –** работает с классами, аннотированными **@Configuration**.  
  

У него есть методы:

* **getBean() –** получить бин по id либо по классу, а можно и по id и по классу

**Аннотации**

**@Component –** указывает, что нужно создать бин этого класса. Этот бин затем может использоваться для автовнедрения в другие бины. Затем можем в XML конфигурации или при создании AplicationContext указать в каких пакетах проводить поиск.



**Stereotypes:**

* **@Scope -**указать скоуп бина.
* **@Controller –** указывает, что класс является контроллером для обработки запросов от клиента.
* **@RestController –** указывает, что класс является контроллером для REST. Содержит аннотации @Controller и @ResponseBody.
* **@Service -**  Указывает, что класс является сервисом для выполнения бизнес-логики.
* **@Repository -** Указывает, что класс является репозиторием для работы с бд
* **@Configuration -** Указывает, что класс содержит Java-конфигурацию(@Bean-методы)

**@Lazy –** сделать ленивую загрузку бина.

**@DependsOn –** указать, что бин зависит от другого бина.

**@Import –** импортировать другую Java-конфигурацию.



**@ImportResource –** импортировать конфигурацию XML в Java конфигурацию

**@ComponentScan –** сканировать дополнительные пакеты для поиска бинов.

**@Autowire –** помечает поле (setter или параметр конструктора), указывая что Spring должен внедрить эту зависимость в бин. DI через конструктор и сеттер является лучшим способом, т.к. для него не нужно использовать рефлексию.

Этой аннотацией можно пометить несколько конструкторов, тогда для инжекта выберется конструктор с максимальным количеством параметров, бины которых спринг может предоставить.

Если внедряемый объект массив или коллекция с дженериком, то Spring внедрит все бины, подходящие по типу в этот массив.

Имеет параметр **required,** который указывает является ли зависимость обязательной (по умолчанию true). Если установлено false, то Spring не будет выбрасывать исключение, если бин не найден.

**@Inject** – аналогична @Autowired. Но @Autowired специфична для спринга, а @Inject стандартная, и будет работать не только в спринге, но и в другом DI контейнере.

**@Bean –** аннотация @Bean прописанная перед методом, информирует Spring о том, что возвращаемый данным методом объект должен быть зарегистрирован как бин. Имеет параметры *scope, destroyMethod, initMethod, name(*по умолчанию имя метода*), value(*алиас для name*).*

**@Value** – аннотация для внедрения простых значений в свойства. Такие значения можно получать из property файлов, из бинов и т.п. Аргумент, переданный аннотации @Value может быть только String, Spring преобразует значение в указанный тип.



Можно также указать значение по умолчанию



Можно использовать эту аннотацию над сеттером, над параметром метода

**ClassPathBeanDefinitionScanner**

Специальный класс, который сканирует пакеты в поисках аннотаций Component. Он создает дополнительные BeanDefinitions из всех классов, над которыми стоит @Component либо другая аннотация, аннотированная @Component (@Service, @Repository и т.п.).

**Решение конфликтов в @Autowired**

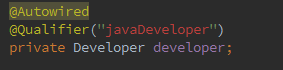
Autowired внедряет бины основываясь на их типе. Если могут быть использовано несколько бинов возникает конфликт. Для решения можно воспользоваться аннотациями:

**@Primary –** устанавливается над бином, чтобы указать что он должен использоваться как основной. По сути мы повышаем его приоритет перед другими бинами того же типа.

**@Order** определяет порядок сортировки бинов. Значение по умолчанию Ordered.LOWEST\_PRECEDENCE указывает что бин имеет наименьший приоритет. Ordered.HIGHEST\_PRECEDENCE – для максимального приоритета.

Может использоваться например при построении цепочки фильтров. Либо для указания порядка в коллекциях. Так же этой аннотацией можно контролировать порядок выполнения BeanPostProcessor-ов и BeanFactoryPostProcessor-ов.

**@Qualifier –** устанавливается непосредственно в месте внедрения бина через Autowired**.** В ней указывается имя нужного бина.



При **Инжекте коллекций** спринг может сам создавать коллекции, вместо тех что мы явно прописываем как бины. Например в следующем примере в сервис заинжектится Мапа созданная самим спрингом, ключом в которой будет имя бина, а значением сам бин. Чтобы избежать этого можно использовать конкретную реализацию коллекции (например HashMap) либо иcпользовать **@Qualifier** (более предпочтительный способ).



**Конфигурация**

**С помощью XML**

Недостатки:

* Если бин нуждался в кастомной логике, приходилось писать костыли.

**С помощью Java-кода**

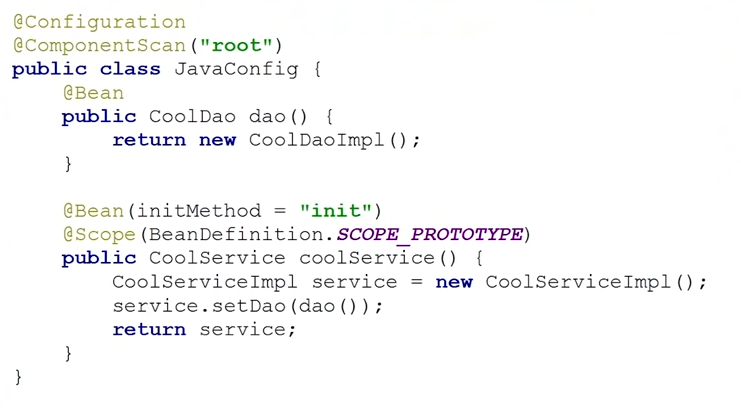
Плюсы:

* Дает возможность настраивать кастомно бины, запускать какие-нибудь методы.

Контекст создается следующим образом:



Класс, который занимается сканированием и регистрированием JavaConfig-ов называется **AnnotatedBeanDefinitionReader**, который является частью ApplicationContext. Когда нужно будет получить бин, будет происходить делегирование фабричным методам.



**ConfigurationClassPostProcessor**  - особый BeanFactoryPostProcessor. Он создает Bean-Definitions по аннотации @Bean.

**С помощью аннотаций**

Spring сканирует пакеты, указанные в @ComponentScan и создает бины помеченные @Component и т.д.

Если в приложении используется как xml так и аннотации, то xml будет перебивать бины созданные с помощью аннотаций, так как инъекция аннотациями выполняется раньше чем XML/

**Профили**

Профили в Spring позволяют кастомизировать приложение для работы в различном окружении. В зависимости от активного профиля можно инстанциировать различные реализации одного и того же бина, а также присваивать различные значения свойствам приложения.

Аннотация **@Profile** позволяет использовать разные настройки для Spring в зависимости от указанного профиля. Ее можно ставить на @Configuration и @Component классы, а также на @Bean методы.

public interface SomeService {

…

}

@Service(SomeService.NAME)

@Profile("dev")

public class SomeDevServiceBean implements SomeService {

…

}

@Service(SomeService.NAME)

@Profile("prod")

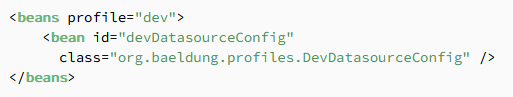
public class SomeProdServiceBean implements SomeService {

…

}

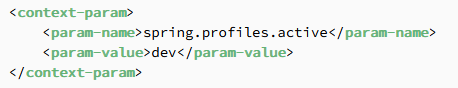
Имя профиля можно писать с восклицательным знаком. Это используется чтобы исключить создание бина в указанном профиле.

В xml указать профиль можно атрибутом profile тега beans.



Активный профиль можно указать несколькими способами.

Для веб-приложений можно использовать WebApplicationInitializer или web.xml файл.



Можно использовать системный JVM параметр



В maven можно использовать свойство **spring.profiles.active**

В тестах удобно использовать аннотацию @ActiveProfile.

Также можно создавать различные properties файлы для различных профилей.

**События в AplicationContext**

**ApplicationContextEvent** - основной класс для событий, возникающих в процессе жизненного цикла ApplicationContext. Его подклассы:

* **ContextRefreshedEvent** - публикуется автоматически после поднятия контекста
* **ContextStartedEvent** - публикуется методом ApplicationContext#start
* **ContextStoppedEvent** - публикуется методом ApplicationContext#stop
* **ContextClosedEvent** - публикуется автоматически перед закрытием контекста

**ApplicationListener** - интерфейс который позволяет обрабатывать ApplicationEvent события. Можно использовать аннотацию **@EventListener** вместо интерфейса.

**Механизм перезапуска приложений**

Перезапустить приложение может понадобиться в случаях:

* Если нам нужно загрузить измененную метаинформацию
* Если нужно изменить текущие активные профили
* Если контекст упал и мы хоти иметь возможность автоматически поднять его

Мы можем перезапустить наше приложение **закрыв контекст приложения и создать новый контекст с нуля**.

В некоторых случаях мы можем перезагрузить контекст вызвав его метод **refresh**. Однако не все типы контекста это поддерживают. Например, **FileSystemXmlApplicationContext**, **GroovyWebApplicationContext** и некоторые другие поддерживают его. Однако мы должно избегать вызова этого метода, так как он разработан как внутренний метод, используемый фреймворком для инициализации контекста приложения.