Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Конспект лекции №3
Фреймворк Spring. Конфигурация при помощи аннотаций.

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

1 Предисловие

В данном конспекте будут рассмотрены не только аннотации, но и некоторые темы, которые не уместились в рамки первого конспекта.

2 Область действия компонентов

2.1 Паттерн Singleton (Одиночка)

Паттерн проектирования «Одиночка» (Singleton) — один из порождающих паттернов проектирования и один из простейших паттернов в языке Java. Согласно этому паттерну класс использует для всех вызовов один и тот же объект, то есть создание экземпляров класса ограничивается одним объектом с предоставлением глобальной точки доступа к данному классу. Это значит следующее: класс отвечает за создание объекта, а также гарантирует, что для всех клиентских обращений к этому объекту будет создан лишь один объект. Данный класс не допускает непосредственного создания своих объектов. Получить его экземпляр можно только с помощью предоставляемого им статического метода.

Этот паттерн проектирования удобен, когда необходим ровно один объект для согласования действий в масштабах всей системы. Создать экземпляр можно двумя способами:

- немедленное создание экземпляра (early instantiation) — экземпляр создается во время загрузки;
- отложенное создание экземпляра (lazy instantiation) — экземпляр создается при необходимости.

Фреймворк Spring предлагает в качестве реализации паттерна «Одиночка» одиночную область видимости для компонентов. Она схожа с данным паттерном, но не идентична таковому в языке Java. Компонент с одиночной областью видимости в Spring означает один экземпляр компонента в расчете на компонент и в расчете на контейнер. Если описать один компонент определенного класса в отдельном контейнере Spring, то последний создаст один и только один экземпляр класса с данным описанием компонента.

2.2 Паттерн Prototype (Прототип)

Паттерн проектирования «Прототип» (Prototype) относится к семейству порождающих. Он используется для создания объектов путем клонирования на основе экземпляра-прототипа. В корпоративных приложениях создание объекта требует значительных затрат ресурсов в смысле непосредственного создания объектов и инициализации начальных значений их свойств. В случае подобного объекта имеет смысл воспользоваться паттерном проектирования «Прототип». При этом просто копируется уже существующий подобный объект вместо требующего временных затрат создания нового.

Этот паттерн включает реализацию прототипного интерфейса в целях создания клона существующего объекта. Используется в случаях, когда непосредственное создание объекта требует значительных затрат ресурсов,

например, когда объект создается после дорогостоящей операции с базой данных. Объект можно закэшировать, вернуть его клон при следующем запросе и обновлять базу по мере необходимости, уменьшая таким образом количество требуемых обращений к ней.

2.3 Область действия компонентов Spring

По умолчанию все компоненты Spring единичны. Когда контейнер передает компонент (либо через связывание, либо как результат вызова метода контейнера getBean()), всегда будет передан тот же самый экземпляр компонента. Однако иногда бывает необходимо получить уникальный экземпляр компонента при каждом обращении.

При объявлении <bean> компонента можно определить область его действия. Чтобы заставить фреймворк Spring создавать новый экземпляр при каждом обращении, в объявление компонента следует добавить атрибут scope со значением prototype. Например:

```
<bean id="engineBean"
    class="ru.firstapp.GasEngine" scope="prototype">
    <property name="volume" value="${gasEngine.volume}"/>
</bean>
```

Помимо значения prototype, атрибуту scope можно присвоить ряд других значений, определяющих область действия компонента.

В основном вполне достаточно будет оставить область действия в значении по умолчанию singleton, но в ситуациях, когда желательно использовать Spring как фабрику новых экземпляров доменных объектов, может пригодиться область действия prototype. Если доменные объекты определены как прототипы компонентов, вы легко сможете настроить их в Spring как любые другие компоненты. При этом фреймворк Spring всегда будет возвращать уникальный экземпляр при обращении к прототипу компонента.

3 Инициализация и уничтожение компонентов

При создании компонента может потребоваться выполнить некоторые операции по его инициализации, чтобы привести его в пригодное для использования состояние. Аналогично, перед уничтожением и удалением компонента из контейнера, может потребоваться выполнить некоторые заключительные операции. Для этой цели фреймворк Spring позволяет определять методы управления жизненным циклом компонента.

Чтобы определить методы, вызываемые при создании и уничтожении компонента, просто добавьте в элемент <bean> атрибуты init-method и/или destroy-method. Атрибут init-method определяет метод, вызываемый сразу после создания экземпляра компонента. Аналогично атрибут destroy-method определяет метод, вызываемый непосредственно перед удалением компонента из контейнера.

Примерно жизненный цикл бина можно проиллюстрировать следующим образом:

Запуск Spring приложения → Запуск Spring контейнера → Создание объекта бина → Внедрение зависимостей в компоненту (бин) → Вызов init-метода → Работа приложения с готовыми бинами → Вызов destroy-метода → остановка Spring приложения.

Для примера создадим новый класс Lamp:

```
public class Lamp {
	public void turnOn(){
		System.out.println("Да будет свет!");
	}

	public void turnOff() {
		System.out.println("Настала тьма!");
	}
}
```

И создадим наш конфигурационный файл:

```
<bean id="lampBean"
    class="ru.firstapp.Lamp"
    init-method="turnOn"
    destroy-method="turnOff">
</bean>
```

Также изменим наш класс, который запускает приложение

```
Lamp lamp = context.getBean("lampBean", Lamp.class);

context.close();
```

Видим результат в консоли:

```
Да будет свет!
Настала тьма!
```

Обратите внимание, что destroy-метод не вызывается для бинов имеющих scope prototype, этот функционал программист должен взять на себя. Spring берет ответственность только за singleton бины.

4 Фабричные методы

«Фабрика» (Factory) — порождающий паттерн проектирования. Известен также как паттерн проектирования «Фабричный метод». Согласно ему для получения объекта класса можно не раскрывать клиенту нижележащую логику. Новый объект назначается вызывающей стороне с помощью общего интерфейса или абстрактного класса. Это значит, что данный паттерн

проектирования скрывает фактическую логику реализации объекта, способ его создания и то, какой класс он воплощает. В результате клиента не будет заботить создание объекта, управление им и его уничтожение — паттерн «Фабрика» позаботится обо всем этом. Данный паттерн — один из наиболее используемых паттернов проектирования Java.

По-простому, это создание объектов не через new, а через вызов особого фабричного метода.

4.1 Создание компонентов с помощью фабричных методов

Иногда единственный способ создать объект заключается в том, чтобы использовать статический фабричный метод. Фреймворк Spring поддерживает возможность связывания компонентов, объявленных с помощью элементов <bean>, которые содержат атрибут factory-method.

Для примера оставим наш класс Lamp и создадим ему пустой приватный конструктор и создадим фабричный метод:

```
private Lamp() {
}

public static Lamp getLamp(){
	System.out.println("Фабричный метод. Хоба!");
	return new Lamp();
}
```

```
<bean id="lampBean"
    class="ru.firstapp.Lamp"
    init-method="turnOn"
    destroy-method="turnOff"
    factory-method="getLamp">
</bean>
```

Тогда в консоли получаем:

```
Фабричный метод. Хоба!
Да будет свет!
Настала тьма!
```

Достаточно глупый пример, но на то он и пример. :)

Элемент <bean> имеет атрибут factory-method , позволяющий определить статический метод, который должен вызываться для создания экземпляра класса вместо конструктора.

На этом завершим рассмотрение способов построения конфигурации при помощи xml-конфигурации и перейдем к аннотациям.

5 Аннотации

Аннотации – это пометки, с помощью которых программист указывает компилятору Java и средствам разработки, что делать с участками кода помимо

исполнения программы. Аннотировать можно переменные, параметры, классы, пакеты. Можно писать свои аннотации или использовать стандартные – встроенные в Java.

Например ранее уже используемая нами аннотация @Override – стандартная аннотация Java, которая предупреждает, что ниже мы что-то переопределим.

Аннотации позволяют:

- автоматически создавать конфигурационные XML-файлы и дополнительный Java-код на основе исходного аннотированного кода;
- документировать приложения и базы данных параллельно с их разработкой;
- проектировать классы без применения маркерных интерфейсов;
- быстрее подключать зависимости к программным компонентам;
- выявлять ошибки, незаметные компилятору;
- решать другие задачи по усмотрению программиста.

Поясним понятие «маркерный интерфейс». Интерфейсы без каких-либо методов действуют как маркеры. Они лишь говорят компилятору, что объекты классов, которые имплементируют такой интерфейс без методов, должны иметь отличительные черты, восприниматься иначе. Например, java.io.Serializable, java.lang.Cloneable, java.util.EventListener. Маркерные интерфейсы ещё известны как «теги» — они добавляют общий тег ко всем унаследованным классам и объединяют их в одну категорию.

5.1 Обработка аннотации в Java

На основе аннотаций компилятор может с помощью специальных обработчиков генерировать новый код и файлы конфигурации.

Обработчиками обычно выступают библиотеки и утилиты, которые можно брать у сторонних авторов (или создавать самостоятельно) и прикреплять к проекту в среде разработки. Способ подключения зависит от IDE или системы сборки. В Maven обработчики подключают с помощью модуля annotation-user или плагина maven-compiler-plugin.

Парсинг аннотаций происходит циклически. Компилятор ищет их в пользовательском коде и выбирает подходящие обработчики. Если вызванный обработчик на основе аннотации создаёт новые файлы с кодом, начинается

следующий этап, где исходным материалом становится сгенерированный код. Так продолжается до тех пор, пока не будут созданы все необходимые файлы.

6 Связывание посредством аннотаций в Spring

С выходом версии Spring 2.5 появился один из самых интересных способов связывания компонентов, основанный на автоматическом связывании свойств с использованием аннотаций. Автоматическое связывание с использованием аннотаций мало чем отличается от использования атрибута autowire в XML-файле конфигурации. (О котором мы не говорили в предыдущей лекции, так как предполагаем, что слушатель будет пользоваться связыванием при помощи аннотаций) Но он обеспечивает возможность более точного управления автоматическим связыванием, позволяя выборочно объявлять свойства, доступные для автоматического связывания.

Поддержка автоматического связывания посредством аннотаций по умолчанию отключена. Поэтому, прежде чем использовать этот способ, необходимо включить его поддержку в конфигурационном файле. Проще всего это сделать с помощью элемента <context:annotation-config> из пространства имен context :

```
<context:annotation-config />
```

Элемент <context:annotation-config> сообщает фреймворку Spring, что для связывания свойств должен использоваться механизм на основе аннотаций. Добавив этот элемент, можно приступать к добавлению соответствующих аннотаций в программный код, чтобы определить свойства, методы и конструкторы для автоматического связывания.

В Spring 3 поддерживаются несколько аннотаций для автоматического связывания:

- аннотация @Autowired, определяемая самим фреймворком Spring;
- аннотация @Inject из JSR-330;
- аннотация @Resource из JSR-250.

6.1 Автоматическое определение компонентов

Когда в конфигурацию Spring добавляется элемент <context: annotation-config>, он сообщает фреймворку о необходимости учитывать аннотации в компонентах при их связывании. Даже при том, что применение элемента

<context:annotation-config> способно помочь избавиться от большинства элементов <property> и <constructor-arg> в файле конфигурации, это не избавляет от необходимости объявлять компоненты с помощью элемента <bean>.

Но у фреймворка Spring есть в запасе еще одна хитрость. Элемент <context:component-scan> делает все то же, что и элемент <context: annotation-config>, плюс он настраивает фреймворк на автоматическое определение компонентов и их объявление. Это означает, что большинство (если не все) компонентов в приложении на основе Spring можно объявить и связать без использования элемента <bean>. Чтобы включить режим автоматического определения компонентов, необходимо вместо элемента <context:annotation-config> добавить элемент <context:component-scan>.

Обратимся к первому конспекту, где приводился пример с автомобилем и двигателем и применим автоматическое определение компонентов. Для этого в applicationContext.xml пропишем

```
<context:component-scan base-package="ru.firstapp"/>
```

В поле base-package укажите имя вашего пакета. Остальные элементы данного xml-файла удаляем.

Элемент <context:component-scan> заставляет фреймворк просмотреть пакет и все вложенные в него пакеты package и отыскать классы, которые можно автоматически зарегистрировать в виде компонентов в контейнере Spring. Атрибут base-package элемента <context:component-scan> определяет пакет, откуда следует начинать поиск.

Как элемент <context:component-scan> определяет, какие классы регистрировать в виде компонентов Spring?

6.2 Аннотирование компонентов для автоматического определения

По умолчанию элемент <context:component-scan> требует выполнить поиск классов, отмеченных одной из нескольких специальных аннотаций:

@Component – универсальная аннотация, указывающая, что класс является компонентом Spring;

@Controller – указывает, что класс определяет контроллер Spring MVC;

@Repository – указывает, что класс определяет репозиторий данных;

@Service – указывает, что класс определяет службу; любая пользовательская аннотация, определенная с помощью аннотации @Component.

Вернемся к практическому примеру первой лекции. Теперь пропишем аннотацию @Component для класса GasEngine.

```
@Component ("gasEngine")
public class GasEngine implements Engine {
...
}
```

Тогда для получения данного бина из контекста необходимо выполнить следующее:

```
GasEngine engine = context.getBean("gasEngine", GasEngine.class);
```

Когда фреймворк будет просматривать пакет ru.firstapp, он обнаружит, что класс GasEngine отмечен аннотацией @Component, и автоматически зарегистрирует его как компонент. По умолчанию идентификатор компонента генерируется из имени класса, где первый символ имени замещается этим же символом в нижнем регистре.

Но мы для практики решили сделать это самостоятельно и явно определили идентификатор в скобках. Мы также могли задать любое другое имя идентификатора.

6.3 Аннотация @Autowired

Вспоминаем пример из первого конспекта, ранее мы внедряли зависимость с помощью конструктора. Изменим наш пример следующим образом (зачем будет рассказано позже):

```
@Component ("carBean")
public class Car {
private GasEngine engine;

@Autowired
public void setEngine(GasEngine engine) {
      this.engine = engine;
}
```

Когда фреймворк обнаружит, что метод (или конструктор) снабжен аннотацией @Autowired, он попытается выполнить автоматическое связывание по типу (byType). Самая интересная особенность аннотации @Autowired состоит в том, что ее необязательно применять к методу записи.

Аннотацию @Autowired можно даже применить к конструктору:

```
@Autowired
```

```
public Car(GasEngine engine) {
	this.engine = engine;
}
```

При использовании с конструктором аннотация @Autowired указывает, что механизм автоматического связывания должен использовать конструктор при создании компонента, даже если в описании компонента в конфигурационном XML-файле отсутствует элемент <constructor-arg>. Кроме того, аннотировать можно свойства непосредственно и вообще избавиться от методов записи:

```
@Autowired
private GasEngine engine;
```

Как видите, ключевое слово private не является препятствием для аннотации @Autowired. Несмотря на то что свойство engine объявлено частным, оно будет доступно для автоматического связывания. Неужели для аннотации @Autowired нет никаких препятствий?

В действительности существуют несколько обстоятельств, мешающих работе аннотации @Autowired. В частности, должен существовать точно один компонент, подходящий для связывания со свойством или параметром, аннотированным с помощью @Autowired. Если имеется несколько подходящих компонентов или нет ни одного, аннотация @Autowired столкнется с определенными проблемами.

К счастью, имеется возможность помочь аннотации @Autowired в подобных ситуациях. Сначала посмотрим, как помочь аннотации @Autowired в случае, когда нет ни одного подходящего компонента.

6.3.1 Уточнение неоднозначных зависимостей. Аннотация @Qualifier

С другой стороны, проблема может быть обусловлена не отсутствием компонентов, а неоднозначностью выбора. Может так случиться, что компонентов будет слишком много (по крайней мере, два), каждый из которых одинаково пригоден для внедрения в свойство или параметр.

Например, вспомним наш автомобиль и его двигатели, в предыдущем примере мы схитрили и внедряли зависимость с конкретным классом GasEngine. В случае, когда мы осуществляли связывание через xml мы же принимали в конструктор интерфейсную переменную типа Engine.

```
public Car(Engine engine) {
	this.engine = engine;
}
```

В этом случае аннотация @Autowired (если бы мы её прописали выше примере) не сможет определить, какой из них (GasEngine или ElectricalEngine) действительно требуется внедрить. Поэтому, вместо того чтобы пытаться угадать, фреймворк инициирует исключение NoSuchBeanDefinitionException.

Чтобы помочь аннотации @Autowired выбрать требуемый компонент, можно добавить аннотацию @Qualifier . Например, чтобы гарантировать, что Spring выберет бензиновый двигатель для компонента car, даже если существуют другие компоненты, которые могут быть связаны с его свойством engineBean, можно добавить аннотацию @Qualifier, ссылающуюся на компонент с именем engineBean. Для конструктора это будет выглядеть следующим образом:

```
@Autowired
public Car(@Qualifier("gasEngine") Engine engine) {
      this.engine = engine;
}
```

Для поля:

```
@Autowired
@Qualifier("gasEngine")
private Engine engine;
```

Как показано в этом примере, аннотация @Qualifier определяет, что для связывания должен использоваться компонент с идентификатором gasEngine.

На первый взгляд может сложиться впечатление, что аннотация @Qualifier выполняет переход от автоматического связывания по типу к автоматическому связыванию по имени. В данном приме ре именно это и происходит. Но в действительности аннотация @Qualifier лишь сужает выбор компонентов, доступных для автоматического связывания. Просто так получается, что ссылка на идентификатор компонента является одним из способов сузить выбор до единственного компонента.

6.4 Внедрение значений с помощью аннотаций

Помимо внедрения ссылок на компоненты с помощью аннотаций автоматического связывания, может появиться желание использовать аннотации для внедрения простых значений. В версии Spring 3.0 появилась новая аннотация @Value , позволяющая внедрять простые значения.

Аннотация @Value проста в использовании, но обладает широкими возможностями. Ее применение заключается в том, чтобы добавить аннотацию @Value перед свойством, методом или параметром метода и передать ей строку с выражением, результат которого должен быть внедрен. Например:

```
@Value("1.6")
private double volume;
```

Здесь в числовое поле внедряется обычное значение типа double. Но строковый параметр аннотации @Value в действительности является выражением, которое может возвращать значение любого типа, благодаря чему аннотация @Value может применяться к свойствам любых типов.

Внедрение жестко определенных значений с использованием аннотации @Value само по себе малоинтересно. Если заранее известные значения можно

присваивать непосредственно в программном коде Java, почему бы тогда не отказаться от аннотации @Value и не присвоить требуемое значение свойству напрямую? В данном примере аннотация @Value выглядит как излишество.

Как оказывается, внедрение простых значений не является коньком аннотации @Value. Полная ее мощь заключается в возможности использовать выражения на языке SpEL. Язык SpEL позволяет динамически вычислять сложные выражения во время выполнения и использовать полученные значения для внедрения в свойства компонентов. Это делает аннотацию @Value мощным инструментом внедрения.

О языке SpEL мы пока говорить не будем. Пока вернемся к способу внедрения значений из файла.

Например, вместо внедрения жестко определенного значения в свойство volume попробуем задействовать внедрить значение из внешнего файла:

```
@Value("${gasEngine.volume}")
private double volume;
```

Не забывайте, в конфигурационном xml должен находиться следующий тег:

```
<context:property-placeholder location="classpath:gasEngine.properties"/>
```

6.5 Область видимости

По аналогии с xml-конфигурацией при аннотировании можно установить область видимости компонента при помощи аннотации @Scope, который может принимать свойства “singleton”, “prototype” и другие.

6.6 Управление жизненным циклом при помощи аннотаций

При рассмотрении xml-конфигурации мы пользовались тегами init-method и destroy-method, для управления жизненным циклом компонента задавая собственное поведение при инициализации и разрушении компонента.

При использовании аннотаций можно воспользоваться @PostConstruct, чтобы определить метод, который будет вызван для завершения инициализации после создания компонента и внедрения всех зависимостей или @PreDestroy, чтобы определить метод, который будет вызван перед удалением компонента из контейнера.

7 Конфигурация Spring в программном коде на Java

Хотите верьте, хотите нет, но не все разработчики являются ярыми поклонниками XML. В действительности некоторые из них яв ляются членами клуба Настоящих Мужчин, Ненавистников XML. Они очень хотели бы избавить мир от угловых скобок. Длинная история использования XML в Spring приучила некоторых противников XML использовать его.

Тем, кто принадлежит числу противников XML, версия Spring 3.0 может предложить кое-что особенное. Теперь есть возможность конфигурировать приложения на основе Spring практически без XML, исключительно в

программном коде на языке Java. И даже у тех, кто не является противником XML, может возникнуть желание попробовать выполнить конфигурирование на языке Java, потому что, как будет показано чуть ниже, этот прием обладает некоторыми возможностями, недоступными в XML.

Несмотря на то что прием конфигурирования на языке Java позволяет описать конфигурацию приложения практически без использования XML, тем не менее некоторый объем XML-кода все же необходим, чтобы подготовить к использованию конфигурацию на языке Java:

```
<context:component-scan
        base-package="ru.sfu" />
```

Когда мы начинали знакомиться с конфигурациями Spring в формате XML, у нас был фрагмент с элементом <beans> из пространства имен beans, играющим роль корневого элемента. Его эквивалентом на языке Java является класс, отмеченный аннотацией @Configuration. Например:

```
@Configuration
public class SpringConfig {

}
```

Аннотация @Configuration подсказывает фреймворку Spring, что данный класс содержит одно или более определений компонентов. Объявления компонентов – это обычные методы, отмеченные аннотацией @Bean. Посмотрим, как использовать аннотацию @Bean для связывания компонентов в описании конфигурации на языке Java.

Для объявления компонента типа GasEngine с идентификатором gasEngine ранее мы использовали элемент <bean>. В конфигурации на языке Java компонент gasEngine можно определить как метод с аннотацией @Bean:

```
@Bean
public Engine gasEngine() {
      return new GasEngine();
}
```

Этот простой метод в конфигурации на языке Java является эквивалентом элемента <bean>, созданного ранее. Аннотация @Bean сообщает фреймворку Spring, что данный метод вернет объект, который должен быть зарегистрирован в контексте приложения Spring как компонент. Компонент получит идентификатор, совпадающий с именем метода. Все операции, выполняемые внутри метода, в конечном итоге должны создавать компонент.

В данном случае объявление компонента выглядит очень просто. Метод просто создает и возвращает экземпляр класса GasEngine. Этот объект будет зарегистрирован фреймворком Spring в контексте приложения с идентификатором gasEngine.

Хотя этот метод объявления компонента в значительной степени является эквивалентом объявления в формате XML, тем не менее он иллюстрирует одно из самых больших преимуществ реализации конфигурации на языке Java перед оформлением конфигурации в формате XML. В XML-версии оба значения, тип и идентификатор компонента, определяются строковыми атрибутами. Недостатком строковых идентификаторов является невозможность их проверки на этапе компиляции. Можно переименовать класс GasEngine в программном коде и забыть внести соответствующие изменения в конфигурацию в формате XML.

В конфигурации на языке Java не используются строковые атрибуты. Идентификатор компонента и его тип являются частью сигнатуры метода. Фактическое создание компонента выполняется в теле метода. Поскольку в данном случае конфигурация описывается программным кодом, появляется дополнительное преимущество, обусловленное дополнительными проверками, выполняемыми на этапе компиляции и гарантирующими, что компонент будет иметь действительный тип, а его идентификатор будет уникальным.

7.1 Внедрение зависимостей в конфигурации на языке Java

Если объявление компонента в конфигурации на языке Java – не более чем метод, возвращающий экземпляр класса, тогда как в этом случае реализовать внедрение зависимостей? Фактически, если следовать идиомам программирования на языке Java, внедрение реализуется очень просто.

Рассмотрим сначала, как реализовать внедрение значений в компонент. Ранее было показано, как с помощью элемента <constructor-arg> в конфигурации в формате XML создать бензиновый двигатель (компонент типа GasEngine), с объемом двигателя 3.0. В конфигурации на языке Java достаточно просто передать требуемое число конструктору:

```
@Bean
public Engine gasEngine() {
	return new GasEngine(3.0);
}
```

Как видите, конфигурация на языке Java выглядит вполне естест венно и позволяет определять компоненты любыми доступными способами. Внедрение через метод записи на языке Java выглядит не менее естественно:

```
@Bean
public Engine gasEngine() {
	GasEngine engine = new GasEngine();
	engine.setVolume(3.0);
	return engine;
}
```

Внедрение простых значений реализуется очень просто. А как обстоит дело со ссылками на другие компоненты? Все так же просто.

```
@Bean
public Engine gasEngine() {
	return new GasEngine(3.0);
}
```

```
@Bean
private Car car{
	return new Car(gasEngine());
}
```

Внедрение другого компонента заключается в использовании ссылки на метод определения внедряемого компонента. Но эта простота кажущаяся. В действительности все немного сложнее.

В конфигурации Spring на языке Java ссылка на компонент через метод с его объявлением – это не то же самое, что вызов метода. Если бы это было так, то всякий раз, вызывая метод gasEngine(), мы получали бы новый экземпляр компонента. Фреймворк Spring действует немного тоньше.

Пометив методgasEngine() аннотацией @Bean, мы сообщаем фреймворку, что этот метод определяет компонент для регистрации в контексте приложения. Поэтому при каждом обращении к этому методу внутри другого метода объявления компонента Spring будет перехватывать вызов метода и пытаться отыскать компонент в контексте приложения, не позволяя этому методу создать новый экземпляр.

7.2 Внедрение из файла свойств при помощи аннотаций

@PropertySource("classpath:gasEngine.properties")

Аннотация @PropertySource служит для того чтобы указать Spring-конфигурации путь к файлу из которого будет производиться чтение свойств.

8 AnnotationConfigApplicationContext

Ранее для конфигурации Spring-приложения мы использовали xml-файл конфигурации и класс ClassPathApplicationContext, выглядело это следующим образом:

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
```

Теперь, для работы через файл конфигурации на языке Java контекст приложения необходимо конфигурировать следующим образом:

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);
```

Ранее в xml-файле мы указывали где Spring должен искать компоненты следующим образом:

```
<context:component-scan
	base-package="ru.sfu" />
```

Теперь нам необходимо прописать аннотацию @ComponentScan("ru.sfu") в нашем SpringConfig и работать с компонентами в привычном режиме.