1.8 Context features

На данный момент у нас создано небольшое приложение, где статические данные, такие как ID и имя клиента, вынесены в spring.xml.

Не всегда может быть удобно делать именно так - spring.xml имеет свойство расти, и что-то находить в нем становится все сложнее. Также одинаковую информацию может понадобиться подключать в нескольких местах, а потом, возможно, часть изменять, в зависимости от переменных окружения.

Очень часто такого рода статическую информацию выносят в отдельный property-файл, а потом его подключают к spring.xml и по ключам обращаются к нужным свойствам. Особенно это полезно, когда конфигурация Spring у вас задается не в XML файле, а с помощью аннотаций, которые определяются в коде.

Как же подгрузить отдельный property-файл?

Давайте создадим сначала такой property файл. Назовем его **client.properties** и поместим рядом со spring.xml. В нем пропишем свойства клиента:

id=1
name=John Smith
greeting=Hello there!

Теперь сделаем так, чтобы Spring брал свойства из этого файла и инжектил в объект клиента.

Для начала подключим property файл в spring.xml, испоьзуя специальный утилитарный бин **PropertyPlaceholderConfigurer**

У этого бина есть property <property name="**locations**">, которое принимает список всех файлов, которые вы хотите, чтобы он загрузил. Имена файлов можно задать через абсолютный или относительный путь, либо с помощью директивы classpath, например <value>classpath:client.properties</value>, которая укажет, что нужно искать в classpath.

Также у данного бина есть еще парочка интересных свойств:

```
1. <property name="ignoreResourceNotFound" value="true" />
```

По умолчанию – это значение = false, т.е. если файл свойств не будет найден, то выскочит ексепшн. Если ресурсы у вас опциональны, тогда поставьте значение в true.

2. Второе свойство это

```
<property name="systemPropertiesMode" value=”override”>
```

По умолчанию его значение – fallback. Если ключ не будет найден в property-файлах, то Spring попробует найти его в системных свойствах и переменных окружения. Это может быть полезным, чтобы оставить возможность переопределять свойства объектов через SystemProperty, которые передаются при запуске приложения, при этом задав их значения по умолчанию в property-файле. А значение “never” говорит само за себя – системные свойства не рассматриваются.

Давайте создадим бин PropertyPlaceholderConfigurer в нашем spring.xml.

Полный текст spring.xml может выглядеть так:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="
                http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                http://www.springframework.org/schema/util
                http://www.springframework.org/schema/util/spring-util-3.2.xsd">
```

```
        <import resource="loggers.xml" />

        <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
              <property name="locations">
                            <list>
                                    <value>classpath:client.properties</value>
                            </list>
              </property>
              <property name="ignoreResourceNotFound" value="true" />
              <property name="systemPropertiesMode">
                            <util:constant

static-field="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer.SYSTEM_PROPERTIES_MODE_OVERRIDE" />
              </property>
        </bean>

        <bean id="client" class="ua.epam.spring.core.beans.Client"
              p:greeting="${greeting}">
              <constructor-arg index="0" value="${id}" />
              <constructor-arg value="${name}" />
        </bean>

        <util:map id="loggerMap">
              <entry key="INFO" value-ref="consoleEventLogger" />
              <entry key="ERROR" value-ref="combinedEventLogger" />
        </util:map>

        <bean id="app" class="ua.epam.spring.core.App">
              <constructor-arg ref="client" />
              <constructor-arg ref="cacheFileEventLogger" />
              <constructor-arg ref="loggerMap"/>
        </bean>

        <bean id="event" class="ua.epam.spring.core.beans.Event" scope="prototype">
              <constructor-arg>
                            <bean class="java.util.Date" />
              </constructor-arg>
              <constructor-arg ref="dateFormat" />
        </bean>

        <bean id="dateFormat" class="java.text.DateFormat" factory-method="getDateTimeInstance" />

        <bean id="monitor" class="ua.epam.spring.core.util.Monitor"/>
        <bean id="awared" class="ua.epam.spring.core.util.AwareBean" init-method="init"/>
    </beans>
```

Тепепрь давайте проинжектим property из файла в наш бин Client.

Делается это следующим образом: вы просто пишете `${greeting}`

указывая ключ, который находится в property файле.

Таким образом, когда Spring создаст бин PropertyPlaceholderConfigurer, тот прочитает все файлы и все значение сделает доступными для контекста. Дальше вы сможете указать их как value аргумента конструктора либо property бина.

Какие еще интересные штучки нам может предложить Spring?

При использовании XML конфигурации мы можем подключить дополнительные схемы и namespaceы, что позволит расширить функции контекста по определению бинов.

Давайте подключим к нашему проекту namespace util: для этого пропишем `xmlns:util="http://www.springframework.org/schema/util"`

В нем есть много дополнительных тегов, которые позволяют расширить возможности контекста.

Какие возможности нам предоставляет namespace утил?

Напрмиер, использование константы, определенной в каком-то классе в качестве свойства или аргумента конструктора:

<property name="type">
  <**util:constant** static-field="java.lang.Math.PI">
</property>

Это очень удобно, чтобы не дублировать информацию, которая хранится в публичном статическом файнал поле класса.

Также с помощью утил namespace можно определять списки, мапы и наборы свойств как отдельные бины.

Вы уже знаете, как в Spring определить список в свойстве бина, и, например в аргументе конструктора, но это не позволяло нам переиспользовать список несколько раз. С помощью специальных тегов util:properties или util:map, util:list вы можете создать отдельный бин соответствующего класса и указать ему значение точно так же, как мы это делали в прошлой теме.

А что касается util:properties, то можно также указать и имя файла свойств и получить все property загруженными в отдельный объект класса java.util.Properties. А для map можно также указать класс, например, TreeMap, и получить отсортированные значения.

Все эти созданные бины будут находиться в контейнере и вы сможете к ним обращаться либо из кода, либа инжектить в другие бины.

Чтобы данный код сохранился у вас и лучше запомнился, давайте в нашей программе вынесем мапу логгеров в отдельный бин, определенный с помощью тега <util:map id="…"> и проинжектим его в класс App.

Для сокращенной записи свойств бинов, а также агументов конструктора в Spring есть два специальных namespace:

p – for properties

c – for constructor arguments

Подключив соответствующие схемы в теге <beans>, вы сможете указать свойства бина не отдельными тегами, а с помощью атрибутов тега <bean>.

```
<bean id="client" class="…" p:greeting="Hello" />
```

Ссылка на объект

```
<bean id="client" class="…" c:client-ref="client" />
```

Существует такая функция Spring контекста, как автоматическое связывание. Она хороша и опасна одновременно. Добавив атрибут **autowire**="…" к тегу <bean>, можно указать Springy, чтобы он автоматически нашел подходящие бины в контексте и проинжектил их.

Существует 3 типа связывания:

· byName –по имени Property name

· byType – по типу Bean class in property

· constructor – по типу через конструктор Bean class in constructor

Если вы указали по имени, то Spring будет искать в контексте бин, имя которого соответствует имени в property классе.

Связывание по типу указывает Springy искать бин, класс которого подходит для этого property.

Связывание через конструктор – это то же самое, что и связывание по типу, толко для внедрения используется конструктор, а не property.

Получается, что написав атрибут autowire вы можете не указывать кучу тегов property или констрактор арг – Spring сам найдет и подставит нужные бины.

Конечно, если несоклько бинов будут подходить для внедрения, то Spring запутается, поэтому с autowire нужно быть очень осторожным и подходит он только для внедрение бинов с уникальным именем или классом. К тому же не рекомендуется использовать autowire для внедрения одного или двух значений, когда остальные значения вы непосредственно определяете – это очень запутывает и делает конфигурацию непонятной. Если уже испоьзовать autowire, то везде и для всех бинов, хоть это и не всегда возможно.

У autowire также есть некоторые ограничения, наприме, нельзя использовать простые типы, т.к. сложно понять – вы хотите внедрить бин или какую-то константу, или значение из property файлов. Также явное указание референса на какой-то бин имеет больший приоритет перед автоматическим связыванием.

В дополнение к XML конфигурации, autowire также работает и с аннотациями, но об этом в следующей теме.

До сих классы, которые мы определяли как бины в контексте, ничего не знали о Spring.

Иногда может возникнуть ситуация, когда классу необходимо знать, в каком контексте он находится, или какое у него имя в контексте. Для таких случаев Spring предоставляет набор так называемых aware-интерфейсов:

- **ApplicationContextAware**
- **ApplicationEventPublisherAware**
- **BeanFactoryAware**
- **BeanNameAware**
- **ResourceLoaderAware**
- **ServletContextAware**
- **…**

Реализовав их, вы можете получить различную информацию от самого Springa во время инициализации бина.

Это может быть, например, ссылка на контекст, имя бина, или сервлет контекст для веб приложения.

Конечно, такое нужно не часто, и вы скорее всего будете создавать ваше приложение независимым от Spring. Но в то же время, если такая ситуация возникнет, например, в вашем коде тестирования, то теперь вы знаете, как это реализовать.

Также если вы пишете код, в котором хотите реагировать на события контекста, такие как создание, завершение или обновление, то Spring предоставляет специальный интерфейс - **ApplicationListener.**

Реализовав его, вы будете получать события, когда они будут возникать в контексте. Это еще одна утилитарная функция, которая может пригодиться.

1.9 Аннотации.

При использовании Спринг вы можете задавать конфигурацию контекста используя XML файл. Так же это возможно сделать с помощью аннотаций. От ХМЛ можно отказаться вовсе и с помощью аннотаций полностью конфигурировать контекст.

Но сначала разберемся, как использовать аннотации для внедрения бинов, которые сами по себе могут быть определены в ХМЛ файле.

Для этого нужно указать Спринг, чтобы он включил конфигурацию по аннотациям. Нужно добавить схему и namespace-контекст, а потом использовать специальный тег **<context:annotation-config/>.**

Пример:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="
                  http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                  http://www.springframework.org/schema/context
                  http://www.springframework.org/schema/context/spring-context-3.2.xsd">

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

Написав данный тег, мы говорим Спрингу, что в наших классах есть аннотации. Т.е. когда он будет создавать бины, он будет искать в них определенные аннотации и выполнять действия согласно им.

Какие есть основные аннотации?

Самая главная аннотация Спринга – это **@Autowired.** С помощью нее вы указываете Спрингу, значение какого свойства нужно вставить из контекста.

Данная аннотация является именно Спринг-аннотацией.

Однако фреймфорк поддерживает также использование стандартных аннотаций, которые были представлены в соответствующих JSR, а потом попали в JDK. Использование их позволяет быть полностью независимым от Спринг, а также переиспользовать классы в других контейнерах, напр. JBoss Sim.

Такой аннотацией, аналогичной **@Autowired,** является **@Inject.** Разница лишь в том, что спринговая @Autowired содержит также атрибут @Autowired(required=false). Если его установить в false, то при отсутствиии бина для внедрения, будет установлено значение null, а если required=true, то получим эксепшн.

Интересно, что с помощью аннотации @Autowired можно также внедрять не только бины, но и внутренние объекты Спринг, такие как ApplicationContext, EventPublisher etc. То есть не нужно будет реализовывать aware-интерфейсы, чтобы получить эти объекты, - их теперь можно просто внедрить с помощью аннотаций.

Аннотацию @Autowired или @Inject можно указать в разных местах:

· над полем класса (даже если оно private)

· над сеттером

· над конструктором

Какой способ выбрать - решать вам. Скажу только, что внедряя зависимости через поля класса при отсутствии сеттеров, сильно усложнит юнит-тестирование такого класса, ведь простого способа вставить mock уже не будет. Придется либо поднимать отдельный тестовый контекст, либо как-то через рефлексию.

И еще такой момент. Когда вы добавляете аннотацию @Autowired, то Спринг будет инжектить зависимости по типу. Это не всегда возможно, например, в нашем приложении класс App содержит defaultLogger, под тип которого подходит несколько бинов. В таком случае придется также указывать

имя бина, которого мы хотим внедрить. Для этого используется либо спринговая аннотация @Qualifier(“bean name”) совместно с @Autowired, либо единственная JDK аннотация @Resource(name=”bean_name”) . Они очень похожи между собой и по виду отличаются только способом указания имени.

Но есть еще одно отличие – аннотацию @Resource можно использовать только над полями класса либо сеттерами, а вот @Autowired можно использовать и в других местах. Соответственно, аннотацию @Qualifier можно указать перед аргументами конструктора. Сам Спринг рекомендует использовать аннотацию @Resource, если необходимо внедрять по имени, через сеттер или поле класса.

В Спринг есть еще парочка полезных аннотаций:

· **@PostConstruct**

· **@PreDestroy**

Они выполняют ту же функцию, что и методы init() и destroy(). Ксати, при использовании этих аннотаций, ими можно пометить несколько методов и все они будут вызваны на этапе инициализации или уничтожения бина.

Если необходимо внедрять простые значения, например как имя файла, который мы передаем в конструкторе FileEventLogger, или имя клиента, то для этого используется аннотация @Value. С помощью нее можно также передавать значение из файлов свойств.

Кроме использования аннотаций для внедрения зависимостей, их можно применять и для определения бинов. Дл этого в Спринг есть несколько аннотаций:

· **@Component**

· **@Service**

· **@Repository**

· **@Controller**

Чем они отличаются друг от друга? Ничем! Спринг их обрабатывает одинаково. Основная аннотация здесь – это @Component, а остальные созданы для вашего удобства, чтобы можно было пометить специфические классы отдельно и выполнять с ними дополнительную логику. Например, такую логику содержит Spring MVC, который классы, помеченные аннотацией @Controller, считает контроллерами веб-приложения. Также используя AOP, вы можете вызовы методов класса, помеченного аннотацией @Service, предварять вызовом аспекта безопасности. На самом деле, вы даже можете создать свои собственные аннотации, пометить их аннотацие @Component, а потом использовать их в приложении - в таком случае метаданные принесут еще больше пользы.

Для того, чтобы класс, помеченный одной из таких аннотаций, попал в контейнер, необходимо сказать Спрингу, чтобы он просканировал какой-то пакет и их там нашел. Для этого используется тег **<context:component-scan base-package=”…”>.** При создании контекста Спринг просканирует пакет, найдет все классы с аннотациями @Component, @Service и др. и создаст их бины, внедрив необходимые зависимости.

Чтобы указать имя бина, вы прописываете его в аннотации @Component или другой, поддерживаемой Спринг.

Чтобы задать scope бина, вы используете аннотацию @Scope, где в текстовом виде задаете необходимое значение(singleton, prototype, request, session, global-session).

И на последок, давайте посмотрим, как задать конфигурацию Спринг с помощью джава кода и аннотаций без использования XML.

Для этого необходимо создать отдельный класс и пометить его аннотацией @Configuration. В этом классе могут быть публичные методы, помеченные аннотацией @Bean, которые будут возвращать объекты. Получается, что бины вы можете сами конструировать и делать их доступными в контексте.

Конечно, при использовании аннотации @Component это необязательно. Но такие бины, как PropertyPlaceholderConfigurer придется описывать тут.

Если вы самостоятельно создаете бины, то получить зависимости можно с помощью аннотации @Autowired.

Чтобы использовать такую конфигурацию, в основном приложении необходимо содать AnnotationConfigApplicationContext, передав ему на вход класс с аннотацией @Configuration. Таких классов можно передавать несколько по аналогии с ХМЛ файлами, тогда все их конфигурации объединятся в один контекст. Если вы не желаете передавать классы конфигурации в конструктор контекста, тогда можно использовать метод ctx.register(AppConfig.class); Этот более гибкий подход позволяет программно создавать необходимую конфигурацию в зависимости от определенных условий. После регистрации необходимых классов нужно вызвать ctx.refresh(); чтобы обновить контекст.

Если вы объявили свой класс с помощью аннотаций @Component и других, то просканировать пакет и загрузить бины в контейнер можно с помощью вызова метода ctx.scan(“ua.epam.spring.core.beans”) и вызвать метод ctx.refresh();

Используя Java-конфигурацию вы также можете задавать scope для бинов, указывать их алиасы, импортировать в другие контексты. В принципе, все, что можно сделать в spring.xml, можно сделать и в джава-конфигурации. Плюсы и минусы есть в обоих подходах. Часто бывает так, что оба подхода объединяют, когда часть конфигурации объявляют в ХМЛ, а autowired делают с помощью аннотаций.

В любом случае, постарайтесь сделать так, чтобы описание вашего контекста, а также логика внедрения зависимостей была легко доступна и понятна человеку со стороны.

В качестве практики использования аннотаций джава конфигрурации, добавьте аннотации в наше тестовое приложение и определите его контекст с помощью джава кода.