Видео 1.

Framework vs. Library

Dependency Injection

Inversion of Control

Core container

* Beans
* Core
* Context
* Expression

AOP

* AOP
* Aspects

Instrumentation

* Instrument – Spring Boot??

Data access and integration

* JDBC
* JMS
* ORM
* Transaction

Web and Remoting – MVC тут

* Web
* Servlet
* Struts

Test

* Testing

Модули:

spring-context

spring-context-support

spring-tx - транзакции

spring-jdbc

Вариант приложения 1 – Client – имеет id и name, ConsoleEventLogger – имеет метод logEvent, который выводит сообщение на консоль, App – создает Client и ConsoleEventLogger внутри себя и пользуется их методами.

Проблемы:

1. Сложно редактировать – нужно перекомпилировать при изменении данных
2. Плохое масштабирование – только один логгер, который жестко зашит в приложение. Для изменения нужно перекомпилировать.
3. Сложно тестировать – при тестировании App.eventLogger будет тестироваться и ConsoleEventLogger.

Причина всех проблем – жесткое связывание (tight coupling).

Видео 2.

Решение проблем:

1. Всю статическую информацию лучше вынести во внешние файлы.
2. Сделать интерфейсную развязку - реализация должна зависить от интерфейсов, а не от классов. Но интерфейсы не нужны для классов, которые только хранят информацию (как Client).
3. Внедрение зависимостей – вместо создания экземпляров вручную, просим это сделать у Spring.

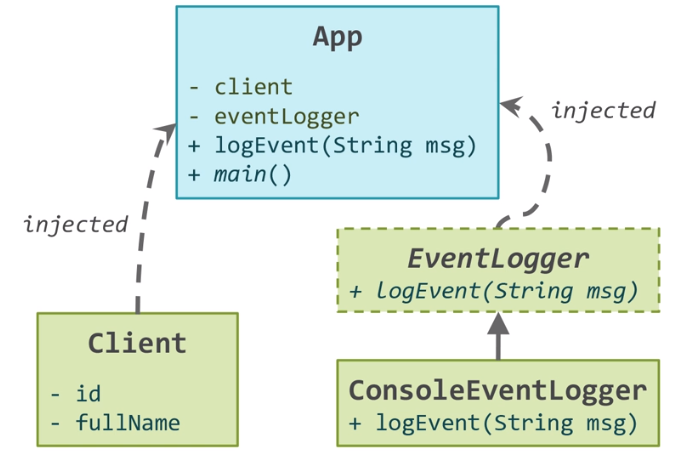
SVN???

Внедрение зависимостей решает проблемы:

1. ~~Плохое масштабирование~~ – можно подсунуть логгер с другой реализацией, рассчитанной на масштабирование.
2. ~~Сложно тестировать~~ – внедряем тестовую реализацию вместо ConsoleEventLogger и можно протестировать App.eventLogger в изоляции

Дорабока приложения

Вариант приложения 2 – ConsoleEventLogger реализует интерфейс EventLogger, App не создает Client и ConsoleEventLogger, а принимает их в качестве параметров конструктора.

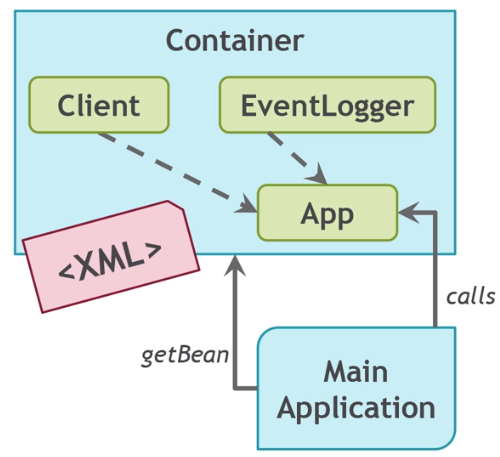


За создание классов и их связывание отвечает spring-контейнер. Мы обращаемся к нему для получения необходимой информации. Экземпляры классов – бины.

Если нужен бин, то возможны 2 варианта:

1. Обращение к контейнеру для получения
2. Навигация по графу объектов (получаем 1й бин, через него следующий и т.д.)

Как узнать какие бины создавать? XML-файл (или аннотации, или java, или groovy).



src/main/resources/Beans.xml

*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="  
 http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>  
  
 <**bean id="client" class="com.epam.spring.core.beans.Client"**/>  
  
 <**bean id="eventLogger" class="com.epam.spring.core.loggers.ConsoleEventLogger"**/>  
  
 <**bean id="app" class="ua.epam.spring.core.com.epam.spring.core.App"**/>  
</**beans**>

Видео 3

2 способа именовать бины (можно вместе):

1. id – должен быть уникальным
2. name – можно указать несколько имен для одного бина. Полезно, когда бин реализует несколько интерфейсов и к нему удобно обращаться по разному.

Если указать несколько имен, то первый будет основным, а остальные будут алиасами. Можно указать алиас вручную:

<alias name=”aaa” alias=”bbb”>

Полезно, когда бины берутся из сторонних библиотек.

Имена нужны для инжектинга. Можно использовать и просто имена интерфейсов, но они часто реализуются несколькими классами, что приводит к ошибкам.

Внедрение зависимостей:

1. Через конструкторы
2. Через геттеры/сеттеры

Через конструкторы:

<**bean id="client" class="com.epam.spring.core.beans.Client"**>  
 <**constructor-arg value="1"**/>  
 <**constructor-arg value="John Smith"**/>  
</**bean**>

Аргументы одного типа инжектятся в том порядке, в котором перечислены.

Можно указать порядок вручную.

<**bean id="client" class="com.epam.spring.core.beans.Client"**>  
 <**constructor-arg index="0" value="1"**/>  
 <**constructor-arg index="1" value="John Smith"**/>  
</**bean**>

Если типы разные, то порядок не важен – спринг поймет.

Но, например тип 1 можно указать и для String и для Integer. Как Спрингу разобраться? Можно указать тип:

<**constructor-arg type="java.lang.Integer" value="1"**/>

<**constructor-arg type="java.lang.String" value="1"**/>

Можно указать имена переменных для параметров:

<**constructor-arg name="arg1" value="1"**/>

<**constructor-arg name="arg2" value="Dude"**/>

Но это хак, который работает через рефлекшн только когда включены дебаг-символы (???) и не используется в проде.

Инжектинг бинов:

<**bean id="app" class="com.epam.spring.core.App"**>  
 <**constructor-arg ref="client"**/>  
 <**constructor-arg ref="eventLogger"**/>  
</**bean**>

Если не указать аргументы, то спринг вызывает конструктор по умолчанию и он должен быть объявлен (если нужно).

Модфикатор доступа в конструкторе м.б. любым – спринг вызовет его даже если он private.

Описание контейнера Спринг – в xml (определены бины и зависимости).

1. Создать контейнер
2. Запустить приложение
3. Спринг создаст бины и проинжектит зависимости

В spring-framework 2 типа контейнеров:

1. BeanFactory – простейший контейнер.
   1. Поднимается
   2. Создает бины
   3. Инжектит их
2. ApplicationContext -
   1. Поднимается
   2. Создает бины
   3. Инжектит их
   4. Предоставляет сервисы для управления бинами

Создание контекста:

ApplicationContext ctx =

**new** ClassPathXmlApplicationContext(**"spring.xml"**);

Получение бина:

1. По имени – класс будет неизвестен, придется делать каст (почему? Все бины типа Object?)
2. По классу – не нужно кастить, используются дженерики
3. По классу и имени – как и у получения по классу
4. По аннотации, которая есть у класса

Можно указать несколько xml при создании контекста. Спринг объединит их.

???контекст == контейнер???

Дорабока приложения

Вариант приложения 3 – теперь в App создается контекст, а создание всех объектов указано в xml конфигурации.

Видео 4

Дорабока приложения

Вариант приложения 4 – появляется класс Event c 4 полями:

- id – генерируется через Random

- msg –через геттеры-сеттер

- date – через конструктор

- df – DateFormat – через конструктор, используется в toString().

EventLogger теперь принимает Event.

Event будет создаваться через Spring. Добавить в Event DateFormat, который принимается консту

Bean Scope / цикл жизни объекта в конейнере:

- scope по умолчанию – singleton

- scope задается в конфигурации, например так:

<**bean id="eventLogger" class="com.epam.spring.core.loggers.ConsoleEventLogger" scope="singleton"**/>

Если scope задан как синглтон, то при вызове getBean() будет возвращаться один и тот же бин.

- prototype – scope, при котором бин будет заново создаваться каждый раз при вызове getBean().

Другие scope:

* request – бин живет только на время запроса;
* session – бин живет пока есть сессия;
* global session – живет, пока есть глобальная http сессия.

Event требует объект типа Date как параметр конструктора. Можно создать его как простой bean и проинжектить. Но можно использовать внутренний бин:

<**bean id="event" class="com.epam.spring.core.events.Event" scope="prototype"**>  
 <**constructor-arg**>  
 <**bean class="java.util.Date"**/>  
 </**constructor-arg**>  
</**bean**>

Внутреннему бину можно назначить name и id, но он виден только там, где он опеределен и достучаться до него нельзя.

Все внутренние бины создаются в момент создания внешнего бина.

DateFormat – абстрактный класс для форматирования даты. У него нет конструктора, но есть статический метод, возвращающий объект этого класса. Для создания бина можно использовать factory-method:

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

Если factory-method требует аргументы, то указываем их через constructor-args.

Видео 5

Дорабока приложения

Появляется новый логгер, который будет писать лог в файл. FileEventLogger (реализует интерфейс EventLogger). Имя файла в который будет записываться событие - полем, значение – через конструктор, создание экземляра – через Spring.

Если файл недоступен, то логгер вылетит с ошибкой при первом появлении эвента. Лучше найти такую ситуацию заранее.

Bean Initializers (правда так называется?) – работают после создания бина, но до его первого использования. Нужны для подготовки бина к работе. Для вызова такого метода нужно написать в xml:

<**bean id="eventLogger" class="com.epam.spring.core.loggers.FileEventLogger" init-method="init"**>  
 <**constructor-arg value="C:\Users\Sergei\_Morozov\Documents\Projects\spring-weaver\logfile.txt"**/>  
</**bean**>

Init-методы:

1. Не могут принимать аргументы
2. Могут иметь любой модификатор доступа (спрингу плевать)
3. Может возвращать что угодно (никто не прочитает) или кидать Exception

Bean Lifecycle (so far):

1. Создаются все зависимые бины (чтобы проинжектить)
2. Вызывается конструктор бина (с параметрами, если есть)
3. Устанавливаются properties (через сеттеры) (как именно??)
4. Вызывается init метод

Дорабока приложения

CacheFileLogger – записывает значение не сразу. Cache – лист eventов, cacheSize – размер листа (через конструктор).

Проблема – если по окончанию работы приложения в кеше остались события, то они не будут записаны. Нужен какой-то деструктор. Для этого есть destroy-method.

<**bean id="eventLogger" class="com.epam.spring.core.loggers.CacheFileEventLogger" init-method="init" destroy-method="destroy"**>  
 <**constructor-arg value="C:\Users\Sergei\_Morozov\Documents\Projects\spring-weaver\logfile.txt"**/>  
 <**constructor-arg value="2"**/>  
</**bean**>

Для destroy метода применяются те же ограничения, что и для init, но вызывется destroy метод только при правильном закрытии контекста.

ConfigurableApplicationContext *ctx* = **new** ClassPathXmlApplicationContext(**"spring.xml"**);

*ctx*.close();

У ApplicationContext нет метода close, он есть у ConfigurableApplicationContext

Ещё контексты???

ConfigurableApplicationContext так же содержит ShutdownHook – метод, для регистрации момента завершения приложения, который вызовется, если приложение завершится средствами java

Видео 6

CacheEventLogger унаследовался от FileEventLogger которому нужен init метод. Для CacheEventLoggerа их тоже придется указать.

Но вместо этого можно использовать **наследование определений** спринга. Описывается бин-родитель, а у потомка просто указывается, кто является его родителем. Тогда нужно будет указывать только те аргументы и методы, которых нет у родителя.

<**bean id="fileLogger"   
 class="com.epam.spring.core.loggers.FileEventLogger"   
 init-method="init"**>  
 <**constructor-arg value="C:\Users\Sergei\_Morozov\Documents\Projects\spring-weaver\logfile.txt"**/>  
</**bean**>  
  
<**bean id="cacheFileLogger"   
 class="com.epam.spring.core.loggers.CacheFileEventLogger"   
 destroy-method="destroy"   
 parent="fileLogger"**>  
 <**constructor-arg value="2"**/>  
</**bean**>

В качестве парента может выступать и абстрактный класс. Если указать параметр abstract=”true”, то спринг не создаст такой бин, но его можно будет использовать для наследования.

Наследование определений не означает, что классы должны быть связаны отношением наследования. Если у двух классов аргументы конструктора, например, принимают одни и те же типы, то можно указать аргументы у одного бина, а для других просто указать его парентом, хотя у этих классов ничего общего.

**Зависимость бинов**. Можно определить, чтобы бин создавался только после того, как создался какой-то другой бин (например, бин мониторинга). Для этого служит параметр depends-on. При этом бины могут быть вообще не связаны между собой.

depends-on=”bean-name1, bean-name2”

**Ленивая инициализация.** По-умолчанию бины создаются сразу, когда читается контекст или когда спринг понимает, что нужно проинжектить один бин в другой. Во втором случае, при указании ref с этим ничего сделать нельзя – нужно чтобы бин был создан до его инжектинга. Но в первом случае, можно указать lazy-init=”true” и тогда бин не будет создан и его init не будет вызван, пока не будет сделан get-bean. Так же можно в теге beans указать

<beans default-lazy-init=”true”>

</beans>

Тогда все бины будут с поздней инициализацией.

**Несколько xml конфигураций.** Можно разделить xml по смыслу. Например, один для работы с БД, другой для бинов с lazy и т.п. Чтобы спинг собрал все xml в один большой контекст, нужно указать:

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

**Родительский контекст.** Возможна ситуация, когда нужно, чтобы было несколько контекстов – один с базовой логикой и несколько с конкретным дополнительным функционалом, который использует и родительские бины.

ApplicationContext parent = new ClassPathXmlApplicationContext(xmls);

ApplicationContext child = new ClassPathXmlApplicationContext(xmls, parent);

Видео 7.

**Property injection**. Если в классе бина есть сеттер, то можно отправлять туда значения указывая проперти для бинов. Для простых величин значение указывается просто через равно, для внедрения бинов используется ref, так же, как для constructor-arg

<**bean id="client" class="com.epam.spring.core.beans.Client"**>  
 ...  
 <**property name="greeting" value="Hello there!"**/>  
</**bean**>

Сначала вызывается конструктор и лишь когда бин уже создан, значения присваиваются пропертям. Поэтому можно присвоить значение полям через конструктор, а потом поменять его, через проперти.

Вызов инит-метода производится после property injection.

**Дорабока приложения**

Добавить строку greeting в Client и подставлять туда значение через проперти.

Создать новый enum EventType, с 2 полями – INFO и ERROR. В logEvent будет отправляться не только сообщение, но и тип. Если сообщение отсутствует, то App должно вызывать CacheFileEventLogger. Если INFO – ConsoleEventLogger, а если ERROR, то и в консоль и в файл и везде, куда можно. Использовать все логгеры нужно через CombinedEventLogger – новый класс, которому через конструктор передается коллекция логгеров, а в методе LogEvent он вызывает все логгеры в цикле и делает в них logEvent.

**Collection Injection.** Списки или массивы инжектятся с помощью тега list. Ещё есть тег set.

<**bean id="combinedEventLogger" class="com.epam.spring.core.loggers.CombinedEventLogger"**>  
 <**constructor-arg**>  
 <**list**>  
 <**ref bean="consoleLogger"**/>  
 <**ref bean="cacheFileLogger"**/>  
 </**list**>  
 </**constructor-arg**>  
</**bean**>

Внутри тега лист можно писать

* <value=”…”/> - для простых значений
* <ref bean=”…”/> - ссылки на другие бины
* <bean class=”...”… - для внутренних бинов

Ещё есть тег map. Он подойдет, чтобы передать в App логгеры для разных событий:

ключ может быть как значением, так и бином.

велью может быть значением, бином, внутренним бином.

<**bean id="app" class="com.epam.spring.core.App"**>  
 <**constructor-arg ref="client"**/>  
 <**constructor-arg ref="cacheFileLogger"**/>  
 <**constructor-arg**>  
 <**map**>  
 <**entry key="INFO" value-ref="consoleEventLogger"**/>  
 <**entry key="ERROR" value-ref="combinedEventLogger"**/>  
 </**map**>  
 </**constructor-arg**>  
</**bean**>

В примере в качестве ключа используется enum. Для него в спринге можно не писать имя класса.

Кроме list, set и map ещё есть props, который реализует класс java.util.properties. Он как map, но и ключи и значения являются строками. В тегах так же пишется key и value.

<props>

<prop key=”” />

<prop key=”” />

</props>

Если нужно что-то установить в null, то есть специальный тег <null/>

Видео 8.

Часто статическую информацию удобно вынести в отдельный property-файл. Чтобы его использовать нужно

1. Создать проперти файл
2. Подключить его в контекст
3. Проинжектить проперти в бин

Для подключения нужно создать специальный бин:

<**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" value="override"**/>  
</**bean**>

???!!!Такой способ не работает!!! Спринг жалуется, что systemPropertiesMode не int!!!??

**locations** – указываем имена файлов для загрузки. Можно указывать абсолютный или относительный пути, либо использовать директиву classpath.

**ignoreResourceNotFound** – по умолчанию false, т.е. если значение не будет найдено, то выйдет эксепшн.

**systemPropertiesMode** – по умолчанию fallback, т.е. если проперти не будет найдено в указанных локейшн, то спринг поищет их в системных файлах и переменных окружения. override указывает, что свойства системы д.б. просмотрены первыми. Т.о. если в системных проперти будет свойство, то его можно будет переопределить в проперти файле. ???проверить!!! Ткач сказал наооборот!!!??? Ещё бывает never - не рассмаривать системные проперти.

Вот так работает:

<**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**>

Как внедрять проперти:

<**bean id="client" class="com.epam.spring.core.beans.Client"**>  
 <**constructor-arg index="0" value="${id}"**/>  
 <**constructor-arg index="1" value="${name}"**/>  
 <**property name="greeting" value="${greeting}"**/>  
</**bean**>

<**constructor-arg index="0"**>${id}</**constructor-arg**> - так не работает!!! Говорит, что Element 'constructor-arg' cannot have character [children], because the type's content type is element-only. А у Ткача почему-то так написано.

Дополнительные неймспейсы.

**xmlns:util="http://www.springframework.org/schema/util"  
xmlns:p="http://www.springframework.org/schema/p"  
xmlns:c="http://www.springframework.org/schema/c"**

Для utils так же указывается:

**xsi:schemaLocation="  
...  
http://www.springframework.org/schema/util  
http://www.springframework.org/schema/util/spring-util-3.2.xsd"**

P-namespace нужен для более компактного внедрения проперти бинов. Имеется ввиду значения полей с геттерами и сеттерами, а не проперти файл.

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

С-namespace – как P, только для аргументов конструктора.

Для внедрения референсов используется ref

<**bean id="app" class="com.epam.spring.core.App"  
 c:client-ref="client"**>

???Зачем util???

**Дорабока приложения**

Добавить проперти файл, указать там значения констант для клиента, подключить его.

Автовайринг.

<bean id=”…” class=”…” autowire=”…”/>

Можно автовайрить:

byName – имени проперти

byType – классу бина в проперти

constructor – классу бина в конструкторе.

???пример???

Подходит только для бинов с уникальным классом или именем. Не рекомендуется смешивать автовайринг с непосредственным указанием.

ContextAwareClasses.

Если нужно, чтобы бин знал о контексте, то можно расширить интерфейс ApplicationContextAware.

**public class** Monitor **implements** ApplicationListener {  
 **public void** onApplicationEvent (ApplicationEvent event)   
 {...}  
}

Можно получить:

* - ссылку на контекст;
* - имя бина;
* - сервлет контекст для веб;
* - и т.д.

Другие классы:

* ApplicationContextAware
* ApplicationEventPublisherAware
* BeanFactoryAware
* BeanNameAware
* ResourceLoaderAware
* ServletContextAware

Для получения информации о событиях контекста, есть интерфейс ApplicationListener

**public class** Monitor **implements** ApplicationListener {  
 **public void** onApplicationEvent (  
 ApplicationEvent event) {...}  
}

События:

* ContextStartedEvent
* ContextStoppedEvent
* ContextRefreshedEvent
* ContextClosedEvent