

# Spring-Cleaning

Tips for clearing the clutter from your Spring configuration

Craig Walls  
Spring Dallas User Group  
July 18, 2007

# About you?

— [ How long have you been working with Spring?

— [ What version of Spring are you using?

— [ What version of Java?

# About me

— [ Over 13 years of experience developing solutions for Telecom, Finance, Retail, and Education

— [ Java fanatic since early 1996; Spring fanatic since 2003

— [ Author XDoclet in Action (Manning 2003), Spring in Action (2005), and Spring in Action 2E (2007)

# About this presentation

— [ You can e-mail me with questions: [craig-sia@habuma.com](mailto:craig-sia@habuma.com)

— [ Download these slides:

— <http://www.habuma.com/spring/springcleaning.pdf>

— [ Download the example code:

— <http://www.habuma.com/spring/springcleaning.zip>

— [ Check out my blog: <http://www.springinaction.com>

# Spring sucks!

— [ Spring == XML

— [ XML == Evil

— [ Evil == Sucks

— [ Therefore, Spring == Sucks

# The so-called solutions

— [ **Forget dependency injection.** I don't need it. What's wrong with letting my objects create their own dependencies?

— [ **I'll do the injection myself.** I know how to call a setter method..I don't need a framework to do it for me.

— [ **Annotations!** Dependency injection is good, but I'd rather declare injections directly in my Java code so I can see everything in one place.

# The truth about Spring/DI

## — [ **Spring != XML**

- The Spring container is decoupled from the configuration strategy

## — [ **Spring is more than DI**

- Spring is a complete application framework

— [ Dependency injection is about making classes unaware of their dependencies

# Nevertheless...

— [ DI is at the core of everything you do in Spring

— [ Spring DI typically involves a lot of XML

— [ Let's see how to reduce/eliminate XML without throwing Spring out with it

# 3 angles of attack

— [ **Smarter XML** - Wire more beans more with less XML

— [ **Annotations** - Use Java 5 annotations to declare bean configurations

— [ **Scripting** - Wire beans using dynamic scripting languages

# Disclaimers

- [ There is no “one size fits all” fix

- An ounce of pragmatism is in order

- [ Time is limited

- So, too, will the examples be

- [ Examples are a bit rough around the edges

# XML Done Smartly

Honey, I Shrunk the XML

# Smarter XML techniques

— [ Shorthand XML

— [ Auto-wiring

— [ Bean inheritance

— [ Property editors

— [ The “p” namespace

— [ Custom configuration

— [ Arid POJOs

# Shorthand XML

— [ Take advantage of Spring's shortcuts for injecting values and references (introduced in Spring 1.2)

— Instead of `<value>` and `<ref>`, use `value=""` and `ref=""`

# Shorthand XML example

## Pre-Spring 1.2:

```
<bean id="myBean" class="com.habuma.MyBeanImpl">  
  <property name="someProperty">  
    <value>This is a string value</value>  
  </property>  
  <property name="someReference">  
    <ref bean="someOtherBean" />  
  </property>  
</bean>
```

## Spring 1.2+:

```
<bean id="myBean" class="com.habuma.MyBeanImpl">  
  <property name="someProperty" value="This is a string value" />  
  <property name="someReference" ref="someOtherBean" />  
</bean>
```

# Shorthand XML: Pros/Cons

- [ **Pros:**

- Definitely more terse

- [ **Cons:**

- Can't be used for values of collections

# Auto-wiring

- [ Let Spring figure out how to inject properties

- [ Comes in 5 varieties:

- No - Do not auto-wire

- `byName` - Inject beans into properties where the bean's ID matches the property's name

- `byType` - Inject beans into properties where the bean's type is assignable to the property's type

- `constructor` - Inject into constructor properties where the beans' types are assignable to the constructor argument types

- `autoDetect` - Try constructor first, then `byType`

# Auto-wiring

— [ Auto-wiring strategy can be declared on a per-bean basis or a per-XML file basis

— Per bean, use the `<bean>` element's `autowire` attribute

— Available in all versions of Spring

— Per XML file, use the `<beans>` element's `default-autowire` attribute

— Available since Spring 2.0

# Auto-wiring example 1

```
<bean id="knight"  
      class="com.springinaction.knight.KnightOfTheRoundTable"  
      autowire="byType">  
    <constructor-arg value="Bedivere" />  
  </bean>
```

This injects all references

Still must explicitly  
configure values

# Auto-wiring example 2

```
<?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-2.0.xsd"
  default-autowire="byType">
  <bean id="knight"
    class="com.springinaction.knight.KnightOfTheRoundTable">
    <constructor-arg value="Bedivere" />
  </bean>
  ...
</beans>
```

All beans will be autowired

No need to explicitly autowire

# Auto-wiring: Pros/Cons

## Pros:

- Can dramatically reduce the amount of XML in a Spring configuration

## Cons:

- Along with terseness comes lack of clarity: What was wired where?

- `byName` auto-wiring couples configuration to implementation details

- `byType` and constructor auto-wiring can be problematic with ambiguities

# Bean inheritance

— [ Available in all (at least since 1.2) versions of Spring

— [ Declare repeated configuration details in a parent bean

— [ Create sub-beans that inherit configuration from parent

# Bean inheritance: Example 1

```
<bean id="knightParent"  
  class="com.springinaction.knight.KnightOfTheRoundTable"  
  abstract="true">  
  <property name="quest" ref="quest" />  
  <property name="horse" ref="horse" />  
  <property name="sword" ref="sword" />  
  <property name="armor" ref="armor" />  
  <property name="shield" ref="shield" />  
</bean>
```

```
<bean id="knight" parent="knightParent">  
  <constructor-arg value="Bedivere" />  
</bean>
```

Type and properties  
are inherited

A diagram consisting of two arrows. The first arrow starts from the right side of the first code block and points to the text 'Type and properties are inherited'. The second arrow starts from the text and points to the right side of the second code block.

# Bean inheritance: Example 2

```
<bean id="knightParent"  
  abstract="true">  
  <property name="quest" ref="quest" />  
  <property name="horse" ref="horse" />  
  <property name="sword" ref="sword" />  
  <property name="armor" ref="armor" />  
  <property name="shield" ref="shield" />  
</bean>
```

```
<bean id="knight"  
  class="com.springinaction.knight.KnightOfTheRoundTable"  
  parent="knightParent">  
  <constructor-arg value="Bedivere" />  
</bean>
```

Just properties  
are inherited



# Bean inheritance:Pros/Cons

## — [ Pros:

— Helps to create DRY Spring configuration

## — [ Cons:

— A little tricky to navigate bean hierarchies, especially without tool support

# Property editors

— [ Available in all versions of Spring

— [ Express complex configuration as Strings

— [ Let Spring figure out how to create complex objects

# Property editors : Built-ins

— [ ByteArrayPropertyEditor

— [ CharacterEditor

— [ CharArrayPropertyEditor

— [ ClassArrayEditor

— [ ClassEditor

— [ CustomBooleanEditor

— [ CustomCollectionEditor

— [ CustomDateEditor

— [ CustomMapEditor

— [ CustomNumberEditor

— [ FileEditor

— [ InputStreamEditor

— [ LocaleEditor

— [ PatternEditor

— [ PropertiesEditor

— [ ResourceBundleEditor

— [ StringArrayPropertyEditor

— [ StringTrimmerEditor

— [ URLEditor

— [ URLEditor

# Property editors example 1

## In Java:

```
public class KnightOnCall implements Knight {  
    ...  
    private URL url;  
    public void setUrl(URL url) { this.url = url; }  
  
    private PhoneNumber phoneNumber;  
    public void setPhoneNumber(PhoneNumber phoneNumber) { this.phoneNumber = phoneNumber; }  
}
```

## In XML:

```
<bean id="knight"  
    class="com.springinaction.knight.KnightOnCall">  
    <property name="url" value="http://www.knightoncall.com" />  
    <property name="phoneNumber" value="940-555-1234" />  
</bean>
```

# Property editors example 2

— [ Registering custom editor (or a non-registered editor):

```
<bean id="customEditorConfigurer"  
  class="org.springframework.beans.factory.config.CustomEditorConfigurer">  
  <property name="customEditors">  
    <map>  
      <entry key="com.springinaction.knight.PhoneNumber">  
        <bean id="phoneEditor"  
          class="com.springinaction.springcleaning.PhoneNumberEditor" />  
      </entry>  
    </map>  
  </property>  
</bean>
```

# Property editors: Pros/Cons

## — [ Pros:

- Complex types can be created from Strings without verbose set of `<property>` elements

## — [ Cons:

- Not always apparent what type is being created
- Looks weird if you don't know what's going on

# The "p" namespace

— [ Introduced in Spring 2.0

— [ Allows for terse injection of properties, as attributes of the `<bean>` element

— [ Made available with:

```
<beans xmlns="http://www.springframework.org/schema/beans"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xmlns:p="http://www.springframework.org/schema/p"  
  xsi:schemaLocation="http://www.springframework.org/schema/beans  
    http://www.springframework.org/schema/beans/spring-beans.xsd">
```

# "p" namespace example

— [ "p" configuration:

```
<bean id="knight"  
  class="com.springinaction.knight.KnightOfTheRoundTable"  
  p:quest-ref="quest"  p:horse-ref="horse"  
  p:sword-ref="sword"  p:armor-ref="armor"  
  p:shield-ref="shield">  
  <constructor-arg value="Bedivere" />  
</bean>
```

# "p" namespace : Pros/Cons

## — [ Pros:

- Less XML to accomplish the same injection

## — [ Cons:

- Super-terse. May look alien to some developers

# Custom elements

- [ Encapsulates more complex bean definitions behind simpler XML elements.

- [ Available since Spring 2.0. Spring 2.0 provides several namespaces:

- aop, jee, lang, tx, util

- More coming in Spring 2.1: context

- [ Spring Web Flow defines a “flow” namespace

- [ DWR provides a “dwr” namespace

- [ Spring Security (Acegi) will provide a custom element namespace

# JNDI configuration example

## Using <bean>:

```
<bean id="dataSource"  
  class="org.springframework.jndi.JndiObjectFactoryBean">  
  <property name="jndiName" value="/jdbc/RantzDatasource" />  
  <property name="resourceRef" value="true" />  
</bean>
```

## Using "jee" namespace:

```
<jee:jndi-lookup id="dataSource"  
  jndi-name="/jdbc/RantzDatasource"  
  resource-ref="true" />
```

# Custom elements:Pros/Cons

## — [ Pros:

- Simplifies complex XML configuration
- Enables domain-specific configuration

## — [ Cons:

- Hides what beans are really being configured (may be a good thing, though)

# Arid POJOs

- [ Turn auto-wiring up a notch

- Automatically declare and auto-wire all classes in a specific package(s)

- Automatically create DAOs from interfaces

- [ Based on the notion that most beans are declared the same

# Arid POJOs

Created by Chris Richardson (POJOs in Action)

Available from: <http://code.google.com/p/aridpojos>

Add to Spring config with:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:arid="http://chrisrichardson.net/schema/arid"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://chrisrichardson.net/schema/arid
    http://chrisrichardson.net/schema/arid.xsd">
```

# Arid POJOs example 1

— [ Automatically declare all beans in a package, then auto-wire them:

```
<arid:define-beans  
  package="com.habuma.dao"  
  autowire="byType" />
```

# Arid POJOs example 2

— [ Automatically create DAOs from a given interface:

```
<bean id="parentDaoFactory" abstract="true"  
  class="net.chrisrichardson.arid.dao.hibernate.GenericDAOFactoryBean">  
  <property name="sessionFactory" ref="sessionFactory" />  
</bean>
```

```
<arid:define-beans package="com.habuma.exampLEDomain"  
  package-scanner=  
    "net.chrisrichardson.arid.InterfaceAndAbstractClassPackageScanner"  
  pattern="net.chrisrichardson.arid.domain.GenericDao+ "  
  bean-generator=  
    "net.chrisrichardson.arid.ChildWithConstructorArgBeanGenerator"  
  parent-bean-name="parentGenericDaoFactoryBean" />
```

# Arid POJOs : Pros/Cons

## Pros:

- Takes auto-wiring a step further by auto-declaring beans

## Cons:

- Just like auto-wiring, it's not clear what's going on. Instead of "what got injected into who?", the question becomes "where are all these beans coming from?"

# Annotating Spring

Dependency injection is where it's @

# Annotating Spring

— [ Use `@AspectJ` for declaring aspects

— [ Use `@Transactional` for transactions

— [ Spring `JavaConfig`

— [ Spring 2.1 annotations

# Life without @AspectJ

Pre-2.0 Spring AOP was a clumsy mess of XML

```
<bean id="knight" class="org.springframework.aop.framework.ProxyFactoryBean">  
  <property name="target" ref="knightTarget" />  
  <property name="interceptorNames" value="pointcutAdvisor" />  
  <property name="proxyInterfaces"  
    value="com.springinaction.knight.Knight" />  
</bean>
```

This is just weird

```
<bean id="knightTarget"  
  class="com.springinaction.knight.KnightOfTheRoundTable">
```

...

```
</bean>
```

```
<bean id="pointcutAdvisor" class="org.springframework.aop.support.RegExpPointcutAdvisor">  
  <property name="pattern" value=".*embark.*" />  
  <property name="advice" value="minstrel" />  
</bean>
```

```
<bean id="minstrel" class="com.springinaction.knight.MinstrelAdvice" />
```

# Life without @AspectJ

— [ Spring 2.0 introduced the "aop" namespace

— Much better...less XML

— Also cleans up the Java advice class...just a POJO

```
<bean id="minstrel" class="com.springinaction.knight.Minstrel" />
```

```
<aop:config>
```

```
  <aop:aspect ref="minstrel">
```

```
    <aop:after-returning
```

```
      method="sing"
```

```
      pointcut="execution(* *.Knight.embarkOnQuest(..))" />
```

```
  </aop:aspect>
```

```
</aop:config>
```

# @AspectJ

- [ But Spring 2.0 also introduced integration with @AspectJ

- [ Now aspects only require minimal XML

- `<aop:aspectj-autoproxy/>`

- One bean declaration for each aspect class

# @AspectJ example

```
@Aspect
public class Minstrel {
    @Pointcut("execution(* *.Knight.embarkOnQuest(..))")
    public void embark() {}

    @AfterReturning("embark()")
    public void sing() {
        System.out.println("Fa la la!");
        System.out.println("The brave knight is embarking on a quest!");
    }
}
```

# @AspectJ example

— [ What is needed in the XML?

```
<aop:aspectj-autoproxy />
```

— [ Yes...that's really all you need!

# @AspectJ : Pros/Cons

## — [ Pros:

— Significantly less XML for aspects

## — [ Cons:

— Aspect classes are now coupled to AspectJ...can't be used anywhere without AspectJ in the classpath

# @Transactional

- [ Pre-2.0 Spring transactions were just as messy as pre-2.0 Spring AOP

- Instead of ProxyFactoryBean, you'd use TransactionProxyFactoryBean

- #1 use for bean inheritance

# @Transactional

— [ Spring 2.0 introduced the “tx” namespace

— [ Much simpler XML:

```
<tx:advice id="txAdvice">
  <tx:attribute>
    <tx:method name="add*" propagation="required" />
    <tx:method name="*" propagation="supports"
      read-only="true"/>
  </tx:attributes>
</tx:advice>
```

# @Transactional

— [ Spring 2.0 also introduced the @Transactional annotation

— [ Now methods and classes can be declared transactional with minimal XML

— Very appropriate use of annotations, BTW

# @Transactional example

## In Java:

```
@Transactional(propagation=Propagation.SUPPORTS)
public class CustomerServiceImpl implements CustomerService {
    @Transactional(propagation=Propagation.REQUIRED)
    public void addNewCustomer(Customer customer) {
        ...
    }
    ...
}
```

## In XML:

```
<tx:annotation-driven transaction-manager="txMgr" />
```

# @Transactional : Pros/Cons

## — [ Pros:

- Like @AspectJ annotations, very very little XML required

## — [ Cons:

- Invasive
- Business classes annotated with @Transactional are now coupled with Spring...can't use them without having Spring in the classpath

# Spring JavaConfig

<http://www.springframework.org/javaconfig>

Currently at version 1.0-M2a

Recreates Spring XML configuration in Java using annotations

Provides several annotations for bean configuration

— **@Configuration** - Declares a class as a config class

— **@Bean** - Declares a method as a bean declaration

— **@ExternalBean** - Declares an abstract method as a reference to an externally defined bean

— **@AutoBean** - Declares an abstract method to serve as a holder for an automatically instantiated/wired bean

— **@ScopedProxy** - Used to declare a scoped proxy for a bean (non-singleton/non-prototype)

# Spring JavaConfig

— [ To load a JavaConfig-configured context, you have two choices:

— Use `AnnotationApplicationContext`

— Simple, no-XML required approach

— Hard to use with web apps

— Can't parameterize configuration instances

— Configure a `ConfigurationPostProcessor` (in the XML)

— Easy to use with web apps (using minimal bootstrap XML)

— Configuration can be parameterized

# Loading JavaConfig

## AnnotationApplicationContext:

```
ApplicationContext ctx =  
    new AnnotationApplicationContext(MyConfig.class.getName());
```

## ConfigurationPostProcessor:

```
<bean class="com.habuma.samples.MyJavaConfig" />  
<bean class=  
    "o.sf.config.java.process.ConfigurationPostProcessor" />
```

# Spring JavaConfig example

## @Configuration

```
public abstract class KnightConfig {
```

### @Bean

```
public Knight knight() {  
    KnightofTheRoundTable knight =  
        new KnightOfTheRoundTable("Bedivere");  
    knight.setQuest(quest());  
    return knight;  
}
```

Private bean

Inject quest into knight

### @Bean

```
private Quest quest() {  
    return new HolyGrailQuest();  
}
```

### @ExternalBean

```
private abstract Horse horse();
```

```
}
```

This bean is defined elsewhere

# JavaConfig : Pros/Cons

## Pros:

- Minimally invasive - Annotations are confined to special context definition classes
- Dynamic - Use Java constructs at will
- Testable - Easily write unit tests against configuration
- Refactorable - No static identifiers
- Offers bean visibility through Java constructs
- Parameterizable if using bootstrap XML

## Cons:

- Non-intuitive - Structured like Spring XML...looks like Java

# Spring 2.1 annotations

— [ Spring 2.1 will add a few new annotations:

— @Component - Indicates that a class is a component that should be registered in Spring

— @Autowired - Indicates a property should be autowired

— @Scoped - Declares scoping on an auto-detected bean

— [ Works with a new `<context:component-scan>` config element

# Spring 2.1:component-scan

- [ Scans a package (and its subpackages), auto-detecting all beans annotated with `@Component`, `@Repository`, or `@Aspect`

- [ Autowires (byType) all properties and methods that are annotated with `@Autowired`

- [ Also supports some JSR-250 annotations:

- `@PostConstruct`, `@PreDestroy`, `@Resource`, `@Resources`

# Spring 2.1 example (Java)

```
@Component("knight")
public class KnightOfTheRoundTable implements Knight {
    private String name;
    private Quest quest;
    private Horse horse;
    ...
    public KnightOfTheRoundTable(String name) {
        this.name = name;
    }

    @Resource ← JSR-250 annotation
    public void setQuest(Quest quest) { this.quest = quest; }

    @Autowired
    private void myKingdomForAHorse(Horse horse) { this.horse = horse; }
} ← Will this work?
```

This isn't a bean setter method

# Spring 2.1 example (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"  
  xmlns:context="http://www.springframework.org/schema/context"  
  xsi:schemaLocation="http://www.springframework.org/schema/beans  
    http://www.springframework.org/schema/beans/spring-beans.xsd  
    http://www.springframework.org/schema/context  
    http://www.springframework.org/schema/context/spring-context-2.1.xsd">
```

```
<context:component-scan  
  base-package="com.springinaction.knight" />
```

← This automatically  
wires almost everything

```
<bean id="knight" class="com.springinaction.knight.KnightOfTheRoundTable">  
  <constructor-arg value="Bedivere" />  
</bean>
```

← Needed to set name  
through constructor

```
</beans>
```

# Spring 2.1 : Pros/Cons

## — [ Pros:

- Moves configuration details closer to the beans being configured (DRY)
- Injection not limited to setter methods

## — [ Cons:

- Moves configuration details closer to the beans being configured (Invasive)
- Could be static identifiers

# Scripting Spring

Cut XML and be buzzword compliant at the same time!

# Scripting Spring

— [ Springy (JRuby)

— [ Grails Spring Builder (Groovy)

# Springy

- [ Enables Spring to be configured through a Ruby DSL

- [ Get it from <http://code.trampolinesystems.com/springy>

- Latest version is 0.2

- Apache license

# Springy : Load the context

— [ Programmatically:

```
ApplicationContext ctx = new JRubyApplicationContext(  
    new ClassPathResource("com/habuma/samples/ctx.rb")  
);
```

— [ No obvious way to use with web applications...bummer...

# Springy example

```
bean :knight,  
  "com.springinaction.knight.KnightOfTheRoundTable" do |b|  
  
  b.new "Bedivere"  
  b.quest = :quest  
  
  ...  
end  
  
bean :quest,  
  "com.springinaction.knight.HolyGrailQuest" do |b|  
  b.new  
end
```

# Springy example 2

— [ Can you do this in Spring XML?

```
for num in (1..10)
  bean : "knight#{num}",
    "com.springinaction.knight.KnightOfTheRoundTable" do |b|
    b.new "Bedivere"
    b.quest = :quest
  end
end
```

# Springy example:inline XML

— [ If you absolutely must use XML, you can inline it in Ruby:

```
inline_xml do <<XML
  <bean id="dragonQuest"
    class="com.sia.knight.SlayDragonQuest" />
XML
end
```

# Springy : Serialize to XML

— [ Cool trick: Get Spring XML from Springy Ruby definition:

```
((JRubyApplicationContext) ctx).getContextAsXml();
```

# Springy : Pros/Cons

## — [ Pros:

- Completely XML-free
- All of Ruby's constructs available to define Spring context

## — [ Cons:

- Serializes context to XML then reloads it-Performance implications

# Grails Spring Builder

— [ Use a Groovy DSL to configure Spring context

— [ Get it at <http://www.grails.org/Spring+Bean+Builder>

— Actually, part of Grails - In grails-core-0.5.6.jar

# Groovy example

```
def bb = new grails.spring.BeanBuilder()

bb.beans {
    quest(HolyGrailQuest) {}
    horse(Horse) {}
    sword(Sword) {}
    shield(Shield) {}
    armor(Armor) {}

    knight(KnightOfTheRoundTable, "Bedivere") {
        delegate.quest = quest
        delegate.horse = horse
        delegate.sword = sword
        delegate.shield = shield
        delegate.armor = armor
    }
}

ApplicationContext ctx = bb.createApplicationContext()
def knight = ctx.getBean("knight")
knight.embarkOnQuest()
```

# Groovy : Pros/Cons

## Pros:

- Completely XML-free
- Can use any of Groovy's constructs to build Spring context

## Cons:

- Not clear how to use it outside of a Groovy script
- Not clear how to use it in a Spring MVC scenario
- (just a nit) Not separate from Grails...must include Grails in classpath

# Recap

He made the XML shorter...too bad he couldn't have done the same with the presentation

# What we have learned

- [ Spring XML sucks...

- ...If you don't take advantage of the tricks to cut the clutter

- [ Spring and Annotations: Not a zero-sum game

- Spring encourages proper use of XML (and tolerates improper use)

- [ Spring != XML

- JRuby and Groovy configuration options are available

# A few final Spring XML tips

- [ You don't have to wire everything

- Case in point: A command controller's `commandName` and `commandClass` properties

- [ Remember that there are two types of configuration

- Use `PropertyPlaceholderConfigurer` to wire in external configuration

- [ Don't put all of your beans in one XML file

- Break your context definition down—perhaps by application layers or functional divisions

# Q&A

[craig-sia@habuma.com](mailto:craig-sia@habuma.com)