Spring Advanced Wiring

Basic Spring wiring solves 90%. This is the other 10%

for the Spring Dallas User Group
by Jack Frosch
20 November 2013

About Me

- Jack Frosch
  - Consultant, Trainer, Entrepreneur
- jackfrosch@gmail.com
- LinkedIn: http://www.linkedin.com/in/jackfrosch/
- Twitter: @jackfrosch

Let's collaborate on something great!

Agenda

- Basic Spring Wiring Review
- Spring Profiles
- Spring Expression Language
- @Conditional (Spring 4)

Objectives

- Review Spring basic wiring using annotations, XML, and code-based configuration
- Understand how to declare and use Spring Profiles
- Understand how to use SpEL to dynamically specify properties

Basic Spring Wiring

- The IoC/DI Value Proposition
- Spring Config – XML
- Spring Config – Annotations
- Spring Config – Code

The IoC / DI Value Proposition

- Core Spring is all about IoC/DI
  - = Inversion of Control / Dependency Injection
- IoC/DI allows us to separate those things that change from those that don’t and only change those classes that change

The IoC / DI Value Proposition

```
public class Collaborator {
  …
}
…
public class UsefulService {
    private Collaborator helper;

    public UsefulService() {
      helper = new Collaborator();
    }
     ...
}
```

The IoC / DI Value Proposition

```
public class Collaborator { … }
public class BetterCollaborator extends Collaborator {
  @Override
  (some behavior)
}
…
// We have to crack open UsefulService to
//  use BetterCollaborator … even though UsefulService
//  hasn’t changed!
public class UsefulService {
    private BetterCollaborator helper;
    public UsefulService() {
       helper = new BetterCollaborator();
    }
    ...
}
```

The IoC / DI Value Proposition

```
// It’s better to use interfaces
public interface UserfulService { … }
public interface Collaborator { … }

public class CollaboratorImpl implements Collaborator {…}

public class BetterCollaboratorImpl extends CollaboratorImpl {
  @Override (some behavior)
}
…
// But we still have to crack open UsefulService
// even though UsefulService hasn’t changed!
public class UsefulServiceImpl implements UsefulService {
    private Collaborator helper;
    public UsefulServiceImpl() {
       helper = new BetterCollaborator();
    }
    ...
}
```

The IoC / DI Value Proposition

```
// This is best

public interface Collaborator { … }
public interface UsefulService { … }
public class CollaboratorImpl implements Collaborator {…}
public class BetterCollaboratorImpl extends CollaboratorImpl {
  @Override (some behavior)
}
…
public class UsefulServiceImpl implements UsefulService {
    private Collaborator helper;
    public UsefulServiceImpl(Collaborator helper) {
       this.helper = helper;
    }
    ...
}
// External process (i.e. Spring) determines which
// implementation is injected. Thus, we can change
// collaborator implementations without touching UsefulService
```

Spring Config – XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=“…”>
   <bean id=“usefulService“
      class=“com.acme.UsefulServiceImpl">
         <constructor-arg ref=”collaborator1” />
    </bean>

    <bean id=“collaborator1"
       class="com.acme.CollaboratorImpl" />

    <bean id=“collaborator2"
       class="com.acme.BetterCollaboratorImpl" />
</beans>
```

Spring Config – XML

Live Code Demo

Spring Config – Annotations

```
// More specifically: @Service, @Repository, @Controller
@Component
public class CollaboratorImpl implements Collaborator {
}

@Service
public class UsefulServiceImpl implements UsefulService {
 private Collaborator helper;

 @Autowired
 public UserServiceImpl(Collaborator helper) {
    this.helper = helper;
 }
}
```

Spring Config – Annotations

Live Code Demo

Spring Config – Code

```
public interface Collaborator { … }
public interface UsefulService { … }
public class CollaboratorImpl implements Collaborator {…}
public class BetterCollaboratorImpl extends CollaboratorImpl {…}
public class UsefulServiceImpl implements UsefulService {…}

@Configuration
public class AppConfiguration {
     @Bean
    public Collaborator helper() {
       return new BetterCollaboratorImpl();
    }

    @Bean(name="myService")
    public UsefulService usefulService() {
       return new UsefulServiceImpl(helper());
    }
}
```

Bean name is "helper"

Bean name is "myService"

Spring Config – Code

Live Code Demo

Spring Profiles

- The environment problem
  Spring Profiles as a solution

The environment problem

- Spring's basic wiring solves problems for using alternate implementations

  But this works only when we know in advance we need to change Spring to use the alternate implementation for all environments
- When the app is executed in different environments, we usually need to change the Spring wiring

The environment problem

- Suppose you create a web app
  - On your development environment, you deploy it to Tomcat. The app needs to
    - Use ehCache
    - Access your local MySQL database
  - After local testing, you commit and a WAR file is built to run on QA's JBoss server. The app needs to
    - Use the JBoss caching solution
    - Access the QA PostgreSQL database
  - After QA testing, a release WAR is built for deployment to the production WebSphere servers. The app needs to
    - Use the WQAS caching solution
    - Access the production Oracle database

The environment problem

- What has changed in your code between these environments?
  - Nothing
- Yet, as we move between environments, we have to crack open the Spring configuration files (XML, annotations, code) and make wiring changes to accommodate the target environment
- Enter Spring Profiles

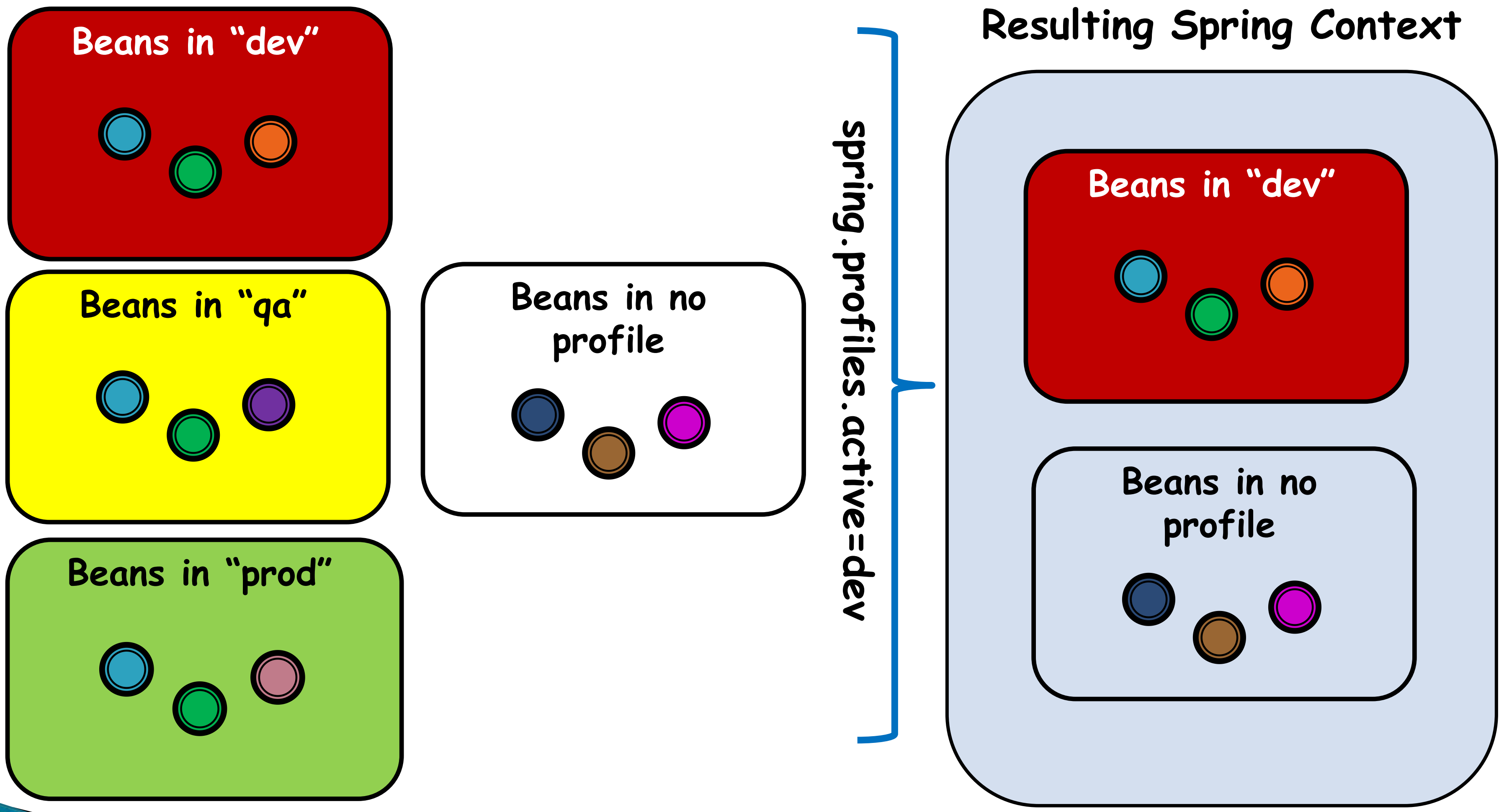
The Spring Profile solution

- Spring Profiles were introduced in Spring 3.1
- Using them is simple
  - Associate beans with a profile (or no profile)
  - Specify default profile(s) in effect when Spring context loads
  - Specify active profile(s) in effect to override the default profiles specified
  - The *effective* profiles are the default profiles if active profiles not specified

The Spring Profile solution

- When the Spring context is loaded
  - All beans not associated with a profile are registered in the context
  - All beans associated with the effective profiles are registered
  - If effective profiles are in effect, beans associated with a profile, but not the effective profiles will not be registered

The Spring Profile solution



The Spring Profile solution

- To support associating beans to profiles
  - Spring XML schema was changed
  - Annotations were changed for declaring the profile

The Spring Profile solution XML Configuration

- Spring XML schema was changed
  - The <beans> element now supports a *profiles* attribute
  - <beans> elements can now be nested inside the outer <beans> element
    - But inner <beans> must be at the end of the outer <beans> element declarations
  - The XML unique id rule in page is relaxed
    - Beans in different profiles can (and will) have the same id

The Spring Profile solution

```
<beans xmlns=“…”> <!-- No profile specified -->
  <bean id=“myBean” … /> <!--belongs to no profile-->

  <beans profile=“dev”>
    <bean id=“cacheProvider” … />
    <bean id=“dataSource” … />
  </beans>

  <beans profile=“qa”>
    <bean id=“cacheProvider” … />
    <bean id=“dataSource” … />
  </beans>

  <beans profile=“prod”>
    <bean id=“cacheProvider” … />
    <bean id=“dataSource” … />
  </beans>
</beans>
```

The Spring Profile solution Annotation Configuration

- Spring added an @Profile annotation to allow Components to be associated with a profile

```
@Service
@Profile(“prod”)
public class ServiceImpl implements Service {…}

// in a different package
@Service
@Profile(“dev”)
public class ServiceImpl implements Service {…}

@Component
public class Consumer {
 @Autowired
 public Consumer(Service service) {…}
}
```

The Spring Profile solution
Specifying Profiles

- Ok, but *how* do we specify which profiles are the *effective* profiles?
- We use key/value pair(s)
  - Two keys are used:
    - spring.profiles.default
    - spring.profiles.active
  - The values associated with these are the profile(s); e.g.

```
spring.profiles.default=prod,fullSecurity
spring.profiles.active=dev,simpleSecurity
```

The Spring Profile solution Specifying Profiles

- Ok, but *where* do we specify these key value pair(s)?
- There are multiple places possible
  - Initialization parameters on DispatcherServlet
  - Context parameters of a web application
  - JNDI entries
  - Environment variables
  - JVM system properties
  - Using the @ActiveProfiles annotation on an integration test class

Spring Config – Code

Live Code Demo

(xml, annotations and code config)

Spring Expression Language (SpEL)

- Basic Spring config solves 90% of the problem
- Profiles solves 90% of the remaining 10%*
- The SpEL solves the rest
- Why so little?
  - SpEL is more useful in XML than in Java code
  - But if we're in Java code, we can do everything SpEL can do anyway
  - So really, it's a mostly solution for the remainder of the Spring config remaining in XML
    - For some organizations, this will be very little

*These numbers are for illustration purposes only. Your mileage may vary.

Spring Expression Language (SpEL)

- The expression language supports the following functionality
  - Literal expressions
  - Boolean and relational operators
  - Regular expressions
  - Class expressions
  - Accessing properties, arrays, lists, maps
  - Method invocation
  - Relational operators
  - Assignment
  - Calling constructors
  - Bean references
  - Array construction
  - Inline lists
  - Ternary operator
  - Variables
  - User defined functions
  - Collection projection
  - Collection selection
  - Templated expressions

You will be tested on all of these at the end of the presentation… ☺

Spring Expression Language (SpEL)

Wow! That's a lot! ☺

Let's see a live code demo

(xml & annotations)

@Conditional

- Profiles solves a common runtime problem
  - However, we still need to declare in advanced that a bean should be loaded for a particular profile
  - What if we want to load a bean only if
    - Is a particular library is available on the class path?
    - An environment variable is set?
    - A particular annotation (with its specified data) is specified for the bean

@Conditional

- To solve these edge cases, we could use some complicated combination of basic bean wiring, Profiles, and SpEL
- Spring 4 offers a simpler solution:

@Conditional

@Conditional

```
@Service
@Conditional(FooCondition.class)
public class ServiceImpl implements Service {…}

// in a different package
@Service
@Conditional(BarCondition.class)
public class ServiceImpl implements Service {…}

@Component
public class Consumer {
 @Autowired
 public Consumer(Service service) {…}
}
```

@Conditional

```
// Spring 4 defines this:

public interface AnnotatedTypeMetadata {
  boolean isAnnotated(String annotationType);

  Map<String, Object> getAnnotationAttributes(
                                     String annotationType);

  Map<String, Object> getAnnotationAttributes(
                                  String annotationType,
                                  boolean classValuesAsString);

  MultiValueMap<String, Object> getAllAnnotationAttributes(
                                        String annotationType);

  MultiValueMap<String, Object> getAllAnnotationAttributes(
                                  String annotationType,
                                  boolean classValuesAsString);
}
```

@Conditional

```
// Spring 4 defines these:

public interface ConditionContext {
  BeanDefinitionRegistry getRegistry();
  ConfigurableListableBeanFactory getBeanFactory();
  Environment getEnvironment();
  ResourceLoader getResourceLoader();
  ClassLoader getClassLoader();
}

public interface Condition {
  boolean matches(ConditionContext ctxt,
                  AnnotatedTypeMetadata metadata);
}
```

@Conditional

```
public class FooCondition implements Condition {
  public boolean matches(ConditionContext ctxt,
                         AnnotatedTypeMetadata metadata) {
     // implement the code that checks for the Foo condition
     // return true if the FooCondition is matched
  }
}

public class BarCondition implements Condition {
  public boolean matches(ConditionContext ctxt,
                         AnnotatedTypeMetadata metadata) {
     // return true if the BarCondition is matched
  }
}
```

@Conditional

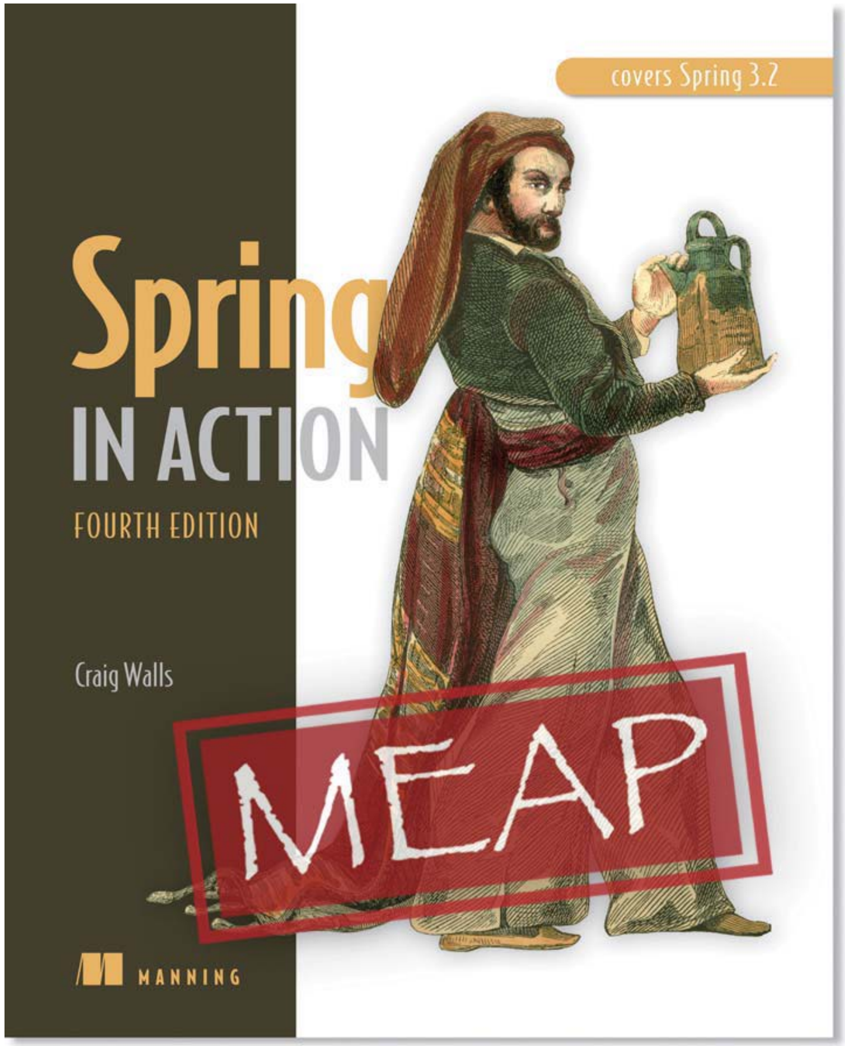
- As empirical proof that @Conditional will solve a broad swath of configuration problems
  - Spring 4 re-implemented @Profile to add @Conditional behavior using Spring 4’s ProfileCondition class
  - ProfileCondition checks if the bean’s declared Profile(s) are in those defined by the runtime environment
    - If so, the bean is included
    - If not, the bean is not included

Summary

- Basic Spring wiring solves 90%+ of the wiring needs
- Sometimes, we need something extra
    - Spring Profiles let us specify an umbrella of common configurations that can be activated, or ignored, with a single runtime active profile indicator
- Sometimes Profiles are too coarse for the fine tuning we need to do
    - Spring Expression Language gives us fine grained, dynamic control over configuration

Next Steps

- Checkout the docs
  - 3.2.5 Reference: http://bit.ly/17iZT7n
- Buy Manning's Spring in Action (4th Edition coming soon)
- Corner Craig Walls and pump him for information



Questions?