Pitchfork - Reference Documentation

1.0-m1

24.05.2006

Table of Contents

Chapter 1. Pitchfork: Spring JEE Support

1.1. Introduction

EJB 3.0 (JSR 220), along with *Common Annotations for the Java Platform (JSR 250)*, define a basic set of annotations for resource injection and interception, as part of the Java EE 5.0 specification release. Java EE 5.0 supports a limited subset of Dependency Injection (DI) called *Resource injection* which provide allows injection of objects from the JNDI environment, such as other Java EE components.

Pitchfork is an Apache License open source project developed collaboratively by Interface21 and BEA Systems, as an add-on for Spring.

The aims of the Pitchfork project are twofold:

- *To provide a basis for implementation of these new features in Java EE 5.0 in existing application servers*, on the basis of Spring's powerful, extensible DI and AOP functionality. An important advantage for users of containers that take this route is that they can easily make use of Spring features that add value beyond the present state of the Java EE specification: examples include DI features such as constructor injection, injection of primitive and complex types; access to existing Spring configurations; ability to work easily with code that does not include Java EE 5.0 annotations; and access to a far more powerful and elegant AOP model.
- *To support Java EE 5.0 annotations inside the Spring container*, allowing classes authored to the EJB 3 and Java EE programming model to be reused with minimal changes (if any) inside Spring-managed applications. It is also possible to mix and match these elements of the Java EE 5.0 programming model with the richer and more powerful capabilities offered by Spring.

This project allows elements of the Java EE 5.0 programming model to be used in Spring; it is not a full implementation of the Java EE 5.0 specifications, nor is that its goal. It is possible that further annotations will be supported in future releases (depending on user feedback); however, while Java EE 5.0 servers may use this project in their implementation of the Java EE specifications, Pitchfork itself will not become a full JEE application server.

This support is used internally in WebLogic Server (since May, 2006) to implement resource injection across Java EE components, and resource injection and interception in EJB 3.0 and components. It is also usable outside the WebLogic platform, as a simple add-on to Spring, with no dependencies besides Spring itself. This project requires the version of Spring JAR it ships with, or Spring 2.0 RC1 or above.

To understand how this support works, consider the following *@Resource* annotation:

```
public class SomeBean {

 private DataSource myDB;

 @Resource(name="jdbc/myCustomDB")
 public void setMyDB(DataSource myDB)
 {
    this.myDB = myDB;
 }
...
}
```

This corresponds to an implicit (or explicit) Spring bean definition like this:

```
<beans>
```

```
 ...
 <bean id="myBean" class="...SomeBean">
   <property name="myOtherDB">
      <jee:jndi-lookup jndi-name="jdbc/myCustomDB" />
   </property>
 </bean>
...
</beans>
```

At the moment, Spring JEE support understands the following annotations, which are part of the JSR 250 and JSR 220:

- JSR 250 injection annotations (javax.annotation): @PostCostruct, @PreDestroy and @Resource
- EJB3 interception annotations (javax.interceptor): @AroundInvoke, @ExcludeClassInterceptors, @ExcludeDefaultInterceptors, @Interceptors, @Invocation
- EJB3 transaction annotations (javax.ejb): @Stateless, @ApplicationException and @TransactionAttribute

In short, Spring can understand Java EE 5.0 injection, interception and transactional metadata.

1.2. Bootstrapping

There are several ways to activate the Pitchfork Java EE 5.0 programming model support in Spring:

- Through specific *PostProcessor*s:

```
<beans>
  <bean class="org.springframework.jee.config.JeeBeanFactoryPostProcessor"/>
  <bean id="bean" class="org.springframework.jee.inject.InterceptedBean"/>
</beans>
```

  The *JeeBeanFactoryPostProcessor* will analyze all the beans declared by the bean factory in which it is declared and will add the appropriate injections and/or interceptions. Simply adding this post processor will change the behaviour of the Spring container overall and it is the recommended way to use Pitchfork. This is a common extension point that should be familiar to Spring users.

- By using *Bootstrap* or *EjbBootstrap* classes:

```
Bootstrap bootstrap = new Bootstrap();
ApplicationContext applicationContext = bootstrap.deploy();
Bean myBean = (Bean) applicationContext.getBean("myBeanName");
...
```

  -- or --

```
EjbBootstrap bootstrap = new EjbBootstrap();
String[] springLocations = new String[] { "classpath:org/springframework/jee/server/springExternal.xml" };
ComponentContributor contributor = new AnnotationComponentContributor(ejbClass);
ApplicationContext context = bootstrap.deploy(springLocations,
                                              new DefaultResourceLoader(), contributor, new DeploymentUnitMeta
...
```

  The Bootstrap classes are mainly used by containers that need control over the deploying process. They offer several methods for specifying what locations, resource loader (ex: classpath or filesystem based) and what deploymentUnit metadata to be used.

This style of use is not primarily intended for developers using Spring, but for those embedding this functionality within an existing container (such as WebLogic Server).

The `DeploymentUnitMetadata` represents a programmatic way of specifing EJB3 descriptor properties like default interceptors or application exceptions which complement the annotations. Applications that can understand EJB3 XML files can plug this information into the JEE support through this class.

Note: It is recommended that the *JeeBeanFactoryPostProcessor* is used since Pitchfork internals may change until a final release.

1.3. Specification support

While JSR 250 is straight forward and it be used in all application types, EJB3-style interception and transaction are just a part of the full JSR 220 that is usually implemented by application containers. Below are listed the existing issues and limitations of the project:

1.3.1. EJB 3.0 deployment descriptors

EJB deployment descriptors (optional XML configuration containing information beyond Java annotations) are not understood by default by Spring or this add-on project. Only annotation processing is performed out of the box. Spring's own metadata is both simpler and far more powerful than the EJB 3 XML metadata, so there would be little motivation for using this style of configuration in a Spring application.

However, through the `DeploymentUnitMetadata`, third parties can add information about the default interceptors or application exceptions. See the javadocs for more information on what properties can be set. Note that XML descriptors contain a lot of information which is important only to the application server and irelevant to Spring (like resource-env).

1.3.2. EJB 3.0-style interception

Ptichfork allows the use of EJB 3.0 style interceptors (annotated with `@AroundInvoke`) to be used in a Spring container. Note that the lifecyle of the interceptor is tightly bound to that of the bean it intercepts. Spring will fulfill this contract but will not support activate or passivate calls (specified by @PostActivate or @PreDestroy). However, hooks are provided so that the outer container which manages the bean lifespan, can inform Spring of these events. Default interceptors can be added programatically through the `DeploymentUnitMetadata` class.

It is possible to mix and match EJB 3 interception with both Spring AOP and Spring 2.0 @AspectJ style functionality. However, in almost all cases either of the latter programming models is wholly superior.

We recommend the @AspectJ programming model in general with Spring 2.0 and above: it is both more elegant and far more powerful than any interception style model. For example, it provides true pointcuts (the core concept of AOP); does not effectively force the use of annotations across a codebase to be used in conjuction with aspects; offers far greater potential for reuse; and offers type safety and robust access to parameters and return types through argument binding. Note that Spring 2.0 also offers an equivalent XML concise namespace schema. For further details about the @AspectJ programming model and XML pointcut expressions, see the AOP chapters of the Spring 2.0 Reference Manual.

1.3.3. EJB 3.0 style declarative transaction management

EJB3-style transactions are understood and applied using Spring transaction support. The application exceptions are parsed at runtime and, based on their annotation (@ApplicationException), the transaction will be commited or rolled back. Transactions are created for beans that are considered session beans: that is, which contain the @Stateless and @Stateful annotations. Again, the `DeploymentUnitMetadata` class can be used to add more application exceptions besides the ones declared already through annotations. Because Spring's transaction support is used under the covers, the EJB style transaction management model is thus supported in any environment: not just a JTA environment.

Note that the functionality provided by the EJB 3 @TransactionAttribute annotation is a subset of that offered by Spring's own Transactional annotation or other Spring metadata. In particular, the EJB 3.0 notion of annotating an exception, rather than a use case, to convey rollback information, is arguably flawed. We do not recommend that Spring users use this model by choice, but see it as a mechanism by which components that use this annotation can benefit from superior Spring functionality.

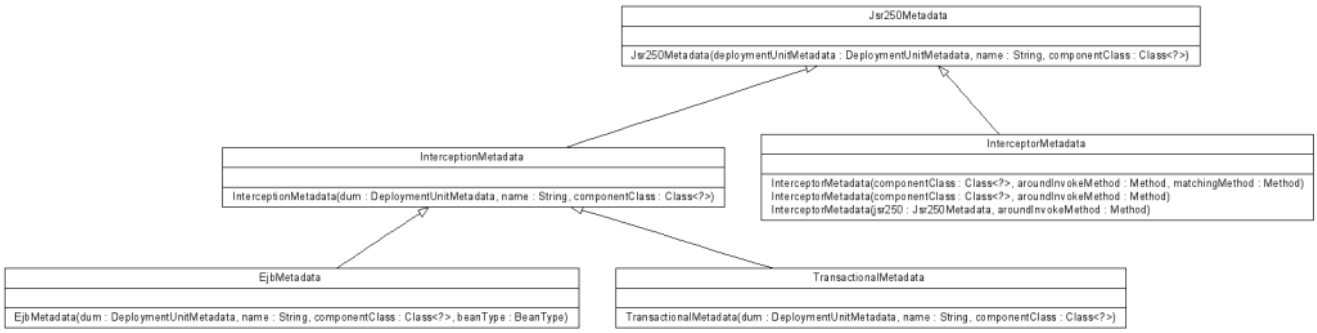
Spring's own transaction annotation support carries more information than EJB transaction metadata and also can support nested transactions on participating resource managers and per use case rollback rules.

1.4. Architecture

The architecture is based on the new ability in Spring 2.0 to attach arbitrary metadata to Spring bean definitions. This new extension point is combined with the existing extension point of a *bean post processor* (an object that can react to the instantiation of each bean in a Spring context).

1.4.1. Metadata

At the core of the Spring JEE project are the metadata classes. All bean definitions contained by Spring bean factory are passed through a chain of processors which, based on various information (usually annotations), create specific metadata that is later on used for applying the injection, creating the interception or applying transactional behavior:



It is important to note that metadata is attached to a Spring bean definition as a custom attribute (a feature of Spring 2.0). Each metadata class holds Java EE specific information regarding the bean definition it is attached to, as well as methods to apply it.

Jsr250Metadata is the base for the current metadata classes - it contains the injection and the lifecycle methods for constructing and destroying the objects along with references to the loading application context, the bean definition registry, the deployment unit metadata and the inspected bean class. It also contains hooks to invoke lifecycle methods on a class instance and apply injection.

IncerceptionMetadata extends the *Jsr250Metadata* and provides hooks for applying EJB3-style interception. It can handle applying the default interceptors, super interceptors and exclusion of interceptors (based on the

inheritance algorithm specified in JSR 250). It also contains hooks for allowing custom interceptors to be added in the weaving process.

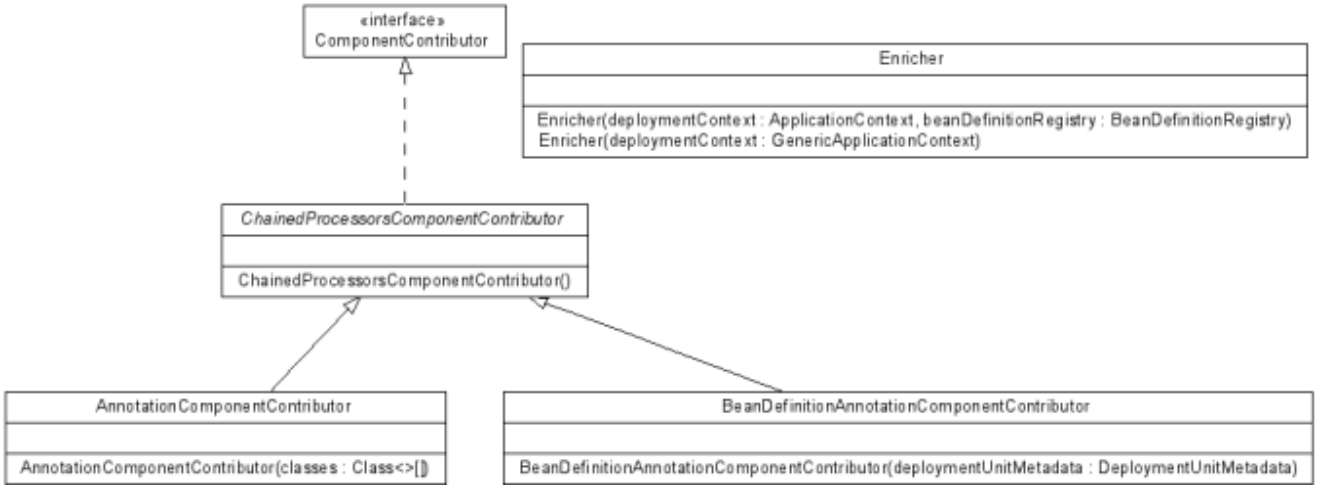
InterceptorMetadata contains information about interceptor beans since they can also be subject to interception and injection. *InterceptorMetadata* hold characteristics like being a default or class interceptor and are usually part of an *InterceptionMetadata* class.

TransactionMetadata is used for holding transactional behavior, like declared application exceptions or transactional methods and transactional attributes. It extends the *InterceptionMetadata* class only to add transactional behavior (through `TransactionInterceptor` and a customized `NameMatchTransactionAttributeSource`).

EjbMetadata is focused on EJB3 functionality but at this point, it is not yet used. (A WebLogic-specific subclass of `TransactionMetadata` is used internally in WebLogic Server.)

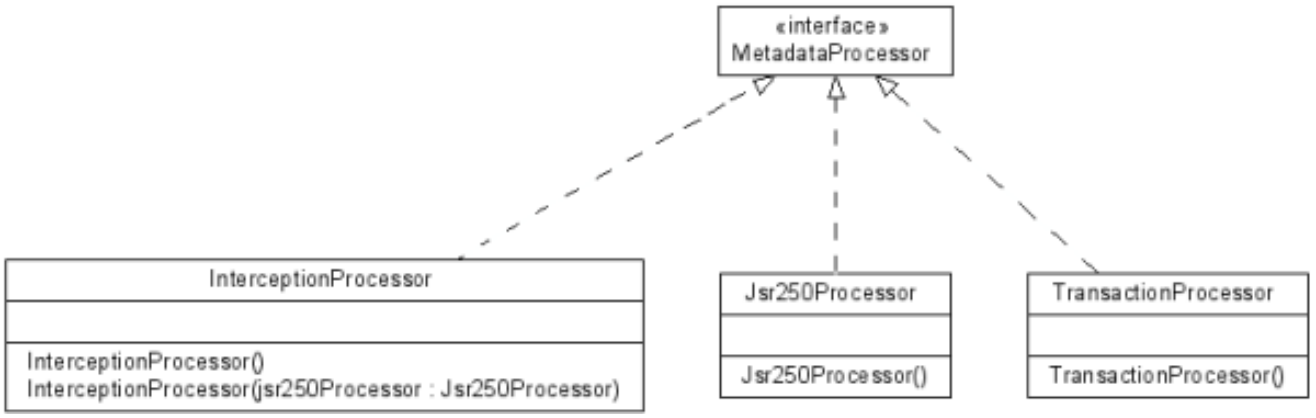
1.4.2. ComponentContributor

However, metadata has to be created after the bean definitions have been read, and later on applied. This is achieved through the *JeeBeanFactoryPostProcessor* which in return, relies on *ComponentContributor* to extract the metadata. *ComponentContributor* interface defines the contract for adding Jee metadata to an existing Spring context - by default, Bean DefinitionAnnotationComponentContributor is used. The process of binding the discovered metadata to the bean definition is done through the *Enricher* class which also validates the metadata (for example, verifies that stateless beans contain a business interface).



1.4.3. Metadata Processors

The discovery metadata process executes several processors which create a chain.



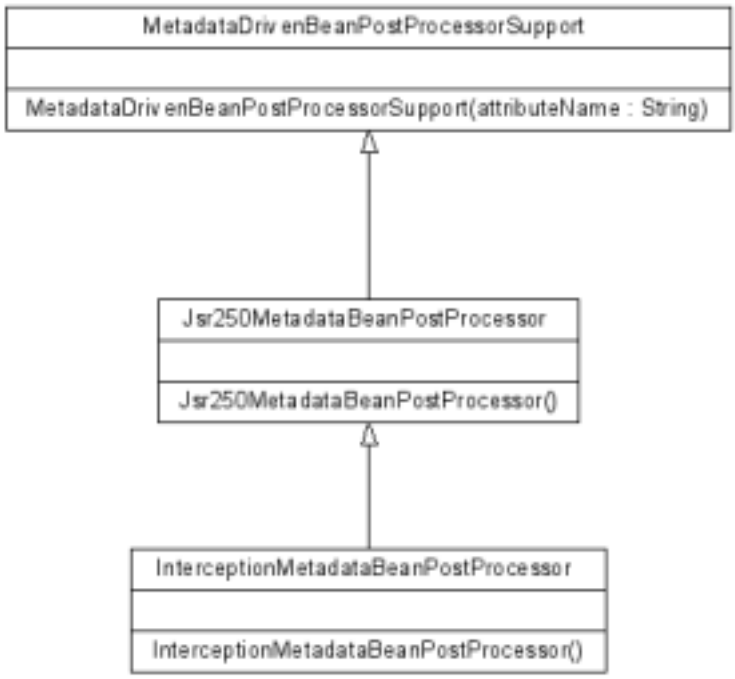
At the moment, there are three metadata processors available out of the box:

1. Jsr250Processor - which inspects the classes for the injection and lifecycle annotations defined by JSR 250
2. InterceptionProcessor - which handles the EJB3 interception annotations. Since the interceptors resulted can also have JSR 250 annotations, the InterceptionProcess uses internally a Jsr250Processor to inspect them.
3. TransactionProcessor - which is aware of the EJB3 @Stateful and @Stateless annotations as well as the transaction attributes.

These three processors are chained inside *ChainedProcessorContributor* and form the default processing chain. It is possible however, to pass a customized chain through *setMetadataProcessors(List<MetadataProcessor> processors)* method. The resulting metadatas are attached to the bean definition by the enricher and later processed.

1.4.4. Metadata PostProcessors

Metadata PostProcessors are used for translating the Jee metadata into injection, interceptors or transaction definitions:

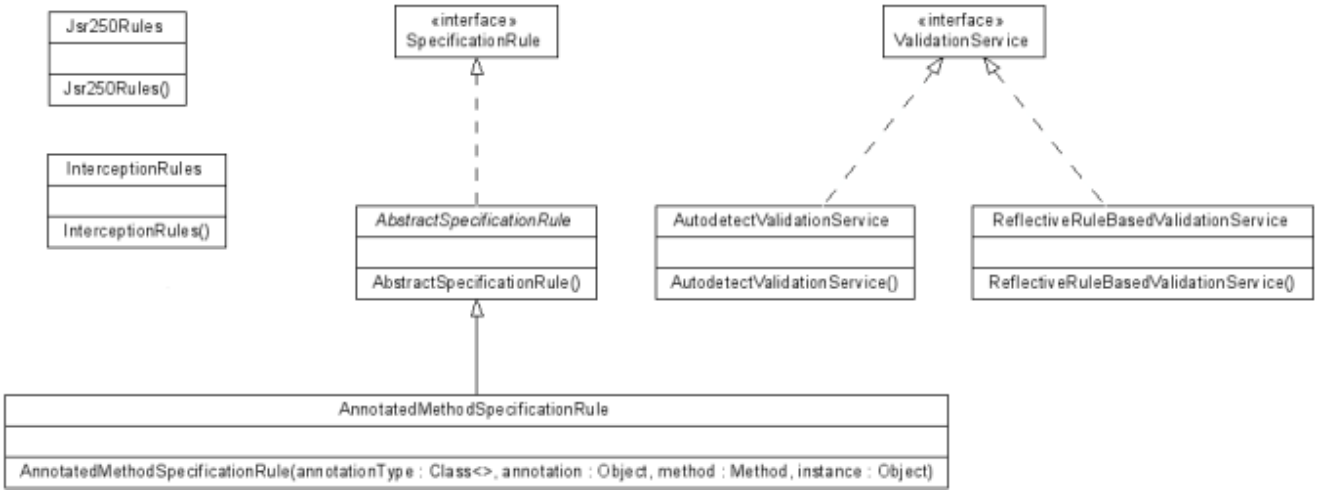


Jsr250MetadataBeanPostProcessor extracts *Jsr250Metadata* object from each bean definition and applies injection and hooks for construction and destruction, while *InterceptionMetadataBeanPostProcessor* takes care of interception. It is recommended that third parties, even when using composition, reuse the *MetadataDrivenBeanPostProcessorSupport* as a base class for reading bean definition metadata. *TransactionMetadata* does not require a special metadata PostProcessor since it is a subclass of *InterceptionMetadata* which is already has a Metadata PostProcessor.

1.4.5. Metadata Validation

Pitchfork defines an extensible mechanism for validation, meaning enforcement of rules in the relevant Java EE 5.0 specifications. This is designed to allow the addition and autodetection of specification rules and error messages unique to a particular host environment, such as an application server.

To execute metadata validation, the current package provides two interfaces:



1. ValidationService - which defines element that can execute validation and

2. SpecificationRule - which defines a validation rule and its relationship to the JSR specifications.

By default, the boostrap classes will use the *AutodetectValidationService* which looks for *@SpecificationRules* annotations which marks a component as a validation rule. Internally, *ReflectiveRuleBasedValidationService* is used to execute the validation method from the detected *SpecificationRule*. See the *InterceptionRules* and *Jsr250Rules* as examples for extending the out-of-the-box validation rules.