CDI --Context dependencies Inject上下文依赖注入，它是一种规范而不是一个具体的实现

一般JavaEE应用服务器中默认会提供CDI的实现，下面是如何在tomcat中使用CDI

The JSF specification was ahead of the pack when it came to working with and annotating *managed beans.* However, as the JSF 2.0 specification was getting wrapped up, another JSR specification, JSR 299, was being developed as well. Frustratingly, JSR 299, the Contexts and Dependency Injection (CDI) specification, provides a much more flexible and extendible implementation than the fairly limited JSF implementation. As a result, it's recommended that if you're developing a JSF application that will be depolyed to a Java EE 6 compliant applicaiton server, then you should be using CDI, and not the standard JSF annotations.

Having two standards and sets of annotations is pretty confusing. Gavin King stepped into the fray on the issue of why we have two specifications, and why one should be used instead of the other. I'd like to take the liberty to quote him on the topic:

|  |
| --- |
| Given that EJB and CDI provide arguably more convenient ways to identify a managed bean, you might wonder precisely what @ManagedBean is needed for. The answer, as alluded to by Dan, is that if you have CDI available in your environment (for example, if you are using EE6), then @ManagedBean is just not really needed. @ManagedBean is really there for use by people who are using JSF2 without CDI...  To summarize, if you do have CDI available to you, it provides a far superior programming model to the @ManagedBean/@ManagedProperty model that JSF2 inherits from JSF1. So superior, in fact, that the EE 6 web profile does not require support for @ManagedProperty etc. The idea being that you should just use CDI instead. |

I'm going to recreate the Rock-Paper-Scissors application and use CDI annotations instead of the JSF ones. To keep things separate, I'm actually going to copy my current, working, JSF application which is saved in a folder named \_easyjsf, and copy it all to a folder named \_easyweld. With that done, I'm going to edit the GameBean to make is Serializable, and to make it use the CDI annotations that correspond to the JSF annotations we've been using up to this point.

So, to redo our GameBean using CDI instead of JSF annotations is fairly simple as far as the coding goes.We simply replace @ManagedBean with @Named and we replace the JSF @SessionScope annotation with one from the javax.enterprise.context package.

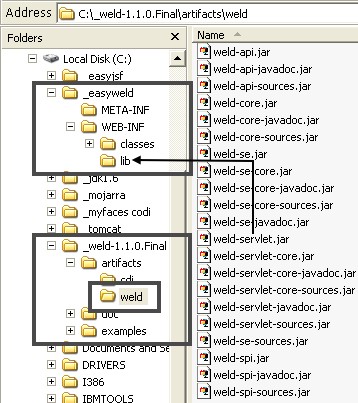
Note than any @Named annotated JavaBean must implement the java.io.Serializable interface. The JSF frameworks isn't quite as strict about serialization, but CDI and Weld is unrelenting.

package com.mcnz.jsf;   
 import javax.inject.Named;   
  import javax.enterprise.context.SessionScoped;   
**@javax.inject.Named   
@javax.enterprise.context.SessionScoped**   
public class GameBean**implements java.io.Serializable**

{ private String computerGesture = null; private String clientGesture = null; public void executeGameLogic(javax.faces.event.ActionEvent event)  
    { this.computerGesture = "rock"; /\* we always choose rock! \*/ }   
public String getResult()   
    { String result = "error"; if (clientGesture != null)  
         { if (clientGesture.equals("paper"))   
              { result = "win"; }   
if (clientGesture.equals("scissors"))   
              { result = "loss"; }   
if (clientGesture.equals("rock"))   
              { result = "draw"; }   
         }   
return result;   
    }

public void reset(javax.faces.event.ActionEvent event)  
    { computerGesture = null; }   
public String getClientGesture()   
    { return clientGesture; }   
public void setClientGesture(String clientGesture)   
    { this.clientGesture = clientGesture; }   
public String getComputerGesture()   
    { return computerGesture; }   
public void setComputerGesture(String computerGesture)   
    { this.computerGesture = computerGesture; }   
}

Now, if you were using a Java EE 6 compliant application server then you could deploy this immediately and it would all work. Unfortuntately, CDI support doesn't come with a standard servlet engine like Tomcat 7, so if you want to deploy a JSF application that leverages CDI annotations, well, there's a little bit of work to do.  
  
First, you need to get an implementation of the CDI specification. The reference implementation of JSR 299 is known as Weld, and can be downloaded from the Seam Framework website:  
  
<http://seamframework.org/Weld/>   
  
When the download is extracted, you'll find an \artifacts\weld subfolder within the distribution, and that weld folder includes a file named weld-servlet.jar that needs to go into the WEB-INF\lib folder of your JSF application. This weld-servlet.jar includes every Java class you need to link to at runtime in order to turn Tomcat 7 into a functional CDI compliant container.



Sadly though, it's not good enough to simply add the weld-servlet.jar file to the lib directory of your application. You also need to tell the web application hosting your JSF code that it's going to be leveraging some CDI functionality, and as such, a listener entry must go inside the web.xml file:

 <listener> <listener-class>org.jboss.weld.environment.servlet.Listener</listener-class> </listener>

Here's what the full faces-config.xml file looks like once the listener entry has been added:

   <?xml version='1.0' encoding='UTF-8'?>   
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<!-- Faces Servlet -->   
<servlet>   
  <servlet-name>Faces Servlet</servlet-name>   
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>   
</servlet>

<!-- Faces Servlet Mapping -->   
<servlet-mapping>   
  <servlet-name>Faces Servlet</servlet-name>   
  <url-pattern>\*.faces</url-pattern>   
</servlet-mapping>

**<listener>   
  <listener-class>org.jboss.weld.environment.servlet.Listener</listener-class>   
</listener>**   
</web-app>

And just so you know, if you mess up the listener entry in the web.xml file, you'll get an error message that looks something like this at runtime:

 Unable to find BeanManager for org.apache.catalina.core.ApplicationContextFacade

And finally, if you want everything to work, you need to add in an empty beans.xml file alongside the web.xml and faces-config.xml file in the WEB-INF folder of the application.

 <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans\_1\_0.xsd"> </beans>

It may seem rather odd and redundant to include an xml file that really doesn't contain anything of significance, but without it, you'll get the following runtime error:

 java.lang.NullPointerException org.jboss.weld.context.ForwardingContextual.toString(ForwardingContextual.java)

or perhaps even this one:

 Target Unreachable, identifier resolved to null

With all of that completed, my web application looks something like this:

|  |  |
| --- | --- |
| https://lh5.googleusercontent.com/_41A-R4AR9qM/TU2WUXxWutI/AAAAAAAAAwc/C2IGr3DO5FA/s800/cdi%20jsf.jpg | \_easyweld\  + META-INF\ + WEB-INF\  ++ beans.xml ++ web.xml ++ faces-config.xml  ++ classes\ +++ com\mcnz\jsf\GameBean.java +++ com\mcnz\jsf\GameBean.class  ++ lib\ +++ jsf-api.jar +++ jsf-impl.jar +++ jstl.jar +++ standard.jar +++ weld-servlet.jar |

From here, you can compile your code with the following command:

**使用示例**

Java EE 6 provides an easy-to-use, standards based, convention-over-configuration, enterprise component model.

Java EE 6 also provides a new dependeny injection, annotation processing and interception system which is better than classic Spring. Some developers who currently use Spring will want to migrate to Java EE 6. This is an easy proposition for greenfield applications. Not all developers who want to migrate will be able to do it in one pass. Some developers will need to combine Spring and CDI. Other developer’s may want to combine Java EE 6 and CDI with Spring based modules and libraries. The first article focuses on using CDI beans in a Spring.application context. The second article focuses using Spring managed beans in CDI.

EJB 3.1 and CDI is a productive convention over configuration model. A lot of people will use EJB 3.1 and CDI because it is the standard. Even if you decide to adopt EJB 3.1 and CDI, you still may need to integrate with 3rd party libraries that use Spring or Spring modules. Thus, Spring integration might be a semi-permanent part of your architecture. Before we go into Spring / CDI integration let’s cover a little bit why CDI is a good option, and why I think you should use it as your primary DI/interception framework.

### Problems with Spring and Improvements to Java EE

Spring was born before annotations in the age of XML. Many projects end up with very large XML files that are difficult to maintain. Spring 2.5 through Spring 3 added annotations but their use is not as widespread or as well understood as Spring’s XML configuration. The Spring annotations are added on top of the existing Spring DI system in what some could argue a non-type safe injection.

Guice created a DI system based on strongly typed annotations. Guice is a well designed strongly typed DI system.

Java EE 6, based on ideas from Guice and Spring, improves further and creates an easier to use, general purpose, strongly typed DI system called Contexts and Dependency Injection (CDI). CDI clears up issues with interceptors in Java EE 5 as well.

One of the major advantages of Java EE 6 over Spring is there is less configuration and less moving parts. Java EE 6 usability surpasses Spring 3 and its testability comparable to Spring 3′s.

Spring is a collection of good ideas and trendy techniques some of which have gone out of style. A big part of Spring 3 was pruning some of these evolutionary dead ends out of the code base. However, many people who use Spring use later versions of Spring but in a style of Spring so to speak from cira 2004, i.e., XML.

It is not enough for a team to say they use Spring. What do they use from Spring? Which parts do they use? Do they use custom classloaders? Do they use classpath scanning? Do they use AspectJ weaving? Do they use a combination of approaches? Do they write there own aspects? If so do they use traditional Spring AOP or AspectJ AOP? This makes integrating Spring with CDI a bit complicated as it is hard to tell exactly what a person means when they say they use Spring.

The complexity and the maintainability of configuring Spring is complicated because of its build-your-own container approach to enterprise development.

Roo, a utility from SpringSource to generate best-practice Spring 2.5 and Spring 3.0 application, on the other hand relies heavily on AspectJ and annotations although AspectJ is not widely adopted and there is little AspectJ expertise in the industry. There is the Spring reality (XML) and the Spring vision of the future (Annotations, Roo and AspectJ). Any integration of Spring and CDI would need to take this into account.



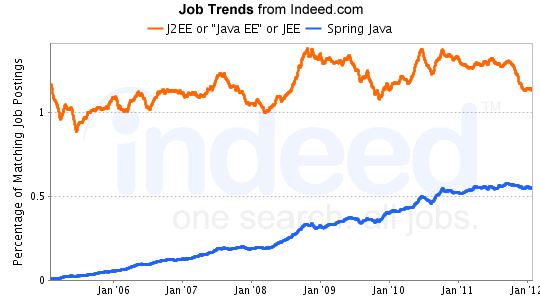
From the graph above, you might come to the conclusion that Spring’s vision via Roo with regards to domain objects, annotations and AspectJ is different from Spring’s user base, which it appears is primarily using XML. For those Spring users who are still using XML, before switching to Spring’s new vision, they may want to first check out standards like Java EE 6, CDI and EJB 3.1 which provide a very productive convention over configuration environment.

Spring’s annotation support for injection is not strongly typed like CDI and Guice. It’s annotations rely heavily on names to lookup other objects for many cases. It is an extension of the existing Spring IOC container which was written before annotations existed and before classpath scanning for annotations was the norm. Integrating CDI and Spring would need to bridge these gaps.

### Integrating Spring and Java EE

Due to the improved useabiltty (type safe injection and convention over configuration) of CDI, a key part of Java EE 6 and the fact that it is the standard. It is very likely that many development shops will adopt Java EE 6 and CDI in some form. Thus, the focus of this two part article series.

Since Spring is more than just an DI container. It is a set of utility classes and modules. Java EE 6 scope is in a similar but different focus. While CDI scope is even more focused. Even if you decide to use CDI and Java EE 6, it is likely they you might need a capability that is part of the Spring ecosystem. Therefore, you will likely need to integrate CDI and Spring. Again referring to the first graph, traditional Spring is in wide use, however, Java EE use is even greater. Compare to the following graph.



Again although Spring use is wide, it is not as wide as Java EE.

For some application, you can replace Spring with Java EE 6 and CDI all the way. This will give you much simpler configuration. Then there are others who need one of the many modules at SpringSource which all not so strangely rely on Spring. Then there are third party frameworks that also use Spring. Even if you are a Java EE shop, it is very likely that you will need to integrate with Spring at some level. Then there are those who can’t afford to switch over whole hog to a new framework, and must do it piecemeal.

Currently, for example, there is no CDI enabled Model 2 style framework like Spring MVC. Not all developers agree with the JSF framework approach. Therefore, there may be a need and desire to integrate Spring MVC on the front end while using Java EE 6 and CDI for the backend.

For other application you may want to start to use Java EE 6 and CDI for some new use cases while keeping Spring for other use cases as you start to adopt the more productive CDI and Java EE features into your development stack. This would be a gradual migration.

Let’s briefly cover how one might integrate Spring and CDI in the same application. We will take two applications, one a Java EE 6 application and the other a modified project that Roo generated (I stripped out the domain objects and the AspectJ mixins).

### Java EE 6 and Spring 3 with annotations more similar than not

Spring due to project pitchfork closely aligns to the style of Java EE 5 in many respects.

For an example of this let’s compare some database access objects written using Spring and EJB. Look at the EJB 3.1 and CDI version of a database access object.

#### TaskRepository class

...

import javax.ejb.Stateless;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

@Stateless // <1>

public class TaskRepository {

@PersistenceContext // <2>

private EntityManager entityManager;

public void persist(Task task) {

this.entityManager.persist(task);

}

...

**<1>** Identify this class as an EJB that has transaction support **<2>** Inject the persistence context from JPA into the EJB

Now without blinking, look at the Spring version of the same database access object using Spring.

##### TaskRepository class

...

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;

import org.springframework.transaction.annotation.Transactional;

@Repository // <1>

@Transactional // <2>

public class TaskRepository {

@PersistenceContext

private EntityManager entityManager; // <3>

public void persist(Task task) {

this.entityManager.persist(task);

}

...

**<1>** Identify this class as Spring bean (must be registered with Spring) **<2>** Mark all of the methods as transactional **<3>** Inject the persistence context from JPA into the Spring bean

The annotations are somewhat functionally equivalent. At first blush, there is no real difference between these. However one of the key differences is the way the EntityManager, which is stateful, is handled and the underlying design of EJBs and Spring. In the EJB world, the entityManager is injected before this EJB is used, and the container worries about the underlying thread and transaction isolation. In the Spring world, the entityManager is a proxy object to an entityManager that must be looked up on every method call to the entityManager (sometimes in map associated with a thread local varaible, sometimes as a JTA transaction resource depending on your configuration). The EJB 3 approach is more efficient and consistent in addition to being the standard.

Also, in the Spring version, you would need a few extra applicationContext.xml files and a few entries in a web.xml file. Here is the applicationContext.xml file to use the Spring version of the database access object.

#### applicationContext.xml

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/schema/context"

xmlns:jee="http://www.springframework.org/schema/jee" xmlns:tx="http://www.springframework.org/schema/tx"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.0.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

<context:property-placeholder location="classpath\*:META-INF/spring/\*.properties" />

<context:component-scan base-package="org.**cdisource**.springapp"> <!-- <1> -->

<context:exclude-filter expression="org.springframework.stereotype.Controller"

type="annotation" />

</context:component-scan>

<!-- <2> -->

<bean class="org.springframework.orm.jpa.JpaTransactionManager"

id="transactionManager">

<property name="entityManagerFactory" ref="entityManagerFactory" />

</bean>

<!-- <3> -->

<tx:annotation-driven mode="aspectj"

transaction-manager="transactionManager" />

<!-- <4> -->

<jee:jndi-lookup id="dataSource" jndi-name="jdbc/basic" />

<!-- <5> -->

<bean

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"

id="entityManagerFactory">

<property name="dataSource" ref="dataSource" />

</bean>

</beans>

**<1>** Look for @Repository, and @Service stereotypes and add them to the application context. **<2>** Configure transaction manager for JPA **<3>** Create an annotation driven transaction support using AspectJ **<4>** Lookup a JNDI datasource **<5>** Configure the EntityManagerFactory

In the Spring version you have to essentially configure your own container. You have many options. Do you want to use a custom Spring classloader? Do you want to use Spring AOP/AspectJ code weaving? etc. Every shop uses some subset of what is available in the Spring universe for solving even basic issues like transaction demarcation of database access object methods. There are many ways to solve the same problem. This is good but can be bad as well. It means if you are hiring someone off the street to do development with knowledge of Spring that it is unlikely that you will find two people using Spring the same way. Are they using the new annotations? Are they using the AspectJ integration? In the last three consulting engagements I was on where folks used Spring, they were all using Spring XML from 2004. There may just be too many options.

To inject the TaskRepository bean into a Spring MVC controller we could use @AutoWire (just like before).

##### Injecting a TaskRepository into a TaskController using Spring.

package org.**cdisource**.springapp.web;

...

import org.**cdisource**.springapp.TaskRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

...

@RequestMapping("/tasks")

@Controller

public class TaskController {

@Autowired // <1>

private TaskRepository repo;

...

**<1>** Injecting a TaskRepository into a Spring MVC controller using Spring. This is similar to injecting a CDI bean into a JSF backing bean as shown below.

##### Injecting a TaskRepository into a TaskController using CDI.

package org.**cdisource**.springapp.task;

import java.io.Serializable;

import javax.inject.Inject;

import javax.inject.Named;

...

@Named("taskHome")

@ConversationScoped

public class TaskHome implements Serializable {

...

@Inject

private TaskRepository repo; // <1>

...

**<1>** Injecting a TaskRepository into a JavaServer Faces (JSF) Backing bean using CDI.

For simple cases like these @Inject from CDI and Spring’s @Autowired are fairly similar. The real difference in this example is that one uses Spring MVC and one uses JavaServer Faces. The reality is that not all people use JavaServer Faces, and Java EE 6 supports non JavaServer Faces applications using JavaServer Pages (JSP) and Servlets. If you are building an application using JSP and Servlets then Spring MVC is a popular option. In addition, Spring is a popular option for integration with many other web frameworks like Tapestry, Wicket, Struts and more.

The question in many developer’s mind is can I use CDI with Spring MVC, Tapestry, Wicket, Struts and more? The answer is yes. You don’t have to wait. If you are already using Spring, then integrating CDI is quite easy with all of these frameworks and we will show you how to do it in this article. (Note there is also a CDI plugin for Struts 2.)

### Working CDI into the mix

One easy way to work CDI into the mix is to use a Java EE 6 container and use the EJB and CDI support with JPA. It is an understatement to say that EJB, CDI and JPA work well together. You see JPA is part of EJB. JPA defines the way modern entity beans are written. If you are like me, you may cringe at the name EJB. Well, EJB is not EJB of yore. EJB + CDI + JPA is a powerful combination of POJO productivity. The past sins of EJB have been fixed with EJB 3.1, CDI and JPA. Let’s look at how easy it is to integrate a true EJB into a Spring application using CDI.

In order to bridge the TaskRepository into Spring you could use a org.springframework.beans.factory.FactoryBean. A FactoryBean is used to bridge from Spring into other object systems. One famous FactoryBean that ships with the Spring framework is the JndiObjectFactoryBean. The JndiObjectFactoryBean uses JNDI to lookup a bean and map it into Spring’s application context, thus bridging the gap between JNDI and Spring. Let’s create a factory bean that get’s a TaskRepository object from CDI and maps it as an injectable bean in the Spring world.

##### TaskRepositoryFactoryBean

package org.**cdisource**.springapp;

import javax.enterprise.inject.spi.BeanManager; // <2>

import org.springframework.beans.factory.FactoryBean;

import org.springframework.beans.factory.InitializingBean;

import org.**cdisource**.beancontainer.BeanContainer; // <3>

import org.**cdisource**.beancontainer.BeanContainerImpl; // <3>

public class TaskRepositoryFactoryBean implements FactoryBean<TaskRepository>, InitializingBean { // <1>

private BeanContainer beanContainer; // <3>

private BeanManager beanManager; // <2>

public void setBeanManager(BeanManager beanManager) {

this.beanManager = beanManager;

}

@Override

public void afterPropertiesSet() throws Exception {

// <3>

beanContainer = new BeanContainerImpl(beanManager);

}

@Override

public TaskRepository getObject() throws Exception {

// <4>

return beanContainer.getBeanByType(TaskRepository.class);

}

@Override

public Class<?> getObjectType() {

return TaskRepository.class;

}

@Override

public boolean isSingleton() {

return true;

}

}

**<1>** TaskRepositoryFactoryBean is both a FactoryBean and an InitializingBean which means in Spring parlance a bean that can create a bean and a bean that will be notified after all of its members have been injected. **<2>** The BeanManager is the main interface from CDI used to lookup beans. **<3>** Since the BeanManager is a bit difficult to use (very low level), a bumper crop of support open source support APIs including Weld Solder and **CDISource** have cropped up. TheBeanContainer is from the **CDISource** project (both listed in the reference section of the article). **<4>** In the getObject method of the FactoryBean we look up the TaskRepository in CDI. Spring calls this method to lookup the bean for injection. The getObject method is from the FactoryBean interface. Since you may or may not ever use **CDISource** or Weld Solder, let’s briefly cover how one would look up a bean in plain CDI by showing you what the getBeanByType method looks like.

##### getBeanByType possible implementation, (with removed NPE checks for brevity)

@Override

public <T> T getBeanByType(Class<T> type, Annotation... qualifiers) {

BeanManager beanManager = ...

Set<Bean<?>> beans = beanManager.getBeans(type, qualifiers);

if (beans.isEmpty()) {

throw new RuntimeException("Could not locate a bean of type "

+ type.getName());

}

Bean<?> bean = beanManager.resolve(beans);

CreationalContext<?> context = beanManager

.createCreationalContext(bean);

@SuppressWarnings("unchecked")

T result = (T) beanManager.getReference(bean, bean.getBeanClass(),

context);

return result;

}

When you use CDI in a Java EE application, you never have to work at this level of the CDI code. However, when you are trying to get Spring to play nice with CDI, it makes sense. (You can download and use this example, see the resources section below).

This technique would not just work for EJB, it would work for any CDI managed bean you wanted to inject inside of Spring application context.

Now that we have integrated CDI with Spring, we can rip out all of the “configure your own container” stuff out of the application context and let the Java EE 6 container do what is was built to do as follows:

##### simplified application context.xml after letting EJB CDI do its thing

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<beans xmlns="http://www.springframework.org/schema/beans" xmlns:jee="http://www.springframework.org/schema/jee"

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-3.0.xsd

http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">

<bean class="org.**cdisource**.springapp.TaskRepositoryFactoryBean" name="taskRespository"> <!-- <1> -->

<property name="beanManager"><jee:jndi-lookup jndi-name="java:comp/BeanManager"/></property> <!-- <2> -->

</bean>

</beans>

**<1>** Configure our TaskRepositoryFactoryBean. **<2>** Use JNDI to lookup the BeanManager.

Ok. That is more like it. We went from forty lines of XML to 10. In most people’s book that is a good. The code for the Spring controller would be the same. As you can see, Java EE 6 is much easier to configure.

### Working CDI into the mix

Now one problem we have with this approach is that every time you want to introduce a new CDI bean into Spring, you need to create a new FactoryBean. It would be better to have a generic CDI FactoryBean. One attempt would look like this:

#### CdiFactoryBean, generic way to inject beans into Spring

package org.**cdisource**.springintegration;

import javax.enterprise.inject.spi.BeanManager;

import org.springframework.beans.factory.FactoryBean;

import org.springframework.beans.factory.InitializingBean;

import org.**cdisource**.beancontainer.BeanContainer;

import org.**cdisource**.beancontainer.BeanContainerImpl;

public class CdiFactoryBean implements FactoryBean<Object>, InitializingBean {

private boolean singleton = true;

private BeanManager beanManager; // <1>

private Class<?> beanClass; // <2>

private BeanContainer beanContainer; // <3>

@Override

public void afterPropertiesSet() throws Exception {

beanContainer = new BeanContainerImpl(beanManager);

}

public void setBeanClass(Class<?> beanClass) {

this.beanClass = beanClass;

}

@Override

public Object getObject() throws Exception {

return beanContainer.getBeanByType(beanClass);

}

@Override

public Class<?> getObjectType() {

return beanClass;

}

@Override

public boolean isSingleton() {

return singleton;

}

public void setSingleton(boolean singleton) {

this.singleton = singleton;

}

public void setBeanManager(BeanManager beanManager) {

this.beanManager = beanManager;

}

}

**<1>** The beanManager is injected via JNDI. **<2>** The beanClass is the type that we are creating. **<3>** The beanContainer is the utility class we spoke of earlier to simplfy the CDI BeanManager access. Now when we are configuring the application context we do the following:

##### simplified application context.xml after letting EJB CDI do its thing

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<beans xmlns="http://www.springframework.org/schema/beans" xmlns:jee="http://www.springframework.org/schema/jee"

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-3.0.xsd

http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">

<bean class="org.**cdisource**.springintegration.CdiFactoryBean" name="taskRespository" >

<property name="beanClass" value="org.**cdisource**.springapp.TaskRepository"/> <!-- <1> -->

<property name="beanManager"><jee:jndi-lookup jndi-name="java:comp/BeanManager"/></property>

</bean>

</beans>

**<1>**

We just have one extra property called beanClass, which tells Spring which class our FactoryBean is creating. Now we have a generic way to bridge the Spring and CDI worlds. But, now we would have to add a new entry every time we want to create the bridge to the CDI world for a new bean. If you have one CDI bean to map to Spring, no problem. If you have 40 or more, well who wants to write that much XML. If only there was some way to look up all of the beans in CDI and automatically map them into Spring. Well there is.

Spring provides an extension mechanism to add beans directly into an application context. We could scan all of the CDI beans, and map them into Spring using a Spring BeanFactoryPostProcessor. To do this we will need to programmatically create Spring bean definition based on CDI bean objects as follows:

##### Bridging the gap from CDI beans into the Spring world, every bean using a BeanFactoryPostProcessor

package org.**cdisource**.springintegration;

import java.util.Set;

import javax.enterprise.inject.spi.Bean;

import javax.inject.Named;

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.config.BeanFactoryPostProcessor;

import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;

public class CdiBeanFactoryPostProcessor implements BeanFactoryPostProcessor { // <1>

...

private BeanManagerLocationUtil beanManagerLocationUtil = new BeanManagerLocationUtil();

@Override

public void postProcessBeanFactory(

ConfigurableListableBeanFactory beanFactory) throws BeansException {

DefaultListableBeanFactory factory = (DefaultListableBeanFactory) beanFactory;

Set<Bean<?>> beans = beanManagerLocationUtil.beanManager().getBeans(Object.class); // <2>

for (Bean<?> bean : beans) {

if (bean.getName()!=null && bean.getName().equals("Spring Injection")){

continue;

}

// <3>

BeanDefinitionBuilder definition = BeanDefinitionBuilder.rootBeanDefinition(CdiFactoryBean.class)

.addPropertyValue("beanClass", bean.getBeanClass()) // <4>

.addPropertyValue("beanManager", beanManagerLocationUtil.beanManager()) // <5>

.setLazyInit(true);

String name = generateName(bean); // <6>

factory.registerBeanDefinition(name, definition.getBeanDefinition());

}

}

private String generateName(Bean<?> bean) {

... // <7>

}

}

**<1>**

CdiBeanFactoryPostProcessor is a BeanFactoryPostProcessor which is to say that it is a Spring extension designed to work with bean definitions from Spring after Spring initializes the application context and before it starts creating actual beans.

**<2>** beanManagerLocationUtil is a helper object used to look up the CDI beanManager (typically in JNDI). **<3>** For every CDI bean (javax.enterprise.inject.spi.Bean) create a Spring BeanDefinition. Note we are defining a bunch of CdiFactoryBean.class bean definition entries. **<4>** Use the CDI bean’s class (javax.enterprise.inject.spi.Bean.beanClass) property to populate the CdiFactoryBean beanClass. **<5>** Use the beanManagerLocationUtil.beanManager() to populate the CdiFactoryBean.beanManager. **<6>** We generate a nice unique bean name based on the bean definition to make things easier to debug. <7> The nice unique bean name is based on the beanClass long or short name. The application context is even simpler now.

##### simplified application context.xml

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<beans xmlns="http://www.springframework.org/schema/beans" xmlns:jee="http://www.springframework.org/schema/jee"

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-3.0.xsd

http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">

<bean class="org.**cdisource**.springintegration.CdiBeanFactoryPostProcessor"/>

</beans>

Now all CDI beans are in the Spring application context.

### Conclusion

Spring was built to solve problems with J2EE. Spring borrowed ideas from J2EE and other projects and made them easier to use and more testable. Spring was a great stop gap for many architects and developers. Java EE 5 and Java EE 6 learned from the examples of Spring, Guice and other like projects. Spring learned from the examples of Java EE 5 & 6 and Guice. Now Java EE 6 has a CDI DI mechanism that rivals Spring’s except it is even easier to use and it is type safe (less error prone). That said, Spring is still a thriving community and has tons of ideas and energy and new modules and projects.

In addition many other useful projects also use Spring. The task going forward for Java EE 6 is how do you combine the ease of use of CDI with the utility of Spring, and the answer is CDI/Spring integration.

This article covered the first part of CDI/Spring integration which is how do you map CDI beans into Spring. The next article will cover the second part, which is how do you map Spring beans into CDI. In the next article we cover building a custom extension to map Spring beans into CDI.

There are 12 code listings in this article

#### References

You can download and install the complete CDI Spring Integration. Just go to [**CDISource** source code](https://github.com/CDISource/cdisource). Use git to check out the code. Then do a mvn clean install (maven). We plan on putting **CDISource** into the maven repo at some point.

A quick and easy description on getting started with CDI Using CDI from Spring by Harald Wellmann. You can also find entries in Rick Hightower’s blog about this at [CDI and Spring living in harmony](http://rick-hightower.blogspot.com/2011/04/cdi-and-spring-living-in-harmony.html),  
and [Spring meet CDI, CDI meet Spring](http://rick-hightower.blogspot.com/2011/04/spring-meet-cdi-cdi-meet-spring.html). There is a lot of background information about this on Rick’s blog. Just look at the blog in 03/2011 and 04/2011.

Posted by [rickhightower](http://cdisource.org/site/author/rickhightower/) at 11:52 pm

## [CDI AOP Tutorial](http://cdisource.org/site/2011/05/cdi-aop-tutorial/)

[Article](http://cdisource.org/site/category/article/), [Tutorial](http://cdisource.org/site/category/tutorial/) [No Responses »](http://cdisource.org/site/2011/05/cdi-aop-tutorial/#respond)

May 232011

# CDI based AOP – An Introductory Tutorial

## Introduction

This article discusses CDI based AOP in a tutorial format. [CDI](http://jcp.org/aboutJava/communityprocess/final/jsr299/index.html) is the Java standard for dependency injection (DI) and interception (AOP). It is evident from the popularity of DI and AOP that Java needs to address DI and AOP so that it can build other standards on top of it. DI and AOP are the foundation of many Java frameworks.

CDI is a foundational aspect of Java EE 6. It is or will be shortly supported by [Caucho’s Resin](http://www.caucho.com/resin/), IBM’s !WebSphere, [Oracle’s Glassfish](http://glassfish.java.net/), [Red Hat’s JBoss](http://www.jboss.org/jbossas/docs/6-x.html)and many more application servers. CDI is similar to core Spring and Guice frameworks. Like JPA did for ORM, CDI simplifies and sanitizes the API for DI and AOP. If you have worked with Spring or Guice, you will find CDI easy to use and easy to learn. If you are new to AOP, then CDI is an easy on ramp for picking up AOP quickly, as it uses a small subset of what AOP provides. CDI based AOP is simpler to use and learn.

One can argue that CDI only implements a small part of AOP that is method interception. And while that is a small part of what AOP has to offer, it is also the part that most developers use.

CDI can be used standalone and can be embedded into any application.

[Source code for this tutorial](http://jee6-cdi.googlecode.com/svn/tutorial/cdi-aop-example), and [instructions](http://code.google.com/p/jee6-cdi/wiki/MavenAOPTutorialInstructions) for use. It is no accident that this tutorial follows many of the same examples in the [Spring 2.5 AOP tutorial](http://java.dzone.com/articles/introduction-spring-aop) written three years ago.

It will be interesting to compare and contrast the examples in this tutorial with the one written three years ago for Spring based AOP.

## Design goals of this tutorial

This tutorial is meant to be a description and explanation of AOP in CDI without the clutter of EJB 3.1 or JSF. There are already plenty of tutorials that cover EJB 3.1 and JSF (and CDI).

We believe that CDI has merit on its own outside of the EJB and JSF space. This tutorial only covers CDI. Repeat there is no JSF 2 or EJB 3.1 in this tutorial. There are plenty of articles and tutorials that cover using CDI as part of a larger [JEE 6 application](http://download.oracle.com/javaee/6/tutorial/doc/gjbnr.html). This tutorial is not that. This tutorial series is CDI and only CDI.

This tutorial only has full, complete code examples with source code you can download and try out on your own. There are no code snippets where you can’t figure out where in the code you are suppose to be.

So far these tutorials have been well recieved and we got a lot of feedback. There appears to be a lot of interest in the CDI standard. Thanks for reading and thanks for your comments and participation so far.

## AOP Basics

For some, AOP seems like voodoo magic. For others, AOP seems like a cureall. For now, let’s just say that AOP is a tool that you want in your developer toolbox. It can make seemingly impossible things easy. Aagin, when we talk about AOP in CDI, we are really talking about interception which is a small but very useful part of AOP. For brevity, I am going to refer to interception as AOP.

The first time that I used AOP was with Spring’s transaction management support. I did not realize I was using AOP. I just knew Spring could apply EJB-style declarative transaction management to POJOs. It was probably three to six months before I realized that I was using was Spring’s AOP support. The Spring framework truly brought AOP out of the esoteric closet into the main stream light of day. CDI brings these concepts into the JSR standards where other Java standards can build on top of CDI.

You can think of AOP as a way to apply services (called cross-cutting concerns) to objects. AOP encompasses more than this, but this is where it gets used mostly in the main stream.

I’ve using AOP to apply caching services, transaction management, resource management, etc. to any number of objects in an application. I am currently working with a team of folks on the CDI implementation for the revived JSR-107 JCache. AOP is not a panacea, but it certainly fits a lot of otherwise difficult use cases.

You can think of AOP as a dynamic decorator design pattern. The decorator pattern allows additional behavior to be added to an existing class by wrapping the original class and duplicating its interface and then delegating to the original. See this article [decorator pattern](http://en.wikipedia.org/wiki/Decorator_pattern) for more detail about the decorator design pattern. (Notice in addition to supporting AOP style interception CDI also supports actual decorators, which are not covered in this article.)

## Sample application revisited

For this introduction to AOP, let’s take a simple example, let’s apply security services to our Automated Teller Machine example from the first the firstin this series.

Let’s say when a user logs into a system that a **SecurityToken** is created that carries the user’s credentials and before methods on objects get invoked, we want to check to see if the user has credentials to invoke these methods. For review, let’s look at the **AutomatedTellerMachine** interface.

##### Code Listing: AutomatedTellerMachine interface

package org.cdi.advocacy;

import java.math.BigDecimal;

public interface AutomatedTellerMachine {

public abstract void deposit(BigDecimal bd);

public abstract void withdraw(BigDecimal bd);

}

In a web application, you could write a **ServletFilter**, that stored this **SecurityToken** in **HttpSession** and then on every request retrieved the token from Session and put it into a **ThreadLocal** variable where it could be accessed from a **SecurityService** that you could implement.

Perhaps the objects that needed the **SecurityService** could access it as follows:

##### Code Listing: AutomatedTellerMachineImpl implementing security without AOP

public void deposit(BigDecimal bd) {

/\* If the user is not logged in, don't let them use this method \*/

if(!securityManager.isLoggedIn()){

throw new SecurityViolationException();

}

/\* Only proceed if the current user is allowed. \*/

if (!securityManager.isAllowed("AutomatedTellerMachine", operationName)){

throw new SecurityViolationException();

}

...

transport.communicateWithBank(...);

}

In our ATM example, the above might work out well, but imagine a system with thousands of classes that needed security. Now imagine, the way we check to see if a user is “logged in” changed. If we put this code into every method that needed security, then we could possibly have to change this a thousand times if we changed the way we checked to see if a user was logged in.

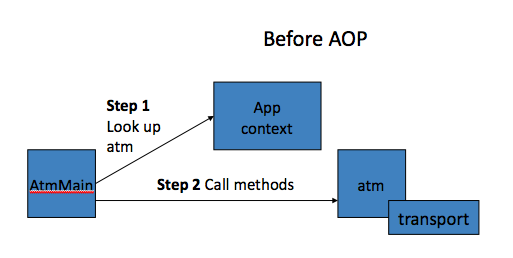
What we want to do instead is to use CDI to create a decorated version of the **AutomateTellerMachineImpl** bean. The decorated version would add the additional behavior to the **AutomateTellerMachineImpl** object without changing the actual implementation of the **AutomateTellerMachineImpl**. In AOP speak, this concept is called a cross-cutting concern. A cross-cutting concern is a concern that crosses the boundry of many objects.

CDI does this by creating what is called an AOP proxy. An AOP proxy is like a dynamic decorator. Underneath the covers CDI can generate a class at runtime (the AOP proxy) that has the same interface as our **AutomatedTellerMachine**. The AOP proxy wraps our existing atm object and provides additional behavior by delegating to a list of method interceptors. The method interceptors provide the additional behavior and are similar to **ServletFilter**s but for methods instead of requests.

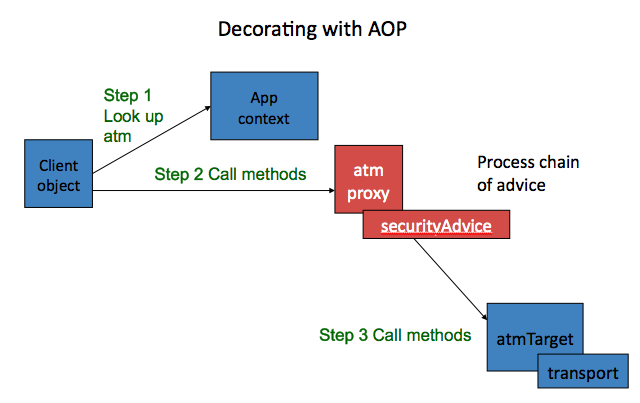
### Diagrams of CDI AOP support

Thus before we added CDI AOP, our atm example was like Figure 1.

##### Figure 1: Before AOP advice

 After we added AOP support, we now get an AOP proxy that applies the **securityAdvice** to the**atm** as show in figure 2.

##### Figure 2: After AOP advice



You can see that the AOP proxy implements the **AutomatedTellerMachine** interface. When the client object looks up the atm and starts invoking methods instead of executing the methods directly, it executes the method on the proxy, which then delegates the call to a series of method interceptor called advice, which eventually invoke the actual atm instance (now called atmTarget).

Let’s actually look at the code for this example.

For this example, we will use a simplified **SecurityToken** that gets stored into a **ThreadLocal** variable, but one could imagine one that was populated with data from a database or an LDAP server or some other source of authentication and authorization.

Here is the **SecurityToken**, which gets stored into a **ThreadLocal** variable, for this example:

##### SecurityToken.java Gets stored in ThreadLocal

package org.cdi.advocacy.security;

/\*\*

\* @author Richard Hightower

\*

\*/

public class SecurityToken {

private boolean allowed;

private String userName;

public SecurityToken() {

}

public SecurityToken(boolean allowed, String userName) {

super();

this.allowed = allowed;

this.userName = userName;

}

public boolean isAllowed(String object, String methodName) {

return allowed;

}

/\*\*

\* @return Returns the allowed.

\*/

public boolean isAllowed() {

return allowed;

}

/\*\*

\* @param allowed The allowed to set.

\*/

public void setAllowed(boolean allowed) {

this.allowed = allowed;

}

/\*\*

\* @return Returns the userName.

\*/

public String getUserName() {

return userName;

}

/\*\*

\* @param userName The userName to set.

\*/

public void setUserName(String userName) {

this.userName = userName;

}

}

The **SecurityService** stores the **SecurityToken** into the **ThreadLocal** variable, and then delegates to it to see if the current user has access to perform the current operation on the current object as follows:

##### SecurityService.java Service

package org.cdi.advocacy.security;

public class SecurityService {

private static ThreadLocal<SecurityToken> currentToken = new ThreadLocal<SecurityToken>();

public static void placeSecurityToken(SecurityToken token){

currentToken.set(token);

}

public static void clearSecuirtyToken(){

currentToken.set(null);

}

public boolean isLoggedIn(){

SecurityToken token = currentToken.get();

return token!=null;

}

public boolean isAllowed(String object, String method){

SecurityToken token = currentToken.get();

return token.isAllowed();

}

public String getCurrentUserName(){

SecurityToken token = currentToken.get();

if (token!=null){

return token.getUserName();

}else {

return "Unknown";

}

}

}

The **SecurityService** will throw a **SecurityViolationException** if a user is not allowed to access a resource. **SecurityViolationException** is just a simple exception for this example.

##### SecurityViolationException.java Exception

package com.arcmind.springquickstart.security;

/\*\*

\* @author Richard Hightower

\*

\*/

public class SecurityViolationException extends RuntimeException {

/\*\*

\*

\*/

private static final long serialVersionUID = 1L;

}

To remove the security code out of the **AutomatedTellerMachineImpl** class and any other class that needs security, we will write an Aspect in CDI to intercept calls and perform security checks before the method call. To do this we will create a method interceptor (known is AOP speak as an advice) and intercept method calls on the atm object.

Here is the **SecurityAdvice** class which will intercept calls on the **AutomatedTellerMachineImpl** class.

##### SecurityAdvice

package org.cdi.advocacy.security;

import javax.inject.Inject;

import javax.interceptor.AroundInvoke;

import javax.interceptor.Interceptor;

import javax.interceptor.InvocationContext;

/\*\*

\* @author Richard Hightower

\*/

@Secure @Interceptor

public class SecurityAdvice {

@Inject

private SecurityService securityManager;

@AroundInvoke

public Object checkSecurity(InvocationContext joinPoint) throws Exception {

System.out.println("In SecurityAdvice");

/\* If the user is not logged in, don't let them use this method \*/

if(!securityManager.isLoggedIn()){

throw new SecurityViolationException();

}

/\* Get the name of the method being invoked. \*/

String operationName = joinPoint.getMethod().getName();

/\* Get the name of the object being invoked. \*/

String objectName = joinPoint.getTarget().getClass().getName();

/\*

\* Invoke the method or next Interceptor in the list,

\* if the current user is allowed.

\*/

if (!securityManager.isAllowed(objectName, operationName)){

throw new SecurityViolationException();

}

return joinPoint.proceed();

}

}

Notice that we annotate the **SecuirtyAdvice** class with an @**Secure** annotation. The @**Secure** annotation is an @**InterceptorBinding**. We use it to denote both the interceptor and the classes it intercepts. More on this later.

Notice that we use @Inject to inject the **securityManager**. Also we mark the method that implements that around advice with and @**AroundInvoke** annotation. This essentially says this is the method that does the dynamic decoration.

Thus, the **checkSecurity** method of **SecurityAdvice** is the method that implements the advice. You can think of advice as the decoration that we want to apply to other objects. The objects getting the decoration are called advised objects.

Notice that the **SecurityService** gets injected into the **SecurityAdvice** and the **checkSecurity** method uses the **SecurityService**\* to see if the user is logged in and the user has the rights to execute the method.

An instance of **InvocationContext**, namely **joinPoint**, is passed as an argument to **checkSecurity**. The **InvocationContext** has information about the method that is being called and provides control that determines if the method on the advised object’s methods gets invoked (e.g., **AutomatedTellerMachineImpl.withdraw** and**AutomatedTellerMachineImpl.deposit**). If \*`joinPoint.proceed()`\* is not called then the wrapped method of the advised object (**withdraw** or **deposit**) is not called. (The proceed method causes the actual decorated method to be invoked or the next interceptor in the chain to get invoked.)

In Spring, to apply an Advice like **SecurityAdvice** to an advised object, you need a pointcut. A pointcut is like a filter that picks the objects and methods that get decorated. In CDI, you just mark the class or methods of the class that you want decorated with an interceptor binding annotation. There is no complex pointcut language. You could implement one as a CDI extention, but it does not come with CDI by default. CDI uses the most common way developer apply interceptors, i.e., with annotations.

CDI scans each class in each jar (and other classpath locations) that has a META-INF/beans.xml. The **SecurityAdvice** get installed in the CDI beans.xml.

##### META-INF/beans.xml

<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/beans\_1\_0.xsd">

<interceptors>

<class>org.cdi.advocacy.security.SecurityAdvice</class>

</interceptors>

</beans>

You can install interceptors in the order you want them called.

In order to associate a interceptor with the classes and methods it decorates, you have to define an **InterceptorBinding** annotation. An example of such a binding is defined below in the @**Secure** annotation.

##### Secure.java annotation

package org.cdi.advocacy.security;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.\*;

import static java.lang.annotation.RetentionPolicy.\*;

import javax.interceptor.InterceptorBinding;

@InterceptorBinding

@Retention(RUNTIME) @Target({TYPE, METHOD})

public @interface Secure {

}

Notice that we annotated the @Secure annotation with the @**InterceptorBinding** annotation.

**InterceptorBindings** follow a lot of the same rules as **Qualifiers** as discussed in the first two articles in this series. **InterceptorBindings** are like qaulifiers for injection in that they can have members which can further qualify the injection. You can also disable **InterceptorBinding** annotation members from qualifying an interception by using the @**NonBinding** just like you can in **Qualifiers**.

To finish our example, we need to annotate our **AutomatedTellerMachine** with the same @**Secure** annotation; thus, associating the **AutomatedTellerMachine** with our**SecurityAdvice**.

##### AutomatedTellerMachine class using @Secure

package org.cdi.advocacy;

...

import javax.inject.Inject;

import org.cdi.advocacy.security.Secure;

@Secure

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

@Inject

@Json

private ATMTransport transport;

public void deposit(BigDecimal bd) {

System.out.println("deposit called");

transport.communicateWithBank(null);

}

public void withdraw(BigDecimal bd) {

System.out.println("withdraw called");

transport.communicateWithBank(null);

}

}

You have the option of use @**Secure** on the methods or at the class level. In this example, we annotated the class itself, which then applies the interceptor to every method.

Let’s complete our example by reviewing the **AtmMain** main method that looks up the atm out of CDI’s **beanContainer**.

Let’s review **AtmMain** as follows:

##### AtmMain.java

package org.cdi.advocacy;

import java.math.BigDecimal;

import org.cdi.advocacy.security.SecurityToken;

import org.cdiadvocate.beancontainer.BeanContainer;

import org.cdiadvocate.beancontainer.BeanContainerManager;

import org.cdi.advocacy.security.SecurityService;

public class AtmMain {

public static void simulateLogin() {

SecurityService.placeSecurityToken(new SecurityToken(true,

"Rick Hightower"));

}

public static void simulateNoAccess() {

SecurityService.placeSecurityToken(new SecurityToken(false,

"Tricky Lowtower"));

}

public static BeanContainer beanContainer = BeanContainerManager

.getInstance();

static {

beanContainer.start();

}

public static void main(String[] args) throws Exception {

simulateLogin();

//simulateNoAccess();

AutomatedTellerMachine atm = beanContainer

.getBeanByType(AutomatedTellerMachine.class);

atm.deposit(new BigDecimal("1.00"));

}

}

Before we added AOP support when we looked up the atm, we looked up the object directly as shown in figure 1, now that we applied AOP when we look up the object we get what is in figure 2. When we look up the atm in the application context, we get the AOP proxy that applies the decoration (advice, method interceptor) to the atm target by wrapping the target and delegating to it after it invokes the series of method interceptors.

### Victroy lap

The last code listing works just like you think. If you use **simulateLogin**, **atm.deposit** does not throw a **SecurityException**. If you use **simulateNoAccess**, it does throw a **SecurityException**. Now let’s weave in a few more “Aspects” to the mix to drive some points home and to show how interception works with multiple interceptors.

I will go quicker this time.

##### LoggingInterceptor

package org.cdi.advocacy;

import java.util.Arrays;

import java.util.logging.Logger;

import javax.interceptor.AroundInvoke;

import javax.interceptor.Interceptor;

import javax.interceptor.InvocationContext;

@Logable @Interceptor

public class LoggingInterceptor {

@AroundInvoke

public Object log(InvocationContext ctx) throws Exception {

System.out.println("In LoggingInterceptor");

Logger logger = Logger.getLogger(ctx.getTarget().getClass().getName());

logger.info("before call to " + ctx.getMethod() + " with args " + Arrays.toString(ctx.getParameters()));

Object returnMe = ctx.proceed();

logger.info("after call to " + ctx.getMethod() + " returned " + returnMe);

return returnMe;

}

}

Now we need to define the Logable interceptor binding annotation as follows:

package org.cdi.advocacy;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.\*;

import static java.lang.annotation.RetentionPolicy.\*;

import javax.interceptor.InterceptorBinding;

@InterceptorBinding

@Retention(RUNTIME) @Target({TYPE, METHOD})

public @interface Logable {

}

Now to use it we just mark the methods where we want this logging.

##### AutomatedTellerMachineImpl.java using Logable

package org.cdi.advocacy;

...

@Secure

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

...

@Logable

public void deposit(BigDecimal bd) {

System.out.println("deposit called");

transport.communicateWithBank(null);

}

public void withdraw(BigDecimal bd) {

System.out.println("withdraw called");

transport.communicateWithBank(null);

}

}

Notice that we use the @**Secure** at the class level which will applies the security interceptor to every mehtod in the **AutomatedTellerMachineImpl**. But, we use @**Logable** only on the **deposit** method which applies it, you guessed it, only on the **deposit** method.

Now you have to add this interceptor to the beans.xml:

##### META-INF/beans.xml

<beans

...

<interceptors>

<class>org.cdi.advocacy.LoggingInterceptor</class>

<class>org.cdi.advocacy.security.SecurityAdvice</class>

</interceptors>

</beans>

When we run this again, we get something like this in our console output:

May 15, 2011 6:46:22 PM org.cdi.advocacy.LoggingInterceptor log

INFO: before call to public void org.cdi.advocacy.AutomatedTellerMachineImpl.deposit(java.math.BigDecimal) with args [1.00]

May 15, 2011 6:46:22 PM org.cdi.advocacy.LoggingInterceptor log

INFO: after call to public void org.cdi.advocacy.AutomatedTellerMachineImpl.deposit(java.math.BigDecimal) returned null

Notice that the order of interceptors in the beans.xml file determines the order of execution in the code. (I added a println to each interceptor just to show the ordering.) When we run this, we get the following output.

##### Output:

In LoggingInterceptor

In SecurityAdvice

If we switch the order in the beans.xml file, we will get a different order in the console output.

##### META-INF/beans.xml

<beans

...

<interceptors>

<class>org.cdi.advocacy.security.SecurityAdvice</class>

<class>org.cdi.advocacy.LoggingInterceptor</class>

</interceptors>

</beans>

In SecurityAdvice

In LoggingInterceptor

This is important as many interceptors can be applied. You have one place to set the order.

### Conclusion

AOP is neither a cure all or voodoo magic, but a powerful tool that needs to be in your bag of tricks. The Spring framework has brought AOP to the main stream masses and Spring 2.5/3.x has simplified using AOP. CDI brings AOP and DI into the standard’s bodies where it can get further mainstreamed, refined and become part of future Java standards like JCache, Java EE 6 and Java EE 7.

You can use Spring CDI to apply services (called cross-cutting concerns) to objects using AOP’s interception model. AOP need not seem a foreign concept as it is merely a more flexible version of the decorator design pattern. With AOP you can add additional behavior to an existing class without writing a lot of wrapper code. This can be a real time saver when you have a use case where you need to apply a cross cutting concern to a slew of classes.

[CDI](http://jcp.org/aboutJava/communityprocess/final/jsr299/index.html) is the Java standard for dependency injection and interception (AOP). It is evident from the popularity of DI and AOP that Java needs to address DI and AOP so that it can build other standards on top of it. DI and AOP are the foundation of many Java frameworks. I hope you share my excitement of CDI as a basis for other JSRs, Java frameworks and standards.

CDI is a foundational aspect of Java EE 6. It is or will be shortly supported by Caucho’s Resin, IBM’s !WebSphere, Oracle’s Glassfish, Red Hat’s JBoss and many more application servers. CDI is similar to core Spring and Guice frameworks. However CDI is a general purpose framework that can be used outside of JEE 6.

CDI simplifies and sanitizes the API for DI and AOP. I find that working with CDI based AOP is easier and covers the most common use cases. CDI is a rethink on how to do dependency injection and AOP (interception really). It simplifies it. It reduces it. It gets rid of legacy, outdated ideas.

CDI is to Spring and Guice what JPA is to Hibernate, and Toplink. CDI will co-exist with Spring and Guice. There are plugins to make them interoperate nicely (more on these shortly).

This is just a brief taste. There is more to come.