DZone JAVALOBBY
Published on *Javalobby* (http://java.dzone.com)

CDI AOP Tutorial: Java Standard Method Interception Tutorial

By *rhightower*
Created *2011/05/25 - 2:51am*

This article discusses CDI based AOP in a tutorial format. CDI [1] 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 already 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 [2], IBM's WebSphere, Oracle's Glassfish [3], Red Hat's JBoss [4] 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—method interception. While this 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.

Here is the source code for this tutorial [5], and instructions [6] for use. It is no accident that this tutorial follows many of the same examples in the Spring 2.5 AOP tutorial [7] 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 [8]. 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 cure-all. 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 [9] 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 first [10] in 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", operationNa
                        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.

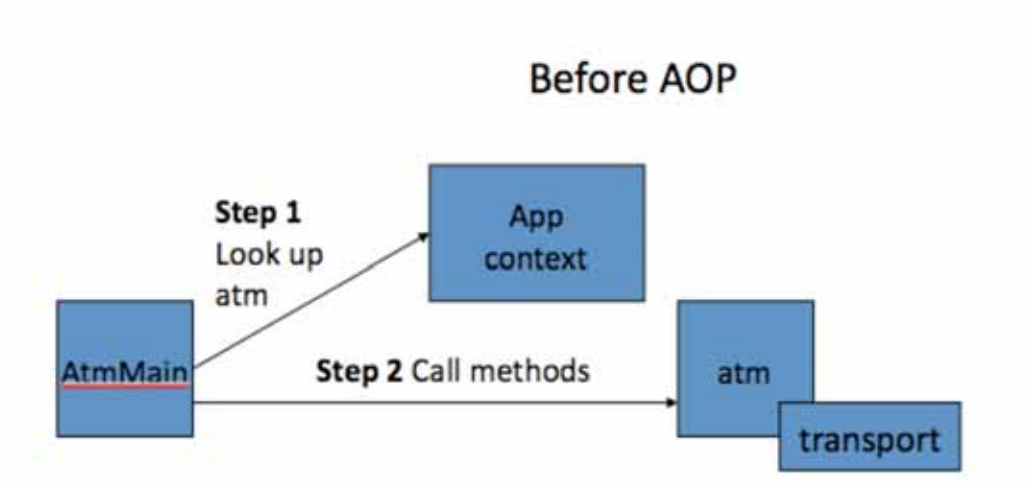
What we want to do instead is to use CDI to create a decorated version of the **AutomateTellerMachineImpl** bean. The decorated version would add the additional behavior to the **AutomateTellerMachineImpl** object without changing the actual implementation of the **AutomateTellerMachineImpl**. In AOP 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

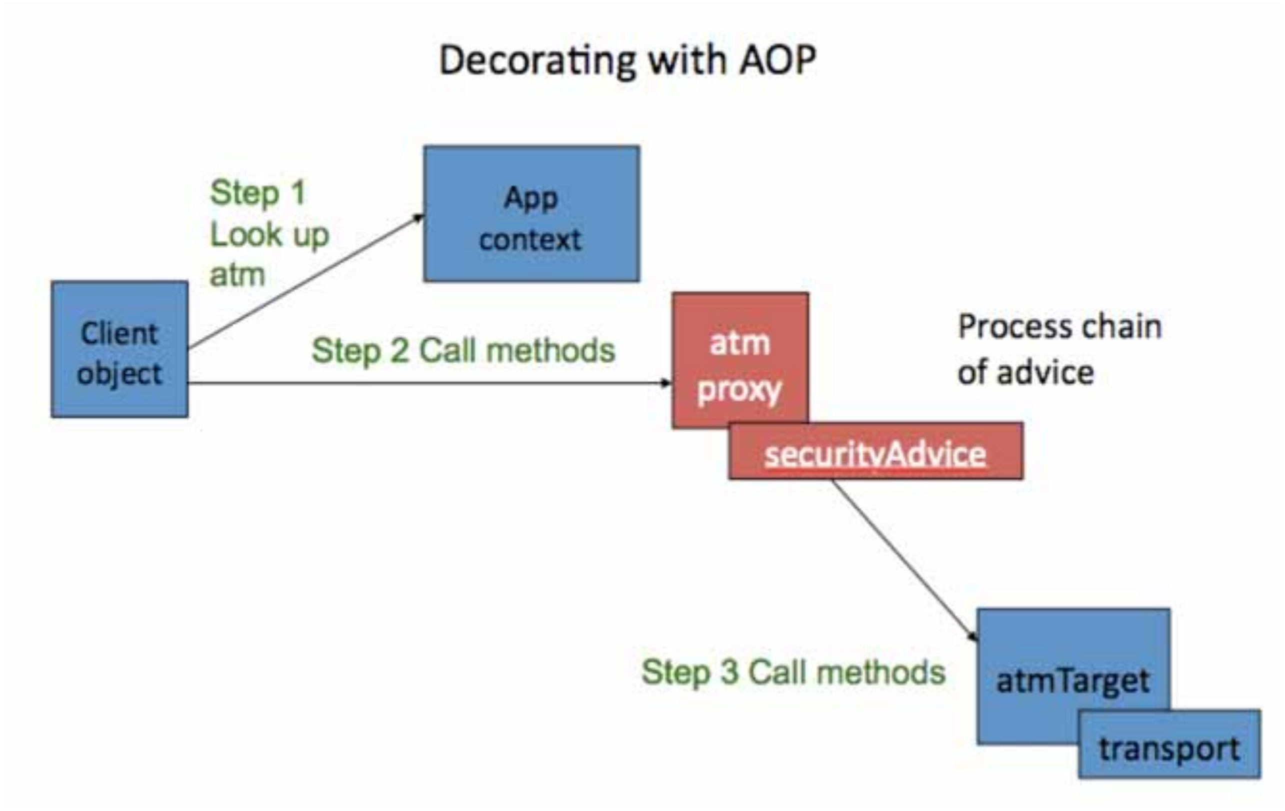
Thus before we added CDI AOP, our atm example was like Figure 1.

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<Sec

        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/X
    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"));
    }

}
```

Continue reading... Click on the navigation links below the author bio to read the other pages of this article.

Be sure to check out part I of this series as well: CDI DI Tutorial [11]!

About the author
This article was written with CDI advocacy in mind by Rick Hightower [12] with some collaboration from others. Rick Hightower has worked as a CTO, Director of Development and a Developer for the last 20 years. He has been involved with J2EE since its inception. He worked at an EJB container company in 1999. He has been working with Java since 1996, and writing code professionally since 1990. Rick was an early Spring enthusiast [13]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [14], Rick is a big fan of the potential of CDI and thinks that EJB 3.1 has come a lot closer to the mark.

CDI Implementations - Resin Candi [15] - Seam Weld [16] - Apache OpenWebBeans [17]

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.toS
        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
May 15, 2011 6:46:22 PM org.cdi.advocacy.LoggingInterceptor log
INFO: after call to public void org.cdi.advocacy.AutomatedTellerMachineImpl.deposit(
```

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 like 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.

To reiterate...

CDI [1] 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.

Resources

- CDI Source [18]
- CDI advocacy group [19]
- CDI advocacy blog [20]
- CDI advocacy google code project [21]
- Google group for CDI advocacy [22]
- Manisfesto version 1 [23]
- Weld reference documentation [24]
- CDI JSR299 [1]
- Resin fast and light CDI and Java EE 6 Web Profile implementation [2]
- CDI & JSF Part 1 Intro by Andy Gibson [25]
- CDI & JSF Part 2 Intro by Andy Gibson [26]
- CDI & JSF Part 3 Intro by Andy Gibson [27]

About the Author

This article was written with CDI advocacy in mind by Rick Hightower [28] with some collaboration from others.

Rick Hightower has worked as a CTO, Director of Development and a Developer for the last 20 years. He has been involved with J2EE since its inception. He worked at an EJB container company in 1999. He has been working with Java since 1996, and writing code professionally since 1990. Rick was an early Spring enthusiast [13]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development. Although not a fan of EJB 3 [14], Rick is a big fan of the potential of CDI and thinks that EJB 3.1 has come a lot closer to the mark.

There are 18 code listings in this article

Source URL: http://java.dzone.com/articles/cdi-aop

Links:
[1] http://jcp.org/aboutJava/communityprocess/final/jsr299/index.html
[2] http://www.caucho.com/resin/
[3] http://glassfish.java.net/
[4] http://www.jboss.org/jbossas/docs/6-x.html
[5] http://jee6-cdi.googlecode.com/svn/tutorial/cdi-aop-example
[6] http://code.google.com/p/jee6-cdi/wiki/MavenAOPTutorialInstructions
[7] http://java.dzone.com/articles/introduction-spring-aop
[8] http://download.oracle.com/javaee/6/tutorial/doc/gjbnr.html
[9] http://en.wikipedia.org/wiki/Decorator_pattern
[10] http://code.google.com/p/jee6-cdi/wiki/utorial
[11] http://java.dzone.com/articles/cdi-di-p1
[12] http://profiles.google.com/RichardHightower/about
[13] http://java.sys-con.com/node/47735
[14] http://java.sys-con.com/node/216307
[15] http://www.caucho.com/
[16] http://seamframework.org/Weld
[17] http://openwebbeans.apache.org/1.1.0-SNAPSHOT/index.html
[18] http://cdisource.org/site/
[19] http://sites.google.com/site/cdipojo/
[20] http://cdi4jadvocate.blogspot.com/
[21] http://code.google.com/p/jee6-cdi/
[22] http://groups.google.com/group/cdiadvocate4j
[23] http://cdi4jadvocate.blogspot.com/2011/03/cdi-advocacy.html
[24] http://docs.jboss.org/weld/reference/1.1.0.Final/en-US/html/
[25] http://www.andygibson.net/blog/tutorial/getting-started-with-jsf-2-0-and-cdi-in-jee-6-part-1/
[26] http://www.andygibson.net/blog/tutorial/getting-started-with-cdi-part-2-injection/
[27] http://www.andygibson.net/blog/tutorial/getting-started-with-jsf-2-0-and-cdi-part-3/
[28] https://profiles.google.com/RichardHightower/about