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

# CDI Dependency Injection - Tutorial II - Annotation Processing and Plugins

By *rhightower*

Created *2011/04/05 - 10:20am*

CDI provides a pluggable architecture allowing you to easily process your own annotations. Read this article to understand the inner workings of CDI and why this JSR is so important.

CDI simplifies and sanitizes the API for DI and AOP like JPA did for ORMs. Through its use of **`Instance`** and @**`Produces`**, CDI provides a pluggable architecture. This is a jaw dropping killer feature of CDI. Master this and you start to tap into the power of CDI. The last article [1] was just to lay the ground work to the uninitiated for this article.

This article continues our tutorial of dependency injection with CDI [1].

This article covers:

- How to process annotations for configuration (injection level and class level)
- How to use an annotation for both injection and configuration (@**`Nonbinding`**)
- Using **`Instance`** to manage instances of possible injection targets
- CDI's plugin architecture for the masses

With this pluggable architecture you can write code that finds new dependencies dynamically. CDI can be a framework to write frameworks. This is why it is so important that CDI was led through the JSR process.

Just like last time, there are some instructions on how to run the examples: Source code for this tutorial [2], and instructions [3] for use. A programming article without working sample code is like a sandwich with no condiments or dry toast without jelly.

# Advanced CDI tutorial

The faint of heart stop here. All of the folks who want to understand the inner workings of CDI continue. So far, we have been at the shallow, warm end of the pool. Things are about to get a little deeper and colder. If you need to master CDI, then this article if for you. If you don't know what CDI is then read the first CDI DI article [1].

## Advanced: Using @Produces and `InjectionPoint` to create configuration annotations

Our ultimate goal is to define an annotation that we can use to configure the retry count on a transport. Essentially, we want to pass a retry count to the transport.

We want something that looks like this:

Code Listing: **`TransportConfig`** annotations that does configuration

```
        @Inject @TransportConfig(retries=2)
        private ATMTransport transport;
```

(This was my favorite section to write, because I wanted to know how to create a annotation configuration from the start.)

Before we do that we need to learn more about @**`Produces`** and **`InjectionPoint`**s. We are going to use a producer to read information (meta-data) about an injection point. A major inflection point for learning how to deal with annotations is the **`InjectionPoint`**s. The **`InjectionPoint`**s has all the metadata we need to process configuration annotations.

An **`InjectionPoint`** [4] is a class that has information about an injection point. You can learn things like what is being decorated, what the target injection type is, what the source injection type, what is the class of the owner of the member that is being injected and so forth.

Let's learn about passing an injection point to @**`Produces`**. Below I have rewritten our simple @**`Produces`** example from the previous article [1], except this time I pass an **`InjectionPoint`** argument into the mix.

Code Listing: **`TransportFactory`** getting meta-data about the injection point

```
package org.cdi.advocacy;

import javax.enterprise.inject.Produces;
import javax.enterprise.inject.spi.InjectionPoint;

public class TransportFactory {


    @Produces ATMTransport createTransport(InjectionPoint injectionPoint) {

        System.out.println("annotated " + injectionPoint.getAnnotated());
        System.out.println("bean " + injectionPoint.getBean());
        System.out.println("member " + injectionPoint.getMember());
        System.out.println("qualifiers " + injectionPoint.getQualifiers());
        System.out.println("type " + injectionPoint.getType());
        System.out.println("isDelegate " + injectionPoint.isDelegate());
        System.out.println("isTransient " + injectionPoint.isTransient());

        return new StandardAtmTransport();
    }

}
```

Now we just run it and see what it produces. The above produces this output. Output

```
annotated AnnotatedFieldImpl[private org.cdi.advocacy.ATMTransport org.cdi.advocacy.
bean ManagedBeanImpl[AutomatedTellerMachineImpl, {@javax.inject.Named(value=atm), @D
member private org.cdi.advocacy.ATMTransport org.cdi.advocacy.AutomatedTellerMachine
qualifiers [@Default()]
type interface org.cdi.advocacy.ATMTransport
isDelegate false
isTransient false
deposit called
communicating with bank via Standard transport
```

It appears from the output that **`annotated`** tells us about the area of the program we annotated. It also appears that **`bean`** tells us which bean the injection is happening on.

From this output you can see that the **`annotated`** property on the **`injectionPoint`** has information about which language feature (field, constructor argument, method argument, etc.). In our case it is the field **`org.cdi.advocacy.AutomatedTellerMachineImpl.transport`**. is being used as the target of the injection, it is the thing that was **`annotated`**.

From this output you can see that the **`bean`** property of the **`injectionPoint`** is being used to describe the bean whose member is getting injected. In this case, it is the **`AutomatedTellerMachineImpl`** whose is getting the field injected.

I won't describe each property, but as an exercise you can.

Exercise: Look up the **`InjectionPoint`** in the API documentation [4]. Find out what the other properties mean. How might you use this meta-data? Can you think of a use case or application where it might be useful? Send me your answers on the CDI group mailing list [5]. The first one to send gets put on the CDI wall of fame. (All others get honorable mentions.)

Drilling further you can see what is in the beans and annotated properties.

Code Listing: **`TransportFactory.createTransport`** drilling further into the meta-data about the injection point

```
    @Produces ATMTransport createTransport(InjectionPoint injectionPoint) {

        System.out.println("annotated " + injectionPoint.getAnnotated());
        System.out.println("bean " + injectionPoint.getBean());
        System.out.println("member " + injectionPoint.getMember());
        System.out.println("qualifiers " + injectionPoint.getQualifiers());
        System.out.println("type " + injectionPoint.getType());
        System.out.println("isDelegate " + injectionPoint.isDelegate());
        System.out.println("isTransient " + injectionPoint.isTransient());

        Bean<?> bean = injectionPoint.getBean();

        System.out.println("bean.beanClass " + bean.getBeanClass());
        System.out.println("bean.injectionPoints " + bean.getInjectionPoints());
        System.out.println("bean.name " + bean.getName());
        System.out.println("bean.qualifiers " + bean.getQualifiers());
        System.out.println("bean.scope " + bean.getScope());
        System.out.println("bean.stereotypes " + bean.getStereotypes());
        System.out.println("bean.types " + bean.getTypes());

        Annotated annotated = injectionPoint.getAnnotated();
        System.out.println("annotated.annotations " + annotated.getAnnotations());
        System.out.println("annotated.annotations " + annotated.getBaseType());
        System.out.println("annotated.typeClosure " + annotated.getTypeClosure());

        return new StandardAtmTransport();
}
```

Now we are cooking with oil. Throw some gas on that flame. Look at the wealth of information that the **`InjectionPoint`** defines.

Output

```
...
bean.beanClass class org.cdi.advocacy.AutomatedTellerMachineImpl
bean.injectionPoints [InjectionPointImpl[private org.cdi.advocacy.ATMTransport org.c
bean.name atm
bean.qualifiers [@javax.inject.Named(value=atm), @Default(), @Any()]
bean.scope interface javax.enterprise.context.Dependent
bean.stereotypes []
bean.types [class org.cdi.advocacy.AutomatedTellerMachineImpl, interface org.cdi.adv
annotated.annotations AnnotationSet[@javax.inject.Inject()]
annotated.annotations interface org.cdi.advocacy.ATMTransport
annotated.typeClosure [interface org.cdi.advocacy.ATMTransport, class java.lang.Obje
...
```

We see that **bean.beanClass** gives up the class of the bean that is getting the injected field. Remember that one, we will use it later.

We can see that **bean.qualifiers** gives up the list of qualifiers for the **AutomatedTellerMachineImpl**.

We can also see that **annotated.annotations** gives us the list of annotations that are associated with the injected field. We will use this later to pull the configuration annotation and configure the transport with it.

Exercise: Look up the **Bean** and **Annotated** in the API documentation [6]. Find out what the other properties mean. How might you use this meta-data? Can you think of a use case or application where it might be useful? Send me your answers on the CDI group mailing list [5]. The first one to send gets put on the CDI wall of fame. (All others get honorable mentions.)

Ok now that we armed with an idea of what an **Injection** point is. Let's get configuring our transport.

First let's define an **TransportConfig** annotation. This is just a plain runtime annotation as follows:

Code Listing: **TransportConfig** an annotation used for configuration

```
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.*;



@Retention(RUNTIME) @Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface TransportConfig {
    int retries() default 5;
}
```

Notice that this annotation has one member retries, which we will use to configure the

**ATMTransport** (**transport**).

Now go ahead and use this to decorate the injection point as follows:

Code Listing: **AutomatedTellerMachineImpl** using **TransportConfig** to configure retries

```
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

    @Inject @TransportConfig(retries=2)
    private ATMTransport transport;
```

Once it is configured when you run it, you will see the following output from our producer:

Output

```
annotated.annotations AnnotationSet[@javax.inject.Inject(), @org.cdi.advocacy.Transp
```

This means the annotation data is there. We just need to grab it and use it. Stop and ponder on this a bit. This is pretty cool. The producer allows me to customize how annotations are consumed. This is powerful stuff and one of the many extension points available to CDI. CDI was meant to be extensible. It is the first mainstream framework that encourages you to consume your own annotation data. This not some obscure framework feature. This is in the main usage.

Please recall that the **injectionPoint.annotated.annotations** gives us the list of annotations that are associated with the injected field, namely, the transport field of the **AutomatedTellerMachineImpl**. Now we can use this to pull the configuration annotation and configure the transport with it. The party is rolling now.

Now we need to change the transport implementations to handle setting retires. Since this is an example, I will do this simply by adding a new setter method for retires (**setRetries**) to the **ATMTranport** interface like so:

Code Listing: **ATMTransport** adding a retries property

```
package org.cdi.advocacy;

public interface ATMTransport {
    public void communicateWithBank(byte[] datapacket);
    public void setRetries(int retries);
}
```

Then we need to change each of the transports to handle this new **retries** property as follows:

Code Listing: **StandardAtmTransport** adding a retries property

```
package org.cdi.advocacy;
```

```
public class StandardAtmTransport implements ATMTransport {

    private int retries;

    public void setRetries(int retries) {
        this.retries = retries;
    }


    public void communicateWithBank(byte[] datapacket) {
        System.out.println("communicating with bank via Standard transport retries="
    }

}
```

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

About the author
This article was written with CDI advocacy in mind by Rick Hightower [7] 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 [8]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [9], 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 [10] - Seam Weld [11] - Apache OpenWebBeans [12]

Now we just change the producer to grab the new annotation and configure the transport as follows: (For clarity I took out all of the Sysout.prinltns.)

Code Listing: **`TransportFactory`** using the annotation configuration to configure a new instance of the transport

```
package org.cdi.advocacy;

...
import javax.enterprise.inject.spi.Annotated;
import javax.enterprise.inject.spi.Bean;
import javax.enterprise.inject.spi.InjectionPoint;

public class TransportFactory {
    @Produces ATMTransport createTransport(InjectionPoint injectionPoint) {

        Annotated annotated = injectionPoint.getAnnotated();

        TransportConfig transportConfig = annotated.getAnnotation(TransportConfig.cl



        StandardAtmTransport transport = new StandardAtmTransport();

        transport.setRetries(transportConfig.retries());
        return transport;
    }

}
```

(Side Note: we are missing a null pointer check. The annotation configuration could be null if the user did not set it, you may want to handle this. The example is kept deliberately short.)

The code just gets the annotation and shoves in the retires into the transport, and then just returns the transport.

We now have a producers that can use an annotation to configure an injection.

Here is our new output:

**Output**

```
...
deposit called
communicating with bank via Standard transport retries=2
```

You can see our retries are there as we configured them in the annotation. Wonderful! Annotation processing for the masses!

Ok we are done with this example. What remains is a victory lap. Let's say we had multiple transports in a single ATM and you wanted to configure all of the outputs at once.

Let's configure the transport based on an annotation in the parent class of the injection target, namely, **`AutomatedTellerMachine`**.

Code Listing: **`TransportFactory`** using the annotation configuration from class not field to configure a new instance of the transport

```
public class TransportFactory {
    @Produces ATMTransport createTransport(InjectionPoint injectionPoint) {

        Bean<?> bean = injectionPoint.getBean();
        TransportConfig transportConfig = bean.getBeanClass().getAnnotation(Transpor

        StandardAtmTransport transport = new StandardAtmTransport();

        transport.setRetries(transportConfig.retries());
        return transport;
```

It is an exercise for the reader to make the injection level annotation (from the last example) override the class level annotations. As always, if you are playing along in the home version of CDI hacker, send me your solution. Best solution gets my admiration.

Output

```
deposit called
communicating with bank via Standard transport retries=7
```

Exercise: Make the injection from the field override the injection from the class. It is a mere matter of Java code. Send me your solution on the CDI group mailing list [5]. The first one to send gets put on the CDI wall of fame. (All others get honorable mentions.)

## Advanced Using @Nonbinding to combine a configuration annotation and a qualifier annotation into one annotation

In the section titled *"Using @Qualfiers with members to discriminate injection and stop the explosion of annotation creation"* we covered adding additional members to a qualifier annotation and then in *"Advanced: Using @Produces and InjectionPoint to create configuration annotations"* we talked about how to write an annotation to configure an injection. Wouldn't be great if we could combine these two concepts into one annotation?

The problem is that qualifier members are used to do the discrimination. We need some qualifier members that are not used for configuration not discrimination.

To make an qualifier member just a configuration member use @**`Nonbinding`** annotation as follows:

Code Listing: **`Transport`** qualifier annotation using @**`Nonbinding`** to add configuration retries param

```
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.enterprise.util.Nonbinding;
import javax.inject.Qualifier;


@Qualifier @Retention(RUNTIME) @Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Transport {
    TransportType type() default TransportType.STANDARD;
    int priorityLevel() default -1;
    String name() default "standard";

    @Nonbinding int retries() default 5;

}
```

Now let's add the **`setRetries`** to the Fast Transport:

Code Listing: **`Transport`** qualifier annotation using @**`Nonbinding`** to add configuration retries param

```
package org.cdi.advocacy;

@Transport(type=TransportType.STANDARD, priorityLevel=1, name="super")
```

```
public class SuperFastAtmTransport implements ATMTransport {
    private int retries=0;

        public void setRetries(int retries) {
        this.retries=retries;
    }


    public void communicateWithBank(byte[] datapacket) {
        System.out.println("communicating with bank via the Super Fast transport ret
    }

}
```

Then we use it as follows:

```
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

    @Inject @Transport(type=TransportType.STANDARD, priorityLevel=1, name="super", r
    private ATMTransport transport;
        ...
```

### Ouptut

```
deposit called
communicating with bank via Standard transport retries=9
```

The final result is we have one annotation that does both qualification and configuration. Booyah!

Exercise: There is an easter egg in this example. There is concept we talked about earlier (in the qualifier discrimination but never added. Please find it and describe it. What are some potential problems of using this approach? Send me your answers on the CDI group mailing list [5]. The first one to send gets put on the CDI wall of fame. (All others get honorable mentions.)

## Advanced: Using Instance to inject transports

The use of the class **`Instance`** allows you to dynamically look up instances of a certain type. This is the plugin architecture for the masses, built right into CDI. Grok this and you will not only understand CDI but have a powerful weapon in your arsenal of mass programming productivity.

These instances can be instances that are in a jar files. For example the **`AutomatedTellerMachine`** could work with transports that did not even exist when the **`AutomatedTellerMachine`** was created. If you don't grok that, read the last sentence again. You are tapping into the scanning capabilities of CDI. This power is there for the taking. The **`Instance`** class is one of the things that makes CDI so cool and flexible. In this section, I hope to give it some justice while still keeping the example small and understandable.

Let's say we wanted to work with multiple transports. But we don't know which transport is configured and on the classpath. It could be that the build was special for a certain type of transport, and it just does not exist on the classpath. Suspend disbelief for a moment and let's look at the code.

Code Listing: **`AutomatedTellerMachineImpl`** using **`Instance`**

```
package org.cdi.advocacy;

import java.math.BigDecimal;

import javax.annotation.PostConstruct;
import javax.enterprise.inject.Default;
import javax.enterprise.inject.Instance;
import javax.inject.Inject;
import javax.inject.Named;

@Named("atm")
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

    @Inject @Soap
    private Instance soapTransport;

    @Inject @Json
    private Instance jsonTransport;

    @Inject @Default
    private Instance defaultTransport;

    private ATMTransport transport;

    @PostConstruct
    protected void init() {
        if (!defaultTransport.isUnsatisfied()) {
            System.out.println("picked Default");
            transport = defaultTransport.iterator().next();
        } else if (!jsonTransport.isUnsatisfied()) {
            System.out.println("picked JSON");
            transport = jsonTransport.iterator().next();
        } else if (!soapTransport.isUnsatisfied()) {
            System.out.println("picked SOAP");
            transport = soapTransport.iterator().next();
        }
    }
```

Notice we are using *`Instance`* as the field type instead of **`ATMTransport`**. Then we look up the actual transport. We can query a **`Instance`** with the **`Instance.isUnsatisfied`** to see it this transport actually exist. There is an **`Instance.get`** method to retrieve a single transport, but I used *`Instance.iterator().next()`* to highlight an important aspect of **`Instance`**, namely, it can return more than one. For example, there could be 20 @**`Default`** based transports in the system.

Imagine if you were implementing a chain of responsibility pattern or a command pattern, and you wanted an easy way to discover the actions or commands that were on the classpath. **`Instance`** would be that way. CDI makes this type of plugin development very easy.

If it could find a single @`Default`, the one we have been using since the start, on the classpath. The output from the above would be as follows:

Output

```
picked Default
deposit called
communicating with bank via Standard transport
```

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

About the author
This article was written with CDI advocacy in mind by Rick Hightower [7] 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 [8]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [9], 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 [10] - Seam Weld [11] - Apache OpenWebBeans [12]

Now to test how the **`Instance.isUnsatisfied`** by commenting out the *`implements ATMTransport`* in StandardAtmTransport class definition. You are essentially taking **`StandardAtmTransport`** out of the pool of possible injection of **`ATMTransport`**. There are no more defaults configured so it should be an unsatisfied.

Code Listing: **`StandardAtmTransport`** commenting out *`implements ATMTransport`* so **`Instance.isUnsatisfied`** returns true

```
package org.cdi.advocacy;

import javax.enterprise.inject.Default;

@Default
public class StandardAtmTransport { //implements ATMTransport {

    public void communicateWithBank(byte[] datapacket) {
        System.out.println("communicating with bank via Standard transport");
    }

}
```

Now the output is this:

```
picked JSON
deposit called
communicating with bank via JSON REST transport
```

Reread this section if you must and make sure you understand why you get the above output.

You can use **`Instance`** to load more than one bean as mentioned earlier. Let's lookup all installed installed @**`Default`** transports. To setup this example remove all of the annotations in the **`ATMTransport`** interfaces and make the beans.xml empty again (so no **`Alternative`** is active).

Code Listing: **`SoapAtmTransport`** making it @**`Default`** by removing @**`Soap`** qualifier

```
package org.cdi.advocacy;

//import javax.enterprise.inject.Alternative;

//@Soap
public class SoapAtmTransport implements ATMTransport {

    public void communicateWithBank(byte[] datapacket) {
        System.out.println("communicating with bank via Soap transport");
    }

}
```

Code Listing: **`JsonRestAtmTransport`** making it @**`Default`** by removing @**`Json`** qualifier

```
package org.cdi.advocacy;

//import javax.enterprise.inject.Alternative;

//@Alternative @Json
public class JsonRestAtmTransport implements ATMTransport {

    public void communicateWithBank(byte[] datapacket) {
        System.out.println("communicating with bank via JSON REST transport");
    }

}
```

Code Listing: **`StandardAtmTransport`** making it @**`Default`** by removing any qualifiers from it

```
package org.cdi.advocacy;


//Just make sure there are no qualifiers
public class StandardAtmTransport implements ATMTransport {

        public void communicateWithBank(byte[] datapacket) {
        System.out.println("communicating with bank via Standard transport");
        }

}
```

We also need to make sure that the **`beans.xml`** file is empty.

Code Listing: *`{classpath}/META-INF/beans.xml`* removing all alternatives

Now use every transport that is installed using the annotation.

```
package org.cdi.advocacy;

import java.math.BigDecimal;
import java.util.Iterator;

import javax.annotation.PostConstruct;
import javax.enterprise.inject.Any;
import javax.enterprise.inject.Instance;
import javax.inject.Inject;
import javax.inject.Named;

@Named("atm")
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

    @Inject
    private Instance allTransports;

    @PostConstruct
    protected void init() {
        System.out.println("Is this ambiguous? " + allTransports.isAmbiguous() );
        System.out.println("Is this unsatisfied? " + allTransports.isUnsatisfied() )
    }

    public void deposit(BigDecimal bd) {
        System.out.println("deposit called");

        for (ATMTransport transport : this.allTransports) {
            transport.communicateWithBank(null);
        }

    }

    public void withdraw(BigDecimal bd) {
        System.out.println("withdraw called");

        for (ATMTransport transport : this.allTransports) {
            transport.communicateWithBank(null);
        }

    }

}
```

In this context ambiguous means more than one. Therefore, CDI found more than one possibility for injection if ambiguous returns true. It should find three defaults.

Your output should look like this (or something close to this).

Output

```
Is this ambiguous? true
Is this unsatisfied? false
deposit called
communicating with bank via JSON REST transport
communicating with bank via Soap transport
communicating with bank via Standard transport
communicating with bank via the Super Fast transport
```

Note that we changed deposit to iterate through the available instances.

Now try something new comment out the *`implements ATMTransports`* in **`SuperFastAtmTransport`**, **`JsonRestAtmTransport`** and **`SoapRestAtmTransport`**. **`JsonRestAtmTransport`** and **`SoapRestAtmTransport`** transport class definition should have this *`//implements ATMTransport {`*.

Now rerun the example. You get this output.

**Output**

```
Is this ambiguous? false
Is this unsatisfied? false
deposit called
communicating with bank via Standard transport
```

Since the only transport left is the standard transport (**`StandardAtmTransport`**), only it is in the output. The **`Instance`** is no longer ambiguous, there is only one so it prints false. CDI finds the one so it is not unsatisfied.

Now comment out all of //implements ATMTransport, and you get this:

```
Is this ambiguous? false
Is this unsatisfied? true
deposit called
```

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

About the author
This article was written with CDI advocacy in mind by Rick Hightower [7] 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 [8]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [9], 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 [10] - Seam Weld [11] - Apache OpenWebBeans [12]

Notice there a no longer any ATMTransport transport implementations in the system at all.

The @**`Any`** qualifier states that you want all instances of an implementation. It does not matter what qualifiers they have, you want them all @**`Json`**s, @**`Soap`**s, @**`SuperFast`**s, whatever.

Add the all of the annotations we commented out back to all of the transports. Add the @**`Any`** to the transport injection as follows:

Code Listing: **`AutomatedTellerMachineImpl`** @**`Inject`** @**`Any`** *`Instance`* to inject all transport instances

```
...
import javax.enterprise.inject.Any;
...
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

    @Inject @Any
    private Instance allTransports;

    private ATMTransport transport;

       ...
}
```

The output of this should be: Output

```
Is this ambigous? true
Is this unsatisfied? false
deposit called
communicating with bank via JSON REST transport
communicating with bank via Soap transport
communicating with bank via Standard transport
communicating with bank via the Super Fast transport
```

@Any finds all of the transports in the system. Once you inject the instances into the system, you can use the **`select`** method of **`instance`** to query for a particular type. Here is an example of that:

Code Listing: **`AutomatedTellerMachineImpl`** using selects to find a particular transport from the list you loaded

```
...
import javax.enterprise.inject.Any;
...
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

    @Inject @Any
    private Instance allTransports;

    private ATMTransport transport;

    @PostConstruct
    protected void init() {
        transport = allTransports.select(new AnnotationLiteral(){}).get();

        if (transport!=null) {
            System.out.println("Found standard transport");
            return;
        }

        transport = allTransports.select(new AnnotationLiteral(){}).get();


        if (transport!=null) {
            System.out.println("Found JSON standard transport");
            return;
        }
        transport = allTransports.select(new AnnotationLiteral(){}).get();


        if (transport!=null) {
            System.out.println("Found SOAP standard transport");
            return;
        }

    }


        public void deposit(BigDecimal bd) {
        System.out.println("deposit called");

        transport.communicateWithBank(...);
    }

       ...
}
```

Here is the expected format. Output

```
Found standard transport
deposit called
communicating with bank via Standard transport
```

Now imagine there being a set of settings that are configured in a db or something and the code might look like this to find a transport (this should look familiar to you by now).

Code Listing: **`AutomatedTellerMachineImpl`** using selects and some business logic to decide which transport to use

```
package org.cdi.advocacy;

import java.math.BigDecimal;

import javax.annotation.PostConstruct;
import javax.enterprise.inject.Any;
import javax.enterprise.inject.Default;
import javax.enterprise.inject.Instance;
import javax.enterprise.util.AnnotationLiteral;
import javax.inject.Inject;
import javax.inject.Named;

@Named("atm")
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

    @Inject @Any
    private Instance allTransports;

    private ATMTransport transport;

    //These could be looked up in a DB, JNDI or a properties file.
    private boolean useJSON = true;
    private boolean behindFireWall = true;

    @PostConstruct
    protected void init() {
        ATMTransport soapTransport, jsonTransport, standardTransport;

        standardTransport = allTransports.select(new AnnotationLiteral(){}).get();
        jsonTransport = allTransports.select(new AnnotationLiteral(){}).get();
        soapTransport = allTransports.select(new AnnotationLiteral(){}).get();

        if (!behindFireWall) {
            transport = standardTransport;
        } else {
            if (useJSON) {
                transport = jsonTransport;
            } else {
                transport = soapTransport;
            }
        }

    }


    public void deposit(BigDecimal bd) {
        System.out.println("deposit called");

        transport.communicateWithBank(...);
    }

    public void withdraw(BigDecimal bd) {
        System.out.println("withdraw called");

        transport.communicateWithBank(...);

    }

}
```

Exercise: Please combine the use of Instance with a producer to define the same type of lookup but have the business logic and select lookup happen in the **`TrasportFactory`**. Send me your answers on the CDI group mailing list [5]. The first one to send gets put on the CDI wall of fame. (All others get honorable mentions.)

## The dirty truth about CDI and Java SE

The dirty truth is this. CDI is part of JEE 6. It could easily be used outside of a JEE 6 container as these examples show. The problem is that there is no standard interface to use CDI outside of a JEE 6 container so the three main implementations Caucho Resin Candi [13], Red Hat JBoss Weld [11] and Apache OpenWebBeans [14] all have their own way to run a CDI container standalone.

As part of the promotion and advocacy of CDI, we (Andy Gibson, Rob Williams, and others) came up with a standard way to run CDI standalone. It is a small wrapper around CDI standalone containers. It works with Resin Candi [13], Weld [11] and OpenWebBeans [14]. If you used the examples, in the CDI DI [15] or this article then you used the first artifact that the CDISource organization put together. We plan on coming up with ways to unit test JPA outside of the container, and a few other things. As we find holes in the CDI armor we want to work with the community at large to fill the holes. CDI, although standard, is very new. We are hoping that the groundwork that CDI has laid down can get used outside of Java EE as well as inside of Java EE (we are not anti-Java EE).

## Conclusion

Dependency Injection (DI) refers to the process of supplying an external dependency to a software component.

CDI [16] 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 vision of CDI as a basis for other JSRs, Java frameworks and standards.

This article discussed more advanced CDI dependency injection in a tutorial format. It covered some of the features of CDI such as processing annotation data and working with multiple instances of various types using the **`Instance`** class to tap into the powerful CDI class scanner capabilities.

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. Through its use of **`Instance`** and @Produces, CDI provides a pluggable architecture. With this pluggable architecture you can write code that finds new dependencies dynamically.

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. There is more integration option on the way.

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

## Resources

- CDI Depenency Injection Article [1]

- CDI advocacy group [17]
- CDI advocacy blog [18]
- CDI advocacy google code project [19]
- Google group for CDI advocacy [20]
- Manisfesto version 1 [21]
- Weld reference documentation [22]
- CDI JSR299 [16]
- Resin fast and light CDI and Java EE 6 Web Profile implementation [23]
- CDI & JSF Part 1 Intro by Andy Gibson [24]
- CDI & JSF Part 2 Intro by Andy Gibson [25]
- CDI & JSF Part 3 Intro by Andy Gibson [26]

## About the Author

This article was written with CDI advocacy in mind by Rick Hightower [27] 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 [8]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development. Although not a fan of EJB 3 [9], 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 35 code listings in this article
**Subtitle:**
Annotation Processing and Plugins

---

**Source URL:** http://java.dzone.com/articles/cdi-di-p2

**Links:**
[1] http://java.dzone.com/articles/cdi-di-p1
[2] https://jee6-cdi.googlecode.com/svn/tutorial/cdi-di-example
[3] http://code.google.com/p/jee6-cdi/wiki/MavenDITutorialInstructions
[4] http://download.oracle.com/javaee/6/api/javax/enterprise/inject/spi/InjectionPoint.html
[5] http://groups.google.com/group/cdiadvocate4j?pli=1
[6] http://download.oracle.com/javaee/6/api/index.html?javax/enterprise/inject/spi/package-summary.html
[7] http://profiles.google.com/RichardHightower/about
[8] http://java.sys-con.com/node/47735
[9] http://java.sys-con.com/node/216307
[10] http://www.caucho.com/
[11] http://seamframework.org/Weld
[12] http://openwebbeans.apache.org/1.1.0-SNAPSHOT/index.html
[13] http://www.caucho.com/resin/candi/
[14] http://openwebbeans.apache.org/owb/index.html
[15] http://code.google.com/p/jee6-cdi/wiki/rticle
[16] http://jcp.org/aboutJava/communityprocess/final/jsr299/index.html
[17] http://sites.google.com/site/cdipojo/
[18] http://cdi4jadvocate.blogspot.com/
[19] http://code.google.com/p/jee6-cdi/

[20] http://groups.google.com/group/cdiadvocate4j
[21] http://cdi4jadvocate.blogspot.com/2011/03/cdi-advocacy.html
[22] http://docs.jboss.org/weld/reference/1.1.0.Final/en-US/html/
[23] http://www.caucho.com/resin/
[24] http://www.andygibson.net/blog/tutorial/getting-started-with-jsf-2-0-and-cdi-in-jee-6-part-1/
[25] http://www.andygibson.net/blog/tutorial/getting-started-with-cdi-part-2-injection/
[26] http://www.andygibson.net/blog/tutorial/getting-started-with-jsf-2-0-and-cdi-part-3/
[27] https://profiles.google.com/RichardHightower/about