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

Dependency Injection - An Introductory Tutorial Part 1

By *rhightower*
Created *2011/03/28 - 9:08am*

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 and JSRs on top of it. DI and AOP are the foundation of many Java frameworks, and CDI will be the foundation of many future specifications and JSRs.

This article discusses dependency injection in a tutorial format. It covers some of the features of CDI such as type safe annotations configuration, alternatives and more. This tutorial is split into two parts, the first part covers the basis of dependency injection, @Inject, @Produces and @Qualifiers. The next part in this series covers advanced topics like creating pluggable components with Instance and processing annotations for configuration.

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 Dependency Injection (DI), then CDI is an easy on ramp for picking up DI quickly. CDI is simpler to use and learn.

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

Source code for this tutorial [5], and instructions [6] for use.

It is no accident that this tutorial follows this Spring 2.5 DI tutorial (using Spring "new" DI annotations) [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 DI annotations.

Design goals of this tutorial

This tutorial series is meant to be a description and explanation of DI in CDI without the clutter of EJB 3.1 or JSF. There are already plenty of tutorials that cover EJB 3.1 and JSF with CDI as a supporting actor.

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.

We start out slow, step by step and basic. Then once you understand the fundamentals, we pick up the pace quite a bit.

All code examples have actually been run. We don't type in ad hoc code. If it did not run, it is not in our tutorial. We are not winging it.

There are clear headings for code listings so you can use this tutorial as a cookbook when you want to use some feature of CDI DI in the future. This is a code centric tutorial. Again, the code listings are in the TOC on the wiki page [9] so you can find just the code listing you are looking for quickly like an index for a cookbook.

Decorators, Extentions, Interceptors, Scopes are out of scope for this first tutorial. Expect them in future tutorials.

If this tutorial is well recieved and we get enough feedback through, the JavaLobby articles, our google group and comments section of the wiki then we will add a comprehensive tutorial on CDI AOP (Decorators and Interceptors) and one on Extentions. The more positive and/or constructive feedback we get the more encouraged we will be to add more.

Dependency Injection

Dependency Injection (DI) refers to the process of supplying an external dependency to a software component. DI can help make your code architecturally pure.

It aids in design by interface as well as test-driven development by providing a consistent way to inject dependencies. For example, a data access object (DAO) may depend on a database connection.

Instead of looking up the database connection with JNDI, you could inject it.

One way to think about a DI framework like CDI is to think of JNDI turned inside out. Instead of an object looking up other objects that it needs to get its job done (dependencies), a DI container injects those dependent objects. This is the so-called Hollywood Principle, "Don't call us," (lookup objects), "we'll call you" (inject objects).

If you have worked with CRC [10] cards you can think of a dependency as a collaborator. A collaborator is an object that another object needs to perform its role, like a DAO (data access object) needs a JDBC connection object for example.

Dependency Injection-`AutomatedTellerMachine` without CDI or Spring

or Guice

Let's say that you have an automated teller machine (ATM, also known as an automated banking machine in other countries) and it needs the ability to talk to a bank. It uses what it calls a transport object to do this. In this example, a transport object handles the low-level communication to the bank.

This example could be represented by these two interfaces as follows:

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);

}
```

Code Listing: ATMTransport interface

```
package org.cdi.advocacy;

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

Now the **AutomatedTellerMachine** needs a transport to perform its intent, namely withdraw money and deposit money. To carry out these tasks, the **AutomatedTellerMachine** may depend on many objects and collaborates with its dependencies to complete the work.

An implementation of the **AutomatedTellerMachine** may look like this:

Code Listing: **AutomatedTellerMachineImpl** class

```
package org.cdi.advocacy;
...
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

        private ATMTransport transport;

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

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

}
```

The **AutomatedTellerMachineImpl** does not know or care how the transport withdraws and deposits money from the bank. This level of indirection allows us to replace the transport with different implementations such as in the following example:

Three example transports: SoapAtmTransport, StandardAtmTransport and JsonAtmTransport

Code Listing: StandardAtmTransport

```
package org.cdi.advocacy;


public class StandardAtmTransport implements ATMTransport {

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

}
```

Code Listing: SoapAtmTransport

```
package org.cdi.advocacy;

public class SoapAtmTransport implements ATMTransport {

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

}
```

Code Listing: JsonRestAtmTransport

```
package org.cdi.advocacy;

public class JsonRestAtmTransport implements ATMTransport {

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

}
```

Notice the possible implementations of the **ATMTransport** interface. The **AutomatedTellerMachineImpl** does not know or care which transport it uses. Also, for testing and developing, instead of talking to a real bank, you could easily use Mockito [11] or EasyMock [12] or you could even write a **SimulationAtmTransport** that was a mock implementation just for testing.

The concept of DI transcends CDI, Guice and Spring. Thus, you can accomplish DI without CDI, Guice or Spring as follows:

Code Listing: `AtmMain`: DI without CDI, Spring or Guice

```
package org.cdi.advocacy;

public class AtmMain {

        public void main (String[] args) {
                AutomatedTellerMachine atm = new AutomatedTellerMachineImpl();
                ATMTransport transport = new SoapAtmTransport();
                /* Inject the transport. */
                ((AutomatedTellerMachineImpl)atm).setTransport(transport);

                atm.withdraw(new BigDecimal("10.00"));

                atm.deposit(new BigDecimal("100.00"));
        }

}
```

About the author
This article was written with CDI advocacy in mind by Rick Hightower [13] 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 [14]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [15], 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 [16] - Seam Weld [17] - Apache OpenWebBeans [18]

Then injecting a different **transport** is a mere matter of calling a different setter method as follows:

Code Listing: `AtmMain`: DI without CDI, Spring or Guice: setTransport

```
ATMTransport transport = new SimulationAtmTransport();
((AutomatedTellerMachineImpl)atm).setTransport(transport);
```

The above assumes we added a **setTransport** method to the **AutomateTellerMachineImpl**. Note you could just as easily use constructor arguments instead of a setter method. Thus keeping the interface of your **AutomateTellerMachineImpl** clean.

Running the examples

To run the examples quickly, we setup some maven pom.xml files for you. Here are the instructions [6] to get the examples up and running.

Dependency Injection-`AutomatedTellerMachine` using CDI

To use CDI to manage the dependencies, do the following:

1. Create an empty **bean.xml** file under **META-INF** resource folder
2. Use the **@Inject** annotation to annotate a **setTransport** setter method in **AutomatedTellerMachineImpl**
3. Use the **@Default** annotation to annotate the **StandardAtmTransport**
4. Use the **@Alternative** to annotate the **SoapAtmTransport**, and **JsonRestAtmTransport**.
5. Use the **@Named** annotation to make the **AutomatedTellerMachineImpl** easy to look up; give it the name "atm"
6. Use the CDI **beanContainer** to look up the **atm**, makes some deposits and withdraws.

Step 1: Create an empty **bean.xml** file under **META-INF** resource folder

META-INF/beans.xml

CDI needs an bean.xml file to be in META-INF of your jar file or classpath or WEB-INF of your web application. This file can be completely empty (as in 0 bytes). If there is no beans.xml file in your META-INF or WEB-INF then that war file or jar file will not be processed by CDI. Otherwise, CDI will scan the jar and war file if the beans.xml file exists even if it is 0 bytes.

Code Listing: META-INF/beans.xml just as empty as can be

Notice that we included a starter beans.xml file with a namespace and a element. Although **beans.xml** could be completely empty, it is nice to have a starter file so when you need to add things (like later on in this tutorial) you can readily. Also it keeps the IDE from complaining about ill formed xml when you actually do have a 0 byte beans.xml. (I hate when the IDE complains. It is very distracting.)

Step 2: Use the **@Inject** annotation to annotate a **setTransport** setter method in **AutomatedTellerMachineImpl**

The **@Inject** annotation is used to mark where an injection goes. You can annotate constructor arguments, instance fields and setter methods of properties. In this example, we will annotate the **setTransport** method (which would be the setter method of the transport property).

Code Listing: AutomatedTellerMachineImpl using **@Inject** to inject a transport

```
package org.cdi.advocacy;

...

import javax.inject.Inject;

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

        private ATMTransport transport;

        @Inject
        public void setTransport(ATMTransport transport) {
                this.transport = transport;
        }

       ...

}
```

By default, CDI would look for a class that implements the **ATMTransport** interface, once it finds this it creates an instance and injects this instance of **ATMTransport** using the setter method **setTransport**. If we only had one possible instance of **ATMTransport** in our classpath, we would not need to annotate any of the **ATMTransport** implementations. Since we have three, namely, **StandardAtmTransport**, **SoapAtmTransport**, and **JsonAtmTransport**, we need to mark two of them as **@Alternative**'s and one as **@Default**.

Step 3: Use the **@Default** annotation to annotate the **StandardAtmTransport**

At this stage of the example, we would like our default transport to be **StandardAtmTransport**; thus, we mark it as **@Default** as follows:

Code Listing: StandardAtmTransport using **@Default**

```
package org.cdi.advocacy;

import javax.enterprise.inject.Default;

@Default
public class StandardAtmTransport implements ATMTransport {
    ...
```

It should be noted that a class is **@Default** by default. Thus marking it so is redundant; and not only that its redundant.

Step 4: Use the **@Alternative** to annotate the **SoapAtmTransport**, and **JsonRestAtmTransport**.

If we don't mark the others as **@Alternative**, they are by default as far as CDI is concerned, marked as **@Default**. Let's mark **JsonRestAtmTransport** and **SoapRestAtmTransport @Alternative** so CDI does not get confused.

Code Listing: JsonRestAtmTransport using **@Alternative**

```
package org.cdi.advocacy;

import javax.enterprise.inject.Alternative;

@Alternative
public class JsonRestAtmTransport implements ATMTransport {

...
}
```

Code Listing: SoapAtmTransport using **@Alternative**

```
package org.cdi.advocacy;

import javax.enterprise.inject.Alternative;

@Alternative
public class SoapAtmTransport implements ATMTransport {
   ...
}
```

Step 5: Use the **@Named** annotation to make the **AutomatedTellerMachineImpl** easy to look up; give it the name "atm"

Since we are not using **AutomatedTellerMachineImpl** from a Java EE 6 application, let's just use the **beanContainer** to look it up. Let's give it an easy logical name like "atm". To give it a name, use the **@Named** annotation. The **@Named** annotation is also used by JEE 6 application to make the bean accessible via the Unified EL [19] (EL stands for Expression language and it gets used by JSPs and JSF components).

Here is an example of using @Named to give the **AutomatedTellerMachineImpl** the name "atm"as follows:

Code Listing: AutomatedTellerMachineImpl using **@Named**

```
package org.cdi.advocacy;

import java.math.BigDecimal;

import javax.inject.Inject;
import javax.inject.Named;

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

}
```

About the author
This article was written with CDI advocacy in mind by Rick Hightower [13] 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 [14]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [15], 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 [16] - Seam Weld [17] - Apache OpenWebBeans [18]

It should be noted that if you use the **@Named** annotations and don't provide a name, then the name is the name of the class with the first letter lower case so this:

```
@Named
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {
       ...

}
```

makes the name automatedTellerMachineImpl.

Step 6: Use the CDI **beanContainer** to look up the **atm**, makes some deposits and withdraws.

Lastly we want to look up the **atm** using the **beanContainer** and make some deposits.

Code Listing: AtmMain looking up the atm by name

```
package org.cdi.advocacy;

...

public class AtmMain {

        ...
        ...

        public static void main(String[] args) throws Exception {
                AutomatedTellerMachine atm = (AutomatedTellerMachine) beanContainer
                                .getBeanByName("atm");

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

        }

}
```

When you run it from the command line, you should get the following:

Output

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

You can also lookup the **AtmMain** by type and an optional list of Annotations as the name is really to support the Unified EL (JSPs, JSF, etc.).

Code Listing: AtmMain looking up the atm by type

```
package org.cdi.advocacy;

...

public class AtmMain {

        ...
        ...

    public static void main(String[] args) throws Exception {
        AutomatedTellerMachine atm = beanContainer.getBeanByType(AutomatedTellerMach
        atm.deposit(new BigDecimal("1.00"));
    }

}
```

Since a big part of CDI is its type safe injection, looking up things by name is probably discouraged. Notice we have one less cast due to Java Generics [20].

If you remove the **@Default** from the **StandardATMTransport**, you will get the same output. If you remove the **@Alternative** from both of the other transports, namely,

JsonATMTransport, and **SoapATMTransport**, CDI will croak as follows:

Output

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: javax.enterprise.inject.AmbiguousResolutionException: org.cdi.advocacy.Au
Too many beans match, because they all have equal precedence.
See the @Stereotype and  tags to choose a precedence.  Beans:
    ManagedBeanImpl[JsonRestAtmTransport, {@Default(), @Any()}]
    ManagedBeanImpl[SoapAtmTransport, {@Default(), @Any()}]
    ManagedBeanImpl[StandardAtmTransport, {@javax.enterprise.inject.Default(), @Any(
   ...
```

CDI expects to find one and only one qualified injection. Later we will discuss how to use an alternative.

Using @Inject to inject via constructor args and fields

You can inject into fields,constructor arguments and setter methods (or any method really).

Here is an example of field injections:

Code Listing: **AutomatedTellerMachineImpl.transport** using @Inject to do field injection.

```
...
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

        @Inject
        private ATMTransport transport;
```

Code Listing: **AutomatedTellerMachineImpl.transport** using **@Inject** to do constructor injection.

```
...
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

        @Inject
        public AutomatedTellerMachineImpl(ATMTransport transport) {
                this.transport = transport;
        }
```

About the author
This article was written with CDI advocacy in mind by Rick Hightower [13] 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 [14]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [15], 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 [16] - Seam Weld [17] - Apache OpenWebBeans [18]

This flexibility allows you to create classes that are easy to unit test.

Using simple @Produces

There are times when the creation of an object may be complex. Instead of relying on a constructor, you can delegage to a factory class to create the instance. To do this with CDI, you would use the @Produces from your factory class as follows:

Code Listing: **TransportFactory.createTransport** using @Produces to define a factory method

```
package org.cdi.advocacy;

import javax.enterprise.inject.Produces;

public class TransportFactory {

        @Produces ATMTransport createTransport() {
                System.out.println("ATMTransport created with producer");
                return new StandardAtmTransport();
        }

}
```

The factory method could use qualifiers just like a class declaration. In this example, we chose not to. The **AutomatedTellerMachineImpl** does not need to specify any special qualifiers. Here is the **AutomatedTellerMachineImpl** that receives the simple producer.

Code Listing: **AutomatedTellerMachineImpl** receives the simple producer

```
import javax.inject.Inject;
import javax.inject.Named;

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

        @Inject
        private ATMTransport transport;
        ...
```

Check your understanding by looking at the output of running this with **AtmMain**.

Output

```
ATMTransport created with producer
deposit called
communicating with bank via Standard transport
```

Using @Alternative to select an Alternative

Earlier, you may recall, we defined several alternative transports, namely, **JsonRestAtmTransport** and **SoapRestAtmTransport**. Imagine that you are an installer of ATM machines and you need to configure certain transports at certain locations. Our previous injection points essentially inject the default which is the **StandardRestAtmTransport** transport.

If I need to install a different transport, all I need to do is change the /META-INF/beans.xml file to use the right transport as follows:

Code Listing: **{classpath}/META-INF/beans.xml**

```
                org.cdi.advocacy.JsonRestAtmTransport
```

You can see from the output that the JSON REST transport is selected.

Output

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

Alternatives codifies and simplifies a very normal case in DI, namely, you have different injected objects based on different builds or environments. The great thing about objects is they can be replaced (Grady Booch said this). Alternatives allow you to mark objects that are replacements for other objects and then activate them when you need them.

If the DI container can have alternatives, let's mark them as alternatives. Think about it this way. I don't have to document all of the alternatives as much. It is self documenitng. If someone knows CDI and they know about Alternatives they will not be surprised. Alternatives really canoncalizes the way you select an Alternative.

You can think of CDI as a canonicalization of many patterns that we have been using with more general purpose DI frameworks. The simplifcation and canonicalization is part of the evoluiton of DI.

Code Listing: Using @Qualifier to inject different types

All objects and producers in CDI have qualifiers. If you do not assign a qaulifier it by default has the qualifier @**Default** and @**Any**. It is like a TV crime show in the U.S., if you do not have money for a lawyer, you will be assigned one.

Qualifiers can be used to discriminate exaclty what gets injected. You can write custom qualifiers.

Qualifiers work like garanimal [21] tags for kids clothes, you match the qualifier from the injection target and the injection source, then that is the type that will be injected.

If the tags (Qualifiers) match, then you have a match for injection.

You may decide that at times you want to inject **Soap** or **Json** or the **Standard** transport. You don't want to list them as an alternative. You actually, for example, always want the **Json** implementation in a certain case.

Here is an example of defining a qualifier for **Soap**.

Code Listing: **Soap** runtime qualifier annotation

```
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.inject.Qualifier;


@Qualifier @Retention(RUNTIME) @Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Soap {

}
```

Notice that a qualifier is just a runtime annotation that is marked with the @**Qualifier** annotation. The @**Qualifier** is an annotation that decorates a runtime annoation to make it a qualifier.

Then we would just mark the source of the injection point, namely, **SoapAtmTransport** with our new @**Soap** qualifier as follows:

Code Listing: **SoapAtmTransport** using new @**Soap** qualifier

```
package org.cdi.advocacy;

@Soap
public class SoapAtmTransport implements ATMTransport {

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

}
```

About the author
This article was written with CDI advocacy in mind by Rick Hightower [13] 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 [14]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [15], 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 [16] - Seam Weld [17] - Apache OpenWebBeans [18]

Next time you are ready to inject a **Soap** transport we can do that by annotating the argument to the constructor as follows:

Code Listing: **AutomatedTellerMachineImpl** injecting **SoapAtmTransport** using new **@Soap** qualifier via constructor arg

```
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

        private ATMTransport transport;

        @Inject
        public AutomatedTellerMachineImpl(@Soap ATMTransport transport) {
                this.transport = transport;
        }
```

You could also choose to do this via the setter method for the property as follows:

Code Listing: **AutomatedTellerMachineImpl** injecting **SoapAtmTransport** using new **@Soap** qualifier via setter method arg

```
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

        private ATMTransport transport;

        @Inject
        public void setTransport(@Soap ATMTransport transport) {
                this.transport = transport;
        }
```

And a very common option is to use a field level injection as follows:

Code Listing: **AutomatedTellerMachineImpl** injecting **SoapAtmTransport** using new **@Soap** qualifier via setter method arg

```
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

        @Inject @Soap
        private ATMTransport transport;
```

From this point on, we are just going to use field level injection to simplify the examples.

Using @Qualfiers to inject multiple types into the same bean using

Let's say that our ATM machine uses different transport based on some business rules that are configured in LDAP or config file or XML or a database (does not really matter).

The point is you want it decided at runtime which transport we are going to use.

In this scenario we may want to inject three different transports and then configure a transport based on the business rules.

You are going to want to get notified when the injection is done and the bean is ready to go from a CDI perspective. To get this notifcation you would annotated an init method with the @PostConstruct annotation. Then you could pick which type of transport that you want to use.

Note the name of the method does not matter, it is the annotation that makes it an init method.

Code Listing: **AutomatedTellerMachineImpl** injecting multiple transports using new multiple qualifiers

```
package org.cdi.advocacy;

import java.math.BigDecimal;

import javax.annotation.PostConstruct;
import javax.inject.Inject;
import javax.inject.Named;

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

        private ATMTransport transport;

        @Inject @Soap
        private ATMTransport soapTransport;

        @Inject @Json
        private ATMTransport jsonTransport;

        @Inject @Json
        private ATMTransport standardTransport;


        //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() {
                //Look up values for useJSON and behindFireWall
```

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

	}


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


		transport.communicateWithBank(null);
	}

       ...
}
```

Try to follow the code above. Try to guess the output. Now compare it to this: **Output**

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

How did you do?

Using @Producer to make a decision about creation

This example builds on the last.

Perhaps you want to seperate the construction and selection of the transports from the **AutomatedTellerMachineImpl**.

You could create a **Producer** factory method that makes a decision about the creation and selection of the transport as follows:

Code Listing: **TransportFactory** deciding which transport to use/create

```
package org.cdi.advocacy;

import javax.enterprise.inject.Produces;

public class TransportFactory {

	private boolean useJSON = true;
	private boolean behindFireWall = true;


	@Produces ATMTransport createTransport() {
		//Look up config parameters in some config file or LDAP server or da
```

```
                System.out.println("ATMTransport created with producer makes decisio

                if (behindFireWall) {
                        if (useJSON) {
                                System.out.println("Created JSON transport");
                                return new JsonRestAtmTransport();
                        } else {
                                System.out.println("Created SOAP transport");
                                return new SoapAtmTransport();
                        }
                } else {
                        System.out.println("Created Standard transport");
                        return new StandardAtmTransport();
                }
        }

}
```

The advantage of this approach is that the logic to do the creation, is seperate from the actual **AutomatedTellerMachineImpl** code.

This may not always be what you want, but if it is, then the producer can help you.

The output should be the same as before.

Output

```
ATMTransport created with producer makes decisions
Created JSON transport
deposit called
communicating with bank via JSON REST transport
```

About the author
This article was written with CDI advocacy in mind by Rick Hightower [13] 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 [14]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [15], 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 [16] - Seam Weld [17] - Apache OpenWebBeans [18]

Using @Producer that uses qualifiers to make a decision about creation

This example builds on the last.

You can also inject items as arguments into the producer as follows:

Code Listing: **TransportFactory** injecting qualifier args

```
package org.cdi.advocacy;

import javax.enterprise.inject.Produces;

public class TransportFactory {

        private boolean useJSON = true;
        private boolean behindFireWall = true;


        @Produces ATMTransport createTransport( @Soap ATMTransport soapTransport,
                                                @Json ATMTransport jsonTransport) {
                //Look up config parameters in some config file
                System.out.println("ATMTransport created with producer makes decisio

                if (behindFireWall) {
                        if (useJSON) {
                                System.out.println("return passed JSON transport");
                                return jsonTransport;
                        } else {
                                System.out.println("return passed SOAP transport");
                                return soapTransport;
                        }
                } else {
                        System.out.println("Create Standard transport");
                        return new StandardAtmTransport();
                }
        }

}
```

In the above example, **createTransport** becomes less of a factory method and more of a selection method as CDI actually creates and passes the **soapTransport** and the **jsonTransport**.

Advanced topic: (Ignore this if it does not make sense) You may wonder why I create **StandardAtmTransport** and not inject it as a producer argument like **soapTransport** and **jsonTransport**. The problem is this **createTransport** is by default **@Default** and **Any** but it overrides the **StandardAtmTransport** which is also by default **Default** and **@Any**, but since **StandardAtmTransport** is overidden then if I inject **@Default ATMTransport standardTransport** as an argument then it tries to call **createTransport** since it is the **@Default**, which will then try to inject the argument **standardTransport**, which will then call **createTransport**, ad infinitum until we get a **StackTraceOverflow**. The solution is create a qualifier for the standard, say, **Standard** and use that to do the injection, or create one for the **createProduces** production, say, **@Transport**. The key here is that the injection arguments of a producer have to have different qualifiers than the production method or all hell breaks lose, cats sleeping with dogs, pandimodium. Ok. Okay. The key here is that the injection arguments have to have different qualifiers than the production method or you will get a **StackTraceOverflow** as CDI calls your production method to resovle the injection point of you production method ad infinitum.

Here is the expected output.

Output

```
ATMTransport created with producer makes decisions
return passed JSON transport
deposit called
communicating with bank via JSON REST transport
```

Using multiple @Qualifiers at the same injection point to discriminate further

You can use more than one qaulifier to further discriminate your injection target.

To demonstrate this let's define to qualifiers **SuperFast** and **StandardFrameRelaySwitchingFlubber**. Let's say at the time we have two transports that are **StandardFrameRelaySwitchingFlubber**. Let's also say that you want to inject a **transport** that is not only a **StandardFrameRelaySwitchingFlubber** but also **SuperFast**.

First let's define the qualifier annotations as follows:

Code Listing: **SuperFast** defining a new qualifier

```
package org.cdi.advocacy;

...

@Qualifier @Retention(RUNTIME) @Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface SuperFast {

}
```

Code Listing: **StandardFrameRelaySwitchingFlubber** defining another new qualifier

```
package org.cdi.advocacy;

...

@Qualifier @Retention(RUNTIME) @Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface StandardFrameRelaySwitchingFlubber {

}
```

Ok, here is the code for the **SuperFastAtmTransport** which is marked with both the **SuperFast** and the **StandardFrameRelaySwitchingFlubber** qualifiers.

Code Listing: **SuperFastAtmTransport** uses two qualifiers

```
package org.cdi.advocacy;

@SuperFast @StandardFrameRelaySwitchingFlubber
public class SuperFastAtmTransport implements ATMTransport {
        public void communicateWithBank(byte[] datapacket) {
                System.out.println("communicating with bank via the Super Fast trans
        }
}
```

Ok, we add the **StandardFrameRelaySwitchingFlubber** to the **StandardAtmTransport** as well.

Code Listing: **StandardAtmTransport** changed to use one qualifier

```
package org.cdi.advocacy;


@StandardFrameRelaySwitchingFlubber @Default
public class StandardAtmTransport implements ATMTransport {
        public void communicateWithBank(byte[] datapacket) {
                System.out.println("communicating with bank via Standard transport")
        }

}
```

Next if I want my AutomatedTellerMachineImpl to have **SuperFast** transport with **StandardFrameRelaySwitchingFlubber**, I would use both in the injection target as follows:

Code Listing: **AutomatedTellerMachineImpl** changed to use two qualifier

```
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

        @Inject @SuperFast @StandardFrameRelaySwitchingFlubber
        private ATMTransport transport;
       ...
```

About the author
This article was written with CDI advocacy in mind by Rick Hightower [13] 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 [14]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [15], 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 [16] - Seam Weld [17] - Apache OpenWebBeans [18]

Output:

```
deposit called
communicating with bank via the Super Fast transport
```

Exercise: Create a transport that is @SuperFast, @StandardFrameRelaySwitchingFlubber and an @Alternative. Then use beans.xml to activate this SuperFast, StandardFrameRelaySwitchingFlubber, Alternative transport. Send me your solution on the CDI group mailing list [22]. The first one to send gets put on the CDI wall of fame.

Exercise for the reader. Change the injection point qualifiers to make only the **StandardAtmTransport** get injected. Send me your solution on the CDI group mailing list [22]. Don't get discouraged if you get a stack trace or two that is part of the learning process. The first one to send gets put on the CDI wall of fame (everyone else gets an honorable mention).

Using @Qualfiers with members to discriminate injection and stop the explosion of annotation creation

There could be an explosion of qualifers annotations in your project. Imagine in our example if there were 20 types of transports. We would have 20 annotations defined.

This is probably not want you want. It is okay if you have a few, but it could quickly become unmanageable.

CDI allows you to descriminate on members of a qualifier to reduce the explosion of qualifiers. Instead of having three qualifier you could have one qualifier and an enum. Then if you need more types of transports, you only have to add an enum value instead of another class.

Let's demonstrate how this works by creating a new qualifier annotation called **Transport**. The **Transport** qualifier annotation will have a single member, an enum called **type**. The **type** member will be an new enum that we define called **TransportType**.

Here is the new **TransportType**:

Code Listing: **Transport** qualifier that has an enum member

```
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.inject.Qualifier;


@Qualifier @Retention(RUNTIME) @Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Transport {
        TransportType type() default TransportType.STANDARD;
}
```

Here is the new enum that is part of the **TransportType**.

Code Listing: **TransportType** enum that defines a type

```
package org.cdi.advocacy;

public enum TransportType {
        JSON, SOAP, STANDARD;
}
```

Next you need to qualify your transport instances like so:

Code Listing: **SoapAtmTransport** using **@Transport(type=TransportType.SOAP)**

```
package org.cdi.advocacy;


@Transport(type=TransportType.SOAP)
public class SoapAtmTransport implements ATMTransport {
    ...
```

Code Listing: **StandardAtmTransport** using **@Transport(type=TransportType.STANDARD)**

```
package org.cdi.advocacy;

@Transport(type=TransportType.STANDARD)
public class StandardAtmTransport implements ATMTransport {
    ...
```

Code Listing: **JsonRestAtmTransport** using **@Transport(type=TransportType.JSON)**

```
package org.cdi.advocacy;

@Transport(type=TransportType.JSON)
public class JsonRestAtmTransport implements ATMTransport {
     ...
```

Code Listing: **AutomatedTellerMachineImpl** using **@Inject @Transport(type=TransportType.STANDARD)**

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

        @Inject @Transport(type=TransportType.STANDARD)
        private ATMTransport transport;
```

About the author
This article was written with CDI advocacy in mind by Rick Hightower [13] 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 [14]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development.

Although not a fan of EJB 3 [15], 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 [16] - Seam Weld [17] - Apache OpenWebBeans [18]

As always, you will want to run the example.

Output

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

You can have more than one member of the qualifier annotation as follows:

Code Listing: **Transport** qualifier annotation with more than one member

```
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.inject.Qualifier;


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

Now CDI is going to use both of the members to discriminate for injection.

If we had a transport like so:

Code Listing: **AutomatedTellerMachineImpl** using two qualifier members to discriminate

```
public class AutomatedTellerMachineImpl implements AutomatedTellerMachine {

        @Inject @Transport(type=TransportType.STANDARD, priorityLevel=1)
        private ATMTransport transport;
```

Then we get this:

Output

```
deposit called
```

- communicating with bank via the Super Fast transport

You can match using any type supported by annotations, e.g., Strings, classes, enums, ints, etc.

Exercise: Add a member String to the qualifier annotation. Change the injection point to discriminate using this new string member. Why do you think this is counter to what CDI stands for? Send me your solution on the CDI group mailing list [22]. The first one to send gets put on the CDI wall of fame. (All others get honorable mentions.)

Conclusion

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

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

This article discussed CDI dependency injection in a tutorial format. It covers some of the features of CDI such as type safe annotations configuration, alternatives and more. There was an introduction level and and advacned level.

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 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 advocacy group [23]
- CDI advocacy blog [24]
- CDI advocacy google code project [25]
- Google group for CDI advocacy [26]
- Manisfesto version 1 [27]
- Weld reference documentation [28]
- CDI JSR299 [1]
- Resin fast and light CDI and Java EE 6 Web Profile implementation [2]
- CDI & JSF Part 1 Intro by Andy Gibson [29]
- CDI & JSF Part 2 Intro by Andy Gibson [30]
- CDI & JSF Part 3 Intro by Andy Gibson [31]

About the Author

This article was written with CDI advocacy in mind by Rick Hightower [32] 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 [14]. Rick enjoys bouncing back and forth between C, Python, Groovy and Java development. Although not a fan of EJB 3 [15], 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 53 code listings in this article

Source URL: http://java.dzone.com/articles/cdi-di-p1

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] https://jee6-cdi.googlecode.com/svn/tutorial/cdi-di-example
[6] http://code.google.com/p/jee6-cdi/wiki/MavenDITutorialInstructions
[7] http://java.dzone.com/articles/dependency-injection-an-introd
[8] http://download.oracle.com/javaee/6/tutorial/doc/gjbnr.html
[9] http://code.google.com/p/jee6-cdi/wiki/DependencyInjectionAnIntroductoryTutorial_Part1
[10] http://en.wikipedia.org/wiki/Class-responsibility-collaboration_card
[11] http://mockito.org/
[12] http://easymock.org/
[13] http://profiles.google.com/RichardHightower/about
[14] http://java.sys-con.com/node/47735
[15] http://java.sys-con.com/node/216307
[16] http://www.caucho.com/
[17] http://seamframework.org/Weld
[18] http://openwebbeans.apache.org/1.1.0-SNAPSHOT/index.html
[19] http://java.sun.com/products/jsp/reference/techart/unifiedEL.html
[20] http://download.oracle.com/javase/tutorial/java/generics/index.html
[21] http://www.garanimals.com/about.htm
[22] http://groups.google.com/group/cdiadvocate4j?pli=1
[23] http://sites.google.com/site/cdipojo/
[24] http://cdi4jadvocate.blogspot.com/
[25] http://code.google.com/p/jee6-cdi/
[26] http://groups.google.com/group/cdiadvocate4j
[27] http://cdi4jadvocate.blogspot.com/2011/03/cdi-advocacy.html
[28] http://docs.jboss.org/weld/reference/1.1.0.Final/en-US/html/
[29] http://www.andygibson.net/blog/tutorial/getting-started-with-jsf-2-0-and-cdi-in-jee-6-part-1/
[30] http://www.andygibson.net/blog/tutorial/getting-started-with-cdi-part-2-injection/
[31] http://www.andygibson.net/blog/tutorial/getting-started-with-jsf-2-0-and-cdi-part-3/
[32] https://profiles.google.com/RichardHightower/about