CONTENTS

Contexts & Dependency Injection for the Java EE Platform 1.2

UPDATED BY ANTOINE SABOT-DURAND

ABOUT CDI

Contexts and Dependency Injection for the Java EE Platform (CDI) introduces a standard set of component management services to the Java EE platform. CDI manages the lifecycle and interactions of stateful components bound to well-defined contexts. CDI provides typesafe dependency injection between components. CDI provides interceptors and decorators to extend the behavior of components, an event model for loosely coupled components, and an SPI allowing portable extensions to integrate cleanly with the Java EE environment.

CDI AND JAVA EE

CDI is included in Java EE since Java EE 6 (CDI 1.0). The EE 6 platform was designed to make sure all EE components make use of CDI services, putting CDI directly at the heart of the platform, welding together the various EE technologies.

In Java EE 7 (and CDI 1.2) this integration goes further with the automatic enablement of CDI in the platform and more integration with other specs.

This document covers the main features of CDI 1.2, included in all Java EE 7 application servers.

CDI IMPLEMENTATIONS

CDI 1.2 has 2 known implementations:

- JBoss Weld, the reference implementation, used by JBoss EAP, WildFly, Glassfish, Oracle WebLogic, and IBM Websphere, among others
- Apache OpenWebBeans, used by Apache TomEE server

GETTING STARTED WITH CDI

The easiest way to start writing and running CDI code is to write a Java EE 7 application and deploy it on your favorite server. (WildlFly is a good choice if you don't have any).

The simplest Maven POM to start developing a Java EE 7 application is only 21 lines long:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.dzone</groupId>
  <artifactId>javaee7-basic-pom</artifactId>
  <version>7.0</version>
  <packaging>war</packaging>
  <dependencies>
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-api</artifactId>
      <version>7.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
    <failOnMissingWebXml>false</failOnMissingWebXml>
  </properties>
</project>
```

ACTIVATING AND CONFIGURING CDI

In Java EE 7, CDI is activated automatically: you don't need to add anything to your application to have CDI ready in it.

When your app is launched, CDI will scan your application to find its components (i.e. managed beans); this mechanism is called *bean discovery.*

Types discovered by this mechanism will be analyzed by the container to check if they meet the requirements for becoming beans, as explained below. The bean discovery mode can be set by adding a *beans.xml* file in the module:

- */META-INF/beans.xml* for a JAR
- */WEB-INF/beans.xml* for a WAR

Bean discovery is defined for each module (e.g. JAR) of the application, also called "*bean archive*," and can have 3 modes:

- Annotated (default mode when no beans.xml file is present): only classes having specific annotations called *Bean Defining Annotations* will be discovered
- All: all of the classes will be discovered
- None: none of the classes will be discovered

Keep in mind that there is no global configuration for bean discovery, it is set only for the current bean archive.

Example of a beans.xml file setting bean discovery mode to all:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.1" bean-discovery-mode="all">
</beans>
```

BEAN-DEFINING ANNOTATIONS

When a bean archive has its bean discovery mode set to

Annotated (the default mode when no beans.xml is present), only the types with one of these annotations will be discovered:

- `@ApplicationScoped`, `@SessionScoped`, `@ConversationScoped` and `@RequestScoped` annotations
- All other normal scope types
- `@Interceptor` and `@Decorator` annotations
- All stereotype annotations (i.e. annotations annotated with `@Stereotype`),
- The `@Dependent` scope annotation.

Note that EJB session beans are an exception to the bean discovery mechanism as they are always discovered as CDI beans (unless explicitly excluded.)

EXCLUDING TYPES FROM DISCOVERY

Managed beans and session beans can be excluded from the discovered beans by adding the `@Vetoed` annotation on their defining class or package.

It can also be done in the *beans.xml* file as explained in the spec.

THE CDI CONTAINER

The container is the heart of CDI: you can think of it as the invisible conductor of your application.

It checks all possible CDI code at boot time, so exceptions at runtime are very rare in CDI—you know that something is wrong in your code at launch.

The container manages your components' lifecycle and services. It'll create class instances for you when needed and add CDI features on the provided object. This enriched object will be automatically destroyed when the scope it is bound to is destroyed.

That's why you'll never use the "new" operator on a bean class unless you want to forego all CDI features on the resulting instance.

BEANS AND CONTEXTUAL INSTANCES

CDI, at the most basic level, revolves around the notion of beans. The container discovers them at startup time by scanning classes in the deployment. A bean is defined by a set of attributes obtained by reading annotations and type on the bean definition. As we said above, the CDI container is in charge of creating and destroying bean instances according to their context, hence the term *contextual instance*. The table below introduces these attributes—they'll be detailed later in this Refcard.

TABLE 1. BEANS ATTRIBUTES

Types set	The set of Java types that the bean provides. This set is used when performing typesafe resolution (find the candidate bean for an injection point).
Qualifiers	Developer-defined annotations provide a typesafe way to distinguish between multiple beans sharing the same type. They are also used by the typesafe resolution mechanism.
Scope	Also known as "context". Determines the lifecycle and visibility of a bean. The container uses this attribute to know when to create and destroy a bean instance.
Alternative status	A bean can be defined as an alternative for an other bean. This feature can be used to simplify test creation, for instance.
Name	This optional value, is the only way to resolve a bean in a non typesafe way (i.e. with a String identifier). It allows bean access from the UI layer (JSF or JSP) or when integrating a legacy framework with CDI.

BEAN VS. CONTEXTUAL INSTANCES

In a lot of blog posts and documentation, the term *Bean* is often used instead of *contextual instance*. It's important to understand the difference. A bean is a collection of metadata associated with some code (usually a class) used by the container to provide a *contextual instance*. A *contextual instance* is the object that the container creates from the Bean attributes when an *injection point* has to be satisfied.

In short, unless you're developing an advanced CDI feature, your code will only deal with *contextual instances* at runtime.

DEFINING AN INJECTION POINT

As we just said, *contextual instances* are created and managed by the CDI container. When creating such an instance, the container may perform injection of other instances in it if it has one or more injection point.

Keep in mind that injection occurs only when the instance is created by the container.

Injection points are declared using the `@javax.inject.Inject` annotation. `@Inject` can be used in 3 places:

FIELD INJECTION

When a field is annotated with `@Inject`, the container will look for a bean with a matching type and will provide a *contextual instance* of this bean to set the field value.

Example: Injecting in private a field

```
public class MyBean {

@Inject
private HelloService service;

    public void displayHello() {
        display(service.hello();
    }
}
```

CONSTRUCTOR INJECTION

Only one constructor in a bean class may be annotated with `@Inject`. All parameters of the constructor will be resolved by the container to invoke it.

Example: injecting in a constructor

```
public class MyBean {
    private HelloService service;

    @Inject
     private MyBean(HelloService service) {
        this.service = service;
    }
}
```

METHOD INJECTION

A bean class can one or more methods annotated with `@Inject`. These methods are called *initializer methods*.

Example: injecting in a method

```
public class MyBean {
    private HelloService service;

    @Inject
    public void initService(HelloService service) {
        this.service = service;
    }
}
```

OTHER INJECTION POINTS

Two specific CDI elements always have injection point without the need of being annotated with `@Inject`:

- Producer methods
- Observer methods

See below for their usage.

DIFFERENT KINDS OF CDI BEANS

CDI provides different ways to define Beans. The type set of the bean will vary with its kind.

If needed, using the `@Typed` annotation on bean definition can further restrict this type set. `Object` will always be part of bean type set.

CDI is not affected by type erasure, so List and List will correctly be treated as two different types.

MANAGED BEANS

Managed beans are the most obvious kind of bean available in CDI. They are defined by a class declaration in a bean archive.

A class is eligible to become a managed bean if it follows the following conditions:

- It is not a non-static inner class.
- It is a concrete class, or is annotated with **`@Decorator`**.
- It has an appropriate constructor - either:
    - The class has a constructor with no parameters, or
    - The class declares a constructor annotated with @Inject.

Note: If the class is in an implicit bean archive (no beans.xml or bean discovery set to annotated) it should also have at least one of the following annotations to become a CDI managed bean:

- `@ApplicationScoped`, `@SessionScoped`, `@ConversationScoped` and `@RequestScoped` annotations
- All other normal scope types
- `@Interceptor` and `@Decorator` annotations
- All stereotype annotations (i.e. annotations annotated with `@Stereotype`)
- The `@Dependent` scope annotation

BEAN TYPES OF A MANAGED BEAN

The set of bean types for a given managed bean contains:

- The bean class
- Every superclass (including Object)
- All interface the class implements directly or indirectly

SESSION BEANS (EJB)

Local stateless, singleton, or stateful EJBs are automatically treated as CDI session beans: they support injection, CDI scope, interception, decoration, and all other CDI services. Remote EJB and MDB types cannot be used as CDI beans.

When using EJB in CDI, you have the features of both specifications. You can for instance have asynchronous behavior and observer features in one bean.

BEAN TYPES OF A SESSION BEAN

The set of bean types for a given CDI session bean depends on its definition:

If the session bean has local interfaces, it contains:

- All local interfaces of the bean
- All super interfaces of these local interfaces
- Object class

If the session bean has a no-interface view, it contains:

- The bean class
- Every superclass (including Object)

EXAMPLES

```
@ConversationScoped
@Stateful
public class ShoppingCart { ... } (1)

@Stateless
@Named("loginAction")
public class LoginActionImpl implements LoginAction {
    ... } (2)


@ApplicationScoped
@Singleton (3)
@Startup (4)
public class bootBean {
    @Inject
    MyBean bean;
}
```

1. A stateful bean (with no-interface view) defined in `@ConversationScoped` scope. It has ShoppingCart and Object in its bean types.
2. A stateless bean in `@Dependent` scope with a view. Usable in EL with name “loginAction”. It has LoginAction in its bean types.
3. It’s javax.ejb.Singleton defining a singleton session bean.
4. The EJB will be instantiated at startup triggering instantiation of MyBean CDI bean.

PRODUCERS

Producers are the way to transform classes you don’t own into CDI beans.

By adding the `@Produces` annotation to a field or a non void method of a bean, you declare a new producer and so a new Bean.

Fields or methods defining a producer may have any modifier—even `static`.

Parameters in producer methods become injection points, and are resolved by the container before invocation.

Producers are also used to defined Java EE resources (like Persistence Context or Resource) as a CDI bean.

BEAN TYPES OF A PRODUCER

It depends on the type of the producer (field type or method returned type):

- If it’s an interface, the bean type set will contain the interface all interface it extends (directly or indirectly) and Object.
- If it’s a primitive or array type, the set will contain the type and Object.
- If it’s a class, the set will contains the class, every superclass, and all interfaces it implements (directly or indirectly).

EXAMPLES

```
public class ProducerBean {
    @Produces
    @ApplicationScoped
    private List<Integer> mapInt = new ArrayList<>(); (1)

    @Produces @RequestScoped @UserDatabase
    public EntityManager create(EntityManagerFactory emf) {
      (2)
        return emf.createEntityManager();
    }
}
```

This producer field defines a bean with Bean types List, Collection, Iterable and `Object`

1. This producer method defines an EntityManager with `@UserDatabase` qualifier in `@RequestScoped` from an EntityManagerFactory bean produced elsewhere.

QUALIFIERS

Sometimes an injection point has more than one bean candidate for injection.

For instance, the following code will cause startup to fail with an "Ambiguous dependency" error:

An ambiguous injection point

```
public class MyBean {
    @Inject
    HelloService service; (1)
}

public interface HelloService {
    public String hello();
}

public class FrenchHelloService implements HelloService {
    public String hello() {
        return "Bonjour tout le monde!";
    }
}

public class EnglishHelloService implements HelloService {
    public String hello() {
        return "Hello World!";
    }
}
```

1. Both implementations of HelloService are candidates for injection here

When bean type is not enough to resolve a bean, we can create a qualifier and add it to a bean. Because qualifiers are annotations, you maintain the benefits of the CDI strong typing approach.

One qualifier by language

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

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

Qualifiers are used on bean definitions or injection points.

```
@French
public class FrenchHelloService implements HelloService {
    public String hello() {
        return "Bonjour tout le monde!";
    }
}

@English
public class EnglishHelloService implements HelloService {
    public String hello() {
        return "Hello World!";
    }
}

public class MyBean {
    @Inject
    @French
    HelloService serviceFr;

    @Inject
    @English
    HelloService serviceEn;
}
```

To match a given bean, an injection point must have a non-empty subset of the bean qualifiers (and of course a type present in its typeset.

Qualifiers can also have members. We could have solved our language problem like this:

A qualifier to qualify the language of the bean

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

    LangChoice value();

    public enum LangChoice {
        FRENCH, ENGLISH
    }
}

@Language(FRENCH)
public class FrenchHelloService implements HelloService {
    public String hello() {
        return "Bonjour tout le monde!";
    }
}
```

```
@Language(ENGLISH)
public class EnglishHelloService implements HelloService {
    public String hello() {
        return "Hello World!";
    }
}

public class MyBean {
    @Inject
    @Language(value = FRENCH)
    HelloService serviceFr;

    @Inject
    @Language(value = ENGLISH)
    HelloService serviceEn;
}
```

`@Nonbinding` annotations can be applied to a qualifier member to exclude it from the qualifier resolution.

A qualifier with a non binding member

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD, PARAMETER})
public @interface MyQualifier {
    @Nonbinding
    String comment(); (1)
}
```

1. The CDI container will treat two instances of MyQualifier with different comment() values as the same qualifier.

BUILT-IN QUALIFIERS

CDI includes the following built-in qualifiers:

TABLE 2. BUILT-IN QUALIFIERS

`@Named`	set bean name for weak typed environment (EL, Javascript)
`@Default`	added to all beans without other qualifiers, or having only the @Named qualifier
`@Any`	added to all beans for programmatic lookup and decorators
`@Initialized`	to qualify events when a context is started
`@Destroyed`	to qualify events when a context is destroyed

CONTEXTS

All Beans have a scope defined by an annotation. When there's no scope annotation on a bean its scope is `@Dependent`.

A scope should be seen as a label to design a context object. Through its scope, a bean is bound to a context. Contexts are in charge of creating, storing, and destroying contextual instances

- The Container is in charge of creating and destroying contexts.
- A context may be inactive without being destroyed.

To make it short, a contextual instance for a given Bean is always a singleton in its context. Remember that contextual instances are created by the container when they are requested, not when their context is created. Scopes are not used to distinguish beans: if two beans with the same type and qualifiers exist in two scopes, there will be ambiguity when injecting them.

While it's possible to create new contexts and scopes with portable extensions, CDI provides the following built-in scopes and their matching contexts out of the box:

`@Dependent` (default) bean has the same scope as the one in which it's injected `@ApplicationScoped` instance is linked to application lifecycle `@SessionScoped` instance is linked to http session lifecycle `@RequestScoped` instance is liked to http request lifecycle `@ConversationScoped` lifecycle manually controlled within session.

SCOPE EXAMPLES

```
public class BaseHelloService implements HelloService
    { ... } (1)

@RequestScoped (2)
public class RequestService {
    @Inject HelloService service;
}

@ApplicationScoped (3)
public class ApplicationService {
    @Inject RequestService service; (4)
}
```

1. Bean has default scope @Dependent, instances are created for each injection
2. Bean is `@RequestScoped`. Instance is created by request context and destroyed with request context
3. Bean is `@ApplicationScoped`. Instance is created by application context and will live for the duration of the application itself
4. No problem to inject bean from another scope: CDI will provide the right bean

TYPESAFE RESOLUTION

When resolving beans for a given injection point, the container considers the set of types and qualifiers of all available beans to find the right candidate.

The actual process is a bit more complex with integration of Alternatives, but the general idea is here.

If the container succeeds in resolving the injection point by finding one and only one eligible bean, the `create()` method of this bean will be used to provide an instance for it.

PROGRAMMATIC LOOKUP

Sometimes it is useful to resolve a bean at runtime or find all beans that match a given type. Programmatic lookup brings this powerful feature thanks to the `Instance<T>` interface.

Request an instance at runtime with Instance

```
public class MyBean {
    @Inject
    Instance<HelloService> services; (1)

    public void displayHello() {
        if(!(services.isUnsatisfied() ||
          services.isAmbiguous())) (2)
            display(services.get().hello()); (3)
    }
}
```

1. `Instance<T>` injection points are always satisfied and never fail at deployment time
2. `Instance<T>` provides test methods to know if requesting an instance is safe
3. With `Instance<T>` you control when a bean instance is requested with the `get()` method

As instance extends the Iterable interface, you can use it to loop on instances of all beans of the specified type and qualifiers.

```
public class MyBean {
    @Inject
    @Any (1)
    Instance<HelloService> services;

    public void displayHello() {
        for (HelloService service : services) {
            display(service.hello());
        }
    }
}
```

1. All beans have `@Any` qualifier so this injection point gets an Instance pointing to all beans having the type HelloService

Finally, programmatic lookup helps you to select a bean by its type and qualifier.

```
public class MyBean {
    @Inject
    @Any
    Instance<HelloService> services;

    public void displayHello() {
        display(
            services.select(
                 new AnnotationLiteral()<French>{})
                 .get()); (1)
    }
}
```

1. `Select()` also accepts a type

The CDI spec provides the `AnnotationLiteral` and `TypeLiteral` classes to help you create instances of an annotation or parameterized type.

ACCESSING CDI FROM NON-CDI CODE

When you need to retrieve a CDI bean from non-CDI code, the CDI class is the easiest way:

Using CDI.current() to access the bean graph

```
public class NonManagedClass {

    public HelloService getHelloService() {
        Instance<HelloService> services =
          CDI.current().select(HelloService.class,
          new AnnotationLiteral()<French> {});
        if (!(services.isUnsatisfied ||
          services.isAmbiguous))
            return services.get();
        else
          return null;
    }
}
```

The `CDI.current()` static method returns a CDI object that extends `Instance<Object>`. As all beans have `Object` in their type set, it allows you to perform a programmatic lookup on your entire beans collection.

CDI can also return the BeanManager—a class giving you access to advanced CDI features, including bean resolution.

For backwards compatibility, the BeanManager is also accessible through JNDI with the name java:comp/BeanManager.

You can learn more on BeanManager in the *spec*.

EVENTS

Events provide a mechanism for loosely coupled communication between components. An event consists of an event type, which may be any Java object, and optional event qualifiers.

THE EVENT OBJECT

Events are managed through instances of `javax.enterprise.event.Event`. Event objects are injected based on the event type.

```
@Inject Event<LoggedInEvent> normalEvent;
@Inject @Admin Event<LoggedInEvent> adminEvent;
```

Events are fired by calling `fire()` with an instance of the event type to be passed to observers.

```
event.fire(new LoggedInEvent(username));
```

OBSERVERS

Observers listen for events with observer methods. An observer methods should be defined in a bean, and must have one of its parameters annotated with `@javax.enterprise.event.Observes`—the annotated parameter defines the type to be observed.

Additional parameters to an observer method are normal CDI injection points.

```
public void afterLogin(@Observes LoggedInEvent event) {
}

public void afterAdminLogin(@Observes @Admin LoggedInEvent
event) {
}
```

CONDITIONAL OBSERVERS

If a *contextual instance* of a bean with an observer method doesn't exist when the corresponding event is fired, the container will create a new instance to handle the event. This behavior is controllable using the receive value of `@Observes`.

TABLE 4. VALUES OF RECEIVE MEMBER IN @OBSERVES

RECEPTION VALUE	MEANING
`IF_EXISTS`	The observer method is only called if an instance of the component already exists.
`ALWAYS`	The observer method is always called. If an instance doesn't exist, one will be created. This is the default value.

TRANSACTIONAL OBSERVER

Event observers are normally processed when the event is fired. For transactional methods, however, it is often desirable for the event to fire at a certain point in the transaction lifecycle, such as after the transaction completes. This is specified with the during value of `@Observes`.

If a transaction phase is specified but no transaction is active, the event is fired immediately.

TABLE 5. VALUES OF DURING MEMBER IN @OBSERVES

TRANSACTIONPHASE VALUE	MEANING
`IN_PROGRESS`	The event is called when it is fired, without regard to the transaction phase. This is the default value.
`BEFORE_COMPLETION`	The event is called during the `before completion` phase of the transaction.
`AFTER_COMPLETION`	The event is called during the `after completion` phase of the transaction.
`AFTER_FAILURE`	The event is called during the `after completion` phase of the transaction, but only when the transaction fails.
`AFTER_SUCCESS`	The event is called during the `after completion` phase of the transaction, but only when the transaction completes successfully.

INTERCEPTORS AND DECORATORS

CDI supports two mechanisms for dynamically adding or modifying the behavior of beans: interceptors and decorators.

INTERCEPTORS

Interceptors provide a mechanism for implementing functionality across multiple beans, and bean methods, that is orthogonal to the core function of those beans.

It is often used for non-business features like logging or security. For instance, in Java EE 7 the JTA specification provides the `@Transactional` interceptor to control transactions for the current invocation.

INTERCEPTOR BINDING TYPE

An interceptor binding is an annotation annotated with the `@javax.interceptor.InterceptorBinding` meta-annotation.

Its goal is to bind the interceptor code to the bean or method to intercept.

Defining an interceptor binding

```
@Inherited
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@InterceptorBinding
    public @interface Loggable {
}
```

INTERCEPTOR DEFINITION

An interceptor is a bean declared with the `@javax.interceptor.Interceptor` annotation.

Its matching interceptor binding should also be added to its declaration.

Since CDI 1.1, the interceptor can be enabled with `@javax.annotation.Priority` annotation—this defines its resolution order.

Method interceptors should have a method annotated `@javax.interceptor.AroundInvoke` that takes the `javax.interceptor.InvocationContext` as a parameter.

Defining an interceptor

```
@Interceptor
@Loggable (1)
@Priority(Interceptor.Priority.APPLICATION) (2)
public class TransactionInterceptor {
    @AroundInvoke (3)
    public Object logMethod(InvocationContext ctx) {
        ...
    }
}
```

1. The interceptor binding to bind this code to this annotation
2. The `@Priority` annotation to enable and prioritize the interceptor.
3. The `@AroundInvoke` annotation indicates which method does the interception

USING INTERCEPTORS

Thanks to interceptor binding it is very easy to apply an interceptor to bean or method.

```
public class MyBean {
    @Logabble
    public void doSomething() {
        ...
    }
}

@Logabble
public class MyOtherBean {
    public void doSomething() {
        ..
    }
}
```

When applied on a bean, all the bean's methods will be intercepted.

ACTIVATING AND ORDERING INTERCEPTORS

In Java EE 7 the easiest way to activate an interceptor in to use the `@Priority` annotation.

It is also possible to do it in beans.xml file as explained in the spec.

DECORATORS

Decorators also dynamically extend beans but with a slightly different mechanism than interceptors. Where interceptors deliver functionality orthogonal to potentially many beans, decorators extend the functionality of a single bean-type with functionality that is specific to that type.

Decorators are an easy way to change the business operation of an existing bean.

A decorator is a bean annotated with `@javax.decorator.Decorator`.

A decorator only decorates the interfaces that it implements (i.e. to be decorated a bean must implement an interface).

Example: a decorator firing an event in addition of expected code execution

```
@Decorator (1)
@Priority(Interceptor.Priority.APPLICATION) (2)
public abstract class EventingDecorator implements
  MyBusiness (3) {
    @Inject
    @Delegate (4)
    MyBusiness business;

    @Inject
    Event<String> evt;

    public void doSomething(String message) {
        business.doSomething(message);
        evt.fire(message)
    }
}
```

1. The decorator is defined with the matching annotation
2. Decorators are enabled and prioritized like interceptors
3. As all methods don't have to be decorated (i.e. implemented), the decorator is allowed to be an abstract class
4. The decorated bean is injected with the specific `@Delegate` annotation.

A decorator must declare a single delegate injection point annotated `@javax.decorator.Delegate`. The delegate injection point is the bean to be decorated. Any calls to the delegate object that correspond to a decorated type will be called on the decorator, which may in turn invoke the method directly on the delegate object. The decorator bean does not need to implement all methods of the decorated types and may be abstract.

ACTIVATING AND ORDERING DECORATORS

In Java EE 7 the easiest way to activate a decorator in to use the `@Priority` annotation.

It is also possible to activate a decorator in beans.xml, as explained in the *spec*.

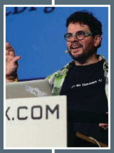
Decorators are always called after interceptors.

GOING FURTHER WITH CDI

This document is only an introduction to CDI—many topics are not covered here.

To go further, you can go to the *learn section* of the CDI specification website where a lot of resources are available to go deeper in CDI learning.

ABOUT THE AUTHOR



After having worked 16 years as an IT consultant, **ANTOINE SABOT-DURAND** joined Red Hat in 2013 and became the CDI specification leader for CDI 1.2 and 2.0. He's also working as a CDI evangelist by giving talk in conferences like Java One or Devoxx, and by helping various open source projects to leverage their CDI integration.

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776