Chapter 1. Use CDI in JOnAS

Table of Contents

1.1. Java Context and Dependency Injection (CDI)

1.1.1. What is this all about ?

In a nutshell, the *Java Context and Dependency Injection* (CDI for short) is a *Java EE 6 specification* that defines a powerful set of complementary services that help improve the structure of application code.

- A well-defined lifecycle for stateful objects bound to lifecycle contexts, where the set of contexts is extensible
- A sophisticated, typesafe dependency injection mechanism, including the ability to select dependencies at either development or deployment time, without verbose configuration
- Support for Java EE modularity and the Java EE component architecture - the modular structure of a Java EE application is taken into account when resolving dependencies between Java EE components
- Integration with the Unified Expression Language (EL), allowing any contextual object to be used directly within a JSF or JSP page
- The ability to decorate injected objects
- The ability to associate interceptors to objects via typesafe interceptor bindings
- An event notification model
- A web conversation context in addition to the three standard web contexts defined by the Java Servlets specification
- An SPI allowing portable extensions to integrate cleanly with the container

CDI can be seen as a standard alternative to proprietary containers (Spring framework, Guice, ...).

1.2. CDI Services

Here is a quick description of the major features of the CDI programming model.

1.2.1. Injection

CDI injection is based on the Dependency Injection specification (JSR 330 [http://jcp.org/en/jsr/detail?id=330]) that defines basic injection capabilities. CDI extends this specification by adding the Qualifier annotation concept that helps to disambiguate between 2 possible injection targets.

It provides means to assemble your application components very easily, using only annotations, in a type-safe way.

Definition of a qualifier `@Synchronous` annotation:

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

Placed on top of a bean class, it qualifies the bean.

```
/**
 * Declaration of a PaymentProcessor bean qualified with @Synchronous.
 */
@Synchronous
class SynchronousPaymentProcessor
        implements PaymentProcessor {
    ...
}
```

It's used at the injection point to help the container choose the right `PaymentProcessor` bean to inject.

```
/**
 * Declared a dependency on a PaymentProcessor bean qualified with @Synchronous
 */
@Inject @Synchronous PaymentProcessor paymentProcessor;
```

1.2.2. Interception

Interception in the CDI model is also annotation based. Any CDI bean can be intercepted using the Java EE 5 interception mechanism (`@AroundInvoke`).

Interceptors allow common, cross-cutting concerns to be applied to beans via custom annotations. Interceptor types may be individually enabled or disabled at deployment time.

The `AuthorizationInterceptor` class defines a custom authorization check:

```
@Secure @Interceptor
public class AuthorizationInterceptor {
    @Inject @LoggedIn User user;
    @Inject Logger log;
    @AroundInvoke
    public Object authorize(InvocationContext ic) throws Exception {
        try {
            if ( !user.isBanned() ) {
                log.fine("Authorized");
                return ic.proceed();
            }
            else {
                log.fine("Not authorized");
                throw new NotAuthorizedException();
            }
        }
        catch (NotAuthenticatedException nae) {
            log.fine("Not authenticated");
            throw nae;
        }
    }
```

```
}
```

The `@Interceptor` annotation, identifies the `AuthorizationInterceptor` class as an interceptor. The `@Secure` annotation is a custom interceptor binding type.

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

The `@Secure` annotation is used to apply the interceptor to a bean:

```
@Model
public class DocumentEditor {
    @Inject Document document;
    @Inject @LoggedIn User user;
    @Inject @Documents EntityManager em;
    @Secure
    public void save() {
        document.setCreatedBy(currentUser);
        em.persist(document);
    }
}
```

Note

By default interceptors are not activated. They have to be explicitely declared in the `beans.xml` file.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <interceptors>
    <!-- Interceptors are only activated if listed in the beans.xml file -->
    <class>org.ow2.jonas.examples.cdi.interceptor.AroundMethodLogInterceptor</
class>
  </interceptors>
</beans>
```

1.2.3. Decoration

The CDI programming model eases the usage of the decorator pattern [http://en.wikipedia.org/wiki/Decorator_pattern].

An abstract bean type (implementing an interface) can be annoted with `@Decorator`, it has a `@Delegate` annotated member and overrides one (or more) methods of the interface.

```
public interface User {
    String getName();
    String getCompanyName();
}
```

The `LongNameUserDecorator` will be applied on all `User` beans.

```
@Decorator
public abstract class LongNameUserDecorator implements User {

    @Inject
    @Delegate
    private User delegate;

    public String getName() {
        return delegate.getName() + " (Last name added with @Decorator)";
    }
```

Note

Like interceptors, decorators may be easily enabled or disabled at deployment time with the `beans.xml` file.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <decorators>
    <!-- Decorators are only activated if listed in the beans.xml file -->
    <class>org.ow2.jonas.examples.cdi.user.decorator.LongNameUserDecorator</
class>
  </decorators>
</beans>
```

1.2.4. Eventing

CDI comes with a full featured eventing support. It nicely decouple the observed object from the observers: the container links each of them together.

1.2.4.1. Observed Element

```
@SessionScoped @Model
public class Login implements Serializable {
    @Inject Credentials credentials;
    @Inject @Users EntityManager userDatabase;
    @Inject @LoggedIn Event<User> userLoggedInEvent;
    ...
    private User user;

    @Inject
    void initQuery(@Users EntityManagerFactory emf) {
        ...
    }

    public void login() {
         List<User> results = ... ;
         if ( !results.isEmpty() ) {
             user = results.get(0);
             userLoggedInEvent.fire(user);
         }
    }
    @Produces @LoggedIn User getCurrentUser() {
         ...
    }
}
```

The Login class is injected with an `Event<User>` instance (notice the `@LoggedIn` qualifier annotation).

When the `User` gets logged in, `userLoggedInEvent.fire(user)` is executed and all observer methods will be synchronously invoked.

1.2.4.2. Observer Method

```
@SessionScoped
public class Permissions implements Serializable {
    @Produces
    private Set<Permission> permissions = new HashSet<Permission>();
    @Inject @Users EntityManager userDatabase;
    Parameter<String> usernameParam;

    ...

    void onLogin(@Observes @LoggedIn User user) {
        permissions = new HashSet<Permission>(
           userDatabase.createQuery(query)
                       .setParameter(usernameParam, user.getUsername())
                       .getResultList() );
```

```
    }
}
```

Notice the `onLogin()` method, it's an observer method because of the `@Observes` parameter annotation. This method will be invoked when a `User` event will be fired by a `@LoggedIn` annotated `Event<User>` field.

1.2.5. Scoping/Context

Instance's life-cycle is bound to a scope (request, session, application).

Beans can de declared to be bound in one scope

```
@SessionScoped
public class Order { ... }
```

Injection/Producers points can also be annotated with scope requirements:

```
public class Shop {
   @Produces @ApplicationScoped @All
   public List<Product> getAllProducts() { ... }
   @Produces @SessionScoped @WishList
   public List<Product> getWishList() { ..... }
}
```

1.3. Activation

In order to activate CDI on JOnAS, the 'cdi' service must be declared in the `$JONAS_BASE/conf/jonas.properties` in the `jonas.services` property.

```
jonas.services    registry,jmx,... ,cdi
```

JOnAS is using Weld (the CDI reference implementation) under the hood.

Warning

Current CDI support in JOnAS is limited to web applications. Future versions will enable EJB and EAR support.

1.4. Resources

- Dependency Injection Specification (JSR 330 [http://jcp.org/en/jsr/detail?id=330])
- @Inject Google project [http://code.google.com/p/atinject/]
- Java Context and Dependency Injection Specification (JSR 299 [http://jcp.org/en/jsr/detail?id=299])
- RedHat JBoss Weld [http://seamframework.org/Weld] (Reference Implementation) + Documentation [http://docs.jboss.org/weld/reference/1.1.0.Final/en-US/html/]
- Apache WebBeans [http://openwebbeans.apache.org/] (ASL2 Licensed Implementation) + Documentation [http://openwebbeans.apache.org/1.0.0-SNAPSHOT/documentation.html]
- JOnAS CDI Sample (in `examples/` or online [http://websvn.ow2.org/listing.php?repname=jonas&path=%2Fsandbox%2Fsauthieg%2Fjonas-cdi-webapp%2F])