Java™ EE 6 introduced a new set of features and services called Contexts and Dependency Injection (CDI). Java EE 7 added more features in this area. CDI relies on and provides services to other technologies introduced in Java EE 6. This article discusses the concept of dependency injection, why it is desirable, the problems it solves, how it is applied in Java EE, and its relationships with these other technologies.

Editor's note (May 2016): This article is updated to reflect Java EE 7, which is fully supported in WebSphere Application Server V8.5.5.6.

Introduction

In Java, classes can have variables (fields) that are non-primitive types. In complex applications, the type of these fields can represent complex technological solutions. For example, a class `Record` can have a field of type `javax.sql.DataSource`, indicating that an instance of `Record` depends on a relational data source for its functioning. In Java EE, you acquire an instance of this data source by looking up an instance through JNDI. An example is shown in Listing 1.

```
Listing 1. Getting a reference to a dependency by using JNDI
InitialContext ic = new InitialContext();  /*
   * This contains references to the environment
   * provided by a J2EE container
   */
DataSource ds = (DataSource)ic.lookup("java:comp/env/jdbc/mydbDS");
```

With dependency injection, you do not have to figure out how to instantiate the dependency, and you can rely on the Java EE container to do more. The example in Listing 1 can now be simplified as shown in Listing 2.

```
Listing 2. Getting a reference to a dependency by using Dependency Injection
public class MyClass
{
   @Resource(name="jdbc/mydbDS" type="javax.sql.DataSource") DataSource ds;
}
```

This is the very basic idea behind dependency injection — a simple way to declare a dependency in your code, and one that enables the run time to take over the mechanics of instantiating the
dependency. This simplicity, however, is only a foundation. Built on top of this idea is a host of solutions and services that are explored in this article.

If this were all there was to dependency injection — a simpler mechanism for dependency declaration — then this wouldn't be an important topic. But this simple syntactical mechanism unlocks a whole host of capabilities and solutions to you, the Java EE programmer. Java EE 5 introduced dependency injection, but that particular technology is more properly called resource injection. The container injects objects that were of specific types known to the container. In dependency injection in Java EE 6 and beyond, the container injects types that are defined by you, the application programmer.

Before delving into this topic further, a word about its pedigree. Dependency injection is a technology that surfaced in various implementations many times before it was introduced into Java EE. The Spring Framework and the Google® Guice library are popular implementations. In JSR 330, an attempt was made to include these capabilities into the J2SE platform. JSR 299 is a specification that used the APIs defined in JSR 330 and added more capabilities to support Java EE needs. IBM® WebSphere® Application Server V8 and V8.5 (full profile) are fully compliant Java EE 6 containers and implement JSR 299. In V8.5.5, WebSphere Application Server Liberty profile introduced support for the Java EE 6 Web Profile, which included support for CDI 1.0. As of V8.5.5.6, Java EE 7 is supported in full compliance mode and includes support for CDI 1.2.

**Injectable types**

To understand the capabilities that CDI brings to the table, and to understand its programming model, you first need to understand the interrelationships between the various specifications that make up Java EE 7 (Figure 1).

**Figure 1. Relationships between Java EE 7 dependency and injection-related specifications**

![Diagram showing relationships between Java EE 7 specifications](image)

The Managed Beans specification defines services and a programming model that all the other specifications rely on and build upon. It defines some resource injection and lifecycle services; most notably, the PostConstruct and the PreDestroy annotations, which are used to indicate lifecycle callback methods to the container that are specified by an application developer.
The PostConstruct and PreDestroy annotations are also part of J2SE as of Java EE 6. As a result, implementations in J2SE that are not Java EE support some dependency injection functions.

The JSF specification also describes a technology known as managed beans. These managed beans are different from the managed beans described in this article. The Java EE foundational managed beans are a generalization of the JSF one, and are not limited to web modules.

Put another way, a managed bean is a component that the container constructs and destroys (as part of managing its lifecycle).

Three kinds of artifacts that can be injected:

- **Managed beans**: EJB session beans, classes annotated with @ManagedBean, and classes that conform to the CDI rules for becoming a managed bean (see [Types that support injection](#)).
- **Java EE resources**: Resources that can be referenced from the component environment naming space.
- **Arbitrary objects** that are returned by producer methods and fields.

Listing 2 showed an example of resource injection. Listing 3 shows an example of a CDI-managed bean. CDI-managed beans are beans by virtue of the rules that are defined in the CDI specification, which supports the injection of an EJB reference.

### Listing 3. Injection into a JAX-RS resource

```java
@Path("/node")
public class NodeView
{
    @EJB(name="ejb/nodeTree", beanInterface=Nodes.class) Nodes nodeTree;
    
    @GET
    @Path("/{id}")
    @Produces("application/xml")
    public Response getNode(@PathParam(value="id") int id)
    {
        TaxonomyNode node = nodeTree.findNodeInCache(id);
    }
}
```

### Types that support injection

The kind of types and component types that can support injection (that is, objects can be injected into them) are enumerated in the Java EE 7 specification, as shown in Table 1.

### Table 1. Component classes that support injection

<table>
<thead>
<tr>
<th>Spec</th>
<th>Classes supporting injection</th>
<th>Supports @PostConstruct?</th>
<th>Supports @PreDestroy?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servlet</td>
<td>servlets, servlet filters, event listeners</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JSP</td>
<td>tag handlers, tag library event listeners</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JSF</td>
<td>scoped managed beans</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Managed beans in CDI

In Java EE 7, a managed bean can be declared by annotating its class with the @ManagedBean annotation. CDI specifies that a Java class qualifies as a managed bean if it is defined to be a managed bean — either by annotation, or because it is a session bean, or if it conforms to all of the following rules:

- It is not a non-static inner class.
- It is a concrete class or is an annotated @Decorator.
- It is not an EJB component.
- It does not implement javax.enterprise.inject.spi.Extension.
- It has an appropriate constructor, either:
  - It has a no-argument constructor, or
  - It has a constructor-annotated @Inject.

As you can see from this list, many classes can become managed beans. However, just because something meets the rules of being a managed bean, it isn't treated as one. Listing 4 shows an example of when a class is treated like a managed bean by the container.

Listing 4. Normal versus managed bean treatment

```java
public class PurchaseOrder {
    @Inject User customer; // Container instantiates and manages instance of User
    User inventoryManager;  // Application instantiates and manages instance of User.
    // This typically happens with a new User() or equivalent.

    ...
}
```

Managed beans are associated with bean types. Bean types are the legal types that can be specified at the injection target for a bean; that is, the client-visible types. Client-visible types can include the bean class, its super-classes, and any interfaces that it implements directly or indirectly. Thus, even though interfaces cannot be managed beans, the injection target can specify an interface as the type to be injected if the bean to be injected implements that interface.
In Listing 4, if `User` is an interface that is implemented by a concrete class called `Customer`, an instance of `Customer` can be injected into the `customer` variable because `User` is one of the bean types of `Customer`.

Generic type parameters are not legal bean types. However, beans can have type parameters. Listing 5 shows examples.

**Listing 5. Legal bean types that use type parameters**

```java
public class Wrapper<T>
{
    @Inject private T myBean;  // Injection fails. This is illegal
    @Inject private MyBean<T> soup; // Legal
}
```

Beans can also be injected by using constructor parameters, as shown in Listing 6.

**Listing 6. Constructor parameter injection**

```java
public class AppContext
{
    private String userName;
    @Inject AppContext(User currentUser)
    {
        userName = currentUser.getUserName();
    }
}
```

A class can specify only one of its constructors with `@Inject`.

Setter methods are another mechanism for injection as shown in Listing 7. Such methods are referred to as initializer methods. Note that the naming conventions used by JavaBeans (“setXYZ”) are irrelevant to how an initializer method works.

**Listing 7. Parameter injection with setter methods**

```java
public class AppContext
{
    private String userName;
    @Inject public void setUserName(User user)
    {
        this.userName = user.getUserName();
    }
}
```

**Bean EL (Expression Language) names**

Managed beans are referenceable from JSPs or JSF artifacts by their **EL name**, which is their name in the unified EL expression language. For example, the bean that is shown in Listings 6 and 7 can be referenced as shown in Listing 8.

**Listing 8. Referencing a managed bean in JSF**

```html
<h:outputText value="#{appContext.userName}"/>
```
The @Named annotation is essential for the bean to be referenceable from JSPs and from JSF. This annotation can be used to specify an EL name as its value. However, if no EL name is specified, the container uses a default name. This name is the unqualified name of the class (for a CDI-managed bean or a session bean) with the first letter changed to a lowercase character.

Only managed beans are referenceable in EL expressions. If a bean type is not a managed bean (for example, if it is an interface), but is a legal bean type, it does not have an EL name and can't be referenced in an EL expression.

**Built-in managed beans**

The CDI specification lists some classes that Java EE containers are required to support as managed beans:

- `javax.transaction.UserTransaction` permits injection of a reference to the JTA User-Transaction.
- `javax.security.Principal` permits injection of a Principal representing the current caller identity.
- `javax.validation.ValidationFactory` permits injection of the default Bean Validation ValidationFactory.
- `javax.validation.Validator` permits injection of a Validator for the default Bean Validation ValidationFactory.

Instances of these classes do not follow the normal rules of what it means to be a CDI-managed bean. Nevertheless, containers support the instantiation, injection, and disposal of such objects just as if they were managed beans.

**Packaging considerations**

You might want to have all of your container-managed classes organized into a separate JAR file for organizational purposes. Java EE containers like this too: at deployment time, containers are required to scan through all the deployed classes to determine which objects need to be managed by the container. The more code that needs to be scanned, the slower the deployment.

To assist containers in this task, the CDI specification defines that only JAR or Java EE modules that contain a file called beans.xml are scanned. Such files or modules are called bean archives. If you define CDI-managed beans but you do not have a beans.xml file, the container does not scan your code for CDI-related annotations, and injection does not work.

The beans.xml file must be placed in one of these locations:

- For a library JAR, EJB JAR, application client JAR, or RAR archive, the file needs to be in the META-INF directory.
- The WEB-INF directory of a WAR.
- For a directory in the JVM classpath that contains classes, the code is scanned if the beans.xml file is in the META-INF subdirectory.

The beans.xml file can be completely empty. See Alternatives for details on what can be put into the file.
Resource bindings

For injection of Java EE resources to work correctly in managed beans, you must specify binding information. The binding information maps the name that is used in the environment context (which is specified in the bean code) to the JNDI name that points to a resource available in a container. In WebSphere Application Server, you specify the binding information in the ibm-managed-beans-bnd.xml file. IBM Rational® Application Developer and IBM Rational Software Architect have easy menu options to create this file (Figure 2). This tooling is not yet available for WebSphere Liberty profile servers.

Figure 2. Creating the ibm-managed-beans-bnd.xml file in Rational Software Architect 8.5

An example ibm-managed-beans-bnd.xml file is shown in Listing 9.

Listing 9. Example contents of a ibm-managed-beans-bnd.xml file

```xml
<managed-beans-bnd xmlns="http://websphere.ibm.com/xml/ns/javaeexm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   version="1.0">
  <managed-bean class="com.ibm.issw.taxonomy.producers.ConnectionManager">
    <resource-ref name="jdbc/supplierDB" binding-name="jdbc/ptiDS" />
    <resource-ref name="jdbc/busPartDB" binding-name="jdbc/ptiDS" />
  </managed-bean>
</managed-beans-bnd>
```
A better service locator

The Service Locator pattern is an application design technique that isolates application code from the details of the service implementation that it deals with. The pattern enables the application to code to interfaces, and to insulate itself from details that might make it less portable or maintainable. For example, an application might want to access a service that renders an image, and this rendering service might work differently depending on the image type. Or, an application might want to access encryption or decryption services, and depending on the hardware or platform it is running on, these services might be available either natively, or through software-only libraries.

The key point here is that the Service Locator itself is part of application code. Dependency injection can provide the same level of isolation from service details, but because injection happens through the Java EE container, your job is easier. The sections Qualifiers and Alternatives describe how this isolation is possible.

Qualifiers

Let’s say that you are writing a program that accesses data from two different external applications. One of the applications provides the data through HTTP. The other provides the data through an FTP location. You are dealing with the classes shown in Listing 10.

Listing 10. Multiple implementations of an interface

```java
public interface Downloader
{
    // returns the local filesystem location to the downloaded data
    public String read(String location, String userName, String password);
}

public class WebDownloadImpl implements Downloader
{
    public String read(String location, String userName, String password)
    {
        // Implementation details
    }
}

public class FTPDownloadImpl implements Downloader
{
    public String read(String location, String userName, String password)
    {
        // Implementation details
    }
}
```

Now, define two annotations: @Ftp and @Web. These annotations are your qualifiers for the managed bean that you plan to inject (Listing 11).
Listing 11. Qualifier definition by using annotations

```java
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
public @interface Ftp{
}

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
public @interface Web{
}
```

In particular, notice the use of the @Qualifier meta-annotation. The @Qualifier tells CDI that the annotation that is defined is a managed bean qualifier. Note the RetentionPolicy. If @Retention is omitted, annotations default to a RetentionPolicy of CLASS, which means that the run time won't be aware of the qualifier that is being applied to a bean. Listing 12 shows how you use these qualifiers.

Listing 12. Using the defined qualifiers

```java
@Web
public class WebDownloadImpl implements Downloader{
    public String read(String location, String userName, String password){
        
    }
}

@Ftp
public class FTPDownloadImpl implements Downloader{
    public String read(String location, String userName, String password){
        
    }
}

public class DatasourceReader{
    @Inject @Ftp Downloader dinoAppReader;
    @Inject @Web Downloader modernAppReader;
    
    public void doStuff(){
        String fileDino = dinoAppReader.read("location", "username", "password");
        String fileModern = modernAppReader.read("location", "username", "password");
    }
}
```

Multiple qualifiers can be specified at an injection point. For example, @Inject @Web @Secure modernAppReader can imply container injection of a bean that supports HTTPS (versus HTTP).

Alternatives

Let's look at a different scenario. In production, you have some data that is being encrypted by using hardware native encryption. On your test system, you don't have this hardware and must encrypt the data by using software only. Your code is similar to Listing 13.
Listing 13. Alternative implementations for a single interface

```java
public interface SecureDataManager {
    public void encrypt(String data);
}
---
@Default
public class NativeSecureDataManager implements SecureDataManager {
    public void encrypt(String data);
}
---
@Alternative
public class SoftwareSecureDataManager implements SecureDataManager {
    public void encrypt(String data);
}
```

@Default and @Alternative are annotations that are introduced by CDI. If your beans.xml file is similar to Listing 14, then the code in Listing 15 instantiates an instance of SoftwareSecureDataManager into the injected field. If the beans.xml file is empty, the @Default type of SecureDataManager is injected.

Listing 14. Defining the active alternative in a beans.xml file

```xml
<beans xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaloation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
    <alternatives>
        <class>com.ibm.issw.cdi.SoftwareSecureDataManager</class>
    </alternatives>
</beans>
```

Listing 15. Using an alternative at the injection point

```java
public class DataHandler {
    @Inject private SecureDataManager sdm;
}
```

Many alternatives can be defined for a type, but only one of them can be active at any point in a bean archive.

**Normal and pseudo scopes**

Just as a variable has a known scope (that is, a period in a thread of execution when it is available for use), a managed bean also has a known scope. Broadly speaking, scopes fall into one of two categories: normal scopes or pseudo scopes. The difference between the two types boils down to whether instances of the managed bean are shared in the scope (normal) or whether they are not shared (pseudo).

Java EE components such as servlets and EJBs do not have a well-defined scope. For example, servlets and stateless session beans are stateless. Therefore, a client of these components cannot differentiate between one instance of the component over another. As a result, it is meaningless to speak of the scope of such components. Stateful EJBs have client-visible state, but the boundaries of scope of the state are not well-defined because the state can be shared between any two clients.
by passing along the reference to the EJB. Scopes have no beginning or ending, except what you enforce.

**Built-in scopes**

The CDI specification defines some built-in scopes. The normal built-in scopes are @RequestScoped, @SessionScoped, @ApplicationScoped, and @ConversationScoped. The only built-in pseudo scope is @Dependent. If scope is not explicitly specified for a bean, then a default scope takes effect. In the absence of stereotypes (not covered in this article), the default scope is @Dependent.

- The request scope is active:
  - During the `service` method of a servlet, during the `doFilter` method of a servlet filter, and when any `ServletRequestListener` or `AsyncListener` is invoked by the container.
  - During any Java EE webservice invocation.
  - During a remote or async method invocation of an EJB, during an EJB timeout method, and during message delivery to a message-driven bean.

- The session scope is active during the `service` method of a servlet, during the `doFilter` method of a servlet filter, and when any `ServletRequestListener`, `AsyncListener`, or `HttpSessionListener` is invoked by the container. The session context is shared between all servlet requests that occur in the same HTTP session. The session context is destroyed:
  - When the `HttpSession` times out.
  - After all `HttpSessionListeners` are called.
  - At the very end of any request in which the `invalidate` method was called, after all filters and `ServletRequestListeners` are called.

- The application scope is active whenever either the request or session scope is active. It is also active when a disposer method is invoked (covered in [Contexts](#)), or during the `@PreDestroy` callback of any bean with @RequestScoped, @SessionScoped, or @ConversationScoped scopes. The application context is shared between all servlet requests, web service invocations, EJB remote method invocations, EJB asynchronous method invocations, EJB timeouts, and message deliveries to message-driven beans that run within the same application. The application context is destroyed when the application is shut down.

**Passivating scopes**

Just like stateful session beans, the container can sometimes passivate and activate some types of managed beans. Passivation capable managed beans must be either:

- A stateful session bean, or
- Implement `java.io.Serializable` if they are a CDI-managed bean.

A producer method is passivation capable if and only if all the types that it returns are passivation capable beans or are primitive types. A producer method with a return type that is declared `final` and does not implement `Serializable` is not passivation capable.

An object that is passivated by the container can contain injected dependencies. These dependencies can be saved away or restored correctly by the container so that when the object is
activated, the dependency that is restored by the container is the same instance that was injected before the passivation event. Such dependencies are called passivation capable dependencies. The following objects are passivation capable dependencies:

- All session beans
- All beans with normal scope
- All passivation capable beans with @Dependent scope
- All Java EE resources.

@SessionScoped and @ConversationScoped built-in scopes are passivating scopes. That is, all beans that are declared with these scopes must be passivation capable.

**EJB scopes**

Stateless session beans are treated as if they are @Dependent scoped. If they are declared by using any other scope, the container reports an error at deployment time. Stateful session beans can be declared by using any scope. EJB singleton beans can either be declared with @ApplicationScope or @Dependent. As with regular CDI-managed beans, if a scope is not declared, the EJB defaults to @Dependent.

**The client proxy**

Consider the managed beans shown in Listing 16.

**Listing 16. Normal scoped instances**

```java
@RequestScoped
public class Employee
{
    //....
}

public class Worker
{
    @Inject Employee manager; // manager of worker
    @Inject Employee worker;
}
```

This code does not do what you intended. Because @RequestScoped is a normal scope, the instance of Employee is shared between the `manager` and the `worker` variables. If the @RequestScoped is replaced by @Dependent or if it is removed (resulting in the default, in this case, of @Dependent), then the two instances for `manager` and `worker` are different.

This behavior occurs because when a managed bean of normal scope is instantiated by the container, the injection point is set to a reference to a proxy object (the client proxy). The client proxy holds a reference to the actual managed bean that was created. Whether the `manager` and the `worker` variables reference the same client proxy or different ones is defined by the implementation, according to the CDI specification. However, the specification does state that, in both cases, the same managed bean should be referenced by the client proxy or proxies. A client proxy is actually necessary to solve the problem of how to handle situations where, for example, a request-scoped bean is injected into a session-scoped bean.
Figure 3. Active scoped instances during two concurrent requests

In Figure 3, you have two requests that share a session and are processed at the same time. A session-scoped managed bean (S1) holds references to an injected request-scoped managed bean (R1 or R2). While the first request is being processed, the reference to the request-scoped object must resolve to R1. When the second request is being processed, it must resolve to R2. This is impossible, unless there is a level of indirection — where the reference from S1 goes to a container-managed object that can switch out the reference to the correct request-scoped object as needed.

Figure 4. The role of the client proxy

Technically, the references to the contextual instance of the bean are resolved at run time by the container from the context. This behavior is described by the specification: "Every time a method of the bean is invoked on a client proxy, the client proxy must obtain a contextual instance of the bean."

The other benefit to having a client proxy is that it enables the container to deal with circular dependencies. Consider the beans shown in Listing 17.
Listing 17. Mutually dependent managed beans

```java
public class Person
{
    @Inject private House home;
}
---
public class House
{
    private Person owner;
    @Inject public House(Person owner)
    {
        this.setOwner(owner);
    }
    //...
}
```

In WebSphere Application Server V8 or V8.5, injecting @Person or @House into a bean leads to a StackOverflowException that is caused by the circular dependency. Note that the scope of these two beans is not specified and therefore the scope defaults to @Dependent. If you change Person to be @RequestScoped, the injection works correctly because of the presence of a client proxy. Indeed the specification requires, in any chain of circular dependencies, that if at least one of the dependencies is a normal scoped bean, then a compliant container must support injection of those beans.

**Contexts**

Take a look at Figure 4 again. The client proxy reference to the R1 and R2 objects are happening at the same time, but in different threads. For this to be possible, the container must maintain contextual information about what is happening in each thread. In fact, for each kind of scope, the container maintains contexts for all injected objects so that when a thread of execution follows a reference to an injected object, the container hands it the right one.

For normal scopes, contexts propagate over all local synchronous method invocations. Conversely, contexts do not propagate over remote method invocations, or asynchronous events (like EJB timer service timeouts).

Be aware that in WebSphere Application Server, the context does propagate across remote EJB invocations if the invoked session bean is present locally. Propagation happens because in WebSphere Application Server, the container optimizes things so that remote method invocations in the same JVM happen on the same thread. In such cases, the contextual reference is already active and doesn't need to be created.

**Producers and disposers**

Another mechanism for producing injectable objects is to use methods as a source for injectable objects. By using methods, you can inject objects whose classes don't conform to the rules of CDI-managed beans. However, in such cases, these classes cannot themselves be targets of injection.
And because the container does not manage the lifecycle of such objects, methods annotated with @PostConstruct or @PreDestroy are not invoked by the container.

Use producer methods if:

- You need injection on objects that are not managed bean instances.
- The concrete type of the objects to be injected varies at run time.
- The objects require some custom initialization that is not done by the bean constructor.

Listing 18 shows an example, adapted from the CDI specification, of a producer method that returns different bean instances depending on the method logic.

**Listing 18. A producer that returns a programmatically-determined type**

```java
@SessionScoped
class PaymentStrategyProducer implements Serializable {
    private PaymentStrategyType paymentStrategyType;
    public void setPaymentStrategyType(PaymentStrategyType type) {
        paymentStrategyType = type;
    }
    @Produces
    @SessionScoped /*
    * This must be a scope that allows one to get to the same
    * contextual reference of the bean (PaymentStrategyProducer)
    * that the setPaymentStrategyType method was invoked on. For
    * example, if the bean was @SessionScoped, then the method
    * could be @RequestScoped
    */
    PaymentStrategy getPaymentStrategy(
        @CreditCard PaymentStrategy creditCard,
        @Cheque PaymentStrategy cheque,
        @Online PaymentStrategy online) {
        switch (paymentStrategyType) {
            case CREDIT_CARD: return creditCard;
            case CHEQUE: return cheque;
            case ONLINE: return online;
            default: throw new IllegalStateException();
        }
    }
}
```

A producer method can be static or non-static. If the containing bean is a session bean, then the non-static producer methods must be business methods of the EJB. When the container determines that a producer method needs to be invoked, the sequence of events depends on whether the method is static or not. For static methods, the method is directly invoked. For non-static methods, the container must first instantiate the managed bean if a contextual instance of the bean does not exist.

If a bean with producer methods has a passivating scope, then all the parameters that are injected into the producer methods must be passivation capable dependencies. In the example in Listing 18, the classes that implement PaymentStrategy for the qualifiers of @CreditCard, @Cheque, and @Online, need to be serializable. Needless to say, if the producer method uses a scope that is a passivating scope, then the return type must also be passivation capable. In the example above, the PaymentStrategy interface needs to extend Serializable.
A disposer method serves the opposite purpose: to do a programmer-controlled cleanup of resources and beans that are injected by a producer method. Just like producer methods, the methods can be static or non-static. For a session bean, the disposer method must be a business method of the EJB.

The container matches disposer methods to producer methods defined in the same class. The container compares the bean types that are returned by the producer methods to the bean types that are declared as being disposed by the disposer method. A disposer method might get matched to multiple producer methods. Disposer methods are automatically invoked by the container when it starts the process of destroying objects created by producer methods (Listing 19).

**Listing 19. A Producer with a paired Disposer method**

```java
public class ConnectionManager
{
    @Resource(name="jdbc/supplierDB", authenticationType=AuthenticationType.CONTAINER,
              type=DataSource.class, shareable=true) DataSource suppDB;

    @Produces @Supplier @RequestScoped
    public Connection getSupplierConnection()
    {
        System.out.println("Producer method called");
        try{
            return suppDB.getConnection();
        }catch(SQLException e){
            throw new RuntimeException(e);
        }
    }

    public void cleanupSupplierConnections(@Disposes @Supplier Connection c)
    {
        System.out.println("Disposer method called");
        try{
            c.close();
        }catch(SQLException e){
            throw new RuntimeException(e);
        }
    }
}
```

A producer method can also be annotated with @Named so that it can be referenced in an EL expression. The default EL name of the returned bean is the producer method name.

**Conclusion**

This article covered some basic concepts and constructs made available to WebSphere Application Server V8 and V8.5 developers through the CDI specification.

**Acknowledgements**

The author thanks Eric Covener, Joseph Bergmark, and Paul Glezen (all of IBM) who reviewed this article and provided insightful feedback.
Related topics

- Read more about CDI in this developerWorks article by Kenneth Stephen: Build better applications with Contexts and Dependency Injection (CDI) in WebSphere Application Server.
- Find out What's new in WebSphere Application Server V8.5.
- Watch this video on the new deployment descriptor editor for CDI: IBM Rational Application Developer v9.0: Context and Dependency Injection (CDI) Tooling Updates.

© Copyright IBM Corporation 2013, 2016
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)