# TheServerSide.com

## Dependency Injection in Java EE 6 (Part 6)

This series of articles introduces Contexts and Dependency Injection for Java EE (CDI), a key part of the Java EE 6 platform. Standardized via JSR 299, CDI is the de-facto API for comprehensive next-generation type-safe dependency injection as well as robust context management for Java EE. Led by Gavin King, JSR 299 aims to synthesize the best-of-breed features from solutions like Seam, Guice and Spring while adding many useful innovations of its own.

Throughout the series, we discussed basic dependency injection, scoping, producers/disposers, component naming, interceptors, decorators, stereotypes, events, conversations and CDI's interaction with JSF in detail. In this last article of the series, we will cover portable extensions, available implementations as well as CDI alignment with Seam, Spring and Guice. We will augment the discussion with a few implementation details of CanDI, Caucho's independent implementation of JSR 299 included in the open source Resin application server.

### CDI Portable Extensions: The New Frontier

An important weakness in Java EE has been the difficulty in integrating third party frameworks at the basic component services level. JCA has long been the only real answer to Java EE pluggability but is limited mostly to providing back-end resource adapters for EIS systems and pluggable messaging providers. Although JNDI, JDBC and JPA have had SPIs (Service Provider Interfaces), EJB never did which made it very difficult to use EJB as a basis for a framework, develop EJB plug-ins or extend EJB container services. As a result, one of the most important value propositions for frameworks like Spring has been the ability to easily extend the framework or integrate third-party solutions.

Enter CDI. While CDI provides a clean, elegant solution to dependency injection and context management, its true power comes from an extensive, well-defined SPI. The CDI SPI allows the creation of portable CDI extensions that can be plugged into any CDI implementation. From the perspective of a majority of developers, what this means is that you can simply drop in CDI plug-in jars into your runtime to integrate third-party APIs into Java EE as well as use extended container services in a completely standard fashion. Figure 1 shows the CDI SPI from a high level. Let's take a closer look at the SPI next.

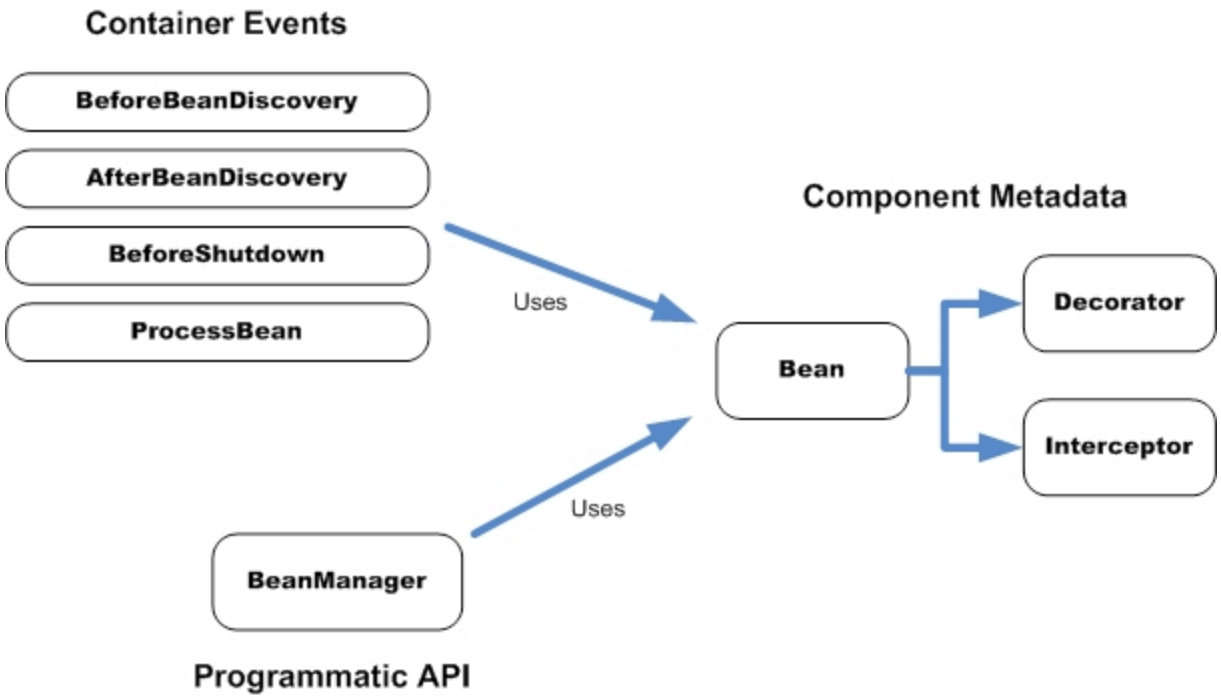


Figure 1: CDI Portable Extension SPI

The SPI allows you to register your own beans, custom scopes, stereotypes, interceptors and decorators with CDI even if is it not included in the automatic scanning process (such as perhaps registering Spring beans as CDI beans), programmatically looking up CDI beans and injecting them into your own objects (such as injecting CDI beans into Spring beans) and adding/overriding annotation-metadata from other sources (such as from a database or property file). The SPI can be segmented into three parts. Interfaces like `Bean`, `Interceptor` and `Decorator` model container meta-data (there are a few other meta-data interfaces such as `ObserverMethod`, `Producer`, `InjectionTarget`, `InjectionPoint`, `AnnotatedType`, `AnnotatedMethod`, etc). Each meta-data object encapsulates everything that the CDI container needs to know about the meta-data type. For example, the Bean interface looks like this:

```
public interface Bean<T> extends Contextual<T> {

  // Java types for the bean

  public Set<Type> getTypes();

  // Qualifiers on the bean

  public Set<Annotation> getQualifiers();
```

```
  // Bean scope

  public Class<? extends Annotation> getScope();

  // EL name, if any

  public String getName();

  // Bean stereotypes, if any

  public Set<Class<? extends Annotation>> getStereotypes();

  // Actual bean implementation class

  public Class<?> getBeanClass();

  // Indicates if a bean is an alternative

  public boolean isAlternative();

  // Indicates if a bean can be null

  public boolean isNullable();

  // Bean injection points

  public Set<InjectionPoint> getInjectionPoints();

}
```

These meta-data interfaces are used by the `BeanManager` API as well as the container events for extensions such as the `BeforeBeanDiscovery` and `AfterBeanDiscory` events.

The `BeanManager API` is a rich façade over the CDI container. It allows you to perform operations such as getting beans, interceptors, decorators and observers by type, qualifiers, injection point or EL name, triggering CDI events, getting the context for a scope and getting EL evaluators as well as utility methods for checking if an annotation is a scope type, qualifier, interceptor binding or stereotype. The bean manager can be injected into a portable extension implementation like this:

```
@Inject

private BeanManager manager;
```

Even if you are not developing a portable extension, you might sometimes need to use the bean manager API (such as for dynamically looking up beans). You can easily do this since the bean manager can be injected anywhere. The following is the entire bean manager interface:

```
public interface BeanManager {

  public Object getReference(Bean<?> bean, Type beanType,

      CreationalContext<?> ctx);

  public Object getInjectableReference(InjectionPoint ij,

      CreationalContext<?> ctx);

  public <T> CreationalContext<T> createCreationalContext(

      Contextual<T> contextual);

  public Set<Bean<?>> getBeans(Type beanType,

      Annotation... qualifiers);

  public Set<Bean<?>> getBeans(String name);

  public Bean<?> getPassivationCapableBean(String id);

  public <X> Bean<? extends X> resolve(Set<Bean<? extends X>> beans);

  public void validate(InjectionPoint injectionPoint);

  public void fireEvent(Object event, Annotation... qualifiers);

  public <T> Set<ObserverMethod<? super T>> resolveObserverMethods(

      T event, Annotation... qualifiers);

  public List<Decorator<?>> resolveDecorators(Set<Type> types,

      Annotation... qualifiers);

  public List<Interceptor<?>> resolveInterceptors(

      InterceptionType type, Annotation... interceptorBindings);

  public boolean isScope(Class<? extends Annotation> annotationType);

  public boolean isNormalScope(
```

```
        Class<? extends Annotation> annotationType);

    public boolean isPassivatingScope(

        Class<? extends Annotation> annotationType);

    public boolean isQualifier(

        Class<? extends Annotation> annotationType);

    public boolean isInterceptorBinding(

        Class<? extends Annotation> annotationType);

    public boolean isStereotype(

        Class<? extends Annotation> annotationType);

    public Set<Annotation> getInterceptorBindingDefinition(

        Class<? extends Annotation> bindingType);

    public Set<Annotation> getStereotypeDefinition(

        Class<? extends Annotation> stereotype);

    public Context getContext(Class<? extends Annotation> scopeType);

    public ELResolver getELResolver();

    public ExpressionFactory wrapExpressionFactory(

        ExpressionFactory expressionFactory);

    public <T> AnnotatedType<T> createAnnotatedType(Class<T> type);

    public <T> InjectionTarget<T> createInjectionTarget(

        AnnotatedType<T> type);

}
```

The events that the container triggers during its life-cycle are the final piece of the SPI. Depending on what the portable extension needs to do, it can listen to one or more events via observer methods. Each event object has specific methods suitable for the container life-cycle stage. The following are the events that the CDI container fires for extensions:

| Event | Description |
|---|---|
| BeforeBeanDiscovery | This event is fired before the container scans for beans. This is an ideal point to add custom qualifiers, scopes, stereotypes, interceptor bindings and annotated types to scan. |
| AfterBeanDiscovery | This event is fired after all beans have been scanned into CDI. This is an ideal point to register additional beans, interceptors, decorators and observer methods. |
| AfterDeploymentValidation | This event is fired after the container is ready to deploy all registered beans. The event is generally only useful for reporting possible deployment problems. |
| BeforeShutdown | This event is fired right before the container shuts down. It is useful for performing any clean-up tasks such as releasing heavy-duty resources. |
| ProcessAnnotatedType | This event is fired after a bean is registered and before the bean metadata is actually processed. This is an ideal event to utilize for creating proxies around the actual bean instance, adding additional bean metadata or stopping the bean from being processed altogether (called vetoing). |
| ProcessInjectionTarget | This event is fired before the CDI container performs injection for a given object. This is a good point to make any changes to the injection target (such as proxying the injection target) or stopping injection from happening. |
| ProcessProducer | This event is fired when the container finds a producer. This is a good point to make any changes to the producer (such as proxying the producer) or stopping the producer from being registered. |
| ProcessBean | Triggered for every bean that is processed by the container. The ProcessBean event can be an instance of ProcessManagedBean, ProcessSessionBean, ProcessProducerMethod or ProcessProducerField. |
| ProcessObserverMethod | This event is fired for every observer method that the CDI container finds. |

Table 1: Scopes in CDI

To put all of this together, let's take a look at what a very simple portable extension might look like. Let's assume that we want to integrate just one object with CDI – that is, make it available for

injection and inject CDI managed beans into it (this is for example how the skeleton for an extension to integrate Spring with CDI might look like). Here is the code:

```
public class IntegrateMyBeanExtension implements Extension {

  public void onBeanDiscovery(

      @Observes AfterBeanDiscovery beanDiscovery,

          BeanManager beanManager) {

    // Create a convenience injection target since we will inject into

    // our custom bean

    InjectionTarget injectionTarget =

        beanManager.createInjectionTarget(

            beanManager.createAnnotatedType(MyBean.class));



    // Register the custom bean

    beanDiscovery.addBean(new Bean<MyBean>() {

      @Override

      public Class<?> getBeanClass() {

        return MyBean.class;

      }



      @Override

      public Set<InjectionPoint> getInjectionPoints() {

        return injectionTarget.getInjectionPoints();

      }
```

```
@Override
public String getName() {
  return "myBean";
}

@Override
public Set<Annotation> getQualifiers() {
  Set<Annotation> qualifiers = new HashSet<Annotation>();
  qualifiers.add(new AnnotationLiteral<Default>(){});
  qualifiers.add(new AnnotationLiteral<Any>(){});
  return qualifiers;
}

@Override
public Class<? extends Annotation> getScope() {
  return ApplicationScoped.class;
}

@Override
public Set<Class<? extends Annotation>> getStereotypes() {
  return Collections.emptySet();
}

@Override
```

```
public Set<Type> getTypes() {
  Set<Type> types = new HashSet<Type>();
  types.add(MyBean.class);
  return types;
}

@Override
public boolean isAlternative() {
  return false;
}

@Override
public boolean isNullable() {
  return false;
}

@Override
public MyBean create(CreationalContext<MyBean> context) {
  // Have CDI instantiate the bean
  MyBean instance = injectionTarget.produce(context);
  // Have CDI perform injection
  injectionTarget.inject(instance, context);
  // Call @PostConstruct
  injectionTarget.postConstruct(instance);
  return instance;
```

```
        }


        @Override
        public void destroy(SecurityManager instance,
            CreationalContext<SecurityManager> context) {
          // Call @PreDestroy
          injectionTarget.preDestroy(instance);
          // Have CDI release the bean
          injectionTarget.dispose(instance);
          // Release any dependent objects
          context.release();
        }
      });
  }
}
```

The first thing you should notice is that all portable extensions must implement the `Extension` interface. Because we are adding beans, it makes the most sense to observe the `AfterBeanDiscovery` event, which is what we are doing in the `onBeanDiscovery method`, which also gets a `BeanManager` instance. The first thing we do is create a convenience `InjectionTarget` instance for our custom bean. This is useful because we want CDI to inject its available beans into our custom object and also as a convenience for implementing the `Bean` interface's `getInjectionPoints` method. We then register our custom bean with CDI using the event's `addBean` method. The rest of the code is pretty unremarkable except for the create and destroy methods that are used to `create` and `destroy` the custom bean instances. In the create method, the `injectionTarget.produce` call is used to have CDI create the actual custom bean instance so that any constructor injections can take place or in case a producer is used. The `injectionTarget.inject` call then performs any other injection on the newly created custom object instance. Conversely, in the destroy method, the `injectionTarget.dispose` call is used so that CDI can call a disposer method if it needs to while the `context.release()` call makes sure any dependent objects are properly cleaned up.

We could of course make the portable extension a lot more sophisticated such as manually looking up beans from the bean manager and injecting them into the custom object ourselves, registering custom stereotypes, scopes, interceptors, configuring the bean meta-data using custom configuration sources and so on. The sky really is the limit in terms of what you can do in a CDI extension.

To learn more about the CDI SPI, check out the official specification or the Weld reference (included in the references section).

**CDI Portable Extensions in Action**

While the CDI portable extension SPI is interesting on its own right, it is unlikely you will be using it directly. As we mentioned, your real exposure to CDI portable extensions will likely be though CDI plug-ins developed by third-party open source or commercial tools. Although CDI is so new, that ecosystem is beginning to evolve already. For example, the ZK framework, an Ajax-based RIA platform already provides a CDI plug-in.

However, it is more likely that a majority of the initial wave of CDI plug-ins will be coming from CDI implementers. For example, JBoss Seam provides a very wide variety of CDI plug-ins for logging, XML configuration, JPA/persistence, JSF, JMS, JAX-RS, JavaScript remoting, security, internationalization, JavaMail, scheduling, document generation, Servlet, Wicket, GWT, Drools, jBPM, JBoss ESB, exception handling, cloud computing and many others. Similarly, the Apache MyFaces CODI project closely related to the OpenWebBeans CDI implementation offers a set of plug-ins for logging, JSF, Bean Validation, JPA/persistence, scripting and testing. We plan to develop a set of plug-ins for JDBC, iBATIS, Struts 2 and Quartz as part of CanDI, the CDI implementation included in Resin. It is obviously not possible to go through such a vast array of choices in the space of a single article (in fact an entire series is probably not even enough). Instead, what we will give you next is a somewhat random sampling of the options to whet your appetite to explore further on your own.

CDI portable extensions can be an important vehicle for Java EE innovation. The Seam 3 persistence module is a great example of this. The module allows you to use EJB transaction management annotations outside the EJB component model in CDI managed beans. The example below shows the `@TransactionAttribute` annotation being used in an `@ApplicationScoped` managed bean:

```
@ApplicationScoped
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class DefaultAccountService implements AccountService {
  @Inject
```

```
  private EntityManager entityManager;


  public void addAccount(Account account) {
    ...
  }
  ...
}
```

This is a model that will likely be standardized in Java EE 7. The Apache MyFaces CODI persistence/JPA module allows for similar capabilities via the `@Transactional` annotation. Note Resin's Java EE 6 Web Profile implementation allows you to use all EJB annotations like `@TransactionAttribute, @Schedule, @Asynchronous, @RolesAllowed, @RunAs, @Lock, @Startup` and `@Remote` outside EJB in CDI managed beans, not just the transaction services.

The sample below comes from the Apache MyFaces CODI core module. This module allows you to inject loggers into CDI beans:

```
@Inject private Logger logger;
```

The core module of Seam 3, named Solder, allows logger injection as well, except that it uses the type-safe, annotation-driven JBoss Logging API instead of standard loggers.

Several Seam 3 modules enhance the injection capabilities of standard APIs like JSF, Servlets, JMS and JAX-RS in important ways. For example, the Seam JMS module allows the injection of connections, sessions, message producers and message receivers in addition to simply queues, topics and connection factories. As you can see from the code below this significantly cuts down the boilerplate code needed for raw JMS:

```
public class TransferSender {
  ...
  @Inject
```

```
  @JmsSession(
      transacted=true,
      acknowledgementType=Session.AUTO_ACKNOWLEDGE)
  private Session session;

  @Inject
  @JmsDestination(jndiName="jms/TransferQueue")
  private MessageProducer producer;
  ...
  public void sendTransfer(Transfer transfer) {
    ...
    producer.send(session.createObjectMessage(transfer));
    ...
  }
  ...
}
```

In a similar fashion, the Seam JSF module allows for injection into Converters and Validators, injection of JSF objects like `FacesContext`, `NavigationHandler` and `Flash`; the Seam REST module enables injection of JAX-RS client proxies; the Seam Servlet module allows the injection of Servlet objects such as `ServletConfig, ServletContext, HttpSession, HttpServletRequest, HttpServletResponse`, HTTP headers, request parameters and cookies. CODI modules have a similar set of capabilities, including supporting injection of the JSF projects stage and Bean Validation objects like `Validator` and `ValidationFactory`.

Both the Seam 3 and CODI modules add a number of custom scopes to CDI including `@ViewScoped, @RenderedScoped, @FlashScoped, @WindowScoped` and grouped conversations. The example below shows the innovative view scope in CODI:

```
@Named

@ViewScoped

public class MyViewScopedBean {

  ...

}
```

The view scope essentially means that a component is alive until it is no longer visible. This means that the component is created when it is first used, stays alive as long as a JSF page references it in the current and subsequent request and is destroyed when the user reaches a page that no longer references the component. In a similar vein, CanDI adds the `@TransactionScoped` and `@ThreadScoped` custom scopes geared towards back-end resources instead of JSF.

One of the very useful things that CODI includes is integration with JSR-223 scripting engines. The example below shows the integration of a JavaScript engine with CDI:

```
@Inject @ScriptLanguage(JavaScript.class)

private ScriptBuilder scriptBuilder;

...

return scriptBuilder.script("x + y").namedArgument(

    "x", a).namedArgument("y", b).eval(Double.class);
```

In a similar vein, the Seam JSF module allows the injection of EL expression evaluators.

A very interesting idea that is evolving fast is using the CDI event/observer model for life-cycle listeners, messaging and even exception handling. For example, the Seam Servlet module allows you to observe Servlet container life-cycle events via CDI like this:

```
public void onServletContextInitialization(

    @Observes @Initialized ServletContext context) {
```

```
  ...
}

public void onSessionInitialization(
    @Observes @Initialized HttpSession session) {
  ...
}

public void onRequestInitialization(
    @Observes @Initialized @Path("/bid")
        HttpServletRequest request) {
  ...
}
```

You can similarly observer JSF life-cycle events via Seam or CODI. The Seam Catch module takes this a step further by allowing you to observe unhandled exceptions. In Resin 4, we are implementing a mechanism to process JMS messages via the CDI event bus as an alternative to the EJB Message Driven Bean model.

The few examples above are just the very small tip of a very large iceberg. Although not truly CDI portable extensions, projects like Arquillian for Java EE testing and Forge for Java EE RAD (Rapid Application Development) are also worth a close look (both are included in the references section).

**CDI Implementations**

There are three major CDI implementations at the moment – JBoss Weld, Apache OpenWebBeans and Resin's CanDI.

JBoss Weld is the CDI reference implementation. It is included in both GlassFish and JBoss AS. Besides running on GlassFish and JBoss you can also embed Weld into Tomcat, Jetty or even standalone Java SE (via a pretty simple and elegant bootstrap API).

OpenWebBeans is the Apache licensed implementation of CDI included in Geronimo. It's closely related to the Apache OpenEJB and Apache MyFaces CODI projects. Besides being included in Geronimo, OpenWebBeans and OpenEJB can also be integrated with Tomcat or be run in Java SE.

CanDI is the independent Caucho CDI implementation included in the Resin 4 Java EE 6 Web Profile application server. CanDI forms the core of Resin itself. In a sense, Resin 4 is probably the most significant CDI based application to-date. Both the Resin Servlet 3 and EJB 3.1 Lite containers are based on CanDI. In fact, CanDI XML is used to configure Resin via its resin.xml, resin-web.xml, beans.xml and resin-beans.xml files. Figure 2 shows the Resin CDI based architecture. Note, while Resin implements CDI, Servlet 3, EJB 3.1 Lite, JTA and JMS, it integrates the reference implementations for JSF 2, JPA 2 and Bean Validation.

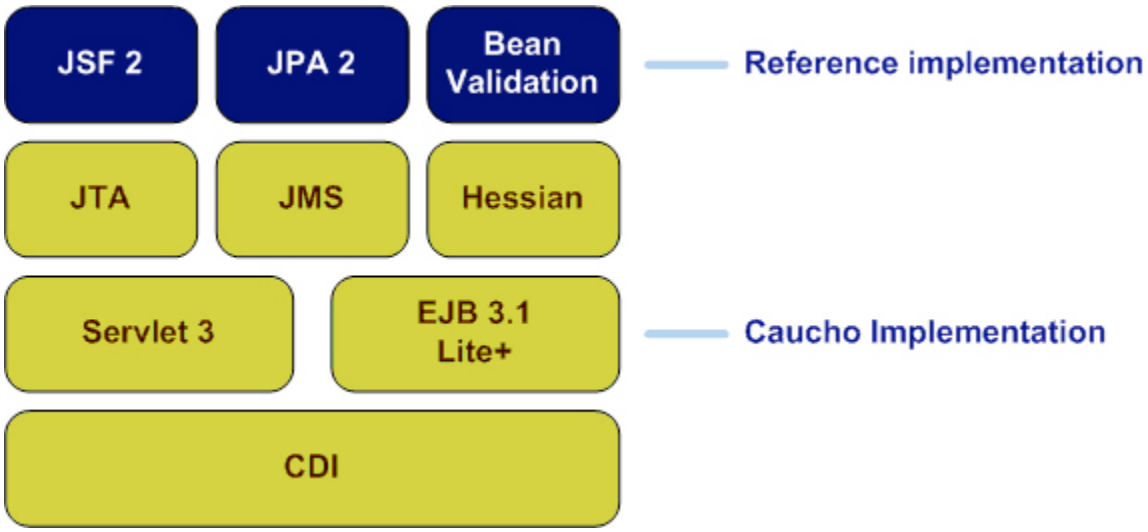


Figure 2: CanDI is the core of Resin 4

Besides these "big three" implementations, other Java EE 6 implementations like WebLogic, WebSphere and JOnAS may also implement their own CDI containers.

**Relationship with Seam, Spring and Guice**

CDI, Weld and Seam 3 essentially evolved from Seam 2. Seam 2 core had the basic ideas in CDI in terms of dependency injection and context management. Those ideas were evolved through the JCP by incorporating the best-of-breed ideas from Seam 2, Guice and Spring with additional unique innovations added on the way. As we mentioned, Weld is reference implementation of CDI from JBoss and essentially supersedes Seam 2 core. As a result, the current Seam project does not include any core dependency injection features. Instead, Seam 3 is basically a set of portable CDI extensions that either enhance Java EE APIs in some way or integrate third-party tools. In fact, each Seam 3

module is separately led, developed and documented, although there are still “umbrella” Seam releases. Figure 3 shows the relationship between CDI, Weld and Seam 3.

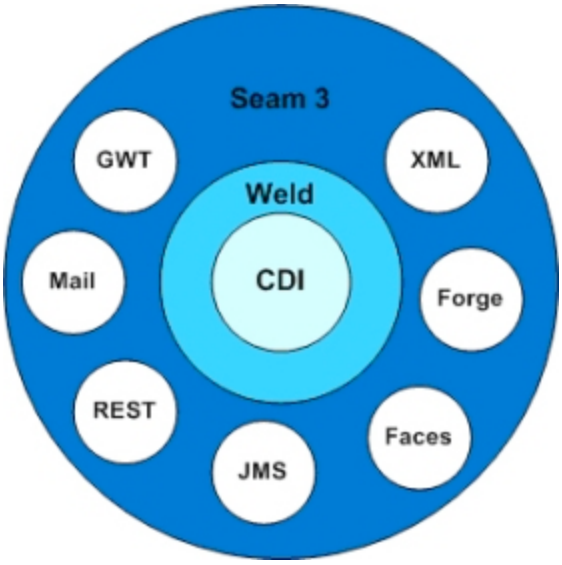


Figure 3: Relationship between CDI, Weld and Seam 3

Guice and Spring do not implement CDI. However both Guice and Spring support the JSR 330 Dependency Injection Annotations for Java, as do all CDI implementations. In a sense, JSR 330 forms the baseline between CDI, Spring and Guice. In addition, Spring also supports some of the JSR 250 (Common Annotations for the Java) annotations (as well as some EJB annotations like @TransactionAttribute). Table 2 shows the major JSR 330 and JSR 250 annotations. You’ll notice that we have actually discussed most of the annotations throughout the series.

| Annotation | JSR | Description |
|---|---|---|
| @PostConstruct | 250 | Call-back after a component is constructed. |
| @PreDestroy | 250 | Call-back before a component is destroyed. |
| @Resource | 250 | Injection of container resources into the component. |

| | | |
|---|---|---|
| @RunAs | 250 | Defines the runtime security role to be used for component execution. |
| @RolesAllowed | 250 | Specifies security roles permitted to access a component. |
| @PermitAll | 250 | Specifies that all security roles are permitted to access a component. |
| @DenyAll | 250 | Specifies that no security roles are allowed access a component. |
| @DataSourceDefinition | 250 | Defines a data source. |
| @ManagedBean | 250 | Defines a component. |
| @Inject | 330 | Injects a component. |
| @Named | 330 | Names a component |
| @Qualifier | 330 | Defines an injection qualifier. |
| @Scope | 330 | Defines a scope. |

Table 2: JSR 330 and JSR 250 Annotations

Guice users should be immediately comfortable with CDI because most of the core philosophies in CDI – loose-coupling, strong-typing and annotations over XML come from Guice. Unfortunately there are no currently slated plans for Guice to implement CDI and no interoperability effort exists. However, it would be a relatively simple task to write a Guice CDI plug-in if there is interest.

While Spring has no current plans to support CDI, Seam 3 does include a Spring integration module. The goals of the Spring integration module are injection support from Spring to CDI (and vice-versa), native qualifier handling, scope/context interoperability, the ability to work with Spring transactions and the ability to share persistence contexts between the containers. We are also planning to add Spring integration support in CanDI for the Resin community.

**CDI 1.1 Coming Soon!**

The work for CDI 1.1 has just begun as part of Java EE 7. You should expect a JSR request to be submitted and an expert group to be formed very soon. As the release number indicates, this is

mostly expected to be a minor release with a few enhancements and bug fixes. Below are some of the possibilities being considered:

Global ordering of interceptors and decorators, as well as global enablement of alternatives.

- An API for managing built-in contexts, allowing the built-in implementation of the conversation context to be used outside of JSF.
- An embedded mode allowing startup outside of a Java EE container.
- Bean declaration at constructor level.
- Static injection.
- Inclusion of @Unwraps from Seam Solder.
- Enhancements to the Portable Extensions SPI.
- Client controlled contexts allowing for SaaS style multi-tenancy.
- Better support for CDI in libraries when used in the Java EE platform.
- Send CDI events for Servlet events.
- Application lifecycle events.

For more details on these, see http://bit.ly/cdi11features and http://bit.ly/cdi10bugs. Besides these changes, the real CDI related work in Java EE 7 should be better alignment with managed beans, EJB, JPA, JSF, EL, Servlet, JMS, JAX-RS and JAX-WS. Do you have other changes in mind? If so, now is really the time to get involved. You can send an email to jsr-299-comments@jcp.org. Also, feel free to email us at reza@caucho.com or ferg@caucho.com.

**Many Thanks!**

Though arduous and time-consuming, writing this article series has been a pleasure and a privilege for us. We hope that it was helpful to you in some way or encouraged you to know more about the vibrant CDI ecosystem. CDI is key to the continuing evolution of Java EE and we are always trying our best to make it a truly outstanding Java server-side technology (hopefully with some of your help).

**References**

1. JSR 299: Contexts and Dependency Injection for Java EE, http://jcp.org/en/jsr/detail?id=299.

2. JSR 299 Specification Final Release, https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_JCP-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=web_beans-1.0-fr-oth-JSpec@CDS-CDS_JCP.

3. CDI Code Examples, http://sites.google.com/site/cdipojo/.

4. Weld, the JBoss reference implementation for JSR 299: http://seamframework.org/Weld.

5. Weld Reference Guide, http://docs.jboss.org/weld/reference/1.0.0/en-US/html/.

6. CanDI, the JSR 299 implementation for Caucho Resin, http://caucho.com/projects/candi/.

7. OpenWebBeans, Apache implementation of JSR 299, http://openwebbeans.apache.org.

8. Seam, http://seamframework.org.

9. Apache MyFaces CODI, http://myfaces.apache.org/extensions/cdi/.

10. ZK Framework, http://books.zkoss.org/wiki/Small_Talks/2010/March/Getting_started_with_ZK_CDI.

11. Arquillian, http://www.jboss.org/arquillian.

12. Seam Forge, http://seamframework.org/Documentation/SeamForge.

**About the Authors**

Reza Rahman is a Resin team member focusing on its EJB 3.1 Lite container. Reza is the author of *EJB 3 in Action* from Manning Publishing and is an independent member of the Java EE 6 and EJB 3.1 expert groups. He is a frequent speaker at seminars, conferences and Java user groups, including JavaOne and TSSJS.

Scott Ferguson is the chief architect of Resin and President of Caucho Technology. Scott is a member of the JSR 299 EG. Besides creating Resin and Hessian, his work includes leading JavaSoft's WebTop server as well as creating Java servers for NFS, DHCP and DNS. He lead performance for Sun Web Server 1.0, the fastest web server on Solaris.

*15 Mar 2011*