Dependency Injection in Java EE 6 - Part 1 by Reza Rahman

November 2009

Discuss this Article

This is the first part of a multi-article series. For part 2 please see: Dependency Injection in Java EE 6 - Part 2

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

In this first article of the series, we are going to take a high-level look at CDI, see how it fits with Java EE overall and discuss basic dependency management as well as scoping. In the course of this series, we will cover features like component naming, stereotypes, producers, disposers, decorators, interceptors, events, the CDI API for portable extensions and many more. We will also talk about how CDI aligns with Seam, Spring as well as Guice and augment the discussion with some implementation details using CanDI, Caucho's independent implementation of JSR 299 included in the Resin application server.

A Quick Glance Back

The primary focus of Java EE 5 was ease-of-use via POJO programming, annotations and convention-over-configuration. Java EE 5 did have a basic form of dependency injection perhaps most appropriately termed *resource injection*. Specifically, you could inject container resources such as JMS connection factories, data sources, queues, JPA entity managers, entity manager factories and EJBs via the @Resource, @PersistenceContext, @PersistenceUnit and @EJB annotations into Servlets, JSF backing beans and other EJBs. This model was adequate for applications comprising of JPA domain objects, services and DAOs written as EJBs and JSF.

However, you had to resort to more general purpose dependency injection technologies like Seam, Spring or Guice for a number of use cases. For example, you could not inject EJBs into Struts Actions or JUnit tests and you could not inject DAOs or helper classes that were not written as EJBs because they do not necessarily need to be transactional. More broadly, it was difficult to integrate third-party/in-house APIs or use Java EE 5 as a basis to build such APIs that are not just strictly business components. These are exactly the class of problems that CDI is designed to solve in a highly type-safe, consistent and portable way that fits the Java philosophy well. In fact, a lot of the Resin container itself is written using JSR 299, something that would have been unimaginable with Java EE 5!

If you are familiar with Spring IoC, CDI is likely to feel more type-safe, futuristic and annotation-driven. Seam developers will find that CDI has a lot more advanced features. CDI is perhaps more geared towards enterprise development than Guice is. Finally, Java EE 5 developers will likely feel that CDI is much more complex at first glance but will see that most of the complexity is well justified and can be ignored when not needed.

Besides the strictly dependency injection centric features, CDI enhances the Java EE programming model in two more important ways - both of which come from Seam. First, it allows you to use EJBs directly as JSF backing beans. We won't cover that this time-around, but in a coming article in the series we will see how this can remove some common glue-code from the presentation tier and make Java EE APIs far more cohesive. Second, CDI allows you to manage the scope, state, life-cycle and context for objects in a much more declarative fashion, rather than the programmatic way most web-oriented frameworks handle managing objects in the request, session and application scopes. We will see this in action in a coming article in the series showing CDI's interaction with JSF.

What's in a Name?

These last two aspects of JSR 299 was its original scope, which is why it was called "WebBeans" initially. As it turns out, JSR 299 in its current form encompassed far beyond this initial scope covering integration and dependency injection concerns in general. This gradual change in scope is reflected in its current, more fitting name, Contexts and Dependency Injection for Java EE. This "scope creep" in CDI has been subject to some controversy but is probably not that

big of a deal in the scheme of things...

How the Pieces Fit Together

It is important to see where CDI fits with Java EE overall to properly grasp what it does and how it does it. CDI has no component model of its own but is really a set of services that are consumed by Java EE components such as *managed beans*, Servlets and EJBs.

Managed beans are a key concept introduced in Java EE 6 to solve some of the limitations we talked about in the previous section with Java EE 5 style resource injection. A managed bean is just a bare Java object in a Java EE environment. Other than Java object semantics, it has a well-defined create/destroy life-cycle that you can get callbacks for via the @PostConstruct and @PreDestroy annotations. Managed beans can be explicitly denoted via the @ManagedBean annotation, but this is not always needed, especially with CDI. From a CDI perspective, this means that almost any Java object can be treated as managed beans and so can be full participants in dependency injection.

In fact, managed beans are intended to become the fundamental building block for all Java EE components. Traditional JSF backing beans are now managed beans (likely annotated with @ManagedBean). All EJB session beans are now also redefined to be managed beans with additional services (thread-safety and transactions by default). Servlets are not yet redefined to be managed beans, but this is very likely to happen in the near future. In essence, all other Java EE APIs will add various services to the foundational managed bean concept, primarily via declarative annotations.

Is the EJB Component Model Needed?

It is certainly an interesting question to ask whether EJB needs it's own component model in the form of the @Stateless, @Stateful, @Singleton and @MessageDriven annotations either. After all, can it also not be positioned simply as business component services on top of managed beans? Resin appreciates this perspective by allowing you to use EJB declarative service annotations such as @TransactionAttribute, @Remote, @DeclareRoles, @RolesAllowed, @Asynchronous, @Schedule and @Lock on managed beans that are not EJBs (you can of course use EJB in a standards defined fashion). We believe this is one direction the EJB specification could easily move towards in the future...

Other than providing dependency injection services to managed beans of all flavors, as we mentioned previously CDI also integrates with JSF via EL bean name resolution from view technologies like Facelets and JSP as well as automatic scope management. CDI's integration with JPA consists of honoring the @PersistenceContext and @PersistenceUnit injection annotations, in addition to @EJB and @Resource. Note CDI does not directly support business component services such as transactions, security, remoting, messaging and the like that are in the scope of the EJB specification.

JSR 299 utilizes the Dependency Injection for Java (JSR 330) specification as its foundational API, primarily by using JSR 330 annotations such as @Inject, @Qualifier and @ScopeType. Led by Rod Johnson and Bob Lee, JSR 330 defines a minimalistic API for dependency injection solutions and is primarily geared towards non-Java EE environments. In particular, it does not define scope types common in server-side Java such as request, session and conversation. It also does not define specific integration semantics with Java EE APIs like JPA, EJB and JSF. CDI essentially adapts JSR 330 for Java EE environments while also adding a number of additional features useful for enterprise applications. Figure 1 shows how CDI fits with the major APIs in the Java EE platform.

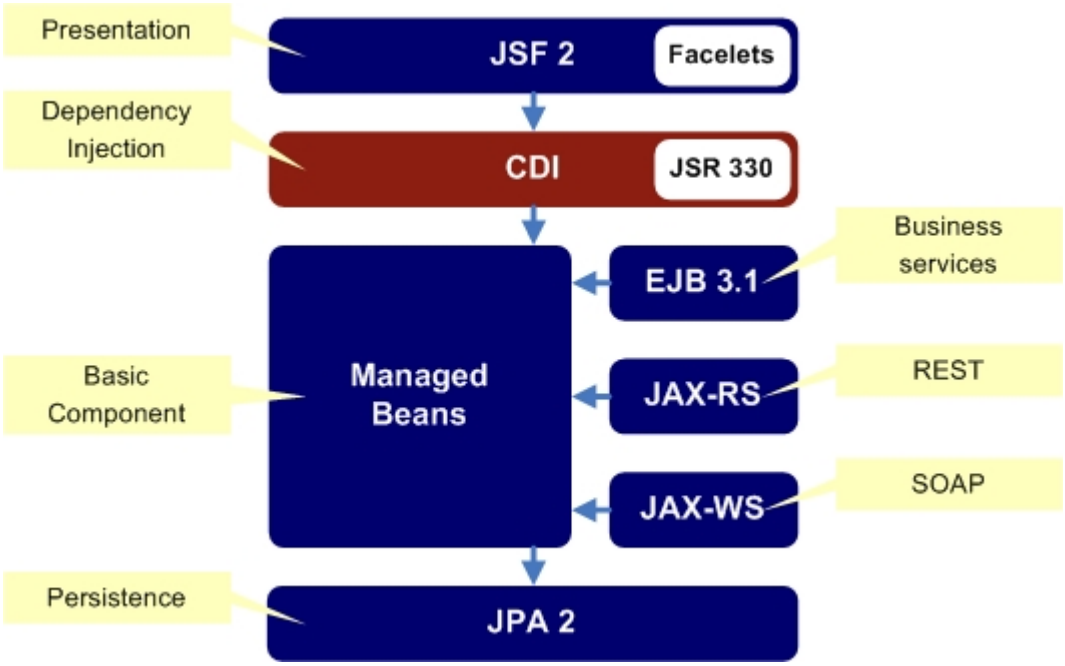

Figure 1: CDI and Java EE

Dependency Injection Basics

The very basic dependency injection concepts in CDI are rather simple but powerful. For most people doing enterprise Java development for a few years, it should be familiar, just with more of a bend for Java centric type-safety and annotations for metadata. The following example shows CDI injection in its most basic form (the example is derived from ActionBazaar of *EJB 3 in Action*):

```
@Stateless
public class BidService {
    @Inject
    private BidDao bidDao;

    public void addBid (Bid bid) {
        bidDao.addBid(bid);
    }
}

public class DefaultBidDao implements BidDao {
    @PersistenceContext
    private EntityManager entityManager;

    public void addBid (Bid bid) {
        entityManager.persist(bid);
    }
}

public interface BidDao {
    public void addBid (Bid bid);
}
```

In the example above, the bid DAO managed bean is being injected into the bid service EJB session bean via the @Inject annotation. CDI resolves the dependency by looking for any class that implement the BidDao interface. When CDI finds the DefaultBidDao implementation, it instantiates it, resolves any dependencies it has (like the JPA entity manager injected via @PersistenceContext) and injects it into the EJB bid service. Since no explicit bean scope is specified for either the service or the DAO, they are assumed to be in the implicit *dependent* scope. We'll discuss the dependent scope shortly, but it basically means the injected object belongs to the object instance it is being injected into. Note that none of this uses string names that can be mistyped and all the code is in Java and so is checked at compile time, probably through an IDE. The power of this will become even more apparent when we look at *qualifiers*.

Qualifiers are additional pieces of meta-data that narrow down a particular class when more than one candidate for injection exists. Let's assume that there are two flavors of DAOs for ActionBazaar -- the default one that uses JPA and a legacy one that uses JDBC. For the bid service, let's assume we need to use the legacy JDBC DAO instead of the one that used JPA. This is how you would do it

using qualifiers:

```
@Stateless
public class BidService {
    @Inject @JdbcDao
    private BidDao bidDao;
    ...
}

@JdbcDao
public class LegacyBidDao implements BidDao {
    @Resource(name="jdbc/ActionBazaarDB")
    private DataSource dataSource;
    ...
}

@Qualifier
@Retention(RUNTIME)
@Target({TYPE})
public @interface JdbcDao {}

public class DefaultBidDao implements BidDao {
    @PersistenceContext
    private EntityManager entityManager;
    ...
}
```

In this case, there are two classes, LegacyBidDao and DefaultBidDao that are candidates for injection into the bid service EJB. CDI determines that it is the JBDC based legacy DAO that should be injected by looking at the JdbcDao qualifier placed on the bidDao variable. Note that the qualifier itself is a custom annotation with the @Qualifier marker on it and so is type-safe, as opposed to a string value that is used in case of the older @Resource annotation attempting to resolve the data source dependency via the value "jdbc/ActionBazaarDB".

These are the foundational concepts on which the more advanced features of CDI are built. We'll look at the more advanced dependency injection features in CDI in later articles in the series like stereotypes, producers, disposers and events.

Context Management Basics

Every object managed by CDI has a well-defined scope and life-cycle that is bound to a specific context. As CDI encounters a request to inject/access an object, it looks to retrieve it from the context matching the scope declared for the object. If the object is not already in the context, CDI will get reference to it and put it into the context as it passes the reference to the target. When the scope corresponding to the context expires, all objects in the context are removed. Table 1 describes the contexts defined by CDI:

Scope	Description
Dependent	A dependent reference is created each time it is injected and the reference is removed when the injection target is removed. This is the default scope for CDI and makes sense for the majority of cases.
ApplicationScoped	An object reference is created only *once* for the duration of the application and the object is discarded when the application is shut down. This scope makes sense for service objects, helper APIs or objects that store data shared by the entire application.
RequestScoped	An object reference is created for the duration of the HTTP request and is discarded when the request ends. This makes sense for things like data transfer objects/models and JSF backing beans that are only needed for the duration of an HTTP request.
SessionScoped	An object reference is created for the duration of an HTTP session and is discarded when the session ends. This scope makes sense for objects that are needed throughout the session such as maybe user login credentials.
ConversationScoped	A conversation is a new concept introduced by CDI. It will be familiar to Seam users, but new to most others. A conversation is essentially a truncated session with start and end points determined by the application, as opposed to the session which is controlled by the browser, server and session timeout. There are two

	types of conversations in CDI -- *transient* which basically corresponds to a JSF request cycle and *long-running* which is controlled by you via the Conversion.begin and Conversion.end calls. A transient conversation can be turned into a long-running one as needed. Conversations are very useful for objects that are used across multiple pages as part of a multi-step workflow. An order or shopping cart is a good example. Conversations deserve detailed discussion so we will take a much closer look at them in a later article in the series.

Table 1: Scopes in CDI

Besides the built-in scopes above, it is also possible to create custom scopes via the @Scope annotation. This is mostly useful for extending CDI in a standard way. For example, we are considering adding @TransactionScoped for Resin's CanDI implementation for back-end components that should only live in the context of a transaction such as a set of abstraction APIs to send messages to JMS queues or to perform basic JDBC operations. We'll leave this as a thought experiment for you that will probably make a lot more sense when we cover producer and disposer methods in a later article. Similarly, custom scopes for workflows, BPM etc might make sense too.

Let's take a quick look at a code example to help crystallize some of this. We'll take the bid service example from before, enhance it a bit, add appropriate scope types and see how it might be used in the view layer using CDI:

```
@Named
@RequestScoped
public class BidManager {
  @Inject
  private BidService bidService;
  ...
}

@Stateless
public class BidService {
    @Inject
    private BidDao bidDao;

    @Inject
    private BiddingRules biddingRules;
    ...
}

public class DefaultBidDao implements BidDao {
    @PersistenceContext
    private EntityManager entityManager;
    ...
}

@ApplicationScoped
public class DefaultBiddingRules implements BiddingRules {
    ...
}
```

In the example above, the bid manager is a backing bean/controller that is useful for handling incoming requests from a JSP or Facelet, so HTTP request is the most appropriate context for it. The bid service EJB is in the dependent scope as before. This is the only scope that is allowed and makes sense for a stateless session bean. The DAO instance remains in the dependent scope too and so is bound exclusively to a bid service EJB instance that is likely pooled. This makes sense because a DAO managed bean instance really should not be shared since it has a reference to a non-thread-safe entity manager. It is fine as long as it is used in the scope of the EJB bid service instance since the EJB is guaranteed to be thread-safe and transactional (this is not a big issue in Resin since all injected entity managers are wrapped in a thread-safe proxy so that you don't need the thread-safety of an EJB and can use a transactional, application-scoped managed bean for the DAO if you want). The bidding rules strategy object injected into the bid service is in the application scope, however. This is because it simply encapsulates some shared business rules that are read-only, does not modify any data and so can be shared safely across the application. CDI will only create one instance of the bidding rules for the entire application and inject it wherever it is needed. Most helper APIs like this one and EJB singleton session beans are great candidates for the application scope.

These context management features of CDI work with JSF in powerful ways to remove a lot of the boilerplate programmatic work you need to do in order to maintain state in a typical web framework. This deserves detailed discussion on its own so we will show you some of these capabilities in a later article in the series. For now, note that the @Named annotation makes the bid manager accessible from EL and is also best discussed in the context of JSF integration.

Continued in Dependency Injection in Java EE 6 - Part 2

Much More to Come

The CDI features we talked about in this first article really is the tip of a pretty large iceberg. As the series progresses, we will show you how to group metadata in stereotypes, create object factories via producer/disposer methods, lightweight event management via dependency injection, using the new conversation scope to manage application state, EL name resolution, CDI's interaction with JSF, interceptors, decorators and much more.

It is a little late in the game to make big changes at this point, but you are still welcome to send your comments on JSR 299 by sending an email to jsr-299-comments@jcp.org. You can also send general comments on Java EE 6 to jsr-316-comments@jcp.org. For comments on Resin or CanDI, our JSR 299 implementation, feel free to email us at reza@caucho.com or ferg@caucho.com. Cheers until next time!

Part 2: Dependency Injection in Java EE 6 - Part 2

References

1. JSR 299: Contexts and Dependency Injection for Java EE, http://jcp.org/en/jsr/detail?id=299.
2. Weld, the JBoss reference implementation for JSR 299: http://seamframework.org/Weld.
3. CanDI, the JSR 299 implementation for Caucho Resin, http://caucho.com/projects/candi/.
4. OpenWebBeans, Apache implementation of JSR 299, http://incubator.apache.org/openwebbeans/.

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.

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.

PRINTER FRIENDLY VERSION

TheServerSide.COM
Your Enterprise Java Community