BIO



Part 4 Contexts and Dependency Injection: The New Java EE Toolbox

ANTONIO GONCALVES Integration with Java EE

This series has attempted to demystify Contexts and Dependency Injection (CDI). In the previous articles, which appeared in the last three issues, I discussed what strong typing really means in dependency injection, how to use CDI to integrate third-party frameworks, and how to create loose coupling with interceptors, decorators, and events. This final article covers the integration of CDI with Java EE.

Java EE is an extension of the Java runtime. It provides a managed environment in which containers provision components with a certain number of services. These services can be lifecycle management, security, validation, persistence, or, of course, injection. Persistence and transactions are often bundled together to develop the back end of an application.

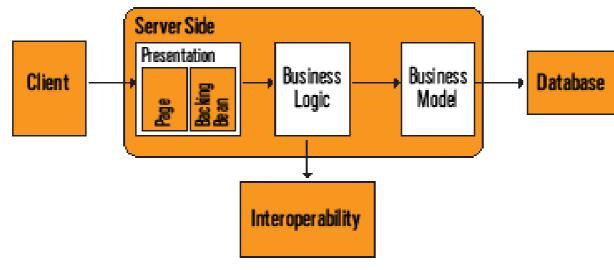


Figure 1. Standard tiers of an application

On the web tier, Java EE comes with servlets, WebSockets [See accompanying article. -Ed.], and JavaServer Faces (JSF), which are related to the user interface. CDI, whose workings I've explained in the last three articles, can bring the web tier and service tier together to create a homogeneous and integrated application.

Bringing the Web Tier and Service Tier Together

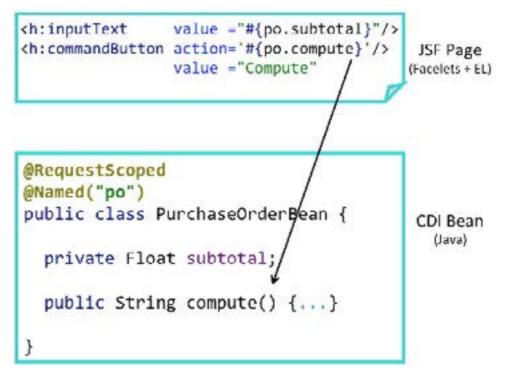
Java EE bundles several technologies that enable us to create any kind of architecture, including web application, REST interfaces, batch processing, asynchronous messaging, persistence, and so on. As shown in Figure 1, all these applications can be organized in several tiers: presentation, busi-

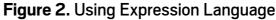
> ness logic, business model, or interoperating with external services. Depending on our needs, any kind of architecture is possible from stateless to stateful, from flat layered to multitiered. One problem, however, is that the web tier and service tier each has its own paradigm, its own language. Because of this, CDI is an important resource to bring them together.

Java for the service tier. Except for the web client (which uses HTML) and the database (which uses Database Definition Language), most of Java EE uses Java as its primary language, and, therefore, we find Java in most of the application tiers:

Java Persistence API entities in the business model or a simple bean on the business logic tier. We even use Java as part of our presentation tier: JSF backing beans are written in Java. EL for the presentation tier. When I say that Java is the primary language, that's because JSF pages are written using Facelets and Expression Language, or EL. EL provides an important mechanism for enabling the presentation layer to communicate with the application logic. It is used by both JavaServer Faces technology and JavaServer Pages. It uses the # symbol. Figure 2 shows that EL uses simple expressions to dynamically access data from components—for example, where the purchase order subtotal is displayed on the page or the compute method is invoked when a button is clicked. CDI to bind service and presentation tiers. To bind both Java and Expression Language, CDI comes to the rescue with a @Named annotation. As you can see in Figure 2, it basically gives a name to a CDI bean so the bean can be bound in EL. So here, where PurchaseOrderBean is annotated with

CDI takes the concept of state management much further, applying it to the entire application, not just to the HTTP layer. Plus, CDI does this in a declarative way: by using a single annotation, the state of the bean is managed by the container. @Named("po"), it means that the bean can be bound in EL with the name po. CDI to manage state. CDI goes further by managing the state of a bean for us using scopes. Let's say that on the top right corner of our web application, we need to display the login of the user. We want this information to remain until the user's session ends. In such a case, we just annotate the bean with @SessionScoped and CDI will manage the state by destroying the bean when the session ends. On the other hand, computing and displaying the





total of a purchase order should be done each time the page is refreshed. Because the scope of the PurchaseOrderBean must be shorter than the session, we can annotate it with @RequestScoped. CDI will maintain the state of the bean only on a per-request basis, which means that this bean is stateless. With just a few annotations, CDI unifies the web tier and service tier, eliminating glue code and letting the developer think about the business problem. CDI defines a uniform model for all our tiers bringing well-defined contexts, which is preserved across multiple requests in a user interaction.

Binding

Binding is the basic service for bringing together the web tier and the service tier. If we want to reference a bean in non-Java code that supports EL, such as a JSF page, we must assign the bean an EL name. The EL name is specified using the @Named built-in qualifier. Then we can easily use the bean in any JSF page through an EL expression. EL was orig-

inally inspired by both ECMAScript and the XPath expression languages. It was introduced in Java EE to make it easy for web page developers to access and manipulate Java in the back end without having to use JavaScript.

Expression Language. EL has a very simple syntax. It uses the hash symbol and curly brackets to identify an expression that needs to be evaluated. These expressions can be more or less complex (see Listing 1) and use arithmetic operators, lambda expressions, and so forth.

Listing 1.

// Value Expressions
#{purchaseOrderBean.subtotal}
#{purchaseOrderBean.customer.name}

// Array Expressions
#{purchaseOrderBean.orders[2]}

// Method Expressions
#{purchaseOrderBean.compute}

// Parameterized Method Calls
#{purchaseOrderBean.compute('5')}

Value expressions are the most common because they can read and write data. Here, our page would access the subtotal attribute or the customer name attribute of the PurchaseOrderBean. The syntax also allows access to items in an array or list, using the square bracket notation and specifying an index. As here, the expression returns the second purchase order of the bean. Another useful feature of EL is its support of method expressions. A method expression is used to invoke a public method of a bean, which can return a result. Here, the expression invokes the compute method of the PurchaseOrderBean. Parameterized method calls can use parameters. Here, the number 5 is passed as the compute value. JSF pages. Coming back to our presentation tier, EL is present in JSF pages in different forms. In Listing 2, for example, value expressions are used to display the subtotal or valueadded tax (VAT) rate of a purchase order. This binding is bidirectional, meaning that these expressions can also change the value of these attributes once the page is posted to the server. Method expressions are handy when we need to perform an action when a button is clicked, such as computing the amount of the purchase order. In this case, clicking the compute button will invoke the compute method of the PurchaseOrderBean.

Listing 2.

CDI beans. The PurchaseOrderBean in Listing 3 has subtotal and vatRate attributes, with getters and setters. It also has a compute method that is in charge of computing the total amount of the purchase order given a certain VAT rate. There is nothing special except the @Named annotation —without it, the bean would not have an EL name and, therefore, could not be bound to the page.

f •

Listing 3.

```
@Named
public class PurchaseOrderBean {
    private Float subtotal = 0F;
    private Float vatRate = 5.5F;
    // ...
    public String compute() {
        Float vat = subtotal * (vatRate / 100);
        Float discount =
            subtotal * (discountRate / 100);
        total = subtotal + vat - discount;
        return null;
        }
        // ...
}
```

@Named. The **@Named** annotation makes it possible to reference the bean from EL and, therefore, its attributes and methods. We can let CDI choose a name for us by not specifying an argument to the **@Named** annotation. The name defaults to the unqualified class name, decapitalized—in this case, purchaseOrderBean with a lowercase *p*. But we can specify an argument to the **@Named** annotation to use a non-default name. With **@Named("order")**, the expression needs to be renamed accordingly.

Binding with Producers and Alternatives

As we've just seen, the @Named annotation allows the binding between an expression and a bean. Coupled with a producer, anything can then be referenced in EL. For example, we produce an integer, we name it, and it can then be referenced in an expression. Alternatives can also be used to switch the implementation not only in Java, but also in EL. Naming a producer. To illustrate named producers and alternatives, let's take a NumberProducer class, the role of which is to produce numbers (see Listing 4). It has vatRate and discountRate attributes, both of type Float. The idea is to produce these attributes so they can be managed by CDI and injected somewhere else. As you know by now, this code could be ambiguous because both attributes have the same data type, Float. To differentiate them, we use a @VAT qualifier on one, and a @Discount qualifier on the other. Now, if we want to access the VAT rate directly on a JSF page, we just annotate the produced attribute with @Named. By default the EL name is vatRate, so the JSF page just references the vatRate directly, without having to prefix the name of the class: NumberProducer (<h:inputText</pre> value="#{vatRate}"/>). Remember that @Named uses a default name that we can override. For example, instead of vatRate we can change the name to vat and reference it in this expression: (<h:inputText value="#{vat}"/>).

Listing 4.

}

public class NumberProducer {

```
@Produces
@VAT
@Named("vat")
private Float vatRate = 5.5F;
```

@Produces
@Discount
@Named("discount")
private Float discountRate = 2.25f;

Alternative producer. Now, let's say we have a different use case. VAT rate and discount rate need to change depending on external configuration. For example, the VAT rate is 5.5 percent in certain countries and 19.6 percent in others, or the discount rate is usually 2.25 percent, but for Christmas it is set to 4.75 percent. This is the typical use case where alterna-

```
f
•
•
```

tives can be used. First, we still need to produce, qualify, and name both the VAT and discount rate attributes (see Listing 5). Then we add an @Alternative annotation. As you can see, CDI is very expressive. Each annotation has its own meaning, and we can read the code very easily. Then it's just a matter of enabling or disabling the alternatives in beans.xml.

Listing 5.

```
public class NumberProducer {
```

```
@Produces @VAT @Named("vat")
private Float vatRate = 5.5F;
```

@Produces @VAT @Named("vat") @Alternative private Float vatRateAlt = 19.6F;

```
@Produces @Discount @Named("discount")
private Float discountRate = 2.25f;
```

@Produces

@Discount @Named("discount") @Alternative
private Float discountRateAlt = 4.75f;

}

State Management

We're all used to the concept of an HTTP session and an HTTP request. These are two examples of the broader problem of managing state that is associated with a particular context, while ensuring that all needed cleanup occurs when the context ends—for example, when the HTTP session ends, it needs to be cleaned up. Traditionally, this state management has been implemented manually, by getting and setting servlet session and request attributes. CDI takes the concept of state management much further, applying it to the entire application, not just to the HTTP layer. Plus, CDI does this in a declarative way: by using a single annotation, the state of the bean is managed by the container. No more

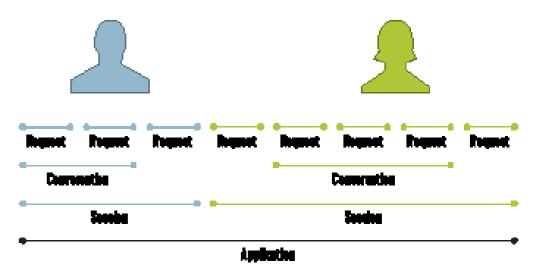


Figure 3. The four built-in CDI scopes

memory leaks when the application fails to clean up session attributes; the CDI container does it automatically. CDI extends the context model defined by the Servlet specification—application, session, request—to another context: a conversation. It then applies the context to the entire business logic, not just to the web tier. Built-in scope. Before looking at some code, let's first examine the four built-in CDI scopes shown in Figure 3. Let's say we have an application that has a lifespan of several months. We boot the server and leave it up and running for a few months before we shut it down. In this case, the application scope lasts for a very long time. One user logs in and remains logged in for a few minutes. The session scope spans from the moment he logs in until the moment he logs out. A second user logs in but her session stays active for a bit longer. Each session is independent and belongs to a single user, and the lifespan can be totally different. In the meantime, both users click at their own pace. Each click creates a request that is handled on the server. The last scope is the conversation and is slightly different because it can span for as long as needed. It's just a matter of beginning a conversation, which can span several requests, and ending it. Each user will have his or her own conversation. Each of these scopes

f ♪

is represented by an annotation.

Application scope. For example, let's say the application needs a global cache—a simple one, with just a map of key-value objects and a few methods to add data to the cache, get a value depending on the key, and remove a cache entry. We want this cache to be shared across all users' interactions within the application. For that, we just annotate the bean with @ApplicationScoped (see Listing 6). This cache will be automatically created by the CDI container when it is needed, and automatically destroyed when the context in which it was created ends. That is when the server is shut down. If we want this cache to be referenced directly from a JSF page, just add a @Named annotation.

Listing 6.

```
@Named
@ApplicationScoped
public class Cache implements Serializable {
```

```
private Map<Object,Object> cache =
    new HashMap<>();

public void addToCache(
    Object key, Object value) {
    // ...
}
public Object getFromCache (Object key) {
    // ...
}
public void removeFromCache (Object key) {
    // ...
}
```

Session scope. Application-scope beans live during the application and are shared to all users. Session-scoped beans live during the time of the HTTP session and belong only to the current user. This scope is useful, for example, for model-

ing a shopping cart (see Listing 7). Each user has his own list of items and, while he's logged in, he can add items to the shopping cart and check out at the end. This instance of the shopping cart will be automatically created for the first time when the session is created and automatically destroyed when the session ends. The instance is bound to the user session and is shared by all requests that execute in the context of that session. Again, use @Named if invocation from EL is needed.

```
Listing 7.
```

@Named
@SessionScoped
public class ShoppingCart
 implements Serializable {

// ...

```
private List<Item> cartItems =
    new ArrayList<>();
public String addItemToCart() {
    // ...
}
public String checkout() {
```

} }

Request scope. Until now, all the scopes we've covered handle state. For stateless applications, we can use the HTTP request and request scope beans. These beans usually model services (see Listing 8), or controllers, that have no state—for example, creating a book, retrieving all the book cover images, or getting a list of books depending on a category. Usually they have an @Named annotation because they are invoked when a button or a link on a page is clicked. An object that is defined as @RequestScoped is created once for every request and does not have to be serializable.

}

```
Listing 8.
```

```
@Named
@RequestScoped
public class BookService {
    public Book persist(Book book) {
        // ...
    }
    public List<String> findAllImages() {
        // ...
    }
    public List<Book> findByCategory(
        long categoryId) {
        // ...
    }
}
```

Conversation scope. The last built-in scope is the conversation scope. The conversation scope is a bit like the session scope in that it holds the state associated with a user and spans multiple requests to the server. However, unlike the session scope, the conversation scope is demarcated explicitly by the application. Let's say we have several web pages that form a wizard, to allow a customer to create a profile (see Listing 9). For controlling the lifecycle of a conversation, CDI gives us a Conversation API that may be obtained by injection. So, when a user starts to create a profile, a conversation is started by calling the begin method. The user can then go from page to page, go back to the previous page, go to the next page, and so on, until the conversation ends. As you can see, the conversation scope is the only one that needs explicit demarcation. All the other scoped beans are cleaned up by the CDI container; conversations need to be explicitly started and ended or they time out.

Listing 9. @Named @ConversationScoped

```
public class CustomerWizard implements
   Serializable {
    @Inject
    private Conversation conversation;
    private Customer customer =
        new Customer();
    public void initProfile () {
        conversation.begin();
        // ...
    }
    public void endProfile () {
        // ...
        conversation.end();
    }
```

}

Dependent scope. All the scopes we've just seen are contextual scopes. This means their lifecycle is managed by the container, and any injected references to the beans are also contextually aware. The CDI container ensures that the objects are created and injected at the correct time, as determined by the scope that is specified for these objects. The dependent scope is not a contextual scope and is actually called a pseudo-scope. Dependent scope is the default CDI bean scope. If a bean does not declare a specific scope, it will be injected as a dependent-scoped bean. This means that it will have the same scope as the bean where it's being injected. For example in Listing 10, if a request-scoped service (BookService) injects a dependent IsbnGenerator, then the injected IsbnGenerator will be request scoped. An instance of a dependent bean is strictly dependent on some other object. IsbnGenerator is instantiated when BookService is created and destroyed when BookService is destroyed. We can always use the *@Dependent* annotation, but we don't have to, because it is the default scope.

```
Listing 10.
@Dependent
public class IsbnGenerator {
    public String generateNumber() {
        return "13-84356-" +
            Math.abs(new Random().nextInt());
    }
}
```

@RequestScoped
public class BookService {

```
@Inject
private IsbnGenerator generator;
// ...
```

Conclusion

}

In this article, we've examined how to bring the web tier and service tier together thanks to binding with the @Named annotation and state management with scopes. When using CDI, there is no distinction between presentation tier components and business logic components. Both can be scoped, injected, or referenced in EL. We can layer our application according to whatever architecture we need rather than being forced to bend our application logic into a technical layering. And if the architecture layering is too flat, nothing stops us from creating an equivalent layered architecture using CDI. It is possible to write Java EE applications where everything is a CDI bean. </article>

LEARN MORE

- <u>CDI specification</u>
- <u>Beginning Java EE 7</u>
- <u>PluralSight course on CDI 1.1</u>
- <u>Weld CDI reference implementation</u>

//javaone recap /

JAVAONE 2015: Wrap-up and Review

JavaOne, the annual monster shindig for Java developers, was held this year in San Francisco, California, in late October. This year, more than 9,000 developers participated in almost 500 sessions, with the average attendee participating in 14 sessions.

The overarching themes at this year's conference were Java's 20th anniversary and its expanding presence in two areas: the cloud and the Internet of Things (IoT).

Notable among new cloud services unveiled at the show was a new one from Oracle named Java SE Cloud Service, which incorporates Java and a suite of development tools including Git, Maven, and Hudson all aimed at moving programming to the cloud.

Java's ability to scale down to small devices was the focus of the IoT track. This theme was repeated in the Java Lounge in the vendor exhibit area, where attendees could use soldering irons and other tools to build devices using the Raspberry Pi, a small, hobbyist-oriented technology that has become wildly popular in the last few years. The Saturday before the show, JavaOne4Kids, a technical convention for children, hosted 450 attendees who learned how to program robots using Java.

An annual fixture of JavaOne is the <u>Duke's Choice Awards</u>, which recognize particularly meritorious Java projects or community members. The winners this year included AsciidocFX (a document creation tool), Byte Buddy (a library for generating and manipulating bytecodes, discussed in detail on page 19), OmniFaces (a library for web applications), and KumuluzEE (a microservices-enabling technology).

Oracle videotaped most of the sessions and made them available online at no cost. This carefully curated and categorized <u>list</u> provides an excellent way to see videos of the sessions you might have missed.

The next JavaOne conference will be held September 18–22, 2016, in San Francisco. For a listing of other conferences and events, see the Events section in this issue. Antonio Goncalves is a senior developer specializing in Java/Java FF. He has written books covering Java EE 5, Java EE 6, and Java EE 7. A Java Champion who founded the Paris JUG and Devoxx France, Goncalves is an independent JCP member on various JSRs and also does Java EE training for PluralSight.