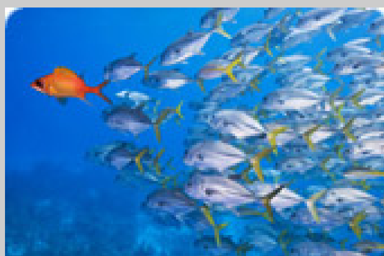
Article

Introducing the Java EE 6 Platform: Part 1

By Ed Ort, December 2009

Articles Index

Part 1 | Part 2 | Part 3



Java Platform, Enterprise Edition (Java EE) is the industry-standard platform for building enterprise-class applications coded in the Java programming language. Based on the solid foundation of Java Platform, Standard Edition (Java SE), Java EE adds libraries and system services that support the scalability, accessibility, security, integrity, and other requirements of enterprise-class applications.

Since its initial release in 1999, Java EE has matured into a functionally rich, high performance platform. Recent releases of the platform have also stressed simplicity and ease of use. In fact, with the

current release of the platform, Java EE 5, development of Java enterprise applications has never been easier or faster.

Progress continues. The next release of the platform, Java EE 6, adds significant new technologies, some of which have been inspired by the vibrant Java EE community. It also further simplifies the platform, extending the usability improvements made in previous Java EE releases.

Java EE 6 adds significant new technologies and extends the usability improvements made in previous Java EE releases.

This article highlights some of the significant enhancements in Java EE 6.

Contents

- Java EE 6 Goals
- Powerful New Technologies
- Enhanced Web Tier Capabilities
- EJB Technology, Even Easier to Use
- A More Complete Java Persistence API
- Further Ease of Development
- Profiles and Pruning
- Summary
- For More Information
- Comments

Java EE 6 Goals

Here are the main goals for the Java EE 6 platform:

More Flexible Technology Stack. Over time, the Java EE platform has gotten big, in some cases too big for certain types of applications. To remedy this, Java EE 6 introduces the concept of *profiles*, configurations of the Java EE platform that are designed for specific classes of applications. A profile may include a subset of Java EE platform technologies, additional technologies that have gone through the Java Community Process, but are not part of the Java EE platform, or both. Java EE 6 introduces the first of these profiles, the Web Profile, a subset of the Java EE platform designed for web application development. The Web Profile includes only those technologies needed by most web application developers, and does not include the enterprise technologies that these developers typically don't need.

- Java EE 6 introduces the Web Profile, a subset of the Java EE platform designed for web application development.

In addition, the Java EE 6 platform has identified a number of technologies as candidates for pruning. These

Get Java EE Training and Certification

- Java EE Training
  Find out about training for architects and web component, business component, and integration developers.
- Certification
  Learn about various Sun certification courses for programmers and enterprise architects, preparation methods, and savings programs.

Ed Ort is a writer on the staff of the Sun Developer Network. He has written extensively about a wide variety of programming topics including relational database technology, programming languages, web services, and Ajax. Read his blog.

candidates include technologies that have been superseded by newer technologies or technologies that are not widely deployed. *Pruning* a technology means that it can become an optional component in the next release of the platform rather than a required component.

Enhanced Extensibility. Over time, new technologies become available that are of interest to web or enterprise application developers. Rather than adding these technologies to the platform — and growing the platform without bounds — Java EE 6 includes more extensibility points and more service provider interfaces than ever before. This allows you to plug in technologies — even frameworks — in your Java EE 6 implementations in a standard way. Once plugged in, these technologies are just as easy to use as the facilities that are built into the Java EE 6 platform.

- More extensibility points and service provider interfaces as well as web tier features such as support for self-registration makes the platform highly extensible.

Particular emphasis on extensibility has been placed on the web tier. Web application developers often use third-party frameworks in their applications. However, registering these frameworks so that they can be used in Java EE web applications can be complicated, often requiring developers to add to or edit large and complex XML deployment descriptor files. Java EE 6 enables these frameworks to self-register, making it easy to incorporate and configure them in an application.

Further Ease of Development. Java EE 5 made it significantly easier to develop web and enterprise applications. For instance, Java EE 5 introduced a simpler enterprise application programming model based on Plain Old Java Objects (POJOs) and annotations, and eliminated the need for XML deployment descriptors. In addition, Enterprise JavaBeans (EJB) technology was streamlined, requiring fewer classes and interfaces and offering a simpler approach to object-relational mapping by taking advantage of the Java Persistence API (informally referred to as JPA).

- Usability improvements in many areas of the platform makes it even easier to develop web and enterprise applications.

Java EE 6 makes it even easier to develop enterprise or web applications. Usability improvements have been made in many areas of the platform. For example, you can use annotations to define web components such as servlets and servlet filters. Furthermore, a set of annotations for dependency injection has been standardized, making injectable classes much more portable across frameworks. In addition, Java EE application packaging requirements have been simplified. For example, you can add an enterprise bean directly to a web archive (WAR) file. You no longer need to package an enterprise bean in a Java archive (JAR) file and then put the JAR file in an enterprise archive (EAR) file.

Powerful New Technologies

Java EE 6 adds significant new technologies that make the platform even more powerful. Three of these are described below:

- Java API for RESTful Web Services (JAX-RS)
- Contexts and Dependency Injection for the Java EE Platform (CDI)
- Bean Validation

Java API for RESTful Web Services (JAX-RS)

Java API for RESTful Web Services (JAX-RS), JSR 311 enables you to rapidly build lightweight web services that conform to the Representational State Transfer (REST) style of software architecture. An important concept in REST is the existence of resources, each of which can be referred to with a global identifier, that is, a URI. In particular, data and functionality are considered resources that can be identified and accessed through URIs. To manipulate these resources, components of the network, clients and servers, communicate through a standardized interface such as HTTP and a small, fixed set of verbs — `GET`, `PUT`, `POST`, and `DELETE` — and exchange representations of these resources.

RESTful web services are web services built according to the REST architectural style. Building web services with the RESTful approach has emerged as a popular alternative to using SOAP-based technologies thanks to REST's lightweight nature and the ability to transmit data directly over HTTP.

JAX-RS furnishes a standardized API for building RESTful web services in Java. The API contributes a set of annotations and associated classes and interfaces. Applying the annotations to POJOs enables you to expose web resources. This approach makes it simple to create RESTful web services in Java.

JAX-RS makes it simple to create RESTful web services in Java.

The specification for the initial release of the technology, JAX-RS 1.0, was finalized in October 2008 and a reference implementation named Jersey is also available. Java EE 6 includes the latest release of the technology, JAX-RS 1.1, which is a maintenance release that aligns JAX-RS with new features in Java EE 6.

Let's take a look at a RESTful web service that uses JAX-RS.

```
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.Get;
import javax.ws.rs.Post;
import javax.ws.rs.Consumes;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core UriInfo;
import javax.ws.rs.core.UriBuilder;
import java.net.URI;

@Path ("items")
@Produces (MediaType.APPLICATION_XML)
Public class ItemsResource {

    @Context UriInfo uriInfo;

    @GET
    Items listItems() {
        Return Allitems();
    }

    @POST
    @Consumes (MediaType.APPLICATION_XML)
    Public Response create(Item item) throws ItemCreationException {
        Item newItem = createItem(item);
        URI newItemURI = uriInfo.getRequestUriBuilder().path(newItem.getId()).build();
        return Response.created(newItemURI).build();
    }

    ...
}
```

In this example, the `ItemsResource` class is a web service that manages a set of items. The imports in the class are for JAX-RS 1.1 annotations, classes, and interfaces.

Annotations add much of the information needed to identify resources and serve HTTP requests.

The `@Path` annotation specifies a relative path for the resource, in this case `"items"`. The URI for the class resource is based on the application context. So if the application context for this example is `http://example.com`, the URI for the class resource is `http://example.com/items`. This means that if a client directs a request to the URI `http://example.com/items`, the `ItemsResource` class will serve it.

The `@GET` annotation specifies that the annotated method, here the `listItems()` method, handles HTTP `GET` requests. When a client directs an HTTP `GET` request to the URI for the `ItemsResource` resource, the JAX-RS

runtime invokes the `listItems()` method to handle the `GET` request.

Notice the `@Produces` annotation. It specifies the MIME media types that the methods in the resource can produce and return to the client. In the `ItemsResource` example, the `@Produces` annotation specifies `MediaType.APPLICATION_XML`. The `MediaType` class is an abstraction of a MIME media type. Constants supplied to the class identify the particular media type to be abstracted. The `MediaType.APPLICATION_XML` specification is an abstraction of the MIME media type for XML content, "application/xml".

JAX-RS automatically translates between Java types and MIME media types.

Annotations such as `@Produces` suggest some of the content type translation that JAX-RS handles automatically. For example, the `listItems()` method returns a Java object of type `Items`. JAX-RS automatically translates that Java type to the "application/xml" MIME type to be used in the HTTP response to the client. Note that the translation is automatic only if the returned type is supported by default. For instance, if `Items` is a JAXB-annotated bean, then the translation would be automatic. However, if `Items` is a POJO, you would need to implement a `MessageBodyReader` to handle the serialization.

You can also specify a `@Produces` annotation on a method. In that case, the MIME type you specify on the method overrides the MIME types in any `@Produces` annotation that you specify on the class. For example, you could specify a `@Produces` annotation for the `listItems()` method as follows:

```
@GET
@Produces (MediaType.TEXT_PLAIN)
Items listItems() {
    Return Allitems();
}
```

JAX-RS would then translate the `Items` Java type to the "text/plain" MIME type, which represents plain text, and return content of that type in the HTTP response to the client.

The `@POST` annotation specifies that the annotated method, in this case, the `create()` method, responds to HTTP POST requests. In this example, the method creates a new item, perhaps in a database, and then returns a response indicating that it created the new item. When a client directs an HTTP `POST` request to the URI for the `ItemsResource` resource, the JAX-RS runtime invokes the `create()` method to handle the `POST` request.

Notice that the `@Consumes` annotation is specified on the `create()` method. The annotation specifies the MIME media types that the methods in the resource can accept from the client. As is the case for the `@Produces` annotation, if you specify `@Consumes` on a class, it applies to all the methods in the class. If you specify `@Consumes` on a method, it overrides the MIME type in any `@Consumes` annotation that you specify for the class. In the example, the

`@Consumes` annotation specifies that the `create()` method can accept XML content, that is, the MIME type "application/xml". Here the type translation is from MIME type to Java type. When a client submits XML content in a `POST` request to the URI for the `ItemsResource` class, JAX-RS invokes the `create()` method and automatically translates the incoming XML to the `Item` Java type required for the method's argument.

JAX-RS also includes a number of utility classes and interfaces that further simplify actions related to building and using RESTful web services in Java. You've already seen one of them: `MediaType`, a class for abstracting MIME media types. Some others are:

Utility classes and interfaces further simplify actions related to building and using RESTful web services.

- `UriInfo`, an interface for accessing URI information. In this example, the `@Context` annotation injects the `UriInfo` interface into the `uriInfo` field of the `ItemsResource` class.
- `UriBuilder`, a class for building URIs from their components
- `Response`, a class represents an HTTP response
- `Response.ResponseBuilder`, a class that builds `Response` objects, in accordance with the well-known Builder Pattern

These classes and interfaces are used in the following statements in the example:

```
URI newItemURI = uriInfo.getRequestUriBuilder().path(newItem.getId()).build();
return Response.created(newItemURI).build();
```

The first statement builds a URI for the new item. The `getRequestUriBuilder()` method is a `UriInfo` method that creates a `UriBuilder` object. The `path()` and `build()` methods are `UriBuilder` methods that together construct the URI for the new item.

The second statement creates a `Response` object for the new item to be returned to the client. The `created` method is a `Response` method that creates a `Response.ResponseBuilder` object. The `build()` method is a `Response.ResponseBuilder` method that creates the `Response` object for the new item. This object delivers metadata to the JAX-RS runtime to construct the HTTP response.

These utility classes and interfaces hide a lot of the complexity of HTTP programming — another reason why using JAX-RS is a simple way to build RESTful web services. However, this simplicity also extends beyond web services. JAX-RS can simplify the development of many types of HTTP-aware web applications. For example, if you need to build a web application that examines HTTP headers, you can probably code it in a much simpler way by

JAX-RS eliminates a lot of the low-level programming required in HTTP-aware web applications.

using JAX-RS rather than other approaches.

JAX-RS has other convenient features. For example, JAX-RS includes a number of parameter-based annotations to extract information from a request. One of these is `@QueryParam`, with which you can extract query parameters from the `Query` component of a request URL. Some other parameter-based annotations are `@MatrixParam`, which extracts information from URL path segments, `@HeaderParam`, which extracts information from HTTP headers, and `@CookieParam` which extracts information from the cookies declared in cookie-related HTTP headers.

For information about all the features in JAX-RS 1.1, see Java API for RESTful Web Services (JAX-RS), JSR 311.

Contexts and Dependency Injection for the Java EE Platform

Contexts and Dependency Injection for the Java EE Platform (CDI), JSR 299 is a technology that supplies a powerful set of services to Java EE components. These services allow Java EE components, including EJB session beans and JavaServer Faces (JSF) managed beans, to be bound to lifecycle contexts, to be injected, and to interact in a loosely coupled way by firing and observing events. Perhaps most significantly, CDI unifies and simplifies the EJB and JSF programming models. It allows enterprise beans to replace JSF managed beans in a JSF application.

> CDI unifies and simplifies the EJB and JSF programming models. It allows enterprise beans to act as JSF managed beans in a JSF application, and brings transactional support to the web tier.

In essence, CDI helps bridge what was a major gap between the web tier of the Java EE platform and the enterprise tier. The enterprise tier, through technologies such as EJB and JPA, has strong support for transactional resources. For example, using EJB and JPA you can easily build an application that interacts with a database, commits or rolls back transactions on the data, and persists the data. The web tier, by comparison, is focused on presentation. Web tier technologies such as JSF and JavaServer Pages (JSP pages) render the user interface and display its content, but have no integrated facilities for handling transactional resources.

Through its services, CDI brings transactional support to the web tier. This can make it a lot easier to access transactional resources in web applications. For example, CDI makes it a lot easier to build a Java EE web application that accesses a database with persistence provided by JPA.

Let's look at some key parts of a web application that uses CDI services. The application, which processes user login and user logout requests, includes both JSF and EJB components. Here is the code for an input form on a JSF page that displays a login prompt for the web application:

```
<f:view>
    <h:form>
        <h:panelGrid columns="2" rendered="#{!login.loggedIn}">
           <h:outputLabel for="username">Username:</h:outputLabel>
           <h:inputText id="username" value="#{credentials.username}"/>
           <h:outputLabel for="password">Password:</h:outputLabel>
           <h:inputText id="password" value="#{credentials.password}"/>
        </h:panelGrid>
        <h:commandButton value="Login" action="#{login.login}" rendered="#{!login.loggedIn}"/>
        <h:commandButton value="Logout" action="#{login.logout}" rendered="#{login.loggedIn}"/>
    </h:form>
</f:view>
```

As you can see from the code, the login prompt displays fields for a user to enter a user name and password. It also displays a Login button and a Logout button. Notice the unified expression language (EL) expressions such as `#{credentials.username}` and `#{login.login}`. These expressions refer to beans, named `credentials` and `login`.

Note that CDI builds on a new concept introduced in Java EE 6 called managed beans, which is designed to unify all of the various types of beans in Java EE 6. A *managed bean* is a Java class that is treated as a managed component by the Java EE container. Optionally, you can give it a name in the same namespace as that used by EJB components. A managed bean can also rely on a small number of container-provided services, mostly related to lifecycle management and resource injection. Other Java EE technologies such as JSF, EJB, and CDI build on this basic definition of a managed bean by adding services. So for example, a JSF managed bean adds lifecycle scopes, an EJB session bean adds services such as support for transactions, and CDI adds services such as dependency injection. In CDI a managed bean or simply a *bean* is a Java EE component that can be injected into other components, associated with a context, or reached through EL expressions.

CDI builds on managed beans, which are designed to unify all of the various types of beans in Java EE 6.

You declare a managed bean by annotating its class with the `javax.annotation.ManagedBean` annotation or by using one of several CDI annotations such as a scope annotation or a qualifier annotation. Scope annotations and qualifier annotations are discussed later in this section. The annotation-based programming model makes it possible for a bean to begin as a POJO and later become another type of Java EE component such as an EJB component — perhaps to take advantage of more advanced functionality, such as transactional and security annotations or the instance pooling facility offered by EJB containers. For example, you can turn a POJO into a stateful session bean by adding a `@Stateful` annotation to the object. Clients that use CDI to access a bean are unaffected by the bean's transition from POJO to EJB.

Any bean can be bound to a lifecycle context, can be injected, and can interact with other beans in a loosely coupled

way by firing and observing events. In addition, a bean may be called directly from Java code, or as in this example, it may be invoked in a unified EL expression. This enables a JSF page to directly access a bean, even a bean that is implemented as an EJB component such as a session bean.

In this application, a bean named `Credentials` has a lifecycle that is bound to the JSF request. The `Credentials` bean is implemented as a JavaBean as follows:

```
@Model
public class Credentials {

   private String username;
   private String password;

   public String getUsername() { return username; }
   public void setUsername(String username) { this.username = username; }

   public String getPassword() { return password; }
   public void setPassword(String password) { this.password = password; }
}
```

To request CDI services, you annotate a Java EE component with CDI annotations. The `@Model` annotation is a CDI annotation that identifies the `Credentials` bean as a model object in an Model-View-Controller (MVC) architecture. The annotation, which is built into CDI, is a stereotype annotation. A stereotype annotation marks a class as fulfilling a specific role within the application.

The application also includes a `Login` bean whose lifecycle is bound to the HTTP session. The `Login` bean is implemented as an EJB stateful session bean, as follows:

CDI services allow Java EE components, including EJB components, to be bound to lifecycle events.

```
@Stateful
@SessionScoped
@Model
public class Login {

   @Inject Credentials credentials;
   @Inject EntityManager userDatabase;

   private User user;

   @TransactionAttribute(REQUIRES_NEW)
   @RolesAllowed("guest")
   public void login() {
      ...
   }

   public void logout() {
      user = null;
   }

   public boolean isLoggedIn() {
      return user!=null;
   }

   @RolesAllowed("user")
   @Produces @LoggedIn User getCurrentUser() {
      ...
   }
}
```

The `@Stateful` annotation is an EJB annotation that specifies that this bean is an EJB stateful session bean. The `@TransactionAttribute` and `@RolesAllowed` annotations are also EJB annotations. They declare the EJB transaction demarcation and security attributes of the annotated methods.

The `@SessionScoped` annotation is a CDI annotation that specifies a scope for the bean. All beans have a scope that determines the lifecycle of its instances and the instances of the bean that are made visible to instances of other beans. This is an important feature because components such as EJB components do not have a well-defined scope. In particular, EJB components are not aware of the request, session, and application contexts of web tier components such as JSF managed beans, and do not have access to the state associated with those contexts. In addition, the lifecycle of a stateful EJB component cannot be scoped to a web-tier context.

All beans have a scope. Among other things, this gives EJB components access to the request, session, and application contexts of web tier components.

By contrast, scoped objects in CDI exist in a well-defined lifecycle context that is managed by the Java EE container. Scoped objects may be automatically created when needed and then automatically destroyed when the context in which they were created ends. Significantly, the state of a scoped object is automatically shared by clients that execute in the same context. This means that clients such as other beans that execute in the same context as a scoped object see the same instance of the object. But clients in a different context see a different instance. The `@SessionScoped` annotation specifies that the scope type for the `Login` bean is session scope. Objects that are not associated with any of the usual scopes, but instead exist for the exclusive benefit of an object that triggered their creation, are said to be *dependents* of their owner. The lifecycle of these dependent objects is tied to that of the owner. In particular, a dependent object is destroyed whenever the owner is destroyed.

CDI services allow Java EE components, including EJB and JPA components, to be injected.

Beans typically acquire references to other beans through dependency injection. The dependency injection mechanism is completely type safe. CDI uses the annotations specified in JSR 330: Dependency Injection for Java for dependency injection. One of those annotations, `@Inject`, identifies a point at which a dependency on a Java class or interface can be injected. The container then provides the needed resource. In this example, the `Login` bean specifies two injection points. The first use of the `@Inject` annotation in the example injects a dependency on the `Credentials` bean. In response, the container will inject the `Credentials` bean into any instance of `Login` created within this context. The second `@Inject` annotation injects a dependency on the JPA `EntityManager`. The container will inject the `EntityManager` to manage the persistence context. Refer to Standardized Annotations for Dependency Injection to learn more about the `@Inject` annotation and other annotations in JSR 330.

The `@Produces` annotation identifies the `getCurrentUser()` method as a producer method. A producer method is called whenever another bean in the system needs an injected object of the specified type. In this case, the injected object is the currently logged-in user, which is injected by the qualifier annotation `@LoggedIn`. A *qualifier* identifies a specific implementation of a Java class or interface to be injected. In order to use a qualifier annotation, you first need to define its type as a qualifier. You use the `@Qualifier` annotation, another JSR 330 annotation, to do that. For example:

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

Let's return to the login prompt discussed earlier. When a user responds to the prompt and clicks the Submit button, CDI technology goes into action. The Java EE container automatically instantiates a contextual instance of the `Credentials` bean and the `Login` bean. An instance of a bean that is bound to a context is called a *contextual*

instance. JSF assigns the user name and password the user entered to the `Credentials` bean contextual instance. Next, JSF calls the `login()` method in the `Login` bean contextual instance. This instance continues to exist for and is available to other requests in the same HTTP session, and provides the `User` object that represents the current user to any other bean that requires it.

This example demonstrates only some of the features in this powerful technology. Another feature enables beans to produce or consume events. Yet another lets you define interceptors that bind additional function across all bean types, or define decorators that apply the additional function to a specific bean type. To learn about these and the other features in CDI technology, see Contexts and Dependency Injection for the Java EE Platform, JSR 299.

Bean Validation

Validating data is a common task that occurs throughout an application. For example, in the presentation layer of an application, you might want to validate that the number of characters a user enters in a text field is at most 20 characters or that the number a user enters in a numeric field is positive. If you set those constraints, you probably want the same data to be validated before it's used in the business logic of the application and when the data is stored in a database.

Developers often code the same validation logic in multiple layers of an application, something that's time consuming and error-prone. Or they put the validation logic in their data model, cluttering it with what is essentially metadata.

Bean Validation, JSR 303 makes validation simpler and reduces the duplication, errors, and clutter that characterizes the way validation is often handled in enterprise applications. Bean Validation affords a standard framework for validation, in which the same set of validations can be shared by all the layers of an application.

Bean Validation affords a standard framework for validation, in which the same set of validations can be shared by all the layers of an application.

Specifically, Bean Validation offers a framework for validating Java classes written according to JavaBeans conventions. You use annotations to specify constraints on a JavaBean — you can annotate a JavaBean class, field, or property. You can also extend or override these constraints through XML descriptors. A validator class then validates each constraint. You specify which validator class to use for a given type of constraint.

Here, for example, is part of a class that declares some constraints through Bean Validation annotations:

```
public class Address {
    @NotNull @Size(max=30)
    private String addressline1;

    @Size(max=30)
    private String addressline2;

     ...

    public String getAddressline1() {
           return addressline1;
    }

    public void setAddressline1(String addressline1) {
           this.addressline1 = addressline1;
    }

     ...
}
```

The `@NotNull` annotation specifies that the annotated element, `addressline1`, must not be null. The `@Size` annotation specifies that the annotated elements, `addressline1` and `addressline2`, must not be longer than the specified maximum, 30 characters.

When an `Address` object is validated, the `addressline1` value is passed to a validator class that is defined for the `@NotNull` constraint as well as to a validator class defined for the `@Size` constraint. The `addressline2` value is also passed to the validator class for the `@Size` constraint. The pertinent validator classes perform the validations.

Both the `@NotNull` and `@Size` constraints are built into the Bean Validation framework so you do not need to define validator classes for them. However, you can add your own constraints to the built-in constraints, in which case, you need to define validator classes. For example, you can define a constraint named `@ZipCode` as follows:

Bean Validation includes a number of built-in constraint definitions. You add your own constraints by declaring an annotation type that specifies a validator class.

```
@Size(min=5, max=5)
@ConstraintValidator(ZipcodeValidator.class)
@Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface ZipCode {
    String message() default "Wrong zipcode";
    String[] groups() default {};
}
```

Then you can specify the `@ZipCode` constraint on a class, field, or property just like any other constraint. Here is an example:

```
public class Address {
     ...

    @ZipCode
    private String zipCode;

    public String getZipCode() {
           return zipCode;
       }

       public void setZipCode(String zipCode) {
           this.zipCode = zipCode;
       }

     ...
}
```

When an `Address` object is validated, the `zipCode` value is passed to the `ZipcodeValidator` class for validation. Notice that the constraint definition includes another constraint: `@Size(min=5, max=5)`. This means that an element annotated by the `@ZipCode` annotation must be exactly 5 characters in length. The element is validated against this constraint in addition to the primary constraint check that `ZipcodeValidator` performs. Bean Validation allows you to create a constraint that is composed of other constraints. In fact, any composing constraint can itself be composed of constraints. Notice too that the constraint definition specifies an error message to be returned if the constraint fails the validation check. Here, the error message is "Wrong zipcode".

You can also use Bean Validation to validate an entire object graph in a straightforward way. An *object graph* is an object composed of other objects. If you specify the `@Valid` annotation on the root object of an object graph, it

directs the pertinent validator to recursively validate the associated objects in the object graph. Consider the following example:

In addition to validating individual objects, you can use Bean Validation to validate an entire object graph.

```
public class Order {
    @OrderNumber private String orderNumber;
    @Valid @NotNull private Address delivery;
}
```

When an `Order` object is validated, the `Address` object and the associated objects in its object graph are validated too.

To meet the objective of sharing the same set of validations across all the layers of an application, Bean Validation is integrated across the Java EE 6 platform. For example, presentation-layer technologies such as JSF and enterprise-layer technologies such as JPA have access to the constraint definitions and validators available through the Bean Validation framework. You no longer need to specify constraints in multiple places and in multiple ways across the layers of an application.

Bean Validation is integrated across the Java EE 6 platform.

To learn more about Bean Validation, see Bean Validation, JSR 303.

» Continue to the next part of this article

Part 1 | Part 2 | Part 3

Rate This Article

Comments

We welcome your participation in our community. Please keep your comments civil and on point. You may optionally provide your email address to be notified of replies - your information is not used for any other purpose. By submitting a comment, you agree to these Terms of Use.

About Sun | About This Site | Newsletters | Contact Us | Employment | How to Buy | Licensing | Terms of Use | Privacy | Trademarks

Copyright Sun Microsystems, Inc.

A Sun Developer Network Site

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this License.

Sun Developer RSS Feeds

Article

Introducing the Java EE 6 Platform: Part 2

By Ed Ort, December 2009

Part 1 | **Part 2** | Part 3

Enhanced Web Tier Capabilities

Some of the most significant enhancements made in Java EE 6 appear in the web tier. As mentioned earlier, one of the goals of Java EE 6 is to make the platform more extensible, and two key improvements in the area of extensibility are web fragments and shared framework pluggability. These two new features are provided in Java EE 6 by Servlet 3.0 technology. Servlet 3.0, JSR 315, the latest version of Servlet technology, offers some other valuable enhancements such as support for asynchronous processing and support for annotations.

Another important Java EE 6 web tier technology is JSF 2.0, the latest version of JSF technology. Among its

benefits, JSF 2.0 simplifies page and component authoring through Facelets, and adds support for asynchronous JavaScript and XML (commonly referred to as Ajax), and annotations.

Support for Web Fragments in Servlet 3.0

Web application developers often use third-party frameworks such as Apache Wicket or Spring MVC in their applications. To use these frameworks, developers need to register the frameworks in the web application, a task that involves configuring framework-specific artifacts such as servlets and listener classes. It's typical for developers to register these frameworks by specifying deployment descriptors for the frameworks in the application's `web.xml file` — the same file that contains deployment descriptors for the web components that constitute the web application. Not only does this make for some very large `web.xml`, files but it also makes it difficult to isolate and maintain the descriptors for the frameworks.

Web fragments, a new feature of Servlet 3.0 technology, solves this problem by modularizing deployment descriptors. A *web fragment* can be considered a logical segment of a `web.xml` file. There can be multiple web fragments, each representing a logical segment, and the set of web fragments can be viewed as constituting an entire `web.xml` file. This logical partitioning of the `web.xml` file enables web frameworks to self-register to the web container. Each web framework that you use in a web application can define in a web fragment all the artifacts that it needs, such as servlets and listeners, without requiring you to edit or add information in the `web.xml` file.

> Web fragments enable web frameworks to self-register, eliminating the need for you to register them through deployment descriptors.

Here is an example of a web fragment that registers a servlet and a listener:

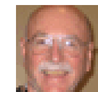
```
<web-fragment>
     <servlet>
         <servlet-name>myFrameworkServlet</servlet-name>
         <servlet-class>myFramework.myFrameworkServlet</servlet-class>
     </servlet>

     <listener>
         <listener-class>myFramework.myFrameworkListener</listener-class>
     </listener>
 </web-fragment>
```

The `<web-fragment>` element identifies a web fragment. A web fragment must be in a file named `web-fragment.xml` and can be placed in any location in a web application's classpath. However, it's expected that a web framework will typically place its web fragments in the `META-INF` directory of the framework's JAR file,

Get Java EE Training and Certification

- Java EE Training
  Find out about training for architects and web component, business component, and integration developers.
- Certification
  Learn about various Sun certification courses for programmers and enterprise architects, preparation methods, and savings programs.



Ed Ort is a writer on the staff of the Sun Developer Network. He has written extensively about a wide variety of programming topics including relational database technology, programming languages, web services, and Ajax. Read his blog.

which will typically reside in the `WEB-INF/lib` directory of the web application.

You use the element `<metadata-complete>` in the `web.xml` file to instruct the web container whether to look for web fragments as well as annotations — see Annotations in More Types of Java EE Components for information about annotations provided by Servlet 3.0 technology. If you set `<metadata-complete>` to false, or do not specify the `<metadata-complete>` element in your `web.xml` file, then during deployment, the container must scan web fragments and annotations to build the effective metadata for the web application. In response, the web container searches for web fragments and annotations in framework JAR files. The web container then uses the configuration information in each web fragment to register the framework for use with the web application. However, setting `<metadata-complete>` to true, causes the deployment descriptors to provide all the configuration information for the web application. In this case, the web container does not search for web fragments and annotations.

Because Servlet 3.0 technology supports web fragments, you can modularize your `web.xml` file. Your web application can still have the traditional, monolithic `web.xml` file, or it can have a logically partitioned `web.xml` file that includes one or more web fragments.

> With its support for web fragments, Servlet 3.0 technology lets you modularize your `web.xml` file.

However, because Servlet 3.0 enables you to modularize your deployment descriptors, the order in which these descriptors are processed can be important. For example, the order in which the descriptors for an application are processed affects the order in which servlets, listeners, and filters are invoked. With Servlet 3.0, you can specify the order in which deployment descriptors are processed.

Servlet 3.0 supports absolute ordering and relative ordering of deployment descriptors. Your specify absolute ordering using the `<absolute-ordering>` element in the `web.xml` file. You specify relative ordering with an `<ordering>` element in the `web-fragment.xml` file.

For example, suppose your application includes two web fragments — `MyFragment2` and `MyFragment3`, and also includes a `web.xml` file. You can declare absolute ordering of the descriptors by specifying the following in the `web.xml` file for the application:

```
<web-app>
    <name>MyApp</name>
    <absolute-ordering>
       <name>MyFragment3</name>
       <name>MyFragment2</name>
    </absolute-ordering>
    ...

</web-app>
```

Here, the processing order would be as follows:

- `web.xml`. The `web.xml` descriptor is always processed first.
- `MyFragment3`.
- `MyFragment2`.

Shared Framework Pluggability

Web fragments and annotations are not the only way that Servlet 3.0 allows you to extend a web application. You can also plug in shared copies of frameworks, such as Java API for XML-Based Web Services (JAX-WS), JAX-RS and JSF, that are built on top of the web container. Servlet 3.0 introduces a new interface called `ServletContainerInitializer` that can be used to plug in a framework.

For example, here's how you can plug in a framework named `A`:

```
@HandlesTypes(AnnotationA.class)

 AServletContainerInitializer implements ServletContainerInitializer
 {
    public void onStartup(Set<Class<A>>c, ServletContext ctx) throws ServletException {
      // Framework-specific code here to initialize the runtime
      // and setup the mapping etc.
      ServletRegistration reg = ctx.addServlet("AServlet", "com.foo.AServlet");
   reg.addServletMapping("/foo");
```

The container discovers the `ServletContainerInitializer` using the JAR services API. It does this when the container or application is started. The framework implementing the `ServletContainerInitializer`

must bundle in the `META-INF/services` directory of its JAR file a file called `javax.servlet.ServletContainerInitializer` that points to the implementation class of the `ServletContainerInitializer`. The `@HandlesTypes` annotation specifies the types that the `ServletContainerInitializer` can handle. Any classes of those types discovered in any JAR contained in the `WEB-INF/lib` directory are passed to the `ServletContainerInitializer`. The `ServletContainerInitializer` is then able to use the same programmatic configuration APIs as `ServletContextListeners`.

Asynchronous Processing in Servlet 3.0

Servlet 3.0 introduces support for asynchronous processing. With this support, a servlet no longer has to wait for a response from a resource such as a database before its thread can continue processing, that is, the thread is not blocked. This support enables long-lived client connections such as those in chat room applications. In these types of applications you don't want a server thread to be blocked for a long period of time serving a request from a single client. You want the servlet to process a request from the client and then free up the server thread as quickly as possible for other work. Among its benefits, support for asynchronous processing makes the use of servlets with Ajax more efficient.

> A servlet no longer has to wait for a response from a resource such as a database before its thread can continue processing.

Servlets and servlet filters that support asynchronous processing must be written with the goal of asynchrony in mind. In particular, several long-standing assumptions about the order in which some methods will be called do not apply for asynchronous processing. To ensure that code written for synchronous processing won't be used in an asynchronous context, Servlet 3.0 requires you to use the `asyncSupported=true` attribute. To make a servlet asynchronous, you specify `asyncSupported=true` in a `@WebServlet` annotation and make asynchronous requests in the servlet. You can also mark a servlet filter as asynchronous by specifying `asynchSupported=true` in a `@WebFilter` annotation. Only after taking these steps are taken the corresponding classes available for asynchronous invocations.

The support for asynchronous processing also includes new `ServletRequest` methods, such as `startAsync()`, to make an asynchronous request, and new classes, such as `AsyncContext`, which provides the execution context for an asynchronous operation.

Here, for example, is a servlet that makes an asynchronous request.

```
@WebServlet(name="CalculatorServlet", asyncSupported=true, urlPatterns={"/calc", "/getVal"})
public class CalculatorServlet extends HttpServlet{
   public void doGet(HttpServletRequest req, HttpServletResponse res) {
   ...
      AsyncContext aCtx = req.startAsync(req, res);

   }
   ...
}
```

Notice that the `startAsync()` method returns an `AsyncContext` object. This object holds the request and response objects that were passed in the call to the method. At this point, the thread that served the original request is available for other operations.

Servlet 3.0 also introduces a new listener class, `AsyncListener`, that notifies you when an asynchronous operation is complete or if a timeout occurs. The `AsyncContext` class includes a `complete()` method, with which you can commit the response after an asynchronous operation is complete. The `AsyncListener` class also has a `dispatch()` method that forwards the asynchronous request to the container so that other frameworks such as JSP can generate the response.

Simplified Page Authoring in JSF 2.0

JavaServer Faces technology provides a server-side component framework that simplifies the development of user interfaces (UIs) for Java EE applications. The latest release of the technology, JSF 2.0, JSR 314, makes UI development for Java EE applications even easier. One area of particular improvement is page authoring. Authoring a JSF page is much easier in JSF 2.0 through the use of the standard JavaServer Faces View Declaration Language, commonly called Facelets.

Facelets

Facelets is a powerful but lightweight declaration language that you can use to present JSF pages. In the Facelets approach, you use HTML-style templates to present a JSF page and to build component trees. Although JSF can be used with different display technologies, most JSF applications use JSP as the display technology. In other words, the UI in a JSF application is typically a JSP page that contains JSF components. However, Facelets offers several advantages over JSP.

Facelets is a powerful but lightweight declaration language that you can use to present JSF pages.

In JSP, elements in a web page are processed and rendered in a progressive order. However, JSF provides its own processing and rendering order. This can cause unpredictable behavior when web applications are executed. Facelets

resolves this mismatch. Facelets also enables code reuse through templating and can significantly reduce the time to develop and deploy UIs. For these reasons, Facelets is now the preferred presentation technology for building JSF-based applications.

Facelets is now the preferred presentation technology for building JSF-based applications.

Facelets are usually written with XHTML markup language. This allows Facelets pages to be portable across diverse development platforms. Here, for example, is a Facelets XHTML page that is part of a sample JSF application provided with the Java EE 6 Tutorial.

```
<xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
        <title>Guess Number JSF Application</title>
</head>
<body>
   <h:form>
    <h2>
    Hi. My name is Duke. I am thinking of a number from <b>
           <h:outputText value="#{UserNumberBean.minimum}"/> to
             <b>
    <h:outputText value="#{UserNumberBean.maximum}"/>.
    <p>
      Can you guess it ?
    </p>
    <h:graphicImage id="waveImg" url="/wave.med.gif" />
    <h:inputText id="userNo"
          value="#{UserNumberBean.userNumber}">
     converterMessage="#{ErrMsg.userNoConvert}">
    <f:validateLongRange
          minimum="#{UserNumberBean.minimum}"
          maximum="#{UserNumberBean.maximum}"/>
    </h:inputText>
    <h:commandButton id="submit"
          action="success" value="submit" />
    <h:message showSummary="true" showDetail="false"
               style="color: red;
                font-family: 'New Century Schoolbook', serif;
                font-style: oblique;
                text-decoration: overline"
                id="errors1"
                for="userNo"/>
    </h2>
```

```
        </h:form>
    </body>
    </html>
```

The page renders the UI shown in Figure 1. The UI prompts a user to guess a number that the system — in the person of Duke, the Java technology mascot — has selected. The UI displays the text `Hi my name is Duke. I am thinking of a number from` *`min`* `to` *`max`*`.` , where *`min`* and *`max`* represent the minimum and maximum values allowable as a guess, respectively. The UI also displays the Duke image, a text field for the user to enter a number, and a button to submit the form.

Figure 1. *A UI Created With Facelets*

This Facelets XHTML page is not very different from an equivalent JSP page. In particular, Facelets supports JSF and JSTL tag libraries. Facelets also includes a Facelets tag library that enables feature-rich page templating. The namespace declaration `xmlns:ui="http://java.sun.com/jsf/facelets"` is for the Facelets tag library — although no tags in that library are used in this example. Facelets also supports the unified expression language.

It might not be evident here what additional value Facelets provides over JSP. To better understand the value of Facelets, lets examine two of its most powerful features: templating and composite components.

Templating

With templating, you can create a page that acts as a template for other pages in an application. This helps you avoid creating similarly constructed pages multiple times. Templating also helps maintain a standard look and feel in an application with a large number of pages.

Templating allows you to create a page that acts as a template for other pages in an application.

The Facelets tag library contains a templating tag, `<ui:insert>`. To implement templating, you create a template page that includes the `<ui:insert>` tag. You then create a client page that uses the template. In the client page, you use a `<ui:composition>` tag to point to the template and `<ui:define>` tags to specify content to insert into the template.

Here is an example of a template page.

```
<xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"

   <head>
      <title><ui:insert name="title">Page Title</ui:insert</title><body>
   </head>
   <body>
       <div>
           <ui:insert name="Links"/>
       </div>
       <div>
           <ui:insert name="Data"/>
      </div>
   </body>
</html>
```

Here is an example of a client page that uses the template.

```
<xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

   <html xmlns="http://www.w3.org/1999/xhtml"
         xmlns:ui="http://java.sun.com/jsf/facelets"
         xmlns:h="http://java.sun.com/jsf/html"
    <body>
      <ui:composition template="/template.xhtml">
          This text will not be displayed.
          <ui:define name="title">
              Welcome page
          </ui:define>
          <ui:define name="Links">
              ... [Links should be here]
          </ui:define>
          <ui:define name="Links">
              ... [Data should be here]
          </ui:define>
      </ui:composition>
          This text also will not be displayed.
    </body>
   </html>
```

When the template is invoked by the client, it renders a page with the title `Welcome Page`. The page also displays two sections: one that lists the links specified in the client, and one that shows the data specified in the client.

Composite Components

Composite components makes it easy to create customized JSF components.

Composite components is a new feature in JSF that makes it easy to create customized JSF components. You can create composite components by using JSF page markup, other JSF UI components, or both. And with the help of Facelets, any XHTML page can become a composite component. In addition, composite components can have validators, converters, and listeners attached to them just like the set of UI components provided by JSF.

After you create a composite component, you can store it in a library and use it as needed.

Let's create a composite component that is rendered as a login panel. When a user logs in, the component reports a login event as shown in Figure 2.

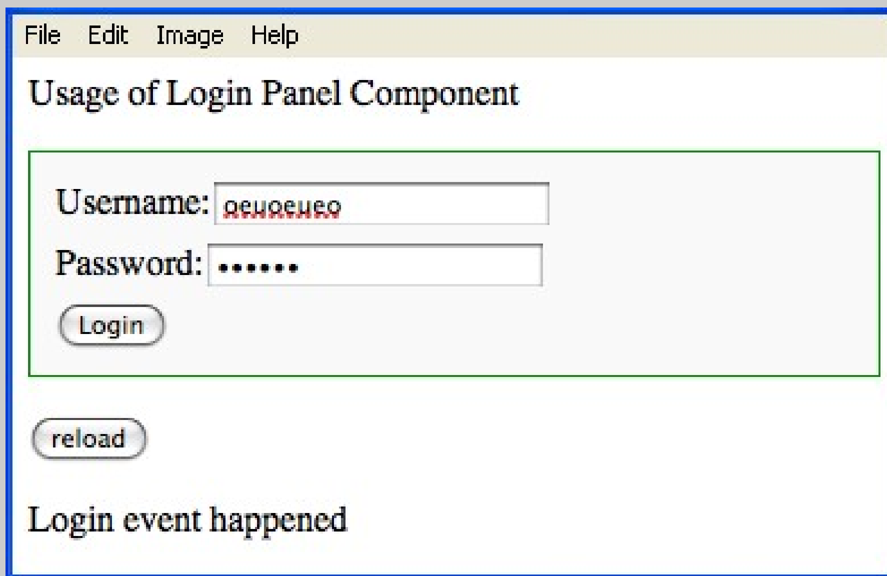


Figure 2. *Composite Component*

Here is the source code for the composite component.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core">
   xmlns:f="http://java.sun.com/jsf/facelets">
   xmlns:composite="http://java.sun.com/jsf/composite">

<h:head>
<title>This content will not be displayed in the rendered output</title>
</h:head>

<h:body>
 <composite:interface>
        <composite:actionSource name="loginEvent"/>
 </composite:interface>
 <composite:implementation>
  <table>
         <tr>
            <td>Username:  <h:inputText id="username" /> </td>
         </tr>
         <tr>
            <td>Password: <h:inputSecret id="password" /></td>
         </tr>
```

```
            <tr>
               <td><h:commandButton value="Login" id="loginEvent" /></td>
            </tr>
     </table>
    </composite:implementation>
   </h:body>
   </html>
```

The declaration `xmlns:composite="http://java.sun.com/jsf/composite"` declares the namespace for composite UI components. The `<composite:interface>` tag declares the usage contract for the composite component, in other words, what a page author needs to know to use the composite component. The `<composite:attribute>` tag in the usage contract specifies a `<composite:actionSource>` tag. This tag indicates that the component can expose an event, making it accessible by any page that uses the composite component.

The `<composite:implementation>` tag defines the implementation for the composite component. Here the implementation is a simple table that contains JSF components for the username and password fields and a login button.

To make the composite component available for use, you save the code in an XHTML file and then store the file in a subdirectory of the `resources` directory under the application root directory. The name of the subdirectory is taken to be the name of the resource library that contains the composite component. The JSF runtime finds the composite component by appending `.xhtml` to the name of the composite component's tag. For example, if you name the tag `loginPanel`, store the code for the composite component in a file named `loginPanel.xhtml`.

You can then use the composite component in a web page. Here, for example, is the code for the web page shown in Figure 2 that uses the composite component.

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:ez="http://java.sun.com/jsf/composite/ezcomp">

<head>
<title>Example 01>/title>
<style type="text/css">
.grayBox { padding: 8px; margin: 10px 0; border: 1px solid #CCC; background-color: #f9f9f9;  }
</style>
</h:head>

<h:body>
  <p>Usage of Login Panel Component</p>

     <ui:debug hotkey="p" rendered="true"/>

  <h:form>
      <div id="compositeComponent" class="grayBox" style="border: 1px solid #090;">
         <ez:loginPanel>
             <f:actionListener for="loginEvent" type="example01.LoginActionListener" />

         </ez:loginPanel>
      </div>
  <p><h:commandButton value="reload" /></p>

  <p><h:outputText value="#{loginActionMessage}" /></p>
  </h:form>

</h:body>
</html>
```

Notice the declaration `xmlns:ez="http://java.sun.com/jsf/composite/ezcomp"`. This specifies the namespace and prefix for the composite component. In this case, `ezcomp` is the name of the subdirectory in the resources directory. JSF uses the following convention: for any namespace URI starting with http://java.sun.com/jsf/composite/, the one and only path segment that ends the namespace URI is taken to be the name of the resource library in which the Facelets XHTML files for the composite components are found.

The `<f:actionListener>` tag associates an action listener with the composite component. The `for` attribute in the tag indicates that this listener is for the action event named `loginEvent` on the composite component. You would also need to provide code to process the event. For example, you might provide code that looks something like

the following:

```
import javax.faces.component.UIComponent;
import javax.faces.component.ValueHolder;
import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

public class LoginActionListener implements ActionListener {

    public void processAction(ActionEvent event) throws AbortProcessingException {
        FacesContext context = FacesContext.getCurrentInstance();
        context.getExternalContext().getRequestMap().put("loginActionMessage",
                "Login event happened");
    }
}
```

Support for Ajax in JSF 2.0

JSF 2.0 has built-in support for Ajax. With Ajax, web applications can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.

JSF 2.0 has built-in support for Ajax, making it easier to develop dynamic web applications that take advantage of both JSF technology and Ajax.

In support of Ajax, JSF's request processing cycle has been expanded to allow partial page updates and partial view traversal. Partial view traversal allows one or more components in a view to be visited, potentially to have them pass through either or both the execute phase or render phase of the request processing lifecycle. This is a key feature in JSF and Ajax frameworks and it allows selected components in the view to be processed, rendered, or both.

To use Ajax with JSF you need to access a JavaScript resource that has the resource identifier `jsf.js`. The resource, which exists under the `javax.faces` resource library, contains the JavaScript API that enables JSF to interact with Ajax. The JavaScript API comprises a standard set of JavaScript functions that facilitate Ajax operations in a JavaServer Faces framework. You rarely need to include this file directly. JSF automatically includes it whenever you use any Ajax-enabled tags or components in your view.

You can then make an Ajax request in either of two ways. You can use the `<f:ajax>` tag or you can invoke functions in the JavaScript API.

Here is an example that uses the `<f:ajax>` tag:

```
<h:commandButton id="button1">
   <f:ajax execute="..." render="..."/>
</h:commandButton>
```

Here, the <f:ajax> tag is nested inside an <h:commandButton> tag. This associates the Ajax action specified in the `execute` attribute with the command button rendered by the <h:commandButton> tag. You can also specify an `event` attribute to identify the JavaScript DOM event to which the Ajax action applies. If you do not specify an `event` attribute, JSF uses the default action for the component. In this case, the default action is `onclick`, so JSF associates the Ajax request specified in the `execute` attribute with the `onclick` event of the rendered button. When a user clicks the button, JSF submits the Ajax request to the server.

One benefit of using the <f:ajax> tag is that you don't have to specifically load the `jsf.js` resource in your page — it is done automatically for you. By comparison, if you invoke the JavaScript API, you first have to make the `jsf.js` resource available to the current view, using an <h:outputScript> tag. For example:

```
<f:view contentType="text/html"/>
  <h:head>
    <meta...
    <title...
  </h:head>
  <h:body>
    ...
    <h:outputScript name="jsf.js" library="javax.faces" target="body"/>
    ...
  </h:body>
  ...
```

You then use functions in the JavaScript API to make Ajax requests. For example, you use the JavaScript function `jsf.ajax.request` to send an Ajax request to the server, as shown in the following code example. The code includes a <h:commandButton> tag that renders a button. When a user clicks the button, an Ajax request is submitted to the server.

```
<h:commandButton id="button1" value="submit">
onclick="jsf.ajax.request(this,event);" />
```

JSF 2.0's built-in support for Ajax makes it a lot easier to develop dynamic web applications that take advantage of both JSF technology and Ajax.

More New Features in Servlet 3.0 and JSF 2.0

This section covered only some of the many new features and enhancements in Servlet 3.0 and JSF 2.0. Another new feature of note in Servlet 3.0 enables you to use methods in the `ServletContext` class to programmatically add servlets and servlet filters to a web application during startup. You use the `addServlet()` method to add a servlet to the web application, and the `addFilter()` method to add a servlet filter. The ability to programmatically add servlets and servlet filters at startup is particularly useful to framework writers. In conjunction with the shared framework pluggability feature by which extension libraries can discover classes listed in the `@HandlesTypes` annotation, with this facility web frameworks can configure themselves with no developer intervention.

In addition, Servlet 3.0 works with a number of enhanced security features. For example, in addition to declarative security, Servlet 3.0 offers programmatic security through the `HttpServletRequest` interface. You can, for example, use the `authenticate()` method of `HttpServletRequest` in an application to perform username and password collection, or you can use the `login()` method to direct the container to authenticate the request caller from within an unconstrained request context. For more information about these and other features in Servlet 3.0, see Servlet 3.0, JSR 315.

Some additional enhancements in JSF 2.0 relate to how resources are packaged and handled. JSF 2.0 standardizes where resources are packaged. All resources now go in a `resources` directory or a subdirectory. Resources are any artifacts that a component may need in order to be rendered properly, such as CSS files or JavaScript files. Figure 3 shows part of a NetBeans IDE project for a JSF application. Notice the `resources` directory in the project and the CSS and image resources it contains.

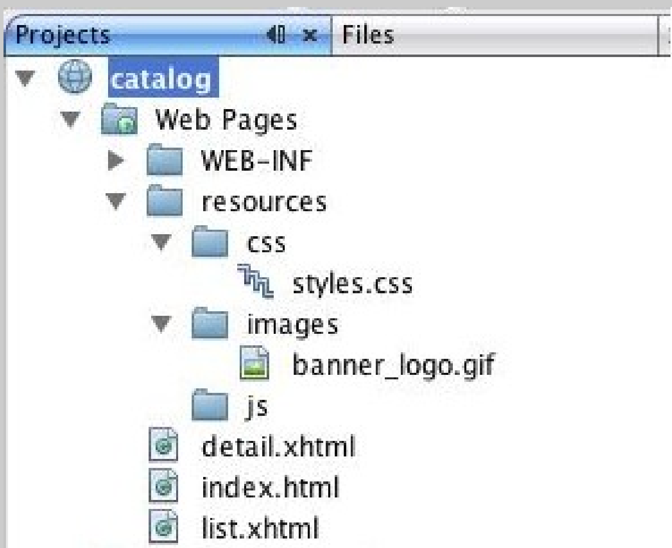


Figure 3. *Resources in the `resources` Directory of a JSF Application*

JSF 2.0 also includes new APIs for representing and handling resources. You use the `javax.faces.application.Resource` class to represent a resource. You use the `javax.faces.application.ResourceHandler` class to create instances of resources as well as to serve resources to the requesting user agent. For more information about these and other features in JSF 2.0, see JavaServer Faces 2.0: A Complete Tour. Also see the JSR 314: JavaServer Faces 2.0 specification.

» Continue to the next part of this article

Part 1 | **Part 2** | Part 3

Rate This Article

Comments

We welcome your participation in our community. Please keep your comments civil and on point. You may optionally provide your email address to be notified of replies - your information is not used for any other purpose. By submitting a comment, you agree to these Terms of Use.

About Sun | About This Site | Newsletters | Contact Us | Employment | How to Buy | Licensing | Terms of Use | Privacy | Trademarks

Copyright Sun Microsystems, Inc.

A Sun Developer Network Site

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this License.

Sun Developer RSS Feeds

Article

Introducing the Java EE 6 Platform: Part 3

By Ed Ort, December 2009

Part 1 | Part 2 | **Part 3**

EJB Technology, Even Easier to Use

Enterprise JavaBeans technology is the server-side component architecture for developing and deploying business applications in Java EE. Applications that you write using EJB technology are scalable, transactional, and secure. EJB 3.0, which is part of the Java EE 5 platform, made the technology a lot easier to use. The latest release of the technology, JSR 318: Enterprise JavaBeans 3.1, which is available in the Java EE 6 platform, further simplifies the technology and makes many improvements that reflect common usage patterns.

Some improvements made in EJB 3.1 are as follows:

- No-interface view. Allows you to specify an enterprise bean using only a bean class without having to write a separate business interface.
- Singletons. Lets you easily share state between multiple instances of an enterprise bean component or between multiple enterprise bean components in an application.
- Asynchronous session bean invocation. Enables you to invoke session bean methods asynchronously by specifying an annotation.
- Simplified Packaging. Removes the restriction that enterprise bean classes must be packaged in an `ejb-jar` file. You can now place EJB classes directly in a WAR file.
- EJB Lite. Is a subset of EJB 3.1 for inclusion in a variety of Java EE profiles.

No-Interface View

The EJB 3.0 local client view is based on a plain old Java interface (POJI) called a local business interface. A local interface defines the business methods that are exposed to the client and that are implemented on the bean class. This separation of interface and implementation is sometimes unnecessarily cumbersome and adds little value — this is especially true for fine-grained components that are accessed locally from clients within the same module.

EJB 3.1 simplifies this approach by making local business interfaces optional. A bean that does not have a local business interface exposes a *no-interface view*. Now you can get the same enterprise bean functionality without having to write a separate business interface.

> Local business interfaces are optional in EJB 3.1. Now you can get the same enterprise bean functionality without having to write a separate business interface.

The no-interface view has the same behavior as the EJB 3.0 local view, for example, it supports features such as pass-by-reference calling semantics and transaction and security propagation. However, a no-interface view does not require a separate interface, that is, all public methods of the bean class are automatically exposed to the caller. By default, any session bean that has an empty `implements` clause and does not define any other local or remote client views, exposes a no-interface client view.

For example, the following session bean exposes a no-interface view:

Get Java EE Training and Certification

- Java EE Training
  Find out about training for architects and web component, business component, and integration developers.
- Certification
  Learn about various Sun certification courses for programmers and enterprise architects, preparation methods, and savings programs.

Ed Ort is a writer on the staff of the Sun Developer Network. He has written extensively about a wide variety of programming topics including relational database technology, programming languages, web services, and Ajax. Read his blog.

```
@Stateless
public class HelloBean {

    public String sayHello() {
        String message = propertiesBean.getProperty("hello.message");
        return message;
    }

}
```

As is the case for a local view, the client of a no-interface view always acquires an EJB reference -- either through injection or JNDI lookup. The only difference is that the Java type of the EJB reference is the bean class type rather than the type of a local interface. This is shown in the following bean client:

```
@EJB
private HelloBean helloBean;

...

String msg = helloBean.sayHello();
```

Note that even though there is no interface, the client cannot use the `new()` operator to explicitly instantiate the bean class. That's because all bean invocations are made through a special EJB reference, or proxy, provided by the container. This allows the container to provide all the additional bean services such as pooling, container-managed transactions, and concurrency management.

Singletons

A singleton bean, also known as a singleton, is a new kind of session bean that is guaranteed to be instantiated once for an application in a particular Java Virtual Machine (JVM)*. With singletons, you can easily share state between multiple instances of an enterprise bean component or between multiple enterprise bean components in the application. Consider, for example, a class that caches data for an application. You might define the class as a singleton and in doing so, ensure that only one instance of the cache and its state is shared in the application.

Singletons help you easily share state between the EJB components in an application.

You define a singleton with the `@Singleton` annotation, as shown in the following code example:

```
@Singleton
public class PropertiesBean {

 private Properties props;

 public String getProperty(String name) { ... }

@PostConstruct
       public void initialize { // props = ...}

}
```

Because it's just another kind of session bean, a singleton can define the same local and remote client views as can stateless and stateful beans. Clients access singletons in the same way as they access stateless and stateful beans, that is, through an EJB reference. For example, a client can access the `PropertiesBean` singleton shown in the previous example as follows:

```
@EJB
private PropertiesBean propsBean;

...

String msg = propsBean.getProperty("hello.message");
```

Here, the container ensures that all invocations through `PropertiesBean` references in the same JVM are serviced by the same instance of the `PropertiesBean`. By default, the container enforces the same threading guarantee as for other component types. In other words, the singleton is fully thread safe. Specifically, no more than one invocation is allowed to access a particular bean instance at any one time. For singletons, that means blocking any concurrent invocations. However, this is just the default concurrency behavior. Additional concurrency options allow more efficient concurrent access to the singleton instance.

Asynchronous Session Bean Invocation

One of the powerful features introduced in EJB 3.1 is the ability to invoke session bean methods

asynchronously. For an asynchronous invocation, control returns to the client before the container dispatches the invocation to an instance of the bean. This allows the client to continue processing in parallel while the session bean method performs its operations.

With EJB 3.1, you can invoke session bean methods synchronously or asynchronously.

You can make a method asynchronous by marking it with the `@Asynchronous` annotation. You apply the annotation to a business method of a bean class. You can also use a deployment descriptor to designate a method as asynchronous.

Asynchronous methods can return a `java.util.concurrent.Future<V>` object or `void`. A `Future<V>` object holds the result of an asynchronous operation. You can access the `Future<V>` object to retrieve a result value, check for exceptions, or cancel an in-progress invocation. The `Future<V>` interface provides a `get()` method to retrieve the value. You can also retrieve the value by using the convenience class, `AsyncResult<V>`, which implements the `Future<V>` interface.

In the following example, the `performCalculation()` method in made asynchronous. The method uses the `AsyncResult<V>` class to retrieve the value returned in the `Future<V>` object.

```
@Stateless
Public class CalculatorBean implements CalculatorService {
   ...

   @Asynchronous
   public Future<Integer> performCalculation(...) {

      // ... do calculation

      Integer result = ...;
      return new AsyncResult<Integer>(result);
   }
}
```

Simplified Packaging

The EJB specification has always required that enterprise beans be packaged in an enterprise module called an `ejb-jar` file. Since it is common for Java EE web applications to use enterprise beans, this packaging requirement can be burdensome. These applications are forced to use a web application archive (`.war`) file for the web application, an `ejb-jar` file for the enterprise beans, and an enterprise archive (`.ear` file) that encompasses the other packages. This is illustrated in Figure 4. This packaging approach is further complicated

by the special handling required for any classes or resources that must be shared among the modules.

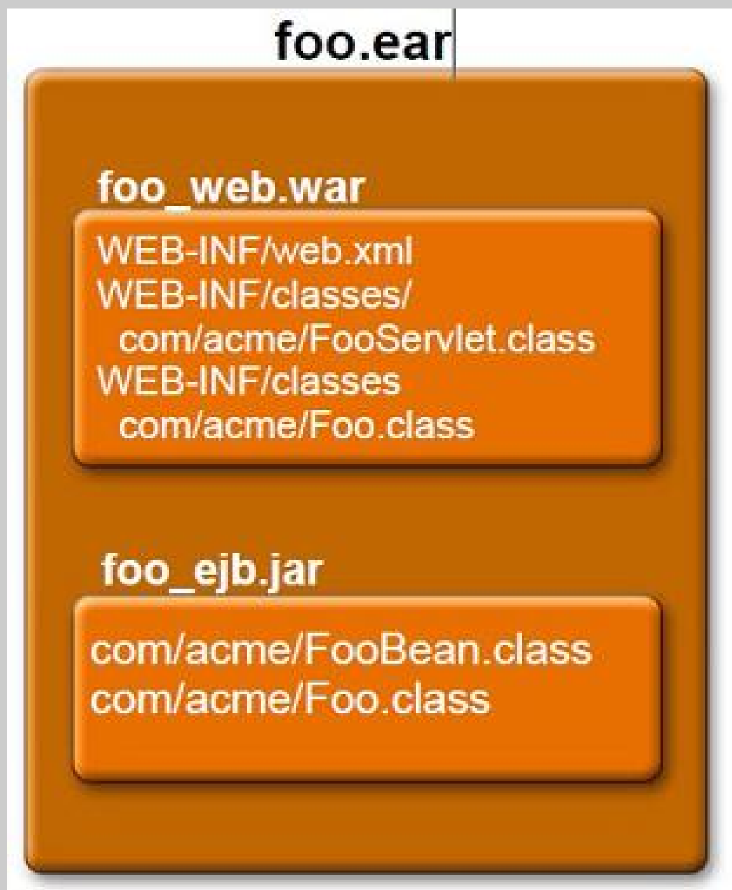


Figure 4. *Traditional Enterprise Application Packaging*

Figure 5. *Simplified Enterprise Application Packaging*

EJB 3.1 addresses this packaging complexity by removing the restriction that enterprise bean classes must be packaged in an `ejb-jar` file. As Figure 5 illustrates, you can now place EJB classes directly in the `.war` file, using the same packaging guidelines that apply to web application classes. This means that you can place EJB classes under the `WEB-INF/classes` directory or in a `.jar` file within the `WEB-INF/lib` directory. The EJB deployment descriptor is also optional. If you need it, you can package the EJB deployment descriptor as a `WEB-INF/ejb-jar.xml` file.

In EJB 3.1, you can place enterprise bean classes in the `.war` file along with web tier components. You don't have to put EJB classes in the `ejb-jar` file.

EJB Lite

For many applications, EJB technology offers a lot more functionality than those applications really need. The typical application that uses EJB only needs a subset of the features provided by the technology. EJB Lite meets the needs of these applications with a small subset of the features in EJB 3.1 centered around the session bean component model.

EJB Lite should simplify the use of EJB technology for many developers. Developers who use EJB Lite in their applications only need to learn to use a small set of features. In addition, applications developed with EJB Lite can run in application servers that implement either EJB Lite or the full EJB 3.1 API. Also, vendor implementations of EJB Lite should be simpler and more lightweight than full EJB implementations.

EJB Lite meets the needs of applications that require only a subset of the features provided by EJB technology.

Note that EJB Lite is not a product or an implementation, but rather a small convenient subset of the EJB 3.1 API. The objective of EJB Lite is to offer a subset of EJB 3.1 features that cover the common requirements for the business logic tier of most applications, one that also gives vendors the flexibility to provide EJB technology across a variety of Java EE profiles. To meet those objectives, EJB Lite offers the following features:

- Stateless, stateful, and singleton session beans
- Local EJB interfaces or no interfaces
- Interceptors
- Container-managed and bean-managed transactions
- Declarative and programmatic security
- Embeddable API

More New Features in EJB 3.1

EJB 3.1 delivers more features and enhancements than those covered in the previous sections. For example, it

includes an embeddable API and container for use in the Java SE environment. These makes it easy to test EJB components outside a Java EE container, and more generally, in Java SE. For another example, the introduction of singletons in EJB 3.1 offers a convenient way for EJB applications to receive callbacks during application initialization or shutdown. By default, the container decides when to instantiate the singleton instance. However, you can force the container to instantiate a singleton instance during application initialization by using the `@Startup` annotation. This allows the bean to define a `@PostConstruct` method that is guaranteed to be called at startup time. In addition, any `@PreDestroy` method for a singleton is guaranteed to be called when the application is shutting down, regardless of whether the singleton was instantiated using lazy instantiation or eager instantiation.

To learn about all the features and enhancements in EJB 3.1, see JSR 318: Enterprise JavaBeans 3.1.

A More Complete Java Persistence API

The Java EE 5 platform introduced the Java Persistence API, which provides a POJO-based persistence model for Java EE and Java SE applications. JPA handles the details of how relational data is mapped to Java objects, and it standardizes Object/Relational (O/R) mapping. JPA has been widely adopted and is recognized as the enterprise standard for O/R persistence.

Java EE 6 includes the latest release of this technology, JSR 317: Java Persistence 2.0. JPA 2.0 adds a number of significant new features and enhancements. These include:

- Object/Relational mapping enhancements, such as the ability to model collections of objects
- Additions to the Java Persistence query language
- A new criteria-based query API
- Support for pessimistic locking

Object/Relational Mapping Enhancements

JPA 1.0 supported the mapping of collections. However, those collections could only contain entities. JPA 2.0 adds the mapping of collections of basic data types, such as `Strings` or `Integers`, as well as collections of embeddable objects. Recall that an embeddable object in JPA is an object that cannot exist on its own, but exists as part of a parent object — that is, its data does not exist in its own table, but is embedded in the parent object's table.

In JPA 2.0, you can map collections of basic data types, such as `Strings` or `Integers`, as well as collections of embeddable objects.

JPA 2.0 adds two annotations in support of the new collection mappings:`@ElementCollection` and

`@CollectionTable`. You use the `@ElementCollection` annotation to specify that the basic or embeddable objects in the collection are stored in a separate table called a collection table. You use the `@CollectionTable` annotation to specify details about the collection table, such as its columns.

Here, for example, is an embeddable class that represents a service visit for a vehicle. The embeddable class stores the date of the visit, a description of the work that was done, and the cost. In addition, the vehicle can be equipped with one or more optional features. Each of the available features is an enumerated value of type `FeatureType`.

```
public enum FeatureType { AC, CRUISE, PWR, BLUETOOTH, TV, ... }

@Embeddable
public class ServiceVisit {
    @Temporal(DATE)
    @Column(name="SVC_DATE")
    Date serviceDate;

    String workDesc;
    int cost;
}
```

The enumerated values and embeddable objects can then be used in an entity that represents a vehicle's service history and its features.

```
@Entity
public class Vehicle {

    @Id int vin;

    @ElementCollection
    @CollectionTable(name="VEH_OPTNS")
    @Column(name="FEAT")
    Set<FeatureType> optionalFeatures;

    @ElementCollection
    @CollectionTable(name="VEH_SVC")
    @OrderBy("serviceDate")
    List<ServiceVisit> serviceHistory;
    ...
}
```

The first pairing of @ElementCollection and @CollectionTable annotations in the Vehicle entity specifies that the FeatureType values are stored in the VEH_OPTNS collection table. The second pairing of @ElementCollection and @CollectionTable annotations in the entity specifies that the ServiceVisit embeddable objects are stored in the VEH_SVC collection table.

Though not shown in the example, the @ElementCollection annotation has two attributes: targetClass and fetch. The targetClass attribute specifies the class name of the basic or embeddable class, and is optional if the field or property is defined using generics, as it is in this example. The fetch attribute is optional and specifies whether the collection should be retrieved lazily or eagerly, using the javax.persistence.FetchType constants of either LAZY or EAGER, respectively. By default, the collection is fetched lazily, which is the case in this example.

There are many more Object/Relational mapping enhancements in JPA 2.0 than the mapping of collections described here. For example, JPA 2.0 supports nested embeddables, embeddables with relationships, and ordered lists. There are also new annotations for generalized map functionality, more flexible support for specific access types through an @Access annotation, more mapping options for entity relationships such as foreign key mapping support for unidirectional one-to-many relationships, support for derived identities through the @MapsId annotation, and support for orphan removal.

Additions to the Java Persistence Query Language

JPA 1.0 defined an extensive Java Persistence query language (informally referred to as JPQL) with which you can query entities and their persistent state. JPA 2.0 makes a number of enhancements to JPQL. For example, you can now use case expressions in queries. In the following query, a case expression increases an employee's salary by a multiplier of 1.1 if the employee has a rating of 1, by a multiplier of 1.05 if the rating is 2, and by a multiplier of 1.01 for any other rating:

JPA 2.0 includes enhancements to Java Persistence query language. For example, you can now use case expressions in queries.

```
UPDATE Employee e
SET e.salary =
   CASE WHEN e.rating = 1 THEN e.salary * 1.1
        WHEN e.rating = 2 THEN e.salary * 1.05
        ELSE e.salary * 1.01
   END
```

JPA 2.0 also adds a number of new operators to JPQL. Two of these are `NULLIF`, and `COALESCE`. The `NULLIF` operator is particularly useful when a database uses something other than nulls to encode missing or non-applicable information. Using `NULLIF`, you can easily convert these values to nulls in queries. If the arguments to `NULLIF` are equal, `NULLIF` returns null, otherwise it returns the value of the first argument.

JPA 2.0 also adds two new operators to Java Persistence query language: `NULLIF`, and `COALESCE`.

For example, suppose that salaries in an employee table are represented as integer values and that missing salaries are encoded by -99999. Here's a query that returns the average value of the salaries. To correctly ignore the missing salaries, the query uses `NULLIF` to convert the -99999 values to null.

```
SELECT AVG(NULLIF(e.salary, -99999))
FROM Employee e
```

The `COALESCE` operator accepts a list of parameters and then returns the first non-null value from the list. It is equivalent to the following case expression:

```
CASE WHEN value1 IS NOT NULL THEN value1
     WHEN value2 IS NOT NULL THEN value2
     WHEN value3 IS NOT NULL THEN value3
     ...
     ELSE NULL
END
```

For example, suppose that an employee table includes columns for a work phone number and a home phone number. A missing phone number is represented by a null value. The following query returns the name and phone number of each employee. The `COALESCE` operator specifies that the query return the work phone number, but if it's null, return the home phone number. If both are null, return a null value for the phone number.

```
SELECT Name, COALESCE(e.work_phone, e.home_phone) phone
FROM Employee e
```

The other new operators that JPA 2.0 adds to JPQL are `INDEX`, `TYPE`, `KEY`, `VALUE`, and `ENTRY`. The `INDEX` operator queries over the ordering in an ordered list. The `TYPE` operator selects an entity's type and restricts a query to one or more entity types. The `KEY`, `VALUE`, and `ENTRY` operators are part of the generalized map functionality in JPA 2.0. You use the `KEY` operator to extract map keys, the `VALUE` operator to extract map values, and the `ENTRY` operator to select a map entry.

In addition, JPA 2.0 adds support for operators in the select list, as well as in collection-valued parameters and non-polymorphic queries.

Criteria API

One of the significant new features introduced in JPA 2.0 is the Criteria API, an API for dynamically constructing object-based queries. In essence, the Criteria API is the object-oriented equivalent of JPQL. With it, you can take an object-based approach to creating queries, rather than using the string manipulation required by JPQL.

You can use the new Criteria API to dynamically construct object-based queries.

The Criteria API is based on a metamodel, an abstract model that provides schema-level metadata about the managed classes of the persistence unit. The metamodel enables you to build queries that are strongly typed. It also enables you to browse the logical structure of a persistence unit.

Typically an annotation processor is expected to use the metamodel to generate static metamodel classes. These classes provide the persistent state and relationships of the managed classes of a persistence unit. However, you can also code the static metamodel classes.

Here, for example, is an entity:

```
@Entity public class Employee {
  @Id Long Id;
  String firstName;
  String lastName;
  Department dept;
}
```

And here is its corresponding static metamodel class:

```
import javax.persistence.meta,model.SingularAttribute;
import javax.persistence.meta,model.StaticMetamodel;


@Generated("EclipseLink JPA 2.0 Canonical Model Generation"
@StaticMetamodel(Employee.class)
public class Employee_ {
  public static volatile SingularAttribute<Employee, Long> id;
  public static volatile SingularAttribute<Employee, String> firstName;
  public static volatile SingularAttribute<Employee, String> lastName;
  public static volatile SingularAttribute<Employee, Department> dept;

}
```

In addition, a JPA 2.0 metamodel API enables you to dynamically access the metamodel. So when you use the Criteria API you can either statically access the metamodel classes or dynamically access the metamodel. However, the Criteria API gives you even more flexibility in that you can navigate the metamodel either in an object-based way or in a string-based way. This means that you have four ways to use the Criteria API:

The Criteria API is based on a metamodel for building queries that are strongly typed.

- Statically with metamodel classes
- Statically with strings
- Dynamically with the metamodel
- Dynamically with strings

No matter which of these approaches you use, you define a Criteria API-based query by constructing a

CriteriaQuery object. You construct the CriteriaQuery using a factory object called CriteriaBuilder. You can get the CriteriaBuilder from either the EntityManager or EntityManagerFactory class.The following code, for example, constructs a CriteriaQuery object:

```
EntityManager em = ... ;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
```

Notice that the CriteriaQuery object is generically typed. You use the createQuery method of CriteriaBuilder to create a CriteriaQuery and to specify a type for the query result. In this example, the Employee.class argument to the createQuery method specifies that the query result type is Employee. Significantly, CriteriaQuery objects and the methods that create them are strongly typed, and this typing continues through the execution of the query.

Next, you specify one or more query roots for the CriteriaQuery object. The query roots represent the entities on which the query is based. You create a query root and add it to a query with the from() method of the AbstractQuery interface. The AbstractQuery interface is one of a number of interfaces, such as CriteriaQuery, From, and Root, that are defined in the Criteria API. The CriteriaQuery interface inherits from the AbstractQuery interface.

The argument to the from() method is the entity class or EntityType instance for the entity. The result of the from() method is a Root object. The Root interface extends the From interface, which represents objects that may occur in the from clause of a query.

The following code adds a single query root to the CriteriaQuery object:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> emp = cq.from(Employee.class);
```

After you add one or more query roots to the CriteriaQuery object, you access the metamodel and then construct a query expression. How you do that depends on whether you issue the query statically or dynamically and whether you use the metamodel or strings to navigate the metamodel. Here's an example of a

static query that uses the metamodel classes:

```
cq.select(emp);
cq.where(cb.equal(emp.get(Employee_.lastName), "Smith"));
TypedQuery<Employee> query = em.createQuery(cq);
List<Employee> rows = query.getResultList();
```

The `select()` and `where()` methods of the `CriteriaQuery` interface specify the selection items that are to be returned in the query result.

Notice that you create a query using the `EntityManager` and you specify the `CriteriaQuery` object as input. This results in a `TypedQuery`, an extension introduced in JPA 2.0 to the `javax.persistence.Query` interface . The `TypedQuery` interface knows the type it returns so that strong typing continues into the query's execution.

In metamodel terminology, `Employee_` is the canonical metamodel class corresponding to the `Employee` entity class. A *canonical metamodel class* follows certain rules described in the JPA 2.0 specification. For example, the name of the metamodel class is derived from the name of the managed class by appending "_" to the name of the managed class. A *canonical metamodel* is a metamodel comprising the static metamodel classes that correspond to the entities, mapped superclasses, and embeddable classes in the persistence unit. This query, in fact, uses the canonical metamodel.

Here is the complete query:

```
EntityManager em = ... ;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> emp = cq.from(Employee.class);
cq.select(emp);
cq.where(cb.equal(emp.get(Employee_.lastName), "Smith"));
TypedQuery<Employee> query = em.createQuery(cq);
List<Employee> rows = query.getResultList();
```

Here's a dynamic version of the query that uses the metamodel API:

```
EntityManager em = ... ;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> emp = cq.from(Employee.class);
EntityType<Employee> emp_ = emp.getModel();
cq.select(emp);
cq.where(cb.equal(emp.get(emp_.getSingularAttribute("lastName", String.class)),"Smith"));
TypedQuery<Employee> query=em.createQuery(cq);
List<Employee> rows=query.getResultList();
```

A criteria query that uses the metamodel API provides the same type safety as one that uses the canonical metamodel, but it can be more verbose than queries based on the canonical metamodel.

The `getModel()` method of `Root` returns the metamodel entity corresponding to the root. It also enables run time access to the persistent attributes declared in the `Employee` entity. Again, the `select()` and `where()` methods of the `CriteriaQuery` interface specify the selection items that are to be returned in the query result.

The `getSingularAttribute()` method is a metamodel API method that returns a persistent single-valued property or field. In this example, it returns the `lastName` property whose value is `Smith`.

Here is a static version of the query that uses string navigation of the metadata:

```
EntityManager em = ... ;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> emp = cq.from(Employee.class);
cq.select(emp);
cq.where(cb.equal(emp.get("lastName"), "Smith"));
TypedQuery query = em.createQuery(cq);
List <Employee>rows = query.getResultList();
```

The string-based approach is relatively easy to use, but you lose the type safety that the metamodel enables.

Support for Pessimistic Locking

Locking is a technique for handling database transaction concurrency. When two or more database transactions concurrently access the same data, locking is used to ensure that only one transaction at a time can change the data.

There are generally two locking approaches: optimistic and pessimistic. Optimistic locking assumes infrequent conflicts between concurrent transactions, that is, they won't often try to read and change the same data at the same time. In optimistic locking, the objective is to give concurrent transactions freedom to process simultaneously, but to detect and prevent collisions. Two transactions can access the same data simultaneously. However, to prevent collisions, a check is made to detect any changes made to the data since the data was last read.

Pessimistic locking assumes that transactions will frequently collide. In pessimistic locking, a transaction that reads the data locks it. Another transaction cannot change the data until the first transaction commits.

JPA 1.0 only supported optimistic locking. You could indicate what type of locking was in effect by specifying a lock mode value of `READ` or `WRITE` in the `lock()` method of the `EntityManager` class. For example:

```
EntityManager em = ... ;
em.lock (p1, READ);
```

For the `READ` lock mode, the JPA entity manager locked the entity and before a transaction committed, checked the entity's version attribute to determine if it had been updated since the entity was last read. If the version attribute had been updated, the entity manager threw an `OptimisticLockException` and rolled back the transaction.

For the `WRITE` lock mode, the entity manager performed the same optimistic locking operations as for the `READ` lock mode. However, it also updated the entity's version column.

JPA 2.0 adds six new lock modes. Two of these are for optimistic locking. JPA 2.0 also permits pessimistic locking and adds three lock modes for that. A sixth lock mode specifies no locking.

These are the two new optimistic lock modes:

- `OPTIMISTIC`. This is the same as the `READ` lock mode. The `READ` lock mode is still supported in JPA

2.0, but specifying `OPTIMISTIC` is recommended for new applications.

- `OPTIMISTIC_FORCE_INCREMENT`. This is the same as the `WRITE` lock mode. The `WRITE` lock mode is still supported in JPA 2.0, but specifying `OPTIMISTIC_FORCE_INCREMENT` is recommended for new applications.

JPA 2.0 adds two new lock modes for optimistic locking. JPA 2.0 also permits pessimistic locking and adds three lock modes for that.

These are the three new pessimistic lock modes:

- `PESSIMISTIC_READ`. The entity manager locks the entity as soon as a transaction reads it. The lock is held until the transaction completes. Use this lock mode when you want to query data using repeatable-read semantics — in other words, when you want to ensure that the data is not updated between successive reads. This lock mode does not block other transactions from reading the data.
- `PESSIMISTIC_WRITE`. The entity manager locks the entity as soon as a transaction updates it. This lock mode forces serialization among transactions attempting to update the entity data. This lock mode is often used when there is a high likelihood of update failure among concurrent updating transactions.
- `PESSIMISTIC_FORCE_INCREMENT`. The entity manager locks the entity when a transaction reads it. It also increments the entity's version attribute when the transaction ends, even if the entity is not modified.

You can also specify the new lock mode `NONE`, in which case, no lock is acquired.

JPA 2.0 also provides multiple ways to specify the lock mode for an entity. You can specify the lock mode in the `lock()` and `find()` methods of the `EntityManager`. In addition, the `EntityManager.refresh()` method refreshes the state of the entity instance from the database and locks it in accordance with the entity's lock mode.

The following code example illustrates pessimistic locking with the `PESSIMISTIC_WRITE` lock mode:

```
// read
Part p = em.find(Part.class, pId);

// lock and refresh before update
em.refresh(p, PESSIMISTIC_WRITE);
int pAmount = p.getAmount();
p.setAmount(pAmount - uCount);
```

The code in this example first reads some data. It then applies a `PESSIMISTIC_WRITE` lock using a call to the `EntityManager.refresh()` method before updating the data. The `PESSIMISTIC_WRITE` lock locks the entity as soon as the transaction updates it. Other transactions cannot update the same entity until the initial transaction commits.

More New Features in JPA 2.0

In addition to the enhancements and new features described in the previous sections, JPA 2.0 can use Bean Validation to automatically validate an entity when it is persisted, updated, or removed. What this means is that you can specify a constraint on an entity, for example, that the maximum length of a field in the entity is 15, and have the field automatically validated when the entity is persisted, updated, or removed. You use the `<validation-mode>` element in the `persistence.xml` configuration file to specify whether automatic lifecycle event validation is in effect.

For more information about all the features in JPA 2.0, see JSR 317: Java Persistence 2.0.

Further Ease of Development

You've seen how new technologies such as CDI and Bean Validation, as well as support for features such as web fragments, Facelets, the no-interface view, and the Criteria API make it even easier to develop enterprise or web applications. However, additional usability improvements have been made in many areas of the Java EE 6 platform. In particular, annotations can now be used in more types of Java EE components. In addition, the set of annotations used for dependency injection has been standardized, making injectable classes much more portable across frameworks.

Annotations can now be used in more types of Java EE components. And the set of annotations used for dependency injection has been standardized.

Annotations in More Types of Java EE Components

The simpler annotation-based programming model that was introduced in Java EE 5 has been extended to other types of Java EE components, such as servlets and JSF components. For example, instead of using a deployment descriptor to define a servlet in a web application, all you need to do is mark a class with the `@WebServlet` annotation, as shown below:

```
@WebServlet(name="CalculatorServlet", urlPatterns={"/calc", "/getVal"})
public class CalculatorServlet extends HttpServlet{
   public void doGet(HttpServletRequest req, HttpServletResponse res) {
   ...
   }

   ...
}
```

The `@WebServlet` annotation is one of the annotations provided by Servlet 3.0 technology. Here are some other Servlet 3.0 annotations:

Instead of creating deployment descriptors, you can annotate classes to specify servlet-related deployment information.

- `@WebFilter`. Defines a servlet filter in a web application
- `@WebInitParam`. Specifies any `init` parameters that must be passed to a servlet or servlet filter
- `@WebListener`. Annotates a listener to get events for various operations on a particular web application context
- `@MultipartConfig`. When specified on a servlet, indicates that the request the servlet expects is of the MIME type `multipart/*`

A good example of the annotation support in JSF 2.0 is the simplified approach to configuring managed beans. Instead of registering a managed bean by configuring it in the JSF configuration file, `faces-config.xml`, all you need to do is mark the bean with the `@ManagedBean` annotation and set its scope with the `RequestScope` annotation, as shown below:

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean(name="userBean")
@RequestScoped
public class UserBean {

   private String name;

   public String getName() {
     return name;
   }
   public void setName(String name) {
     this.name = name;
   }
```

```
        public UserBean() {}
    }
```

Some other JSF 2.0 annotations are:

JSF 2.0 supports various annotations to specify configuration information.

- `@ManagedProperty`. Marks a bean property to be a managed property
- `@ResourceDependency`. Declares the resources that a component will need
- `@ListenFor`. Enables a component to subscribe to particular events with the component as the listener
- `@FacesConverter`. Registers a class as a `Converter`, that is, a class that can perform `Object`-to-`String` and `String`-to-`Object` conversions
- `@FacesValidator`. Registers a class as a `Validator`, that is, a class that can perform validation

If you want the annotations to be processed — whether they are Servlet 3.0 annotations or JSF 2.0 annotations — you need to put the classes that are marked with these annotations in the `WEB-INF/classes` directory of the web application. You can also put the classes in a JAR file in the `WEB-INF/lib` directory of the application.

As is the case for web fragments, you use the `<metadata-complete>` element in the `web.xml` file to instruct the web container whether to look for annotations. If you set `<metadata-complete>` to false or do not specify the `<metadata-complete>` element in your file, then during deployment, the container must scan annotations as well as web fragments to build the effective metadata for the web application. However, if you set `<metadata-complete>` to true, the deployment descriptors provide all the configuration information for the web application. In this case, the web container does not search for annotations and web fragments.

With its support for annotations as well as its new `ServletContext` methods, Servlet 3.0 makes the `web.xml` file optional. In other words, you no longer need to include a `web.xml` file in a WAR file for a web application.

Standardized Annotations for Dependency Injection

Dependency injection is a popular technique in developing enterprise Java applications. In dependency injection, also called inversion of control, a component specifies the resources that it depends on. An injector,

typically a container, provides the resources to the component. Although dependency injection can be implemented in various ways, many developers implement it with annotations. Dependency injection is used heavily in Java development frameworks such as Spring and Guice. Unfortunately, there is no standard approach for annotation-based dependency-injection. In particular, a framework such as Spring takes a different approach to annotation-based dependency injection than that of a framework such as Guice.

However, JSR 330: Dependency Injection for Java, which is part of Java EE 6, changes that. The objective of this specification is to provide a standardized and extensible API for dependency injection.

SR 330, Dependency Injection for Java, provides a standardized and extensible API for dependency injection. You no longer have to work with vendor-specific annotations.

The API comprises a set of annotations for use on injectable classes. The annotations are as follows:

- `@Inject`. Identifies injectable constructors, methods, and fields.
- `@Qualifier`. Identifies qualifier annotations. Qualifiers are strongly-typed keys that help distinguish different uses of objects of the same type. For example, a `@Red Car` and a `@Blue Car` are understood to be different instances of the same type. In this example, `@Red` and `@Blue` are qualifiers.
- `@Scope`. Identifies scope annotations.
- `@Singleton`. Identifies a type that the injector only instantiates once
- `@Named`. A `String`-based qualifier.

For example, the following class named `Stopwatch` uses the `@Inject` annotation to inject a dependency on a class named `TimeSource`:

```
class Stopwatch {
   final TimeSource timeSource;
   @Inject Stopwatch(TimeSource TimeSource) {
     this.TimeSource = TimeSource;
   }
   void start() { ... }
   long stop() { ... }
}
```

The dependency injection can be extended to other dependencies. For example, suppose, you want to create a `StopwatchWidget` class that has a dependency on the `Stopwatch` class. You could define the class as

follows:

```
class StopwatchWidget {
   @Inject StopwatchWidget(Stopwatch sw) { ... }
   ...
}
```

In response, the injector finds a `TimeSource` object, uses the `TimeSource` object to construct a `Stopwatch` object, and then constructs a `StopwatchWidget` object with the `Stopwatch` object. .

The standardized set of annotations specified by JSR 330 should make injectable classes portable across frameworks. You no longer have to work with vendor-specific annotations.

Note that CDI builds on JSR-330 and adds much functionality to dependency injection. This functionality includes automatic discovery and configuration of injectable classes, and an API to define new injectable classes at runtime — for example, to help integrate with third-party frameworks.

Profiles and Pruning

Java EE 6 introduces the concept of profiles as a way to reduce the size of the Java EE platform and better target it for specific audiences. Profiles are configurations of the Java EE platform that are designed for specific classes of applications. A profile may include a subset of Java EE platform technologies, additional technologies that have gone through the Java Community Process, but that are not part of the Java EE platform, or both. For example, consider a hypothetical profile for telephony applications. Such a profile might include Java EE web tier technologies, such as JSP and Servlet, EJB for the enterprise component model, and JPA for persistence. It would likely also include telephony-oriented technologies such as JSR 289: SIP Servlet v1.1, that have gone through the JCP process, but that are not part of the Java EE platform.

> Profiles are Java EE platform configurations that are designed for specific classes of applications.

A profile is defined by following the JCP Community Process. In addition, the Java EE 6 Specification defines the rules for referencing Java EE platform technologies in Java EE Profiles. However, the Java EE 6 specification also underscores the principle that a new profile should be created only when there is a good reason for doing so. The specification states "The decision to create a profile should take into account its potential drawbacks, especially in terms of fragmentation and developer confusion. In general, a profile should

be created only when there is a natural developer constituency and a well-understood class of applications that can benefit from it."

Profiles are intended to evolve independently of each other and independently of the Java EE 6 platform. In particular, profiles are initiated by submitting a Java Specification Request and are released on their own schedule, independently of any concurrent revision of the Java EE platform or of other profiles. This means that a profile such as the hypothetical telephony profile can evolve at a pace that is natural for its targeted industry, without being tied to the evolution of the Java EE platform or any other profile. Note however that it is highly recommended that profiles periodically synchronize with the platform, in particular when a major new platform is released. The objective is to preserve a common programming model and simplify the transfer of developer skills across the entire Java EE 6 family of products.

Web Profile

Java EE 6 defines the first profile, called the Web Profile. This initial profile provides a subset of the Java EE platform and is designed for web application development. The Web Profile includes only those technologies needed by most web application developers, and does not include the enterprise technologies that these developers typically don't need.

Java EE 6 defines the first profile, called the Web Profile. This initial profile is a Java EE platform subset for web application development.

Table 1 lists the technologies in the full Java EE 6 platform and indicates by checkmark (✓) which of those are in the Web Profile.

Table 1: Comparing the Technologies in the Java EE Platform and the Web Profile

Java EE Platform Technology	Web Profile
Web Application Technologies	
JSR 315: Java Servlet 3.0	✓
JSR 314: JavaServer Faces (JSF) 2.0	✓
JSR 245: JavaServer Pages 2.2 and Expression Language (EL) 1.2	✓
JSR 52: A Standard Tag Library for JavaServer Pages 1.2	✓
JSR-45: Debugging Support for Other Languages 1.0	✓

Enterprise Application Technologies	
JSR 299: Contexts and Dependency Injection for the Java EE Platform 1.0	✓
JSR 330: Dependency Injection for Java	✓
JSR 318: Enterprise JavaBeans 3.1	✓(EJB Lite)
JSR 317: Java Persistence API 2.0	✓
JSR 250: Common Annotations for the Java Platform 1.1	✓
JSR 907: Java Transaction API (JTA) 1.1	✓
JSR 303: Bean Validation 1.0	✓
JSR 322: Java EE Connector Architecture 1.6	
JSR 914: Java Message Service (JMS) API 1.1	
JSR 919: JavaMail 1.4	
Web Services Technologies	
JSR 311: JAX-RS: The Java API for RESTful Web Services 1.1	
JSR 109: Implementing Enterprise Web Services 1.3	
JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.2	
JSR 222: Java Architecture for XML Binding (JAXB) 2.2	
JSR 181: Web Services Metadata for the Java Platform	
JSR 101: Java APIs for XML based RPC 1.1	
JSR 67: Java APIs for XML Messaging 1.3	
JSR 93: Java API for XML Registries 1.0 (JAXR) 1.0	
Management and Security Technologies	

JSR 196: Java Authentication Service Provider Interface for Containers 1.0	
JSR 115: Java Authorization Contract for Containers 1.3	
JSR 88: Java EE Application Deployment 1.2	
JSR 77: J2EE Management 1.1	

Notice that the Web Profile includes a servlet container and all the traditional presentation technologies such as JSP, JSF, and the Standard Tag Library for JavaServer Pages (informally referred to as JSTL). EJB 3.1 Lite is available as a component model. There is also JPA for persistence and JTA for transaction management. And with the enhanced extensibility enabled by features such as web fragments, you can easily extend the Web Profile with additional frameworks or libraries such as JAX-RS.

Pruning

Another technique introduced in Java EE 6 for reducing the size of the platform is pruning. Pruning a technology means that the technology can become an optional component in the next release of the platform rather than a required component. Community reaction ultimately decides whether the candidate actually becomes an optional component. Pruning can reduce the size of Java EE platform products because implementors such as Java EE application server vendors may include or exclude a pruned technology in their implementation. However, if Java EE application server vendors do include a pruned technology, they must do so in a compatible way, such that existing applications will keep running.

Another technique introduced in Java EE 6 for reducing the size of the platform is pruning.

These are candidates for pruning:

- JSR 101: Java APIs for XML-Based RPC
- JSR 93: Java API for XML Registries 1.0 (JAXR)
- EJB Entity Beans (defined as part of JSR 153: Enterprise JavaBeans 2.0, and earlier)
- JSR 88: Java EE Application Deployment

Summary

With its support for profiles, significant new technologies such as JAX-RS, enhanced extensibility features such as web fragments, and ease of development improvements such as Facelets and platform-wide adoption of

annotations, Java EE 6 delivers a Java EE platform that is more flexible, more powerful, and more developer friendly than ever before. You can try out an implementation of the Java EE 6 platform by downloading the Java EE 6 SDK. For a more in-depth understanding of the Java EE 6 platform, see the Java EE 6 Tutorial.

Part 1 | Part 2 | **Part 3**

For More Information

- JSR 316: Java Platform, Enterprise Edition 6 (Java EE 6) Specification
- Java EE 6 Technologies
- Java EE 6 SDK
- Java EE 6 Tutorial

* As used on this web site, the terms "Java Virtual Machine" and "JVM" mean a virtual machine for the Java platform.

Rate This Article

Comments

We welcome your participation in our community. Please keep your comments civil and on point. You may optionally provide your email address to be notified of replies - your information is not used for any other purpose. By submitting a comment, you agree to these Terms of Use.

About Sun | About This Site | Newsletters | Contact Us | Employment | How to Buy | Licensing | Terms of Use | Privacy | Trademarks

Copyright Sun Microsystems, Inc.

A Sun Developer Network Site

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this License.