Resource Connections

Java EE components can access a wide variety of resources, including databases, mail sessions, Java Message Service objects, and URLs.

The Java EE 6 platform provides mechanisms that allow you to access all these resources in a similar manner.

This chapter explains how to get connections to several types of resources. The following topics are addressed here:

- Resources and JNDI Naming
- `DataSource` Objects and Connection Pools
- Resource Injection
- Resource Adapters and Contracts
- Metadata Annotations
- Common Client Interface
- Further Information about Resources

Resources and JNDI Naming

In a distributed application, components need to access other components and resources, such as databases.

For example, a servlet might invoke remote methods on an enterprise bean that retrieves information from a database.

In the Java EE platform, the Java Naming and Directory Interface (JNDI) naming service enables components to locate other components and resources.

A resource is a program object that provides connections to systems, such as database servers and messaging systems.

(A Java Database Connectivity resource is sometimes referred to as a data source.)

Each resource object is identified by a unique, people-friendly name, called the JNDI name.

For example, the JNDI name of the JDBC resource for the Java DB database that is shipped with the GlassFish Server is `jdbc/__default`.

An administrator creates resources in a JNDI namespace.

In the GlassFish Server, you can use either the Administration Console or the `asadmin` command to create resources.

Applications then use annotations to inject the resources.

If an application uses resource injection, the GlassFish Server invokes the JNDI API, and the application is not required to do so.

However, it is also possible for an application to locate resources by making direct calls to the JNDI API.

A resource object and its JNDI name are bound together by the naming and directory service.

To create a new resource, a new name/object binding is entered into the JNDI namespace.

You inject resources by using the `@Resource` annotation in an application.

You can use a deployment descriptor to override the resource mapping that you specify in an annotation.

Using a deployment descriptor allows you to change an application by repackaging it rather than by both recompiling the source files and repackaging.

However, for most applications, a deployment descriptor is not necessary.

`DataSource` Objects and Connection Pools

To store, organize, and retrieve data, most applications use a relational database.

Java EE 6 components may access relational databases through the JDBC API.

For information on this API, see http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html.

In the JDBC API, databases are accessed by using `DataSource` objects.

A `DataSource` has a set of properties that identify and describe the real-world data source that it represents.

These properties include such information as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on.

In the GlassFish Server, a data source is called a JDBC resource.

Applications access a data source by using a connection, and a `DataSource` object can be thought of as a factory for connections to the particular data source that the `DataSource` instance represents.

In a basic `DataSource` implementation, a call to the `getConnection` method returns a connection object that is a physical connection to the data source.

A `DataSource` object may be registered with a JNDI naming service.

If so, an application can use the JNDI API to access that `DataSource` object, which can then be used to connect to the data source it represents.

`DataSource` objects that implement connection pooling also produce a connection to the particular data source that the `DataSource` class represents.

The connection object that the `getConnection` method returns is a handle to a `PooledConnection` object rather than being a physical connection.

An application uses the connection object in the same way that it uses a connection.

Connection pooling has no effect on application code except that a pooled connection, like all connections, should always be explicitly closed.

When an application closes a connection that is pooled, the connection is returned to a pool of reusable connections.

The next time `getConnection` is called, a handle to one of these pooled connections will be returned if one is available.

Because connection pooling avoids creating a new physical connection every time one is requested, applications can run significantly faster.

A JDBC connection pool is a group of reusable connections for a particular database.

Because creating each new physical connection is time consuming, the server maintains a pool of available connections to increase performance.

When it requests a connection, an application obtains one from the pool.

When an application closes a connection, the connection is returned to the pool.

Applications that use the Persistence API specify the `DataSource` object they are using in the `jta-data-source` element of the `persistence.xml` file:

```
<jta-data-source>
jdbc/MyOrderDB
</jta-data-source>
```

This is typically the only reference to a JDBC object for a persistence unit.

The application code does not refer to any JDBC objects.

Resource Injection

The `javax.annotation.Resource` annotation is used to declare a reference to a resource; `@Resource` can decorate a class, a field, or a method.

The container will inject the resource referred to by `@Resource` into the component either at runtime or when the component is initialized, depending on whether field/method injection or class injection is used.

With field-based and method-based injection, the container will inject the resource when the application is initialized.

For class-based injection, the resource is looked up by the application at runtime.

The `@Resource` annotation has the following elements:

- `name`: The JNDI name of the resource
- `type`: The Java language type of the resource
- `authenticationType`: The authentication type to use for the resource

- `shareable`: Indicates whether the resource can be shared
- `mappedName`: A nonportable, implementation-specific name to which the resource should be mapped
- `description`: The description of the resource

The `name` element is the JNDI name of the resource and is optional for field-based and method-based injection.

For field-based injection, the default `name` is the field name qualified by the class name.

For method-based injection, the default `name` is the JavaBeans property name, based on the method qualified by the class name.

The `name` element must be specified for class-based injection.

The type of resource is determined by one of the following:

- The type of the field the `@Resource` annotation is decorating for field-based injection
- The type of the JavaBeans property the `@Resource` annotation is decorating for method-based injection
- The `type` element of `@Resource`

For class-based injection, the `type` element is required.

The `authenticationType` element is used only for connection factory resources, such as the resources of a connector, also called the resource adapter, or data source.

This element can be set to one of the `javax.annotation.Resource.AuthenticationType` enumerated type values: `CONTAINER`, the default, and `APPLICATION`.

The `shareable` element is used only for Object Resource Broker (ORB) instance resources or connection factory resource.

This element indicates whether the resource can be shared between this component and other components and may be set to `true`, the default, or `false`.

The `mappedName` element is a nonportable, implementation-specific name to which the resource should be mapped.

Because the `name` element, when specified or defaulted, is local only to the application, many Java EE servers provide a way of referring to resources across the application server.

This is done by setting the `mappedName` element.

Use of the `mappedName` element is nonportable across Java EE server implementations.

The `description` element is the description of the resource, typically in the default language of the system on which the application is deployed.

This element is used to help identify resources and to help application developers choose the correct resource.

Field-Based Injection

To use field-based resource injection, declare a field and decorate it with the `@Resource` annotation.

The container will infer the name and type of the resource if the `name` and `type` elements are not specified.

If you do specify the `type` element, it must match the field's `type` declaration.

In the following code, the container infers the `name` of the resource, based on the class name and the field name:

`com.example.SomeClass/myDB.`

The inferred `type` is

`javax.sql.DataSource.class:`

```
package com.example;
```

```
public class SomeClass {
@Resource
private javax.sql.DataSource myDB;
...
}
```

In the following code, the JNDI name is `customerDB`, and the inferred `type` is `javax.sql.DataSource.class`:

```
package com.example;
public class SomeClass {
@Resource(name="customerDB")
private javax.sql.DataSource myDB;
...
}
```

Method-Based Injection

To use method-based injection, declare a setter method and decorate it with the `@Resource` annotation.

The container will infer the name and type of the resource if the `name` and `type` elements are not specified.

The setter method must follow the JavaBeans conventions for property names: The method name must begin with `set`, have a `void` return type, and only one parameter.

If you do specify the `type` element, it must match the field's type declaration.

In the following code, the container infers the `name` of the resource based on the class name and the field name: `com.example.SomeClass/myDB`.

The inferred `type` is `javax.sql.DataSource.class`:

```
package com.example;
public class SomeClass {
```

```
private javax.sql.DataSource myDB;
...
@Resource
private void
setMyDB(javax.sql.DataSource ds)
{ myDB = ds; }
...
}
```

In the following code, the JNDI name is `customerDB`, and the inferred `type` is `javax.sql.DataSource.class`:

```
package com.example;
public class SomeClass {
private javax.sql.DataSource myDB;
...
@Resource(name="customerDB")
private void
setMyDB(javax.sql.DataSource ds)
```

```
{ myDB = ds; }
...
}
```

Class-Based Injection

To use class-based injection, decorate the class with a `@Resource` annotation, and set the required `name` and `type` elements:

```
@Resource(name="myMessageQueue",
type="javax.jms.ConnectionFactory")
public class SomeMessageBean {...}
```

The `@Resources` annotation is used to group together multiple `@Resource` declarations for class-based injection.

The following code shows the `@Resources` annotation containing two `@Resource` declarations.

One is a Java Message Service message queue, and the other is a JavaMail session:

```
@Resources({
@Resource(name="myMessageQueue",
```

```
type="javax.jms.ConnectionFactory")
,
@Resource(name="myMailSession",
type="javax.mail.Session")
})
public class SomeMessageBean
{...}
```

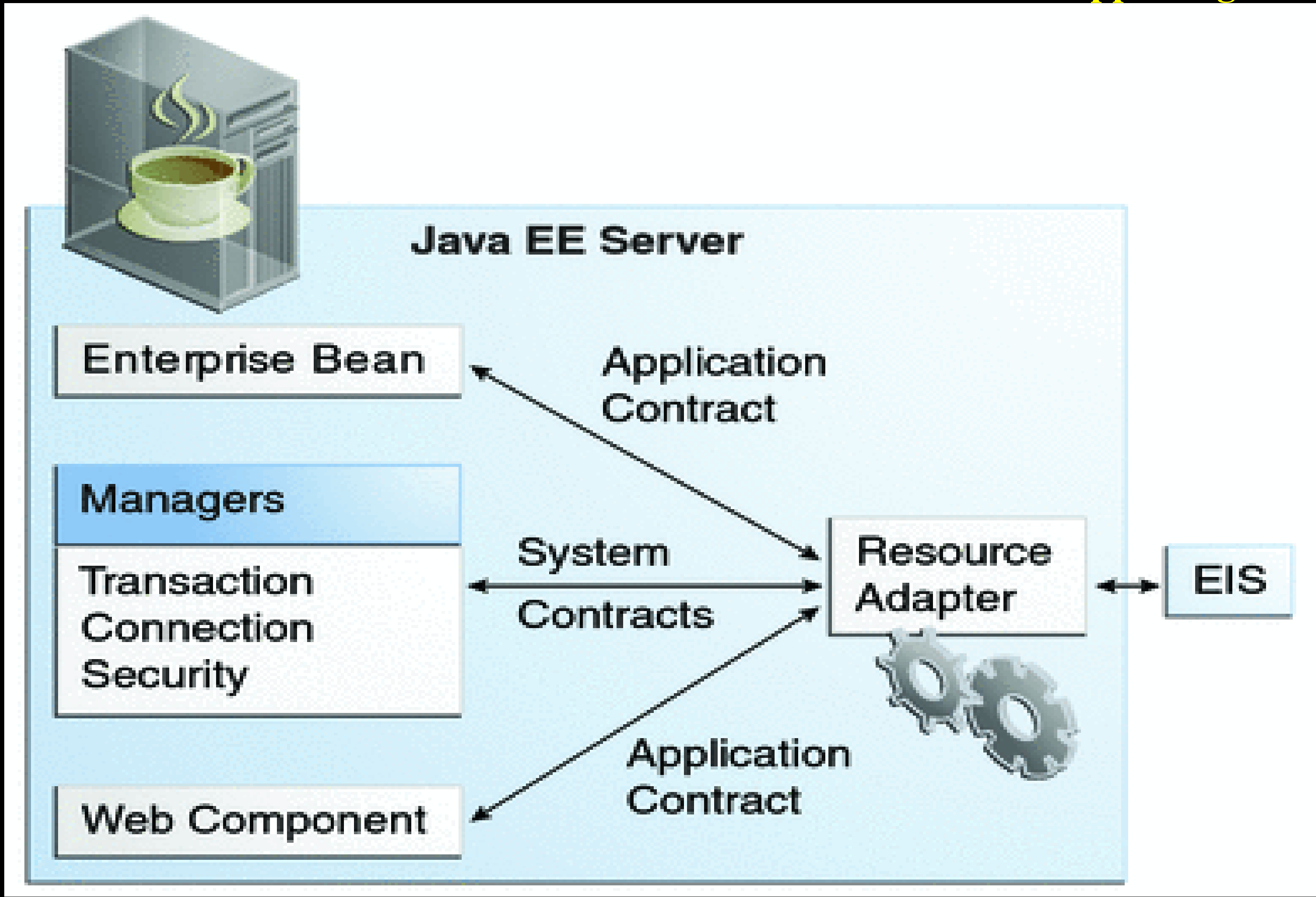
Resource Adapters and Contracts

A resource adapter is a Java EE component that implements the Java EE Connector Architecture for a specific EIS.

Examples of EISs include enterprise resource planning, mainframe transaction processing, and database systems.

As illustrated in Figure 44-1, the resource adapter facilitates communication between a Java EE application and an EIS.

Figure 44-1 Resource Adapters


Java EE Server
Enterprise Bean
Application Contract
Managers
Transaction
Connection
Security
System
Contracts
Resource Adapter
EIS
Application Contract
Web Component

Stored in a Resource Adapter Archive (RAR) file, a resource adapter can be deployed on any Java EE server, much like a Java EE application.

A RAR file may be contained in an Enterprise Archive (EAR) file, or it may exist as a separate file.

A resource adapter is analogous to a JDBC driver.

Both provide a standard API through which an application can access a resource that is outside the Java EE server.

For a resource adapter, the target system is an EIS; for a JDBC driver, it is a DBMS.

Resource adapters and JDBC drivers are rarely created by application developers.

In most cases, both types of software are built by vendors that sell tools, servers, or integration software.

The resource adapter mediates communication between the Java EE server and the EIS by means of contracts.

The application contract defines the API through which a Java EE component, such as an enterprise bean, accesses the EIS.

This API is the only view that the component has of the EIS.

The system contracts link the resource adapter to important services that are managed by the Java EE server.

The resource adapter itself and its system contracts are transparent to the Java EE component.

Management Contracts

The Java EE Connector Architecture defines system contracts that enable resource adapter lifecycle and thread management.

Lifecycle Management

The Connector Architecture specifies a lifecycle management contract that allows an application server to manage the lifecycle of a resource adapter.

This contract provides a mechanism for the application server to bootstrap a resource adapter instance during the deployment or application server startup.

This contract also provides a means for the application server to notify the resource adapter instance when it is undeployed or when an orderly shutdown of the application server takes place.

Work Management Contract

The Connector Architecture work management contract ensures that resource adapters use threads in the proper, recommended manner.

This contract also enables an application server to manage threads for resource adapters.

Resource adapters that improperly use threads can jeopardize the entire application server environment.

For example, a resource adapter might create too many threads or might not properly release threads it has created.

Poor thread handling inhibits application server shutdown and impacts the application server's performance because creating and destroying threads are expensive operations.

The work management contract establishes a means for the application server to pool and reuse threads, similar to pooling and reusing connections.

By adhering to this contract, the resource adapter does not have to manage threads itself.

Instead, the resource adapter has the application server create and provide needed threads.

When it is finished with a given thread, the resource adapter returns the thread to the application server.

The application server manages the thread, either returning it to a pool for later reuse or destroying it.

Handling threads in this manner results in increased application server performance and more efficient use of resources.

In addition to moving thread management to the application server, the Connector Architecture provides a flexible model for a resource adapter that uses threads.

- The requesting thread can choose to block (stop its own execution) until the work thread completes.

- The requesting thread can block while it waits to get the work thread.

When the application server provides a work thread, the requesting thread and the work thread execute in parallel.

- The resource adapter can opt to submit the work for the thread to a queue.

The thread executes the work from the queue at some later point.

The resource adapter continues its own execution from the point it submitted the work to the queue, no matter when the thread executes it.

With the latter two approaches, the submitting thread and the work thread may execute simultaneously or independently.

For these approaches, the contract specifies a listener mechanism to notify the resource adapter that the thread has completed its operation.

The resource adapter can also specify the execution context for the thread, and the work management contract controls the context in which the thread executes.

Generic Work Context Contract

The work management contract between the application server and a resource adapter enables a resource adapter to do a task, such as communicating with the EIS or delivering messages, by delivering `Work` instances for execution.

A generic work context contract enables a resource adapter to control the contexts in which the `Work` instances that it submits are executed by the application server's `WorkManager`.

A generic work context mechanism also enables an application server to support new message inflow and delivery schemes.

It also provides a richer contextual `Work` execution environment to the resource adapter while still maintaining control over concurrent behavior in a managed environment.

The generic work context contract standardizes the transaction context and the security context.

Outbound and Inbound Contracts

The Connector Architecture defines the following outbound contracts, system-level contracts between an application server and an EIS that enable outbound connectivity to an EIS.

- **The connection management contract supports connection pooling, a technique that enhances application performance and scalability.**

Connection pooling is transparent to the application, which simply obtains a connection to the EIS.

- The transaction management contract extends the connection management contract and provides support for management of both local and XA transactions.

A local transaction is limited in scope to a single EIS system, and the EIS resource manager itself manages such transaction.

An XA transaction or global transaction can span multiple resource managers.

This form of transaction requires transaction coordination by an external transaction manager, typically bundled with an application server.

A transaction manager uses a two-phase commit protocol to manage a transaction that spans multiple resource managers or EISs, and uses one-phase commit optimization if only one resource manager is participating in an XA transaction.

- The security management contract provides mechanisms for authentication, authorization, and secure communication between a Java EE server and an EIS to protect the information in the EIS.

A work security map matches EIS identities to the application server domain's identities.

Inbound contracts are system contracts between a Java EE server and an EIS that enable inbound connectivity from the EIS: pluggability contracts for message providers and contracts for importing transactions.

Metadata Annotations

Java EE Connector Architecture 1.6 introduces a set of annotations to minimize the need for deployment descriptors.

- The `@Connector` annotation can be used by the resource adapter developer to specify that the JavaBeans component is a resource adapter JavaBeans component.

This annotation is used for providing metadata about the capabilities of the resource adapter.

Optionally, you can provide a JavaBeans component implementing the `ResourceAdapter` interface, as in the following example:

```
@Connector(
description =
"Sample adapter using the JavaMail API",
displayName =
"InboundResourceAdapter",
vendorName = "My Company, Inc.",
```

```
eisType = "MAIL",
version = "1.0"
)
public class ResourceAdapterImpl
implements ResourceAdapter,
java.io.Serializable {...}
```

- The `@ConnectionDefinition` annotation defines a set of connection interfaces and classes pertaining to a particular connection type, as in the following example:

```
@ConnectionDefinition(
connectionFactory =
samples.mailra..api.
JavaMailConnectionFactory.class,
connectionFactoryImpl =
samples.mailra.ra.outbound.
JavaMailConnectionFactoryImpl.class
,
connection =
samples.connectors.mailconnector.
api.JavaMailConnection.class,
```

```
connectionImpl =
samples.mailra..ra.outbound.
JavaMailConnectionImpl.class
)
public class
ManagedConnectionFactoryImpl
implements
ManagedConnectionFactory,
Serializable {... ...
@ConfigProperty
(defaultValue = "UnknownHostName")
```

```
public void setServerName
 (String serverName) {...}
}
```

- The `@AdministeredObject` annotation designates a JavaBeans component as an administered object.

- The `@Activation` annotation contains configuration information pertaining to inbound connectivity from an EIS instance, as in the following example:

```
@Activation(
messageListeners = {
samples.mailra.api.
JavaMailMessageListener.class
} )
```

```
public class ActivationSpecImpl
implements
javax.resource.spi.ActivationSpec,
java.io.Serializable {...
@ConfigProperty()
// serverName property value
private String serverName =
new String("");
@ConfigProperty()
// userName property value
```

```
private String userName =
new String("");
@ConfigProperty()
// password property value
private String password =
new String("");
@ConfigProperty()
// folderName property value
private String folderName =
new String("Inbox");
// protocol property value
```

```
// Normally imap or pop3
@ConfigProperty(
description =
"Normally imap or pop3"
)
private String protocol =
new String("imap");
...
...
}
```

- The `@ConfigProperty` annotation can be used on JavaBeans components to provide additional configuration information that may be used by the deployer and resource adapter provider.

The preceding example code shows several `@ConfigProperty` annotations.

The specification allows a resource adapter to be developed in mixed-mode form, that is the ability for a resource adapter developer to use both metadata annotations and deployment descriptors in applications.

An application assembler or deployer may use the deployment descriptor to override the metadata annotations specified by the resource adapter developer.

The deployment descriptor for a resource adapter is named `ra.xml`.

The `metadata-complete` attribute defines whether the deployment descriptor for the resource adapter module is complete or whether the class files available to the module and packaged with the resource adapter need to be examined for annotations that specify deployment information.

For the complete list of annotations and JavaBeans components introduced in the Java EE 6 platform, see the Java EE Connector Architecture 1.6 specification.

Common Client Interface

This section explains how components use the Connector Architecture Common Client Interface (CCI) API and a resource adapter to access data from an EIS.

The CCI API defines a set of interfaces and classes whose methods allow a client to perform typical data access operations.

The CCI interfaces and classes are as follows:

- `ConnectionFactory`: Provides an application component with a `Connection` instance to an EIS.
- `Connection`: Represents the connection to the underlying EIS.

- `ConnectionSpec`: Provides a means for an application component to pass connection-request-specific properties to the `ConnectionFactory` when making a connection request.

- `Interaction`: Provides a means for an application component to execute EIS functions, such as database stored procedures.

- `InteractionSpec`: Holds properties pertaining to an application component's interaction with an EIS.

- `Record`: The superinterface for the various kinds of record instances.

Record instances can be `MappedRecord`, `IndexedRecord`, or `ResultSet` instances, all of which inherit from the `Record` interface.

- `RecordFactory`: Provides an application component with a `Record` instance.

- `IndexedRecord`: Represents an ordered collection of `Record` instances based on the `java.util.List` interface.

A client or application component that uses the CCI to interact with an underlying EIS does so in a prescribed manner.

The component must establish a connection to the EIS's resource manager, and it does so using the `ConnectionFactory`.

The `Connection` object represents the connection to the EIS and is used for subsequent interactions with the EIS.

The component performs its interactions with the EIS, such as accessing data from a specific table, using an `Interaction` object.

The application component defines the Interaction object by using an `InteractionSpec` object.

When it reads data from the EIS, such as from database tables, or writes to those tables, the application component does so by using a particular type of `Record` instance: a `MappedRecord`, an `IndexedRecord`, or a `ResultSet` instance.

Note, too, that a client application that relies on a CCI resource adapter is very much like any other Java EE client that uses enterprise bean methods.

Further Information about Resources

For more information about resources and annotations, see

- Java EE 6 Platform Specification (JSR 316):

  http://jcp.org/en/jsr/detail?id=316

- Java EE Connector Architecture 1.6 specification:

  http://jcp.org/en/jsr/detail?id=322

- **EJB 3.1 specification:**
  http://jcp.org/en/jsr/detail?id=318
- **Common Annotations for the Java Platform:**
  http://www.jcp.org/en/jsr/detail?id=250