

You are here: [Home](#) > [Developers](#) > [View Source Articles](#) > Java View Source Article

Developer Program
[Membership](#)
[One-to-One Support](#)
[Newsgroups](#)

Developer Publications
[View Source](#)
[Developer News](#)

Documentation
[Technical Manuals](#)
[White Papers](#)
[TechNotes](#)
[Sample Code](#)
[FAQs](#)
[Books](#)

Technologies
[CORBA](#)
[Directory & LDAP](#)
[Dynamic HTML](#)
[Java](#)
[JavaScript](#)
[Linux](#)
[Security](#)
[SSJS](#)
[XML](#)

Developer Downloads
[Tools & SDKs](#)
[Patches](#)

iPlanet Products
[Technical Resources](#)

Looking Into Enterprise JavaBeans

By [Duane K. Fields](#)

[Send comments and questions](#) about this article to [View Source](#).

The introduction of the Enterprise JavaBeans (EJB) specification by Sun Microsystems and its adoption by major application server companies like Netscape promises to ease and speed the development of mission-critical applications. Enterprise JavaBeans are reusable business logic components for use in distributed, multitier application architectures. You can get up and running quickly by building applications around the growing number of EJB components, which provide functionality that traditionally has represented the biggest challenge to creating web-based applications.

For example, if you were developing a high-end e-commerce application, you might purchase one EJB component that performed real-time credit card approval, another that would manage a customer database, and another that calculated shipping costs. You would then tie these together within your application server by customizing the runtime properties of the beans, and there you would have it - an order processing system. The application server would automatically handle sticky issues like balancing loads, maintaining security, monitoring transaction processes, sharing resources, ensuring data integrity, and so on.

This article will introduce you to Enterprise JavaBeans, give you an overview of how applications are designed around EJB components, and show you how to create and access a simple EJB component. Enterprise JavaBeans are different from the plain JavaBeans that you may be familiar with, so let's start by looking at those differences.

ENTERPRISE JAVABEANS VS. ORDINARY BEANS

You may be wondering how Enterprise JavaBeans relate to the JavaBeans we've been hearing about for the past couple of years that work with application builders. The two types of beans actually don't have much in common from a technical perspective, even if the philosophy behind them - to enable developers to drop reusable components into their applications - is similar.

"Ordinary" beans are Java classes, typically graphical user interface (GUI) components, designed to conform to a series of programming conventions so that integrated development environments like Symantec Visual Cafe or IBM's VisualAge for Java can inspect the beans and allow you to hook them together into applications. Your development environment can then generate appropriate Java code to work with the beans. For example, a bean might represent a special kind of text field or list box, and the development environment could then ease the code development process by graphically allowing you to configure this bean and call the right methods for the desired functionality.

Enterprise JavaBeans are likewise components, but these beans follow a completely different set of conventions and interfaces and aren't for use inside development environments. The purpose of Enterprise JavaBeans is to encapsulate business logic (for example, the steps involved in depositing money into an account, calculating income tax, or selecting which warehouse to ship an order from) into server-side components. In the EJB paradigm, an application is implemented as a set of business-logic-controlling EJB components that have been configured in application-specific ways inside an "EJB container" such as an application server. Clients are then written to communicate with the EJB components and handle the results. The standardized interfaces exist to allow the EJB container to manage security and transactional aspects of the bean.

APPLICATION SERVERS AND EJB CONTAINERS

Enterprise JavaBeans work in concert with an EJB container, typically integrated into an application server

Enterprise JavaBeans work in concert with an EJB container, typically integrated into an application server such as Netscape Application Server (NAS). EJB containers must support Sun's Enterprise JavaBean specification, which details the interface between beans and other application server elements. Enterprise JavaBeans can be used with any application server or other system providing an EJB container that implements these interfaces. EJB containers can also exist in other systems such as transaction monitors or database systems.

Application servers in particular are excellent environments to host EJB containers because they automate the more complex features of multitier computing. Application servers manage scarce resources on behalf of the components involved in the design. They also provide infrastructure services such as naming, directory services, and security. And they provide bean-based applications with the benefit of scalability - most application server environments will let you scale your application through the addition of new clusters of machines.

EJB containers offer their beans a number of important services. While you may not deal with these services directly since they're generally kept under the covers, Enterprise JavaBeans couldn't function without them. These services include the following:

- **Life cycle management services** - Enable initialization and shutdown of beans.
- **Security services** - Enable beans to work with a wide variety of authentication schemes and approval processes.
- **Transaction services** - Manage such things as rolling back transactions that didn't fully complete and handling final commitment of transactions, plus transactions across multiple databases.
- **Persistence and state management services** - Enable beans to keep information between sessions and individual requests, even if the container's server must be rebooted.

The EJB container also provides a communications channel to and from its beans, and it will handle all of its beans' multithreading issues. In fact, the EJB specification explicitly forbids a bean from creating its own threads. This ensures thread-safe operation and frees the developer from often complicated thread management concerns.

ARCHITECTURE OF APPLICATIONS BUILT AROUND BEANS

Now let's examine how we can build an application around some Enterprise JavaBeans. Because the role of these beans is to define the core business logic of your application, you need some other way to deal with presentation issues like generating web pages to communicate results. If you're building a web-based application around Enterprise JavaBeans, you can break up your architecture into three main areas:

- the business logic (the core business transactions and processing), provided by Enterprise JavaBeans
- the presentation logic (the logical connection between the user and EJB services), provided by Java servlets
- the presentation layout (the actual HTML output), provided by Java Server Pages (JSP)

For example, in a banking application a servlet might use the services of an EJB component to determine whether users are business or consumer customers and direct them to an appropriate JSP-controlled web page to show them their account balance.

In this model, Java servlets control the logic and flow throughout the application. The presentation layout (the HTML) is isolated from the program logic through Java Server Pages. This means you can allow your HTML designers to work independently on the look and feel of your pages and keep your engineers focused on code development. And by sticking to standards-based technologies such as Java Server Pages and servlets, you can ensure portability and scalability of all the aspects of your distributed, multitier application.

TYPES OF BEANS

There are two primary types of Enterprise JavaBeans - session and entity beans.

Session beans provide a service, usually work with objects passed in as parameters, and generally disappear between transactions. Session beans take their name from the fact that they're usually instantiated by the client and exist only for the duration of a single client-server session, where the session bean performs operations on behalf of the client. In our e-commerce example, we might have a session bean that calculates shipping charges based on input parameters like the shipping address and the weight of the order.

There are two types of session beans - stateless (the most common type) and stateful. Stateless beans simply perform a service and don't have any persistent data, whereas stateful beans maintain some data

simply perform a service and don't have any persistent data, whereas stateful beans maintain some data between requests but are generally short lived. Stateful beans are responsible for handling their own data persistence issues.

Entity beans represent persistent objects, which are maintained across requests. An entity bean, for example, might represent a particular inventory item or a purchase order. Each entity bean is uniquely identified. One of the key features of entity beans is that they employ container-managed persistence - that is, it's the job of the EJB container (for example, the application server) to maintain the entity. This frees the developer from the often complex job of maintaining persistence across requests and reboots.

Generally, for designs requiring persistence in beans, entity beans are used because the container will provide for persistence of data. However, the EJB specification doesn't require that application servers support entity beans and container-managed persistence of data. Since they're essentially an optional feature, you'll achieve maximum portability and flexibility of deployment by designing applications primarily around session beans, and stateless session beans at that. Until server-managed persistence of entity beans becomes commonplace, you're stuck implementing persistent objects yourself.

You'll also make your application easier to scale if you avoid using persistent objects like stateful session beans and entity beans. If you deploy stateful session beans in a clustered environment, for example, you have to worry about the fact that you must maintain access to beans between requests even if subsequent requests are being handled by different servers from the cluster. Thus, if you had an entity bean representing an invoice that stayed around throughout requests and you deployed that bean as part of an application built on a clustered environment, you would be forced either to make sure that all subsequent user requests were confined to the original server (thus reducing the benefits of a clustered environment) or to manually hand off the information between servers during each request.

CREATING A SIMPLE EJB COMPONENT

Now let's look at the procedure for creating a simple EJB component. In this example, we'll develop a shipping bean that could be used in any application needing to ship packages. Such a bean might be part of an e-commerce package or provided by a shipping agent as a convenience to its customers. This shipping bean will perform services such as shipping the package, calculating charges, and tracking the package. While we won't actually code the entire bean in this example, we'll do enough to give you a feel for what beans look like, how you might interface with them, and how they interact with application servers and client applications.

Because stateless session beans offer the most flexibility and portability, we'll implement our shipping bean as this type.

Creating the Bean Interfaces

When you create a bean, you must first create the home and remote interface classes. These form the contract that client applications will use to access the services of the bean. The interface classes only specify the available services and methods; they don't implement them. Both classes are expressed in RMI syntax, Sun's scheme for remote method invocation. RMI programmers, noting that EJB calling conventions look very similar, may wonder how Enterprise JavaBeans differ from solutions based on RMI. The major difference is that with RMI, the programmer is responsible for handling transaction and security issues, while these issues are handled for Enterprise JavaBeans by their container.

The *remote interface* is how the bean is exposed to clients that want to interact with it. It provides for a set of services that will ultimately be implemented by the container the bean is deployed into and involves very little coding, as shown in Listing 1. You will, however, need to create equivalent methods inside the main JavaBean class, which we'll see later. The remote interface extends `javax.ejb.EJBObject` and `javax.ejb.Remote`.

Listing 1. Creating the remote interface class

```
package commerce.shipping;

import javax.ejb.*;
import java.rmi.RemoteException;

public interface Shipper extends EJBObject, java.rmi.Remote {
    public String getTrackingNum() throws RemoteException;
    public dollarAmount calcCharge(ZipCode origZip, ZipCode destZip)
        throws Remote Exception;
}
```

```
}
```

The *home interface* (also known as the local interface), as shown in Listing 2, provides a set of methods that allow clients to control the life cycle of beans - in other words, the creation and destruction of beans by the container. It extends the `EJBHome` interface and must implement at least one `create()` method, which will be called by the container to invoke the bean into existence. We can overload the `create()` method to allow several different possible initializations. In this case, we'll include an alternate `create()` method that lets us create a shipper bean with a "priority level" (next day, rush, and so on). If we were creating an entity bean, we would have to define a `finder()` method as well, which would be used by the container to locate specific instances of the entity bean.

Listing 2. Creating the home interface class

```
package commerce.shipping;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.rmi.Remote;

public interface ShipperHome extends EJBHome {
    public Shipper create() throws CreateException, RemoteException;
    public Shipper create(int priority) throws CreateException, RemoteException;
}
```

Writing the Enterprise JavaBeans Class

The home and remote interfaces that we just created have very little code and even less functionality. The actual bean class is where the business logic resides. We implement the interface of the type of bean we're creating - for example, a session bean. We then implement the methods defined in our remote interface and any methods specified by the bean type we're implementing, as Listing 3 shows.

Listing 3. Creating the JavaBeans class

```
package commerce.shipping;

import javax.ejb.*;
import java.rmi.*;

public interface ShipperBean implements SessionBean {
    private transient SessionContext context;
    private transient Properties props;
    private int priority;

    // These methods are required by the SessionBean interface and
    // allow the bean to react to life-cycle events such as activation
    // or removal. If you had an advanced bean that needed to manage
    // database connections or other resources, you would open and close
    // them here. Most SessionBeans will use empty implementations.
    public void ejbActivate() { System.err.println("Activated..."); }
    public void ejbRemove() { System.err.println("Removed..."); }
    public void ejbPassivate() { System.err.println("Passivated..."); }

    // These methods are called by the container to associate the
    // bean with its context within the container. Typically, a bean
    // keeps a reference to its context.
    public void setSessionContext(SessionContext context) {
        this.context = context;
        props = context.getEnvironment();
    }
}
```

```

// These methods are analogous to the create() methods we created in our
// home interface and will be called by the container. We therefore need
// to implement the same method signatures as before. Note, however, that
// we don't return an object, as that's handled by the container's
// implementation of the home interface.
public void ejbCreate() {
    System.err.println("New Shipper Bean created");
}

public void ejbCreate(int priority) {
    this.priority = priority;
}

// Now for the "business logic" methods defined in our remote interface.
public String genTrackingNum() throws RemoteException {
    // Insert code to generate the next tracking number.
    return trackingNumber;
}

public dollarAmount calcCharge(ZipCode origZip, ZipCode destZip)
    throws RemoteException {
    // Calculate shipping charges.
    return totalCharges;
}
}

```

Packaging and Deploying the Bean

To deploy an EJB component means to load it into our application server's container. Beans are packaged into Java archive (JAR) files along with a special version of the JAR's manifest file (essentially its table of contents) and a serialized instance of a `DeploymentDescriptor` object, which tells the container how to interact with the bean. The deployment descriptor specifies how the bean is supposed to be used in this particular application. For example, it might dictate who has access to the bean, the specifics of starting and concluding transactions, and such.

While the contents of a bean package are prescribed in the EJB specification, the actual tools that package and build deployable beans are left to the design of the application server developers. Most application server developers will provide tools you can use to build your deployment descriptors.

Accessing the Bean

Clients don't interact directly with beans. Instead, all requests for interaction are handled by wrapper interfaces generated by the container the beans have been deployed into. This shields the client from management services that may be implemented differently from container to container. Access is through the home interface we defined earlier.

Before we can use the home interface to ask the container to create an instance of the bean, we have to acquire a reference to it. This is accomplished using JNDI - the Java Naming and Directory Interface. We access the home interface by placing a lookup call to the server and requesting the class name of the bean. The class name we're looking for was defined in the `DeploymentDescriptor` object we created at deployment time. Once we've acquired this reference, we can ask the server to invoke any of the defined service methods.

I should mention that Enterprise JavaBeans are accessible from CORBA clients as well. The EJB specification will allow CORBA clients to interact with EJB components by presenting them through a CORBA-accessible interface. This provides seamless access between CORBA and Java.

IN SUMMARY

Enterprise JavaBeans are making it possible to build distributed applications with pluggable components from a variety of vendors. They can enable developers to implement solutions that take advantage of previously difficult-to-implement features like distributed transaction management and multithreading, now provided by the EJB container. And the introduction of Enterprise JavaBeans means that we can now tie the outside world of Java technologies - RMI, JDBC, JNDI, and others - into an enterprise computing platform.

the whole family of Java technologies - RMI, JDBC, JNDI - together into an enterprise computing platform well equipped to handle the serious demands of mission-critical computing.

FURTHER RESOURCES

- Anne Thomas, [Enterprise JavaBeans Technology: Server Component Model for the Java Platform](#), Sun Microsystems white paper
- [Enterprise JavaBeans Technical Information](#), Sun Microsystems
- [Deploying and Managing Web-Based Enterprise Applications: The Netscape Application Server Solution](#), Netscape Communications white paper
- Duane Fields, [Java Servlets for JavaScripters: Expanding Your Server-Side Programming Repertoire](#), *View Source*

View Source wants your feedback!
[Write to us](#) and let us know
what you think of this article.

[Duane K. Fields](#) is a senior Internet systems engineer with Tivoli Systems, an IBM company, in Austin, Texas. A Sun-certified Java programmer, Duane has been developing network applications in one form or another since the early days of the Internet.

(4.99)

Related Reading:

- [Manuals](#)
- [TechNotes](#)
- [Sample Code](#)
- [View Source Articles](#)
- [FAQ](#)
- [Newsgroup](#)

Any sample code included above is provided for your use on an "AS IS" basis, under the [Netscape License Agreement - Terms of Use](#)