chapter 3

Introduction to JSPs, Servlets, and EJBs

"I do believe Marsellus Wallace, my husband, your boss, told you to take me out and do whatever I wanted. I wanna dance, I wanna win. I want that trophy, so dance good."

Mia Wallace, Pulp Fiction

So you've got containers and they provide services. You've got a Web server. Great. But what applications are they servicing? A container standing around all day flexing in front of the mirror, murmuring softly "Oh yeah...you've got transaction handling, baby..." doesn't do anyone any good. Containers exist because applications need them.

So there must be J2EE applications. And it's pretty likely that there are some special kinds of technology available to write the J2EE applications in, and some rules about how to write the applications to work correctly with the J2EE application server.

In short, who are the J2EE application server's dancing partners and what are they like?

Chapter in Brief

J2EE gives you more different kinds of technologies to use than Just Plain J2EE code. There are servlets and JSPs to deal with Web-enabled applications, session beans to do business processes, and entity beans to represent your data.

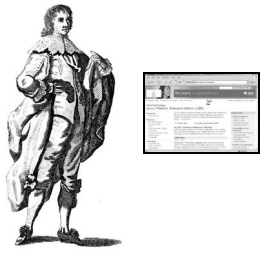
Here's what we talk about in this chapter.

- *Overview of J2EE Code*
- *Client Tier: Nothin' Here But Us Browsers*
- *Web Tier Technologies: JSPs and Servlets*
- *Business Tier Technology: Enterprise JavaBeans*

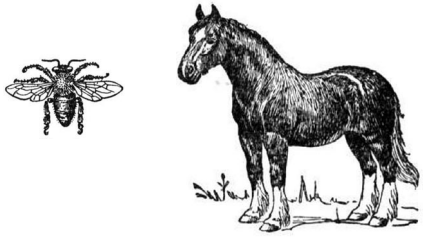
Overview of J2EE Code

Figure 3-1 illustrates the chunks, or tiers, a J2EE application is generally divided into. We'll be focusing on the servlets and JSPs on the Web tier, and the EJBs on the EJB tier.

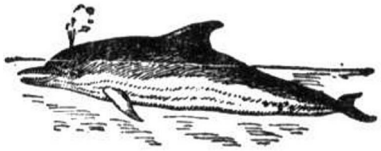
.



The **client tier** just has, of course, customers, but in addition to them, just a **browser**.



This is the **Web tier**, sometimes (but not always) containing the **Web server** and **Web container**, plus all the HTML pages, JSPs, and servlets.



This is the **EJB tier, middle tier,** or **business tier**. It typically has the **EJB container** and all the Enterprise JavaBeans.

This tier is called the **database tier** or **persistence tier.** It's the tier where all the data is stored and where the **database management system** is located. It's in charge of dealing with requests to store and retrieve data, it's loyal, it's always there holding your slippers and/or your invoice records. (Hence the dog analogy).

Figure 3-1 J2EE tiers with different types of technology on each

The database-handling features aren't really the focus of this chapter, but the database does come into the picture since one kind of EJB is the in-memory representation of records in the database.

Client Tier: Nothin' Here But Us Browsers

There's nothing on any of your customers' machines that has anything to do with your application.

Well, that's not *quite* true. The pages from your application get sent over to them by the Web server. They're cached (stored) on the customers' machines for a while, then deleted. There's also the occasional cookie, which is a little text file that tells Amazon who you are when you go back an hour later, and allows Amazon to make, let's be frank, embarrassingly accurate recommendations. You might also, especially in a distributed but not online application, have some J2EE code on your customers' machines.

But for our purposes right now, just take the client tier where your customers hang out and forget it. The real goodies are on the other tiers.

Web Tier Technologies: JSPs and Servlets

JSPs are HTML with a little Java code inside, and servlets are Java code adapted for the Web, sometimes with a little HTML inside.

JavaServer Pages, or JSPs

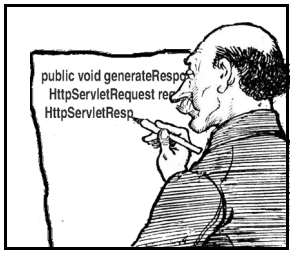
JSPs are basically HTML on Java steroids; you can slip in various kinds of extra codes, plus actual Java code. They go on the Web tier and the Web container sends them to the client tier so users can see the pages.

Here's an example that uses regular HTML, then breaks stride for a moment to use a JSP tag to get a name. This line could appear in the middle of any page that otherwise just has HTML.

```
Thanks for requesting information about <I><%=
request.getParameter("topic") %></I>
```

Sometimes JSPs work on their own, and sometimes they call other parts of the J2EE family, like servlets.

Servlets



Servlets are just Java code for doing normal programming, *but* on the Web. They can handle being out there on the Web because servlets can speak HTTP, the Language of the World Wide Web. They understand all the commands that your browser sends across, including GET and POST commands. They also understand what the Web server says, since the Web server of course speaks HTTP as well.

What Servlets Replaced

Servlets are J2EE's answer to CGI, common gateway interface scripts. CGI scripts are all over the place and great for smaller projects, but don't scale*—grow as your business does—very well.*

Servlets use and extend classes like HttpServlet which are specifically designed to deal with Web stuff and interact with JSPs. The following code appears in a lot of servlets. The doPost and the HttpServletRequest and HttpServletResponse are pretty much dead giveaways. Take a look at Example 3-1, not necessarily to read what the code is doing, but just to see the types of code used in a servlet so you'll recognize these elements in other code you look at on the job.

Example 3-1 Typical servlet code

```
public void doGet (HttpServletRequest myRequest, HttpServletResponse myResponse)
  throws ServletException, IOException
{
    some Java code that does stuff
}
public void doPost (HttpServletRequest myRequest, HttpServletResponse myResponse)
  throws ServletExeption, IOException
{
    doGet (myRequest, myResponse)
}
```

A Bit More About Servlets and JSPs

Servlets and JSPs both answer Web requests, whether the requests are something like "get me the description page for this item" or "find me all the books by Laura Jacobson."

JSPs are really good at providing great layout and some simple code/calculations and can contain all sorts of Java code in a variety of forms. Servlets are pretty powerful code and can generate HTML as well but it's not a great system. So we generally use both; JSPs are used for anything that's mostly layout but then needs a little extra Java punch, and servlets are used for anything that's primarily behind-the-scenes processing, business logic. They often work together; the JSP can ask a servlet to do something, then display the results.

Servlets and JSPs support the idea of *separation of concerns* (the object orientation thing about isolating unrelated chunks of code). JSPs most commonly do presentation logic, which means controlling how things look. Servlets most commonly do processing, the underbeneath stuff of *business logic*, which is just the guts of the program. Each is capable of doing both presentation and business logic, but JSPs are better suited to presentation logic, and servlets are better suited to business logic. We discuss separation of concerns more in *The Attributes of Object Orientation* on page 261.

The Web Deployment Descriptor Keeps Track of How to Run the Application

JSPs and servlets are pretty smart, and so is the Web container, but you can't just write the code and say "go!" You need to give a little bit more information about how to run the application in the form of the *deployment descriptor.* The deployment descriptor is just a big configuration file containing settings for how to run security, how to run the application, and much more. We talk about it more in *Web Deployment Descriptor* on page 107.

Business Tier Technology: Enterprise JavaBeans

EJBs, or Enterprise JavaBeans, are a whole different animal. While servlets and some EJBs can do similar things, the way a programmer writes EJBs is a whole lot different. And the way EJBs run and are managed by their EJB container is different, too.

Types of Enterprise JavaBeans

There are three kinds of EJBs: beans that represent things, beans that do things, and other types of beans that do things.

- Entity beans – Entity beans represent your database. An individual entity bean holds a record from a table in your database, like the record invoice 45991-A, or Baroness Hickenlooper's name and address record.
- Session beans – Session beans do stuff. They compute tax or they send information to another part of your application. They're your business processes. Session beans do what people would do if you were running your business in a world without computers. If you have a use case, you should probably have a session bean or five that correspond to it.
- Message-driven beans – Message-driven beans also do things. But they do things a little differently. Message-driven beans are like a butler who will answer the door when you're in the middle of a project but you're expecting someone to come by. The butler takes care of people who come by so that you can get a little work done.

All right, there might be a little more to it than that. We gave you the gist of each on the first go-around; here's a little more detail.

Entity Beans

Entity beans pretend to be the important things in your business like customers, invoices, orders, pizzas, etc. The things that you store information about in your database. Currently needed portions of the database are loaded into memory, assigned to entity beans, and boom, there's your database, floating around in memory.

A typical entity bean stands around holding a record from the database, and helps change the information. For instance, if an entity bean is holding a person's name and address and the address has changed, the bean might scratch out "910 Harrison Drive" and change it to "191 Mapleton." Or sometimes the bean might just take the record and squish it up and throw it away. Entity beans just do what would be done normally to the database, but in memory.

Nothing the entity bean does in its data-holding-capacity actually affects the database. This is up to the EJB container. The container is the one who pats the entity bean on the head, says "Good job!" and then does the actual corresponding changes in the database itself. The entity bean does the prep work, but the container is really in charge.

Now, you might well be thinking, "Why is this necessary in life? You've got a perfectly good database sitting there with very expensive database people taking care of it. Why duplicate all that in entity beans, especially when you've been telling us that there's not enough memory to go around?" And that's a very good question, especially given how complicated it can be to get data from the entity beans into the database and back again.

The point is the point which we have mentioned and will continue to harp on throughout the book: it's just good object-oriented application design. It's separation of concerns, encapsulation, all those good things.

Something still has to deal with communicating with the tables, of course. You don't bypass that. You just push the interaction with the database to the end of a particular process. That pushing is also one of the first principles of object orientation: put the things that have to do with databases in one place, off to the side by itself.

- J2EE applications are object oriented, and they understand objects. Objects are pieces of code that know information about themselves, and have things they can do. These same applications need to deal with the data from databases. Databases are organized in an entirely different way—a different paradigm, if you will. It's easier for the application to get at data if it's stored in an object-y way rather than in a database-y way.
- Now, granted, you haven't eliminated the need for objects to communicate with the database, by introducing a layer of entity beans on top of the database. But you have reduced the number of things that have to deal with the database, and given them special skills for doing so. These special skills came precoded from an application server, so you didn't have to write them yourself. That saves time and as much as you might hate to admit it, the application server folks might have done it better than you would have.
- Entity beans now are the only things that have to deal with the database and all the complexity therein. If it weren't for them, every session bean and servlet and possibly JSP that wanted to get anything out of or into or changed in the database would have to deal with SQL and database connections and allllll that stuff. Instead, entity beans take over the job, in tandem with the container, so all the other pieces of code, and all the people writing those

pieces of code, can be a little happier, a little more carefree, and get home from work before 10 o'clock.

Here's a condensed list of benefits.

- Entity beans sacrifice themselves so that they bear the brunt of dealing with the database, and no one else has to
- Entity beans separate the messy database connection stuff into one part of the application, so that it doesn't get mixed up with other things
- How entity beans communicate with databases is predetermined so that you don't have to write your own code to do it which might or might not be as well written or consistent

Not only have you encapsulated all this nastiness in one place, but with entity beans and CMP, it's all done for you. You don't have to do diddly. You tell it what the whole messy table thing looks like, and the container takes care of it.

Time Out for Jargon

Entity beans reify the enduring business themes of your domain object model.

Yep, they sure do. Throw this sentence out at the next meeting with a straight face and see how many people nod as if they know what you're talking about.

It just means, entity beans are your data from your database, done up in little Java entity bean packages and running in memory for more convenient access rather than sitting sedately in your database, on the hard disk.

There's also a couple things called *bean-managed persistence* and *container-managed persistence* (BMP and CMP). They relate to entity beans and the database. Specifically. they relate to whether the programmer writes detailed code for how to get data out of the entity beans' big meaty hands and into the database, or whether the programmer leans back in her chair, sips her coffee, and says, "The container can take care of all that stuff." We'll get into that later.

Session Beans

Forget about computers for a few seconds. Imagine Antoine's pizza shop a hundred years ago. People did the work instead of computers. People created the pizza orders, added up the total order and asked for money, assigned delivery people to deliver the pizzas, figured out sales tax, put all the invoices for the day in the right filing cabinet folders, get sleepy in the middle of the afternoon and take naps, and so on.

That's what session beans do in a J2EE application. Session beans do what people used to do before computers. They send the invoice down to Receiving, or calculate sales tax, or schedule a delivery for tomorrow based on what delivery people are available, or say, "Hmm, this guy's ordering a lot of pizzas. Let's give him a coupon for our Gold Members Club." Session beans basically do the work.

Session beans are business logic.

More Business Logic? What About Servlets?

Session beans and servlets do roughly similar kinds of things. However, they're designed to do it in different environments: the Web and a Web container; versus not the Web, and an EJB container. That's not to say you can't have session beans in an online application; it's just that the Web server doesn't talk to the session beans, and session beans don't speak HTTP. If you need to complete a business process and the result will be showing up in the browser interface, then you usually use servlets. But if the results of the business process are instead intended for the database or somewhere else that's not the browser interface, you generally go with session beans.

There are two types of session beans: *stateful session beans* and *stateless session beans*.

- Stateful session beans are faithful to one client for their lifetime. They sit there doing their session bean business logic, and they keep track of everything that has to do with that client and that process. The statefulness in their name has to do with the state, the data, gathered from the client they're working with. If Jeannie starts shopping, her bean stays with her until she orders that pizza and soda special or until she decides to work out and eat a salad instead, and logs off. Stateful session beans can remember everything their client tells them if that's what the application wants, keeps information around like their birthday or what items they've seen, and are ever so faithful.

  They carry around a lot of data with them which makes them useful in that respect, but slower.

  A stateful session bean is like a good waiter in a fancy restaurant who serves you and only you for the entire meal, bringing you menus, brushing the

crumbs off the table, brandishing that enormous pepper grinder, and so on. (Except that the waiter isn't killed or its memory wiped after each meal.)

You could also think of session beans as the ideal boyfriend or girlfriend (though between us, they're a little dull).

- Stateless session beans will service any client's request, and won't remember anything about the last request. You could come to the same stateless session bean five times in a row, and she would greet you with the same vacuous, unrecognizing smile and do the one thing you asked her to do. Stateless doesn't mean they never remember any data; they can have their own private data like a variable that points to a database connection. They just don't remember state gathered from a client; that is, they'll do one thing with one piece of data collected from a client request, but then throw that state away.

  They don't carry around any state with them from one client request to another, so they're generally zippier than stateful session beans when it comes to performance.

  Stateless session beans are like that clerk at the coffee shop who never remembers you but always gives you the kind of coffee you ask for.

Figure 3-2 shows a stateless session bean at left and a stateful session bean at right. Look how closely the stateful session bean is paying attention to the guy she's having the conversation with. Yet the stateless session bean lost interest after the first question he asked her and is now just staring off into space.

Figure 3-2 Stateless and stateful session beans, respectively

Message-Driven Beans

The standard definition is that "Message-driven beans are stateless, server-side, transaction-aware components that process asynchronous messages delivered via the Java Message Service (JMS)."

This just means that you can have the message-driven beans do stuff for you when you're busy. The JMS (Java's post office) sends you the EJB container a message. You're busy processing 10,000 transactions a minute and don't want to have to drop everything and go check on whether there's enough database connections around. So the message-driven bean does it for you.

The message-driven bean is like a butler. You can tell him, "Oh, Jeeves, there'll be some people coming by sometime today to ask about the badgers we've got on sale this week. Could you keep a look out for them and handle it when they arrive? I'm going to be trying to get some work done."

The message-driven beans let the container concentrate on the main task at hand.

To do this in the application, you just specify in the deployment descriptor what messages you're expecting, from who, what message-driven beans will deal with said messages, and so on. You can have lots and lots of message-driven beans.

You might be thinking, after mulling this a bit, why the heck does something as powerful as the J2EE EJB container, which after all we chose to represent with a dolphin, need little butler beans to help it out?

Because J2EE isn't *multithreaded*. That means the container can't do more than one thing at a time. The container is fast, smart, and powerful but not so great with the multitasking. The container is like your geeky Uncle Alfred who can tell you when Easter will be in 2047 but if you ask him where the butter is at the same time, he'll totally lose track of the Easter thing. Or alternately he'll totally ignore you and keep figuring out Easter.

So message-driven beans let you simulate multithreadedness in your J2EE system.

The EJB Deployment Descriptor Keeps Track of How to Run the Application

Just as for the Web container, there's a big configuration file called the deployment descriptor that's kind of like a big Daytimer for the beans. The deployment descriptor keeps track of who's allowed to do what, what the names of all the beans are, and lots of other stuff. We talk about it more in *The EJB Deployment Descriptor* on page 154.

What's up With This "Bean" Name, Anyway?

The term *bean* is confusing since people go around talking about their beans doing this and that, and then about how their application does that and this, and the words get in the way of understanding that their beans *are* their application and vice versa. Beans go together to become an application.

Beans Are Just Java Code With a Twist

Beans are Java code. Beans are your application. A J2EE application, for instance, is often made up of a few JSPs on the front end to make the pages, a servlet or seventeen to do some complicated heavy duty processing for the front end, and a bunch of session beans to do the rest of the program's duties.

So why say *bean*?

The marketing folks and maybe Bill Joy and James Gosling dreamed a long time ago as they invented Java of an ideal world in which every piece of Java code could be reusable. That is, if it were written with the right extra bits of code that would let someone hook up to it, any piece of Java code could be reusable. The reusable code could flit happily from one application to another, doing good wherever it went.

Note: This doesn't happen quite as often as the founders would like but it's possible.

To be one of those reusable components, of course, a piece of Java code can't just be written willy-nilly. There are specific rules for coding them. Beans are written in a way so that, theoretically, you could send people your MarksUpPrices bean, they could pop it into their application, and boom, instant MarksUpPrices functionality.

What's a small unit of Java, in the coffee sense? Well, a bean, of course, so that's the name the big Java programming folks applied to a small unit of reusable Java code.

To make things just a wee bit complicated, there are of course two kinds of beans in Java, and they're entirely unrelated.

The Two Kinds of Entirely Unrelated Beans in Java

There's JavaBeans, and there's Enterprise JavaBeans or EJBs. They have nothing to do with each other, and work differently. How's that for a really silly naming job?

What JavaBeans Are

JavaBeans aren't really much in the news now, but a few years ago they were hot.

JavaBeans are really just very clearly, predictably written Java code and have nothing to do with J2EE.

If you read the object orientation appendix or if you know object orientation, you know that an object knows things about itself. These things are *attributes*. A Java bean, for every one of its attributes like `red` or `Nancy` or `19.99`, should have a way to set those attributes to particular values, and to just go get the value for a particular attribute. After all, if you create an object in Java like a particular red shirt, it would be silly if you couldn't go check to be sure it was red, or go "Whoops!" and change the shirt to blue.

The types of code that set and get the values of attributes are called, fairly logically, *getters* and *setters*. Any piece of Java code that has appropriately named getters and setters for all its attributes can be a JavaBean.

JavaBeans are generally entirely unrelated to EJBs. However, they're not entirely unrelated to J2EE. They've enjoyed a brief fashionable resurgence since JSPs; you can run a JavaBean from a JSP.

What Enterprise JavaBeans Are

Enterprise JavaBeans are written entirely differently from JavaBeans and have a whole nother set of requirements. They have to implement certain interfaces, and they don't interact with Just Any Other Piece of Code. They interact only with an EJB container, or with other EJBs. Nothing else.

EJBs can use all the services we've been talking about: security, data integrity, resource management. JavaBeans get squat.

So JavaBeans are nothing special, and EJBs are. We now want you to forget about JavaBeans for the rest of the book.