PART ONE

Overview

In Part 1, we introduce the server-side development platform that is the *Java 2 Platform, Enterprise Edition* (J2EE), of which the *Enterprise JavaBeans* (EJB) component architecture is a vital piece. J2EE is a conglomeration of concepts, programming standards, and innovations—all written in the Java programming language. With J2EE, you can rapidly construct distributed, scalable, reliable, and portable secure server-side deployments.

Chapter 1 begins by exploring the need for a server-side component architecture such as EJB. You'll see the rich needs of server-side computing, such as scalability, high availability, resource management, and security. We'll look at each of the different parties that are involved in an EJB deployment. We'll also survey the J2EE server-side development platform.

Chapter 2 moves on to the Enterprise JavaBeans fundamentals. We'll look at the concept of *request interception*, which is crucial for understanding how EJB works. We'll also look at the different files that go into a bean and how they work together.

Chapter 3 gets down and dirty with EJB programming. Here, we'll write our first simple bean. We'll show how to code each of the files that compose the bean, and we'll also look at how to call that bean from clients.

CHAPTER 1

Overview

Enterprise JavaBeans (EJB) is a server-side component architecture that simplifies the process of building enterprise-class distributed component applications in Java. By using EJB, you can write scalable, reliable, and secure applications without writing your own complex distributed component framework. EJB is about rapid application development for the server side; you can quickly and easily construct server-side components in Java by leveraging a prewritten distributed infrastructure provided by the industry. EJB is designed to support application portability and reusability across any vendor's enterprise middleware services.

If you are new to enterprise computing, these concepts will be clarified shortly. EJB is a complicated subject and thus deserves a thorough explanation. In this chapter, we'll introduce EJB by answering the following questions:

- What plumbing do you need to build a robust distributed object deployment?
- What is EJB, and what value does it add?
- Who are the players in the EJB ecosystem?

Let's kick things off with a brainstorming session.

The Motivation for EJB

Figure 1.1 shows a typical business application. This application could exist in any vertical industry and could solve any business problem. Here are some examples:

- A stock trading system
- A banking application
- A customer call center
- A procurement system
- An insurance risk analysis application

Notice that this application is a *distributed system*. We broke up what would normally be a large, monolithic application and divorced each layer of the application from the others, so that each layer is completely independent and distinct.

Take a look at this picture, and ask yourself the following question based purely on your personal experience and intuition: *If we take a monolithic application and break it up into a distributed system with multiple clients connecting to multiple servers and databases over a network, what do we need to worry about now* (as shown in Figure 1.1)?

Take a moment to think of as many issues as you can. Then turn the page and compare your list to ours. Don't cheat!

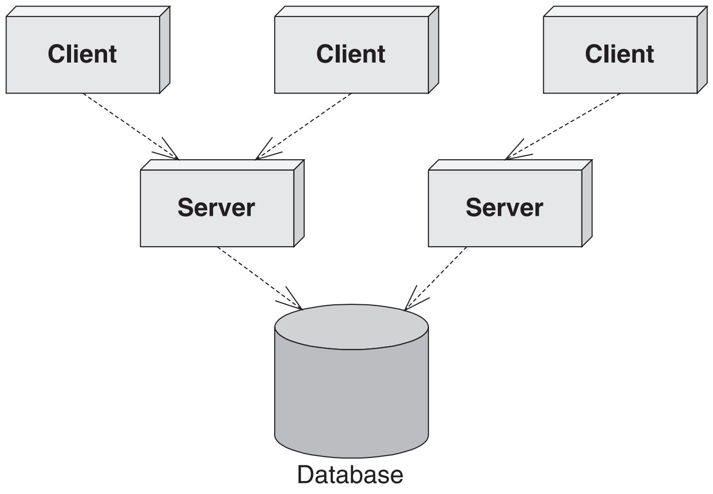

Figure 1.1 Standard multitier deployment.

In the past, most companies built their own middleware. For example, a financial services firm might build some of the middleware services above to help them put together a stock trading system.

These days, companies that build their own middleware risk setting themselves up for failure. High-end middleware is hideously complicated to build and maintain, requires expert-level knowledge, and is completely orthogonal to most companies' core business. Why not buy instead of build?

The *application server* was born to let you buy these middleware services, rather than build them yourself. Application servers provide you with common middleware services, such as resource pooling, networking, and more. Application servers allow you to focus on your application and not worry about the middleware you need for a robust server-side deployment. You write the code specific to your vertical industry and deploy that code into the runtime environment of an application server. You've just solved your business problem by *dividing and conquering*.

Divide and Conquer to the Extreme

We've just discussed how you can gain your middleware from an application server, empowering you to focus on your business problem. But there's even better news: You may be able to buy a partial solution to the business problem itself.

To achieve this, you need to build your application out of *components*. A component is code that implements a set of well-defined interfaces. It is a manageable, discrete chunk of logic. Components are not entire applications—they cannot run alone. Rather, they can be used as puzzle pieces to solve some larger problem.

The idea of software components is very powerful. A company can purchase a well-defined module that solves a problem and combine it with other components to solve larger problems. For example, consider a software component that computes the price of goods. We'll call this a *pricing component*. You hand the pricing component information about a set of products, and it figures out the total price of the order.

The pricing problem can get quite hairy. For example, let's assume we're ordering computer parts, such as memory and hard drives. The pricing component figures out the correct price based on a set of *pricing rules* that may include:

Base prices of a single memory upgrade or a single hard disk

Quantity discounts that a customer receives for ordering more than 10 memory modules

Things to Consider When Building Large Business Systems

By now you should have a decent list of things you'd have to worry about when building large business systems. Here's a short list of the big things we came up with. Don't worry if you don't understand all of them yet—you will.

- **Remote method invocations.** We need logic that connects a client and server via a network connection. This includes dispatching method requests, brokering of parameters, and more.
- **Load balancing.** Clients must be directed to the server with the lightest load. If a server is overloaded, a different server should be chosen.
- **Transparent fail-over.** If a server crashes, or if the network crashes, can clients be rerouted to other servers without interruption of service? If so, how fast does fail-over happen? Seconds? Minutes? What is acceptable for your business problem?
- **Back-end integration.** Code needs to be written to persist business data into databases as well as integrate with legacy systems that may already exist.
- **Transactions.** What if two clients access the same row of the database simultaneously? Or what if the database crashes? Transactions protect you from these issues.
- **Clustering.** What if the server contains state when it crashes? Is that state replicated across all servers, so that clients can use a different server?
- **Dynamic redeployment.** How do you perform software upgrades while the site is running? Do you need to take a machine down, or can you keep it running?
- **Clean shutdown.** If you need to shut down a server, can you do it in a smooth, clean manner so that you don't interrupt service to clients who are currently using the server?
- **Logging and auditing.** If something goes wrong, is there a log that we can consult to determine the cause of the problem? A log would help us debug the problem so it doesn't happen again.
- **Systems Management.** In the event of a catastrophic failure, who is monitoring our system? We would like monitoring software that paged a system administrator if a catastrophe occurred.
- **Threading.** Now that we have many clients connecting to a server, that server is going to need the capability of processing multiple client requests simultaneously. This means the server must be coded to be multi-threaded.
- **Message-oriented middleware.** Certain types of requests should be *message-based* where the clients and servers are very loosely coupled. We need infrastructure to accommodate messaging.
- **Object life cycle.** The objects that live within the server need to be created or destroyed when client traffic increases or decreases, respectively.

- **Resource pooling. If a client is not currently using a server, that server's precious resources can be returned to a *pool* to be reused when other clients connect. This includes sockets (such as database connections) as well as objects that live within the server.**
- **Security. The servers and databases need to be shielded from saboteurs. Known users must be allowed to perform only operations that they have rights to perform.**
- **Caching. Let's assume there is some database data that all clients share and make use of, such as a common product catalog. Why should your servers retrieve that same catalog data from the database over and over again? You could keep that data around in the servers' memory and avoid costly network roundtrips and database hits.**
- **And much, much, *much* more.**

Each of these issues is a separate service that needs to be addressed for serious server-side computing. These services are needed in any business problem and in any vertical industry. And each of these services requires a lot of thought and a lot of plumbing to resolve. Together, these services are called *middleware*.

Bundling discounts that the customer receives for ordering *both* memory and a hard disk

Preferred customer discounts that you can give to big-name customers

Locale discounts depending on where the customer lives

Overhead costs such as shipping and taxes

These pricing rules are in no way unique to ordering computer parts. Other industries, such as health care, appliances, airline tickets, and others need the same pricing functionality. Obviously, it would be a huge waste of resources if each company that needed complex pricing had to write its own sophisticated pricing engine. Thus, it makes sense that a vendor provides a generic pricing component that can be reused for different customers. For example:

1. The U.S. Postal Service can use the pricing component to compute shipping costs for mailing packages. This is shown in Figure 1.2.
2. An automobile manufacturer can use the pricing component to determine prices for cars. This manufacturer may set up a Web site that allows customers to get price quotes for cars over the Internet. Figure 1.3 illustrates this scenario.

3. An online grocery store can use the pricing component as one discrete part of a complete *workflow* solution. When a customer purchases groceries over the Web, the pricing component first computes the price of the groceries. Next, a different vendor's component bills the customer with the generated price. Finally, a third component fulfills the order, setting things in motion for the groceries to be delivered to the end user. We depict this in Figure 1.4.

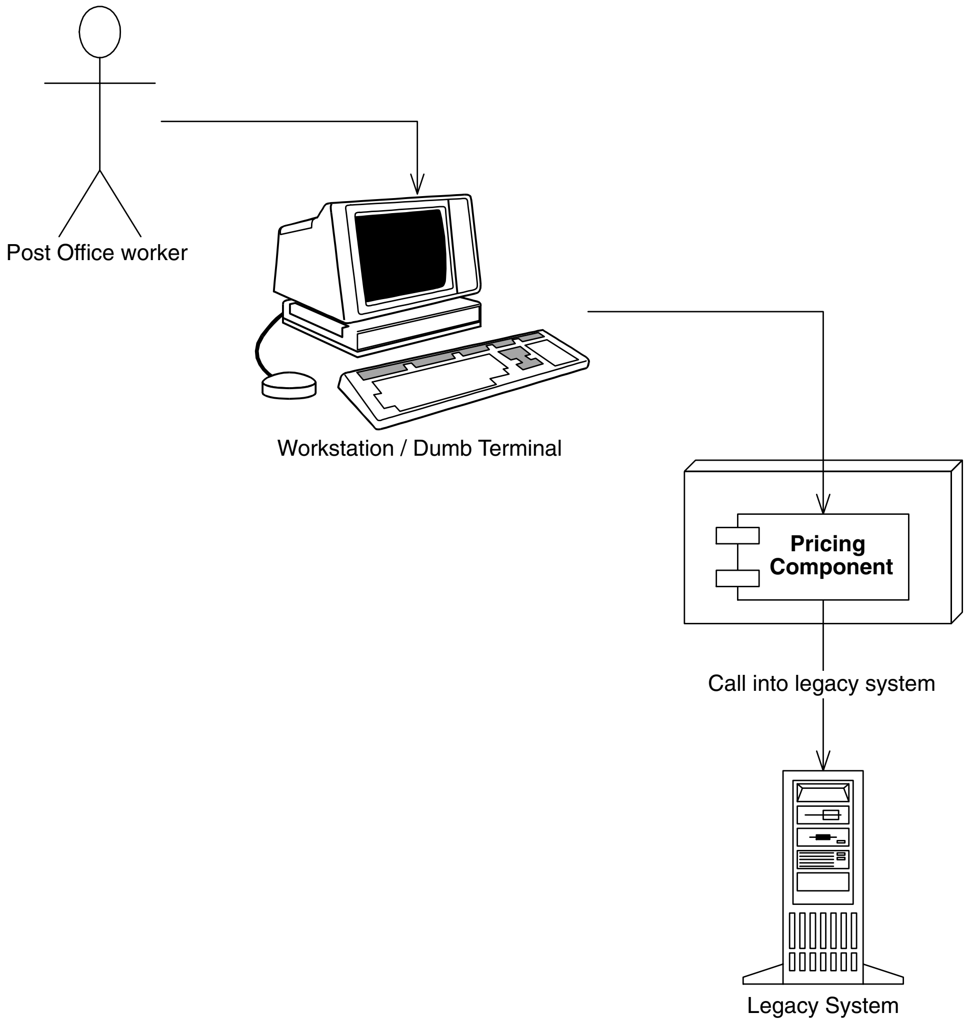

Figure 1.2 Reusing a pricing component for the U.S. Postal Service.

200110_CH01/RomanX 11/14/01 4:14 PM Page 9

Reusable components are quite enticing because components promote rapid application development. An IT shop can quickly assemble an application from prewritten components rather than writing the entire application from scratch. This means:

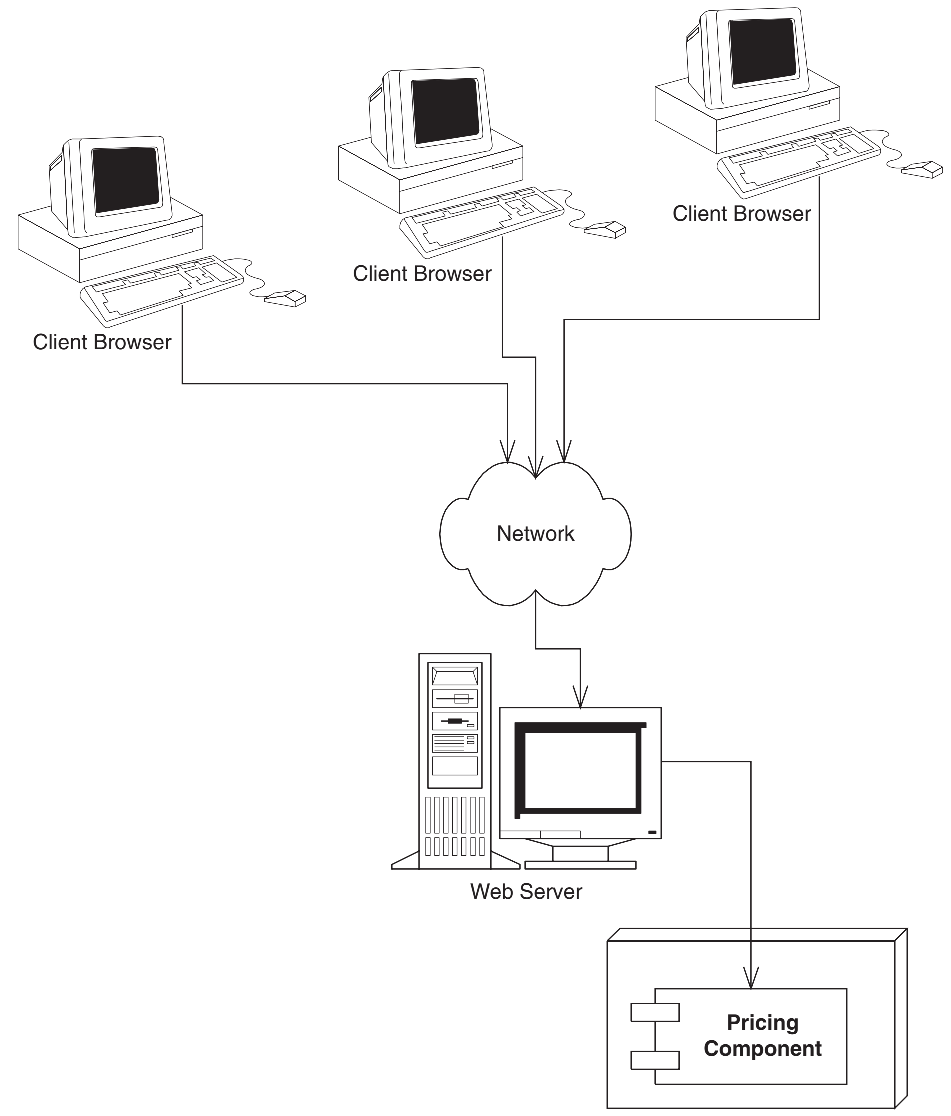

Figure 1.3 Reusing a pricing component for quoting car prices over the Internet.

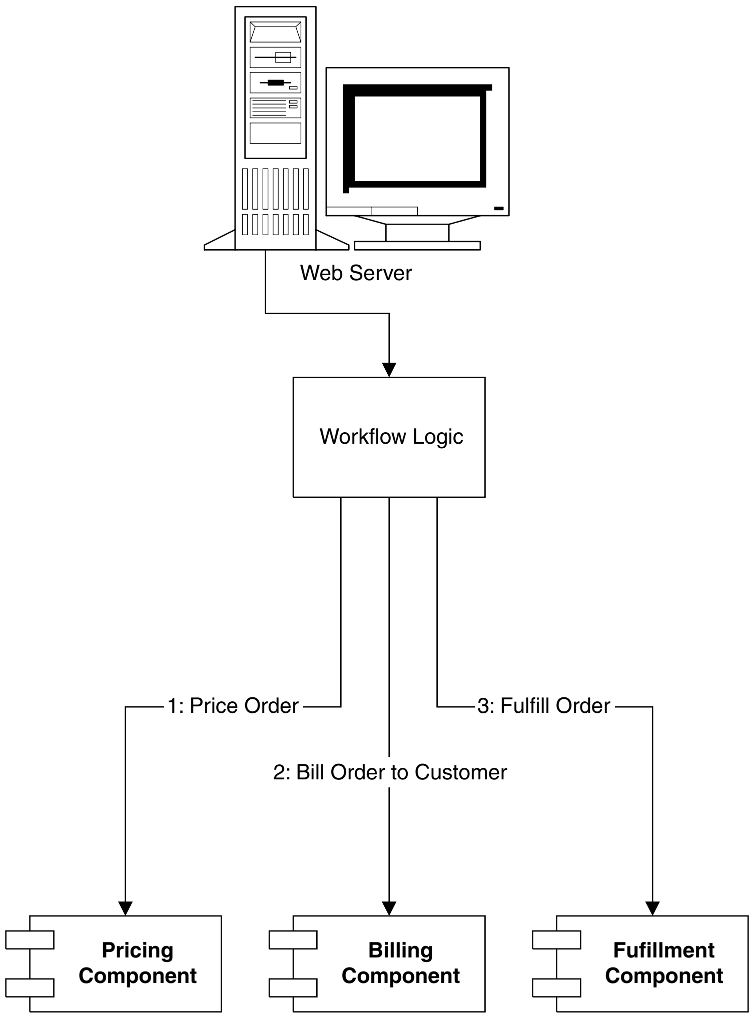

Figure 1.4 Reusing a pricing component as part of an e-commerce workflow solution.

The IT shop needs less in-house expertise. The IT shop can consider the pricing component to be a black box, and it does not need experts in complex pricing algorithms.

The application is assembled faster. The component vendor has already written the tough logic, and the IT shop can leverage that work, saving development time.

There is a lower total cost of ownership. The component vendor's cash cow is its components, and therefore it must provide top-notch documentation, support, and maintenance if it is to stay in business. Because the component vendor is an expert in its field, the component generally has fewer bugs and higher performance than an IT shop's home-grown solution. This reduces the IT shop's maintenance costs.

Once the rules of engagement have been laid down for how components should be written, a *component marketplace* is born, where vendors can sell reusable components to companies. The components are deployed within application servers, which provide the needed middleware.

Is a Component Marketplace a Myth?

There is a very small component marketplace today. For years we've been hoping that the marketplace will explode, but it is behind schedule. There are several reasons for Independent Software Vendors (ISVs) not shipping components:

Maturity. Because components live inside application servers, the application servers must be mature before we see components written to those servers.

Politics. Many ISVs have written their own application servers. Some (falsely) view this as a competitive advantage.

Questionable value. Most ISVs are customer-driven (meaning they prioritize what their customers are asking for). Since components are new to many customers, many of them are not asking for their ISVs to support components.

It is our opinion that the marketplace will eventually explode, and it's just a matter of time. If you represent an ISV, this could be a fantastic opportunity for you.

The good news is that the marketplace already beginning to emerge. Most packaged e-commerce ISVs (Ariba, Broadvision, Vignette, and so on) are shipping or have announced support for server-side Java technologies.

In the meantime, you'll have to build your own components from scratch within your organizations. Some of our customers at The Middleware Company are attempting this by having departments provide components to other departments. In effect, that department is acting as an internal ISV.

Component Architectures

It has been a number of years since the idea of multitier server-side deployments surfaced. Since then, well over 50 application servers have appeared on the market. At first, each application server provided component services in a nonstandard, proprietary way. This occurred because there was no agreed definition of what a component should be. The result? Once you bet on an application server, your code was locked into that vendor's solution. This greatly reduced portability and was an especially tough pill to swallow in the Java world, which promotes openness and portability. It also hampered the commerce of components, because a customer could not combine a component written to one application server with another component written to a different application server.

What we need is an *agreement*, or set of interfaces, between application servers and components. This agreement will enable any component to run within any application server. This will allow components to be switched in and out of various application servers without having to change code or potentially even recompile the components themselves. Such an agreement is called *component architecture* and is shown in Figure 1.5.

If you're trying to explain components to a nontechie, try these analogies:

- **Any CD player can play any compact disc because of the CD standard. Think of an application server as a CD player and components as compact discs.**
- **In the United States, any TV set can tune into any broadcast because of the NTSC standard. Think of an application server as a TV set and components as television broadcasts.**

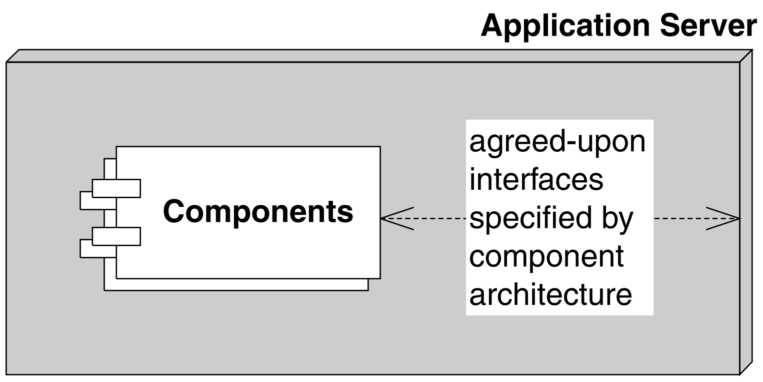

Figure 1.5 A component architecture.

200110_CH01/RomanX 11/14/01 4:14 PM Page 13

Introducing Enterprise JavaBeans

The *Enterprise JavaBeans* (EJB) standard is a component architecture for deployable server-side components in Java. It is an agreement between components and application servers that enable any component to run in any application server. EJB components (called *enterprise beans*) are deployable, and can be imported and loaded into an application server, which hosts those components.

The top three values of EJB are as follows:

1. It is agreed upon by the industry. Those who use EJB will benefit from its widespread use. Because everyone will be on the same page, in the future it will be easier to hire employees who understand your systems (since they may have prior EJB experience), learn best practices to improve your system (by reading books like this one), partner with businesses (since technology will be compatible), and sell software (since customers will accept your solution). The concept of "train once, code anywhere" applies.
2. Portability is easier. The EJB specification is published and available freely to all. Since EJB is a standard, you do not need to gamble on a single, proprietary vendor's architecture. And although portability will never be free, it is cheaper than without a standard.
3. Rapid application development. Your application can be constructed faster because you get middleware from the application server. There's also less of a mess to maintain.

Note that while EJB does have these virtues, there are also scenarios where EJB is inappropriate. See Chapter 15 for a complete discussion of when to (and when not to) use EJB.

Physically, EJB is actually two things in one:

A specification. This is a 500-plus-page Adobe Acrobat PDF file, freely downloadable from http://java.sun.com. This specification lays out the rules of engagement between components and application servers. It constricts how you program so that you can interoperate.

A set of Java interfaces. Components and application servers must conform to these interfaces. Since all components are written to the same interfaces, they all look the same to the application server. The application server therefore can manage anyone's components. You can freely download these interfaces from http://java.sun.com.

Why Java?

EJB components must be written in Java only and require dedication to Java. This is indeed a serious restriction. The good news, however, is that Java is an ideal language to build components, for many reasons.

Interface/implementation separation. We need a clean interface/implementation separation to ship components. After all, customers who purchase components shouldn't be messing with implementation. Upgrades and support will become horrendous. Java supports this at a syntactic level via the *interface* keyword and *class* keyword.

Safe and secure. The Java architecture is much safer than traditional programming languages. In Java, if a thread dies, the application stays up. Pointers are no longer an issue. Memory leaks occur much less often. Java also has a rich library set, so that Java is not just the syntax of a language but a whole set of prewritten, debugged libraries that enable developers to avoid reinventing the wheel in a buggy way. This safety is extremely important for mission-critical applications. Sure, the overhead required to achieve this level of safety might make your application slower, but 90 percent of all business programs are glorified Graphical User Interfaces (GUIs) to databases. That database is going to be your number one bottleneck, not Java.

Cross-platform. Java runs on any platform. Since EJB is an application of Java, this means EJB should also easily run on any platform. This is valuable for customers who have invested in a variety of powerful hardware, such as Win32, UNIX, and mainframes. They do not want to throw away these investments.

If you don't want to go the EJB route, you have two other choices as well:

- **Microsoft's .NET managed components, part of the Microsoft.NET platform**
- **The Object Management Group (OMG's) Common Object Request Broker Architecture (CORBA)**

Note that many EJB servers are based upon and can interoperate with CORBA (see Appendix B for strategies for achieving this).

EJB as a Business Solution

EJB is specifically used to help solve *business problems*. EJB components (enterprise beans) might perform any of the following tasks.

Perform business logic. Examples include computing the taxes on the shopping cart, ensuring that the manager has authority to approve the purchase order, or sending an order confirmation email using the *JavaMail API*.

Access a database. Examples include submitting an order for books, transferring money between two bank accounts, or calling a stored procedure to retrieve a trouble ticket in a customer support system. Enterprise beans achieve database access using the *Java Database Connectivity* (JDBC) *API*.

Access another system. Examples include calling a high-performing *CICS* legacy system written in COBOL that computes the risk factor for a new insurance account, calling a legacy *VSAM* data store, or calling *SAP R/3*. Enterprise beans achieve existing application integration via the *Java Connector Architecture* (JCA).

EJB components are not GUI components; rather, enterprise beans sit behind the GUIs and do all the hard work. Examples of GUIs that can connect to enterprise beans include the following:

Thick clients. Thick clients execute on a user's desktop. They could connect via the network with EJB components that live on a server. These EJB components may perform any of the tasks listed above (business logic, database logic, or accessing other systems). Thick clients in Java include applets and applications.

Dynamically generated web pages. Web sites that are complex need their Web pages generated specifically for each request. For example, the homepage for Amazon.com is completely different for each user, depending on the user's profile. Java servlets and JavaServer Pages (JSPs) are used to generate such specific pages. Both servlets and JSPs live within a Web server and can connect to EJB components, generating pages differently based upon the values returned from the EJB layer.

XML-based Web Service wrappers. Some business applications require no user interface at all. They exist to interconnect with other business partners' applications that may provide their own user interface. For example, Dell Computer Corporation needs to purchase Intel chips to manufacture desktop computers. Intel could expose a Web Service that enables Dell's software to connect and order chips. In this case, Intel's system does not have a user interface of its own, but rather acts as a Web Service. Possible technologies used here include SOAP, UDDI, ebXML, and WSDL. This is shown in Figure 1.6.

The real difference between GUI components (thick clients, dynamically generated Web pages, and Web Service wrappers) and enterprise beans is the domain that each component type is intended to be part of. GUI components are well suited to handle *client-side* operations, such as rendering GUIs (although they don't necessarily need to have one), performing other presentation-related logic, and lightweight business logic operations. They deal directly with the end-user or business partner.

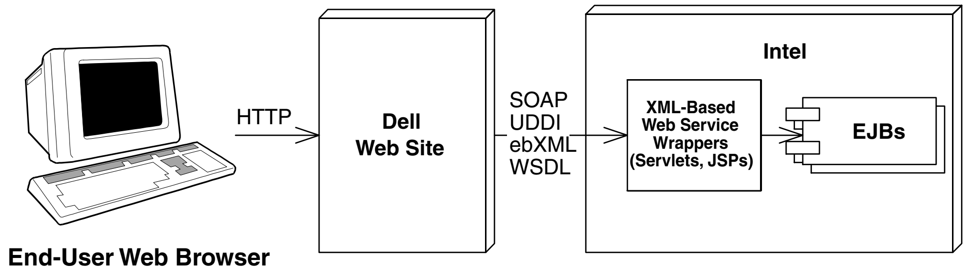

Figure 1.6 EJBs as the back-end to Web services.

Enterprise beans, on the other hand, are not intended for the client side; they are *server-side* components. They are meant to perform server-side operations, such as executing complex algorithms or performing high-volume business transactions. The server side has different kinds of needs from a rich GUI environment. Server-side components need to run in a highly available (24 × 7), fault-tolerant, transactional, and multiuser secure environment. The application server provides this high-end server-side environment for the enterprise beans, and it provides the runtime containment necessary to manage enterprise beans.

The EJB Ecosystem

To get an EJB deployment up and running successfully, you need more than just an application server and components. In fact, EJB encourages collaboration of *more than six* different parties. Each of these parties is an expert in its own field and is responsible for a key part of a successful EJB deployment. Because each party is a specialist, the total time required to build an enterprise-class deployment is significantly reduced. Together, these players form the *EJB Ecosystem*.

Let's discuss who the players are in the EJB Ecosystem. As you read on, think about your company's business model to determine which role you fill. If you're not sure, ask yourself what the core competency of your business is. Also think about what roles you might play in upcoming projects.

The EJB Ecosystem is not for everyone. At my company, we've heard ghastly stories of businesses choosing EJB because everyone else is using it, or because it is new and exciting. Those are the wrong reasons to use EJB and can result in disappointing results. For a complete discussion of when and when not to use EJB, see Chapter 15.

JavaBeans. Enterprise JavaBeans

You may have heard of another standard called *JavaBeans*. JavaBeans are completely different from Enterprise JavaBeans.

In a nutshell, JavaBeans are Java classes that have get/set methods on them. They are reusable Java components with properties, events, and methods (similar to Microsoft's *ActiveX controls*) that can be easily wired together to create (often visual) Java applications.

JavaBeans are much smaller than Enterprise JavaBeans. You can use JavaBeans to assemble larger components or to build entire applications. JavaBeans, however, are *development components* and are not *deployable components*. You typically do not deploy a JavaBean; rather, JavaBeans help you construct larger software that *is* deployable. And because they cannot be deployed, JavaBeans do not need to live in a runtime environment. Since JavaBeans are just Java classes, they do not need an application server to instantiate them, to destroy them, and to provide other services to them. The application itself is made up of JavaBeans.

The Bean Provider

The *bean provider* supplies business components, or enterprise beans. Enterprise beans are not complete applications, but rather are deployable components that can be assembled into complete solutions. The bean provider could be an ISV selling components or an internal department providing components to other departments.

Many vendors ship reusable components today. You can get the complete list from www.componentsource.com or www.flashline.com. In the future, traditional enterprise software vendors (such as sales force automation vendors, enterprise resource planning vendors, financial services vendors, and e-commerce vendors) will offer their software as enterprise beans or provide connectors to their current technology.

The Application Assembler

The application assembler is the overall application architect. This party is responsible for understanding how various components fit together and writes the applications that combine components. An application assembler may even author a few components along the way. His or her job is to build an application from those components that can be deployed in a number of settings. The application assembler is the *consumer* of the beans supplied by the bean provider.

The application assembler could perform any or all of the following tasks:

- From knowledge of the business problem, decide which combination of existing components and new enterprise beans are needed to provide an effective solution; in essence, plan the application assembly.
- Supply a user interface (perhaps Swing, servlet/JSP, application/applet, or Web Service wrapper).
- Write new enterprise beans to solve some problems specific to your business problem.
- Write the code that calls on components supplied by bean providers.
- Write integration code that maps data between components supplied by different bean providers. After all, components won't magically work together to solve a business problem, especially if different vendors write the components.

An example of an application assembler is a systems integrator, a consulting firm, or an in-house programmer.

The EJB Deployer

After the application assembler builds the application, the application must be *deployed* (and go live) in a running operational environment. Some challenges faced here include the following:

- Securing the deployment with a firewall and other protective measures
- Integrating with an LDAP server for security lists, such as Lotus Notes or Microsoft Active Directory
- Choosing hardware that provides the required level of performance
- Providing redundant hardware and other resources for reliability and fault tolerance
- Performance-tuning the system

Frequently the application assembler (who is usually a developer or systems analyst) is not familiar with these issues. This is where the EJB deployer comes into play. EJB deployers are aware of specific operational requirements and perform the tasks above. They understand how to deploy beans within servers and how to customize the beans for a specific environment. The EJB deployer has the freedom to adapt the beans, as well as the server, to the environment in which the beans are to be deployed.

An EJB deployer can be a staff person, an outside consultant, or a vendor. Examples of EJB deployers include Loudcloud and HostJ2EE.com, which both offer hosting solutions for EJB deployments.

The System Administrator

Once the deployment goes live, the system administrator steps in to oversee the stability of the operational solution. The system administrator is responsible for the upkeep and monitoring of the deployed system and may make use of runtime monitoring and management tools that the EJB server provides.

For example, a sophisticated EJB server might page a system administrator if a serious error occurs that requires immediate attention. Some EJB servers achieve this by developing hooks into professional monitoring products, such as Tivoli and Computer Associates. Others are providing their own systems management by supporting the *Java Management Extension* (JMX).

The Container and Server Provider

The container provider supplies an *EJB container* (the application server). This is the runtime environment in which beans live. The container supplies middleware services to the beans and manages them. Examples of EJB containers

Qualities of Service in EJB

Monitoring of EJB deployments is not specified in the EJB specification. It is an optional service that advanced EJB servers can provide. This means that each EJB server could provide the service differently.

At first blush you might think this hampers application portability. However, in reality this service should be provided *transparently* behind the scenes, and should not affect your application code. It is a quality of service that lies beneath the application level and exists at the systems level. Changing application servers should not affect your EJB code.

Other transparent qualities of service not specified in the EJB specification include load balancing, transparent fail-over, caching, clustering, and connection pooling algorithms.

are BEA's WebLogic, iPlanet's iPlanet Application Server, IBM's WebSphere, Oracle's Oracle 9i, Macromedia's JRun, Persistence's PowerTier, Brokat's Gemstone/J, HP's Bluestone, IONA's iPortal, Borland's AppServer, and the JBoss open source code application server.

The server provider is the same as the container provider. Sun has not yet differentiated these (and they may never do so). We will use the terms *EJB container* and *EJB server* interchangeably in this book.

The Tool Vendors

To facilitate the component development process, there should be a standardized way to build, manage, and maintain components. In the EJB Ecosystem, there are several *Integrated Development Environments* (IDEs) assist you in rapidly building and debugging components. Examples are Webgain's Visual Cafe, IBM's VisualAge for Java, or Borland's JBuilder.

Other tools enable you to model components in the Unified Modeling Language (UML), which is the diagram style used in this book. You can then autogenerate EJB code from that UML. Examples of products in this space are Togethersoft's Together/J and Rational's Rational Rose.

There are other tools as well, such as tools to organize components (Flashline, ComponentSource), testing tools (JUnit, RSW Software), and build tools (Ant).

Summary of Roles

Figure 1.7 summarizes the interaction of the different parties in EJB.

You may be wondering why so many different participants are needed to provide an EJB deployment. The answer is that EJB enables companies or individuals to become experts in certain roles, and division of labor leads to best-of-breed deployments.

The EJB specification makes each role clear and distinct, enabling experts in different areas to participate in a deployment without loss of interoperability. Note that some of these roles could be combined as well. For example, the EJB server and EJB container today come from the same vendor. Or at a small startup company, the bean provider, application assembler, and deployer could all be the same person who is trying to build a business solution using EJB from scratch. What roles do you see yourself playing?

For some of the parties EJB merely suggests possible duties, such as the system administrator overseeing the well-being of a deployed system. For other parties, such as the bean provider and container provider, EJB defines a set of

200110_CH01/RomanX 11/14/01 4:14 PM Page 21

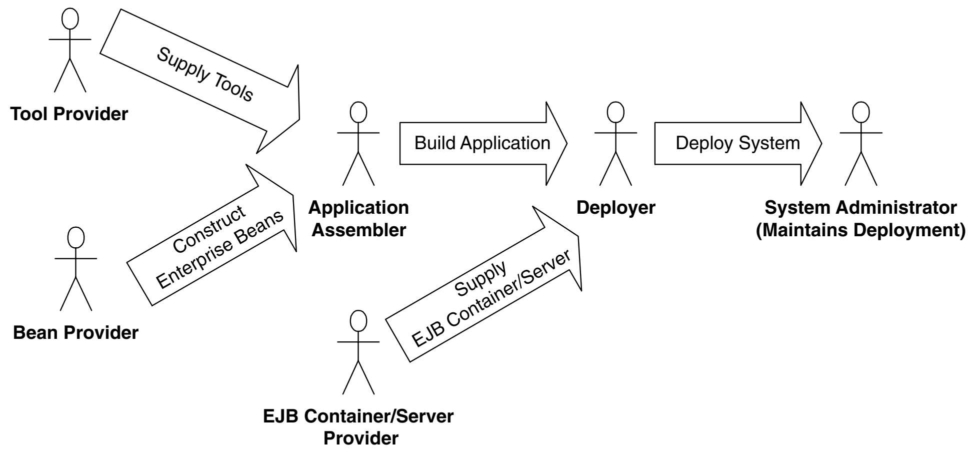
Tool Provider
Supply Tools
Application Assembler
Build Application
Deployer
Deploy System
System Administrator (Maintains Deployment)
Construct Enterprise Beans
Bean Provider
Supply EJB Container/Server
EJB Container/Server Provider

Figure 1.7 The parties of EJB.

strict interfaces and guidelines that must be followed or the entire ecosystem will break down. By clearly defining the roles of each party, EJB lays a foundation for a distributed, scalable component architecture where multiple vendors' products can interoperate.

A future EJB specification will define a new role, called the *persistence manager*, which plugs into an application server. Your components harness the persistence manager to map your business data into storage, such as mapping objects into relational databases.

The persistence manager may be written to understand how to persist business data to any storage type. Examples include legacy systems, flat file systems, relational databases, object databases, or a proprietary system.

The persistence manager provider may be the same as the container/server vendor, such as the case with IBM's WebSphere, which includes built-in persistence capabilities. Examples of ISV persistence manager providers include WebGain's TOPLink and Thought Inc's Cocobase.

Unfortunately, the persistence manager provider role is not explicitly defined in the EJB 2.0 specification. Due to time constraints, a standard for plugging persistence managers into application servers won't exist until a future version of EJB. The good news is this won't affect the portability of your code, because your application doesn't care whether it's being persisted by the container or by some persistence manager that happens to plug into the container. The bad news is that you'll need to rely on proprietary agreements between persistence manager providers and application server vendors, which means that not every persistence manager may work in every application server — for now.

The Java 2 Platform, Enterprise Edition (J2EE)

EJB is only a portion of a larger offering from Sun Microsystems called the Java 2 Platform, Enterprise Edition (J2EE). The mission of J2EE is to provide a platform-independent, portable, multiuser, secure, and standard enterprise-class platform for server-side deployments written in the Java language.

J2EE is a specification, not a product. J2EE specifies the rules of engagement that people must agree on when writing enterprise software. Vendors then implement the J2EE specifications with their J2EE-compliant products.

Because J2EE is a specification (meant to address the needs of many companies), it is inherently not tied to one vendor; it also supports cross-platform development. This encourages vendors to compete, yielding best-of-breed products. It also has its downside, which is that incompatibilities between vendor products will arise—some problems due to ambiguities with specifications, other problems due to the human nature of competition.

J2EE is one of *three different* Java platforms. Each platform is a conceptual superset of the next smaller platform.

The Java 2 Platform, Micro Edition (J2ME) is a development platform for Java-enabled devices, such as Palm Pilots, pagers, watches, and so on. This is a restricted form of the Java language due to the inherent performance and capacity limitations of small devices.

The Java 2 Platform, Standard Edition (J2SE) contains standard Java services for applets and applications, such as input/output facilities, graphical user interface facilities, and more. This platform contains what most people use in standard Java Development Kit (JDK) programming.

The Java 2 Platform, Enterprise Edition (J2EE) takes Java's Enterprise APIs and bundles them together in a complete development platform for enterprise-class server-side deployments in Java.

The arrival of J2EE is significant because it creates a unified platform for server-side Java development. J2EE consists of the following deliverables from Sun Microsystems.

Specifications. Each enterprise API within J2EE has its own specification, which is a PDF file downloadable from http://java.sun.com. Each time there is a new version of J2EE, Sun locks-down the versions of each Enterprise API specification and bundles them together as the de facto versions to use when developing with J2EE. This increases code portability across vendors' products because each vendor supports exactly the same API revision. This is analogous to a company such as Microsoft releasing a new version of Windows every few years: Every time a new version of Windows comes out, Microsoft locks-down the versions of the technologies bundled with Windows and releases them together.

Test suite. Sun provides a test suite for J2EE server vendors to test their implementations against. If a server passes the tests, Sun issues a compliance brand, alerting customers that the vendor's product is indeed J2EE-compliant. There are numerous J2EE-certified vendors, and you can read reviews of their products for free on TheServerSide.com.

Reference implementation. To enable developers to write code against J2EE as they have with the JDK, Sun provides its own free reference implementation of J2EE. Sun is positioning it as a low-end reference platform, as it is not intended for commercial use.

BluePrints Document. Each of the Enterprise APIs has a clear role in J2EE, as defined by Sun's *J2EE BluePrints* document. This document is a downloadable PDF file that describes how to use the J2EE technologies together.

The J2EE Technologies

The Java 2 Platform, Enterprise Edition is a robust suite of middleware services that make life very easy for server-side application developers. J2EE builds on the existing technologies in the J2SE. J2SE includes the base Java support and the various libraries (.awt, .net, .io) with support for both applets and applications. Because J2EE builds on J2SE, a J2EE-compliant product must not only implement all of J2EE, but must also implement all of J2SE. This means that building a J2EE product is an absolutely *huge* undertaking. This barrier to entry has resulted in significant industry consolidation in the Enterprise Java space, with a few players emerging from the pack as leaders.

We will discuss version 1.3 of J2EE, which supports EJB 2.0. Some of the major J2EE technologies are shown working together in Figure 1.8.

To understand more about the real value of J2EE, here is each API that a J2EE 1.3-compliant implementation must provide for you.

Enterprise JavaBeans (EJB). EJB defines how server-side components are written and provides a standard contract between components and the application servers that manage them. EJB is the cornerstone for J2EE and uses several other J2EE technologies.

Java Remote Method Invocation (RMI) and RMI-IIOP. RMI is the Java language's native way to communicate between distributed objects, such as two different objects running on different machines. RMI-IIOP is an extension of RMI that can be used for CORBA integration. RMI-IIOP is the official API that we use in J2EE (not RMI). We cover RMI-IIOP in Appendix A.

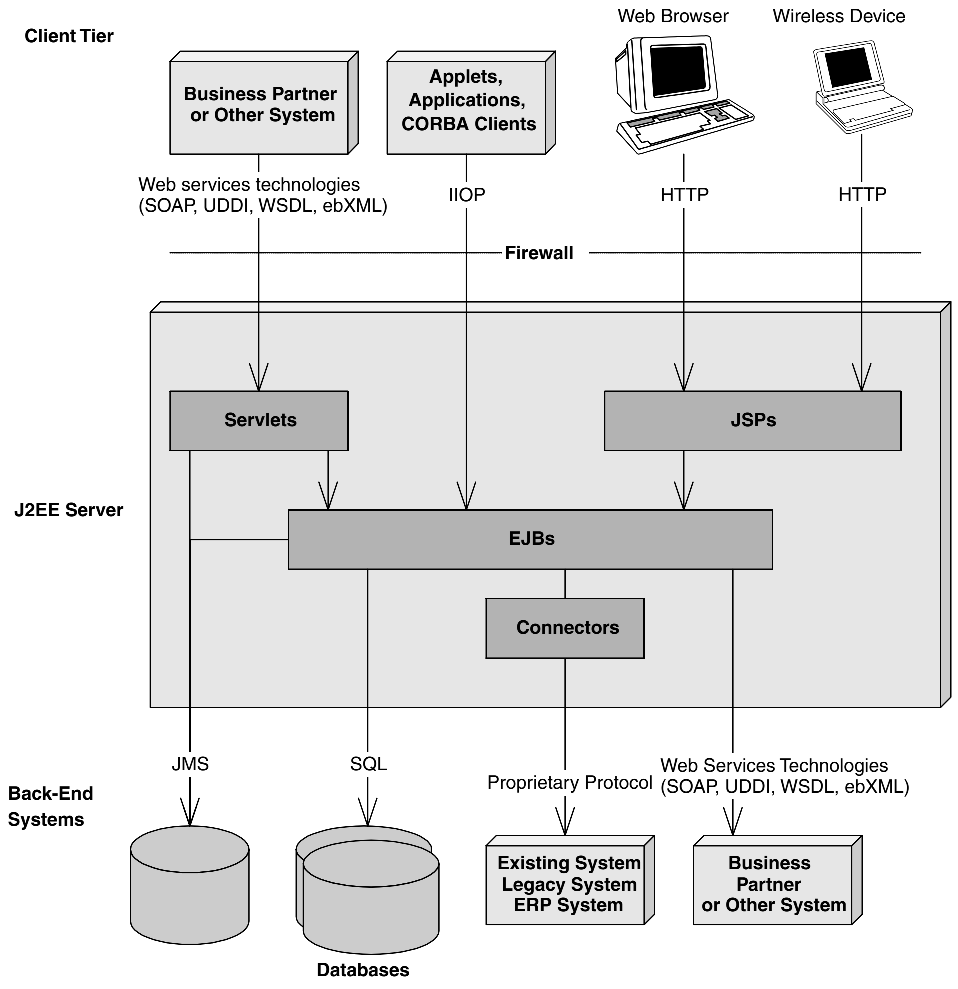
Client Tier
Web Browser
Wireless Device
Business Partner or Other System
Applets, Applications, CORBA Clients
Web services technologies (SOAP, UDDI, WSDL, ebXML)
IIOP
HTTP
HTTP
Firewall
J2EE Server
Servlets
JSPs
EJBs
Connectors
JMS
SQL
Proprietary Protocol
Web Services Technologies (SOAP, UDDI, WSDL, ebXML)
Back-End Systems
Existing System Legacy System ERP System
Business Partner or Other System
Databases

Figure 1.8 A Java 2 Platform, Enterprise Edition deployment.

Java Naming and Directory Interface (JNDI). JNDI is used to access naming and directory systems. You use JNDI from your application code for a variety of purposes, such as connecting to EJB components or other resources across the network, or accessing user data stored in a naming service such as Microsoft Exchange or Lotus Notes. JNDI is covered in Appendix A.

Java Database Connectivity (JDBC). JDBC is an API for accessing relational databases. The value of JDBC is that you can access any relational database using the same API. JDBC is used in Chapter 6.

Java Transaction API (JTA) Java Transaction Service (JTS). The JTA and JTS specifications allow for components to be bolstered with reliable transaction support. JTA and JTS are explained in Chapter 10.

Java Messaging Service (JMS). JMS allows for your J2EE deployment to communicate using messaging. You can use messaging to communicate within your J2EE system as well as outside your J2EE system. For example, you can connect to existing message-oriented middleware (MOM) systems such as IBM MQSeries or Microsoft Message Queue (MSMQ). Messaging is an alternative paradigm to RMI-IIOP, and has its advantages and disadvantages. We explain JMS in Chapter 8.

Java Servlets. Servlets are networked components that you can use to extend the functionality of a Web server. Servlets are request/response oriented in that they take requests from some client host (such as a Web browser) and issue a response back to that host. This makes servlets ideal for performing Web tasks, such as rendering an HTML interface. Servlets differ from EJB components in that the breadth of server-side component features that EJB offers is not readily available to servlets. Servlets are much better suited to handling simple request/response needs, and they do not require sophisticated management by an application server. We illustrate using Servlets with EJB in Chapter 17.

Java Pages (JSPs). JSPs are very similar to servlets. In fact, JSP scripts are compiled into servlets. The largest difference between JSP scripts and servlets is that JSP scripts are not pure Java code; they are much more centered around look-and-feel issues. You would use JSP when you want the look and feel of your deployment to be physically separate and easily maintainable from the rest of your deployment. JSPs are perfect for this, and they can be easily written and maintained by non-Java savvy staff members (JSPs do not require a Java compiler). We illustrate using JSPs with EJB in Chapter 17.

Java IDL. Java IDL is Sun Microsystems' Java-based implementation of CORBA. Java IDL allows for integration with other languages. Java IDL also allows for distributed objects to leverage CORBA's full range of services. J2EE is thus fully compatible with CORBA, completing the Java 2 Platform, Enterprise Edition. We discuss CORBA integration in Appendix B.

JavaMail. The JavaMail service allows you to send email messages in a platform-independent, protocol-independent manner from your Java programs. For example, in a server-side J2EE deployment, you can use JavaMail to confirm a purchase made on your Internet e-commerce site by sending an email to the customer. Note that JavaMail depends on the

JavaBeans Activation Framework (JAF), which makes JAF part of J2EE as well. We do not cover JavaMail in this book.

J2EE Connector Architecture (JCA). Connectors allow you to access existing enterprise information systems from a J2EE deployment. This could include *any* existing system, such as a mainframe systems running high-end transactions (such as those deployed with IBM's CICS or BEA's TUXEDO), Enterprise Resource Planning (ERP) systems, or your own proprietary systems. Connectors are useful because they automatically manage the details of middleware navigation to existing systems, such as handling transaction and security concerns. Another value of the JCA is that you can write a driver to access an existing system once, and then deploy that driver into any J2EE-compliant server. This is important because you only need to learn how to access any given existing system once. Furthermore, the driver needs to be developed only once and can be reused in any J2EE server. This is extremely useful for independent software vendors (ISVs) who want their software to be accessible from within application servers. Rather than write a custom driver for each server, the ISV can write a single driver. We discuss legacy integration more in Chapters 12 and 13.

The Java API for XML Parsing (JAXP). There are many applications of XML in a J2EE deployment. For example, you might need to parse XML if you are performing B2B interactions (such as through Web services), if you are accessing legacy systems and mapping data to and from XML, or if you are persisting XML documents to a database. JAXP is the de facto API for parsing XML documents in a J2EE deployment and is an implementation-neutral interface to XML parsers. You typically use the JAXP API from within servlets, JSPs, or EJB components. There is a free whitepaper available on TheServerSide.com that describes how to build Web services with J2EE.

The Java Authentication and Authorization Service (JAAS). JAAS is a standard API for performing security-related operations in J2EE. Conceptually, JAAS also enables you to plug in a security system to a J2EE deployment. See Chapter 9 for more details on security and EJB.

Summary

We've achieved a great deal in this chapter. First, we brainstormed a list of issues involved in a large, multitier deployment. We then understood that a server-side component architecture allows us to write complex business applications without understanding tricky middleware services. We then dove into the EJB standard and fleshed out its value proposition. We investigated the different players involved in an EJB deployment and wrapped up by exploring J2EE.

The good news is that we're just getting started, and many more interesting and advanced topics lie ahead. The next chapter delves into the concept of *request interception*, which is the mental leap you need to make to understand EJB. Let's go!