CHAPTER 1

Server-side Component Architectures

Enterprise JavaBeans (*EJB*) is a server-side component architecture that enables and simplifies the process of building enterprise-class distributed object applications in Java. By using EJB, you can write scalable, reliable, and secure applications without writing your own complex distributed object 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 made very clear shortly. EJB is a complicated subject and thus deserves a thorough explanation. In this chapter, we'll discusses the main concepts surrounding Enterprise JavaBeans. This starts with a discussion about what's involved in writing enterprise software and why a prepackaged distributed object architecture such as Enterprise JavaBeans simplifies your life. From this discussion, we'll have a greater insight into why a server-side component architecture makes sense, as well as a feature list of what we'd like to see when we choose an architecture for developing server-side distributed object applications.

We'll then examine several endeavors by the industry to address these enterprise needs. The highlight of this discussion—as well as this book—is Sun's *Java 2 Platform*, *Enterprise Edition* (*J2EE*). J2EE is a collection of enterprise technologies, of which EJB is an integral part. By understanding and using J2EE properly, you can build portable, object-oriented, enterprise-class applications in Java.

Go back to the first page for a quick link to buy this book online!

The Need for a Server-Side Component Architecture

To understand the value EJB brings to the table, we first must examine the needs that developers commonly have when authoring and deploying components in a server-side environment. As we uncover the issues surrounding server-side development, we'll begin to see the need for a standardized architecture such as EJB.

Software Components

We begin our discussion with a look at software components. A software 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* such as:

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

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 over and over again for different customers. For example:

Go back to the first page for a quick link to buy this book online!

1. The U.S. Postal Service can use the pricing component to compute shipping costs for mailing packages. This is shown in Figure 1.1.
2. An automobile manufacturer can use the pricing component to discriminate prices for cars. For example, the manufacturer can set up a Web site that allows customers to get price quotes for cars over the Internet. Figure 1.2 illustrates this scenario.
3. An online grocery store can use the pricing component as a 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 1.3.

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:

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.

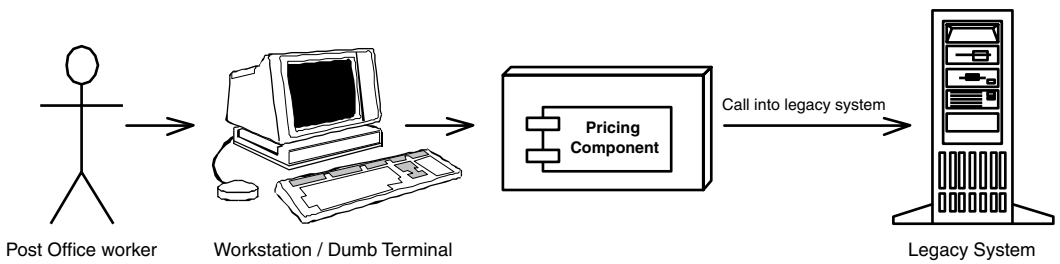

Figure 1.1 Reusing a pricing component for the U.S. Postal Service.

Go back to the first page for a quick link to buy this book online!

Thus, once the rules of engagement have been laid down for how components should be written, a *component marketplace* is born, where vendors can sell re-usable components to companies.

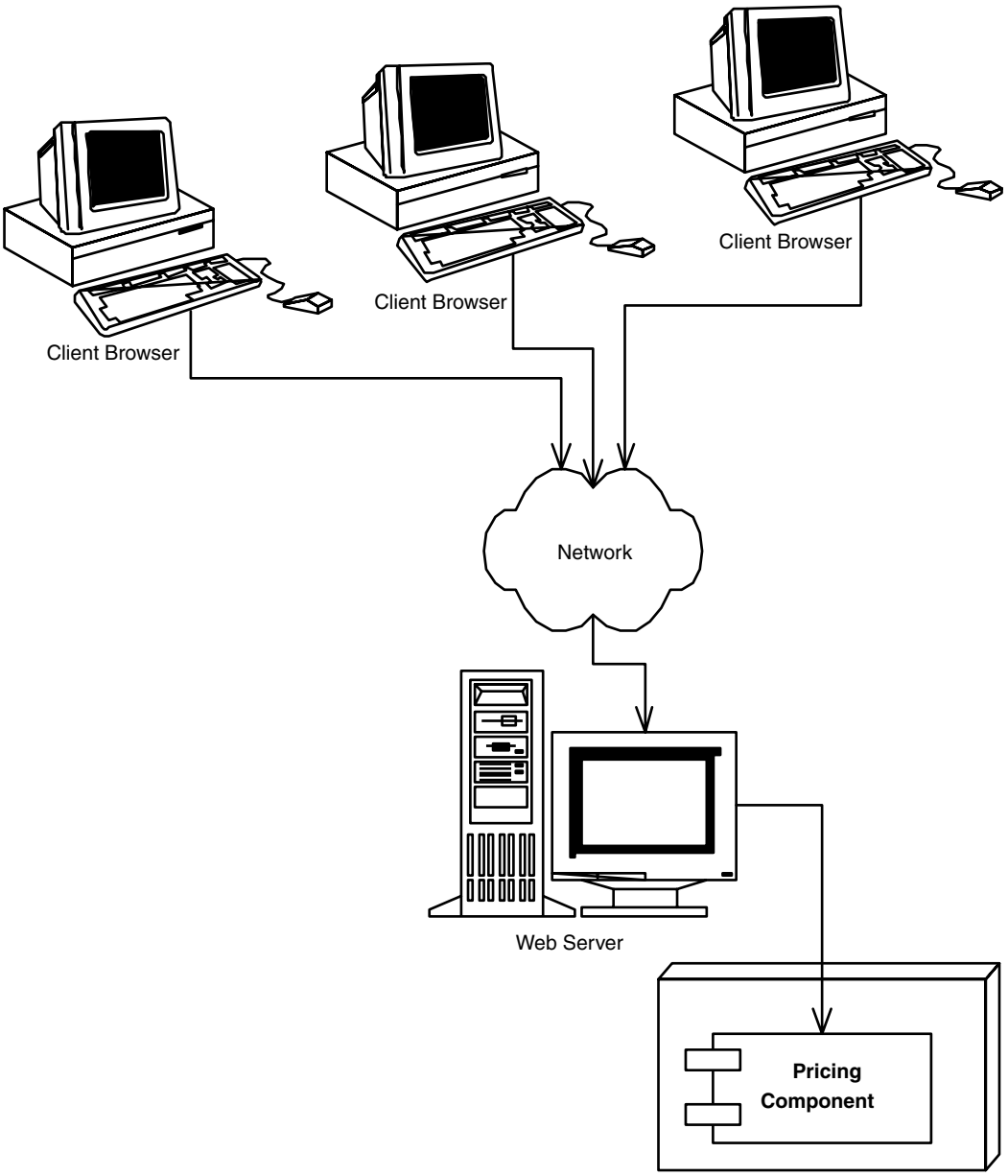

Figure 1.2 Reusing a pricing component for quoting car prices over the Internet.

Go back to the first page for a quick link to buy this book online!

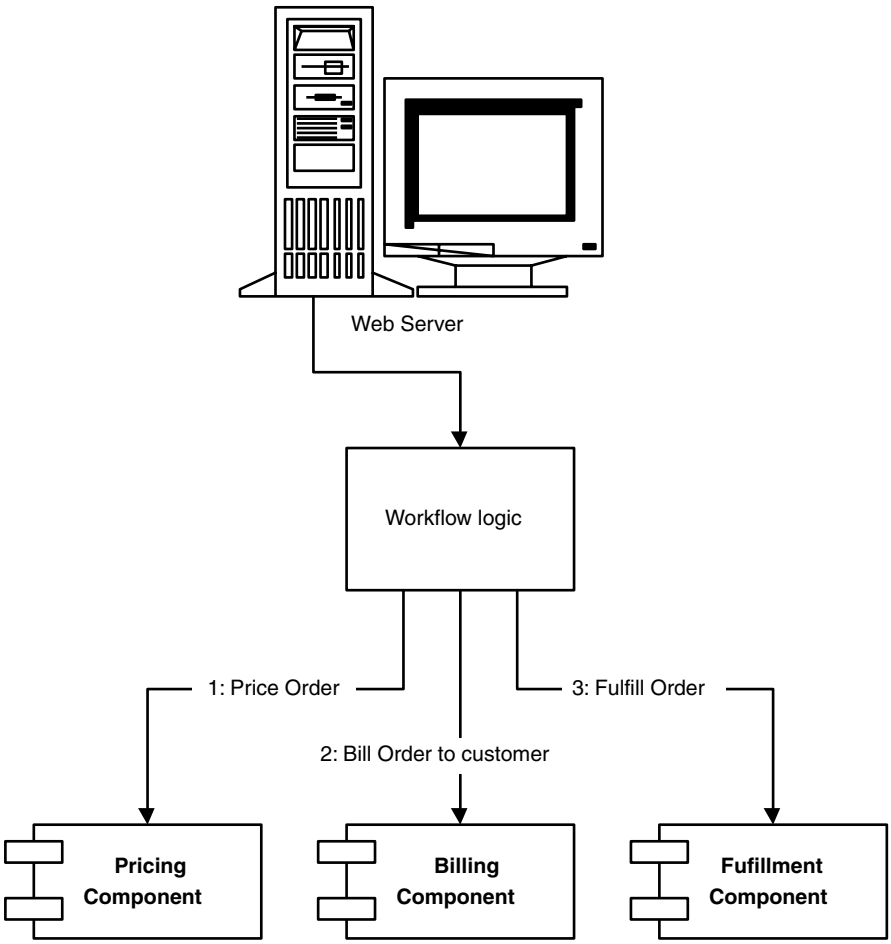

Figure 1.3 Reusing a pricing component as part of an e-commerce workflow solution.

Component Architectures

To facilitate the component development process, there should be a standardized way to build, manage, and maintain components. This approach consists of the following:

Tools for developing components. The process of building components should be streamlined, allowing the component developer to focus on writing the core logic behind the component. This promotes rapid application development and is essential for any component standard to succeed. For example, an *Integrated Development Environment* (*IDE*), such as Symantec's *Visual Cafe*, IBM's *VisualAge for Java*, or Inprise's *JBuilder 2*, assists Java developers in rapidly building and debugging components. Other vendors, such as Inline Software, provide enhanced EJB-specific development tools.

Go back to the first page for a quick link to buy this book online!

A container that manages your deployed components. This component container provides a runtime environment for your components to play in. It also provides a set of common services that most components will need. For example, the container could automatically instantiate new components as necessary, thus removing that burden from the component developer. To combine any container with any component, you must have a well-defined contract between containers and components. This contract allows any container to manage any component.

Tools for deploying and maintaining components. When an organization purchases components from component vendors, there must be a set of tools to aid in the deployment and maintenance of those components. For example, there should be a way to customize the components for a particular environment. In our pricing component example, we could have a tool that assists us in customizing the products we are pricing.

Each of these features is essential in a mainstream component marketplace. And, of course, as a component developer, you would like to focus on writing the components themselves, rather than the ancillary products that are common to all components: the container and the tools. A well-defined *component architecture* supplies the standards necessary for different vendors to write the components, component containers, and tools. Thus, by having a component architecture standard, developers can employ a "divide-and-conquer" approach to programming.

Java: An Ideal Language for Component Architectures

For a component to succeed in solving a business problem, both the component developer and the customer using the component must agree on the syntax and semantics of calling the component's methods. Thus, the component vendor must publish the *contract* (or rules) for calling the component, and the client code must adhere to these rules.

As the vendor releases new versions of the component, that vendor's customers will want to upgrade. This raises a number of issues. Will the new component work with the IT shop's code that called the old component? Do the IT shops need to recompile their client code? Or, even worse, has the component contract changed, necessitating that IT shops modify their client code to map to the new component contract?

Thankfully, object-oriented design introduced a great programming practice to help solve this problem by separating the *interface* of a component from its *implementation*:

A component's interface defines the component's contract with the code that calls it. For example, the interface defines methods and parameters that the component accepts. The interface masks the implementation from clients of

Go back to the first page for a quick link to buy this book online!

the component, so clients deal only with the end result: the methods the component exposes.

A component's implementation is the core programming logic that an object provides. It has some very specific algorithms, logic, and data. This data is private to the component, and it should be hidden from all client code that calls the component.

For interface/implementation separation to be effective, developers must write client code to a component's interface only (this is called *interface-based programming*). If you're writing components, you can force developers into this paradigm by publishing only the interfaces to your components, not your implementations.

By separating interface from implementation, you can vary a component's proprietary logic without changing any client code. For example, you can plug in a different implementation that performs the same task more efficiently. This is possible because the actual implementation is not needed at compile time—only the interface is needed. Hence, there is no specific implementation tied to the client code. This is shown in Figure 1.4.

The Java language supports interface/implementation separation at a syntactic level via the *interface* keyword and *class* keyword. And because Java is an interpreted language, the separation of code into discrete class files ensures that clients do not have to recompile their code if you ship a new version of your component.

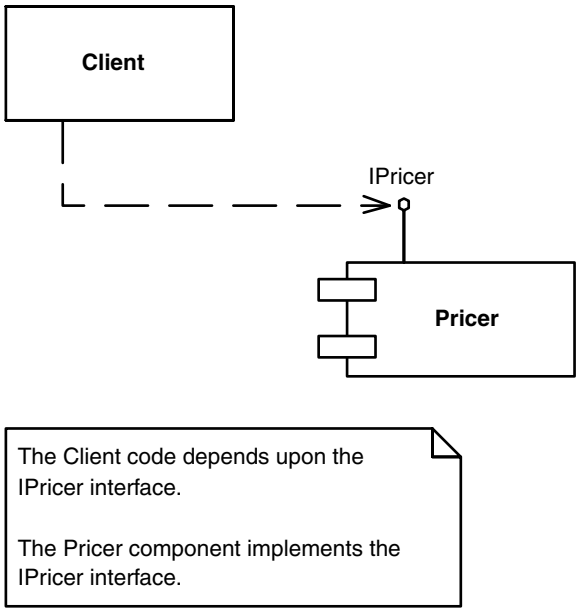

Figure 1.4 Interface-based programming on our pricing component.

Go back to the first page for a quick link to buy this book online!

In addition to the interface/implementation separation, Java is an object-oriented language that has been built from the ground-up as a cross-platform development language and that has wide industry support. This makes the Java language an ideal technology on which you can base components.

Component Architectures in Java

Now that you've seen what a component architecture is, let's look at what component architectures exist in the Java world. The first one you may have heard of is *JavaBeans*. JavaBeans components are small-grained application bits. You can use JavaBeans to assemble larger-grained components or to build entire applications. JavaBeans, however, are *development components* and are not *deployable components*. You typically do not deploy a JavaBean because a JavaBean is not a complete application; rather, JavaBeans help you construct larger software that *is* deployable. And because they cannot be deployed, JavaBeans do not need a runtime environment in which to live. JavaBeans do not need a container to instantiate them, to destroy them, and to provide other services to them because the application itself is made up of JavaBeans.

By way of comparison, the *Enterprise JavaBeans* (*EJB*) standard defines a component architecture for *deployable components* called *enterprise beans*. Enterprise beans are larger, coarser-grained application components that are ready to be deployed. They can be deployed as is, or they can be assembled with other components into larger application systems. Deployable components must be deployed in a container that provides runtime services to the components, such as services to instantiate components as needed.

Enterprise beans are very similar to two other types of Java components: applets and servlets. Applets can be deployed in a Web page, where the browser's applet viewer provides a runtime container for the applets. Servlets can be deployed in a Web server, where the Web server's *servlet engine* provides a runtime container for the servlets. Enterprise beans are deployed in an *application server*, where the application server provides a runtime container for the Enterprise JavaBeans. This is shown in Figure 1.5.

The real difference between applets, servlets, and enterprise beans is the domain of which each component type is intended to be a part.

Applets are portable Java programs that can be downloaded on the fly and can execute in an untrusting environment. For example, an applet can be downloaded from a Web server into a Web browser, and it typically displays a user interface to the end user.

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

Go back to the first page for a quick link to buy this book online!

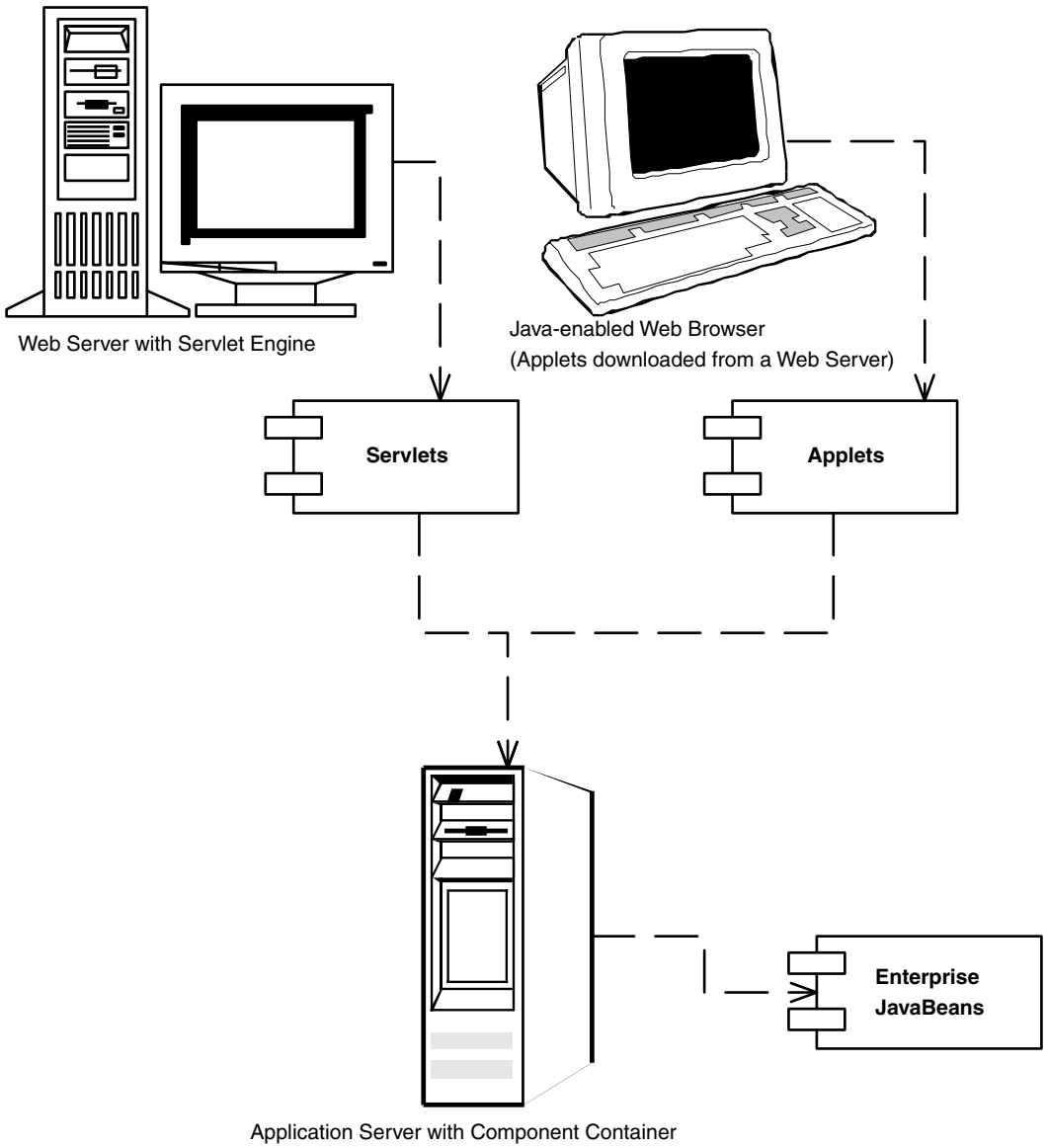

Figure 1.5 Applets, servlets, and Enterprise JavaBeans.

back to that host. This makes servlets ideal for performing Web tasks, such as rendering an HTML interface to an e-commerce catalog.

Both applets and servlets are well suited to handle *client-side* operations, such as rendering graphical user interfaces (GUIs) (although they don't necessarily need to have one), performing other presentation-related logic, and lightweight business logic operations. The client side could be a Web browser, in the case of applets

Go back to the first page for a quick link to buy this book online!

that render user interfaces using the Java Foundation Classes. The client side could also be a Web server, in the case of servlets that render user interfaces in HTML. In both these situations, the components are dealing directly with the end user.

Enterprise beans, on the other hand, are not intended for the client side, but 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 (24x7), fault-tolerant, transactional, and multiuser secure environment. An *application server* provides this high-end server-side environment for the enterprise beans, and it provides the runtime containment necessary to manage enterprise beans.

Finally, note that applets, servlets, and enterprise beans are not "either-or" technologies. You can use JavaBeans as development component building blocks for constructing deployable enterprise beans. You can also provide a user interface to your enterprise beans with applets or servlets (shown in Figure 1.5).

Now that you've seen where EJB fits in with other technologies, let's take a look at the class of problems that EJB addresses. EJB is meant for server-side programming; to appreciate what EJB brings to the table, we must first understand what makes server-side programming difficult.

The Needs of the Server Side

As we've mentioned, a complete component architecture paves the way for the following:

- Developers to write reusable components
- Vendors to write component containers that provide a runtime environment and services to components
- Vendors to provide development, deployment, and maintenance tools, which are necessary complements to the components themselves

This divide-and-conquer approach allows vendors to provide a set of common services that most components will need, thus saving precious development and deployment time. Rather than reimplement the wheel, the component developer can simply outsource the services he needs to other products. Professionals who are experts in providing these services write these products. When harnessed properly, users save time by buying rather than building. Additionally, the overall deployment is strengthened because domain experts are writing these common products.

As we'll see, server-side software opens up a whole new set of problems that require some very high-end services. If you choose to home-grow these services

Go back to the first page for a quick link to buy this book online!

yourself, you'll likely encounter a development and maintenance nightmare. Being able to outsource server-side services is one of the key benefits of a server-side distributed object architecture like EJB.

Multi-tier Architectures

A server-side deployment is software written to support concurrent users performing operations simultaneously, securely, reliably, and efficiently. Examples of server-side deployments include the following:

Banks where many ATM machines connect to a central bank server

Retail outlets such as the Wal-Mart chain of stores, where many Wal-Mart stores send shopping information to a central Wal-Mart server

Support centers where support engineers have terminals that can bring up customer data from a central server

Insurance agencies where insurance sales staff have terminals that connect to a central server

Web sites where thousands or millions of users connect to Web servers and those Web servers need to connect with a central server for data and logic

Robust server-side deployments are not easy to build. Many issues arise, such as scalability, maintainability, security, reliability, and more. With so many clients depending on your central server-side deployment, it would be a catastrophe if your central servers went down, slowed to a crawl, or allowed a hostile party to gain access to the systems. Therefore, server-side deployments need to be well written from the ground up and well tested, and they need to run in a robust environment.

Any well-written deployment has a logical software partitioning into *layers*. Each layer has a different responsibility in the overall deployment, and within each layer there can be one or more components. Note that these layers are purely abstractions, and they may not correspond to physical distribution. A layered system is a well-designed system because each layer is responsible for a separate task. Here is a typical layer partitioning:

A presentation layer contains components dealing with user interfaces and user interaction. For example, the presentation layer of a stand-alone application could be written in Visual Basic. The presentation layer of a Web-based deployment could use Java servlets, Java server pages, and/or Java applets.

A business logic layer contains components that work together to solve business problems. These components could be high-performance engines, such as catalog engines or pricing engines. Typically, these components are written in a type-safe language such as Java or C++.

Go back to the first page for a quick link to buy this book online!

A data layer is used by the business logic layer to persist state permanently. Central to the data layer is one or more databases that house the stored state.

The advantage to partitioning an application into these logical layers is to isolate each layer from the others. For example, it should be possible to plug in a different view (i.e., change the presentation layer) while minimizing impacts on the business logic or data layers. It should similarly be possible to plug in a different set of business rule component implementations within your business logic layer, or to plug in a different database in your data layer, with relatively minor effects on the other layers. In some ways, this is analogous to how the classic model-view-controller separation allows the developer to vary the model, view, and controller independently of one another.

The physical separation of these layers is another story. In a *two-tier architecture*, two of these layers are physically separated from the third, forming two physically separated *tiers*. On the other hand, a *three-tier architecture* separates each of these three abstract, logical layers into three physically distributed tiers. In each of these architectures, the tiers are separated from one another by some physical boundaries, such as machine boundaries, process boundaries, or corporate boundaries. In the discussion that follows, we don't really care what the boundary is—it could be a process boundary, a machine boundary within a local area network, or a boundary across the Internet. And for clarity, we will refer to all deployments with three tiers or more as *N-tiered*, which is used interchangeably with three-tiered occasionally.

So what are the advantages to separating your application into two tiers verses N tiers? There are a number of compelling reasons for both sides, which we'll soon uncover. From this debate, you will begin to see the needs of the server side and why a distributed server-side architecture such as Enterprise JavaBeans is necessary.

Two-Tier Architectures

Traditionally, most high-end deployments have been two-tiered. Two-tier deployments combine the business logic layer with one of the other two layers. There are two combinations possible: combining the presentation layer with the business logic layer, and pushing some of the business logic layer into the data layer. Let's take a look at each scenario.

Combining the Presentation Layer with the Business Logic Layer

Your presentation layer and business logic layer can be coupled together in a single tier, pushing the data access layer into a tier by itself. This draws a tier boundary between the business logic layer and the data layer (Figure 1.6). If you think of the first tier as a "client" and the second tier as a "server," this architecture effectively makes the client "fat" and the server "thin."

Go back to the first page for a quick link to buy this book online!

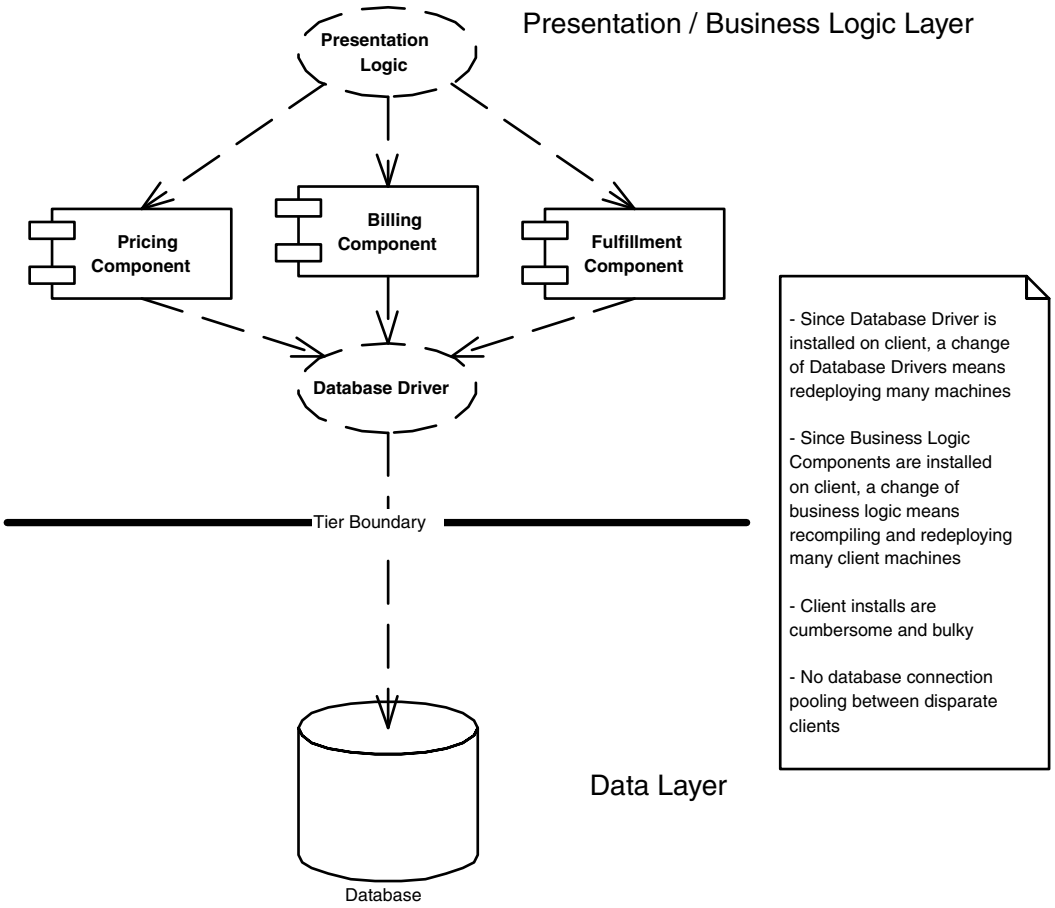

Figure 1.6 Combining presentation with business logic in a two-tier architecture.

In a two-tier architecture, the client application typically communicates with the data layer through a database bridge API such as Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC). This decouples the client from the particular database being used. So that each vendor's proprietary database will conform to the database bridge, each database vendor must supply a database driver that is called from the database bridge API, such as an ODBC driver or JDBC driver.

Two-tier architectures have the following characteristics:

Deployment costs are high. Database drivers must be installed and configured on each of the client tiers, which may mean hundreds or thousands of machines.

Database driver switching costs are high. Switching one database driver with another now requires a reinstallation of every client machine's database

Go back to the first page for a quick link to buy this book online!

drivers. This is very costly from a maintenance perspective because the client tier could reside on the desktops of hundreds or thousands of client machines. For example, the size and mobility of a sales force could make uniform upgrades on all the individual laptops virtually impossible.

Database schema switching costs are high. The fat clients directly access the database via JDBC, SQL/J, or ODBC. This means that the clients are dealing directly with the underlying database schema. If you ever decide to migrate the schema to handle new business processes, you need to redeploy each client.

Database type switching costs are high. The fat clients are bound to a database API, such as a relational database API or an object database API. If you ever decide to switch between database types (such as switching from a relational database to an object database), you must not only redeploy each client, but you must drastically change the client code to suit the new database type.

Business logic migration costs are high. Changing the business logic layer involves recompiling and redeploying the client tier.

Database connection costs are high. Every database client needs to establish its own database connection. These connections are limited in number and costly to establish. When clients are not using the database, the connection is often still held and cannot be used by others.

Network performance suffers. Each time your business logic performs a database operation, you need to make a number of roundtrips across the physical boundary separating the business logic layer and the data layer. If this barrier is a network boundary, it could severely hamper the total time a database operation takes—and it could clog the network, reducing the amount of bandwidth other users have.

Pushing Some of the Business Logic Layer into the Data Layer

More recently, deployments have begun to combine parts of the business logic layer with the data layer into a separate tier. This is illustrated in Figure 1.7. If you think of the first tier as a "client" and the second tier as a "server," this architecture effectively makes the client "thin" and the server "fat."

To actualize this scenario, you push some of your business logic (usually your persistence-related data logic) *within the database*. Databases allow you to execute logic within the database's context by writing a suite of modules known as *stored procedures*. By pushing select parts of your logic into stored procedures, you gain a number of scalability and performance enhancements. For one, because you are pushing some logic inside of the database, the network round trip from the logic to the database is minimized. Rather than performing n database queries, you can call one procedure kept inside the database, which performs n queries for you. This saves you many round trips, increasing the speed

Go back to the first page for a quick link to buy this book online!

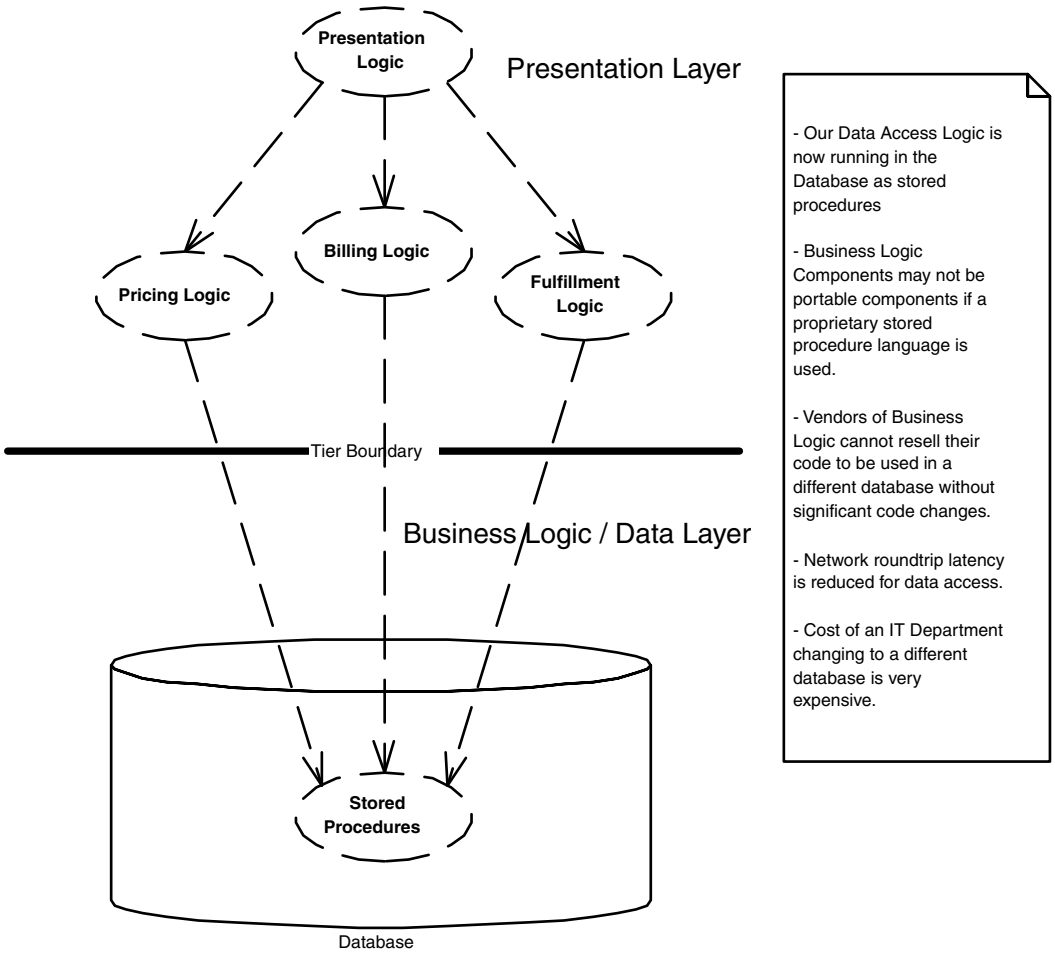

Figure 1.7 Pushing data access logic into the second tier in a two-tier architecture.

of database operations. It also reduces total network traffic, enabling other clients to perform network operations more speedily.

Thus, having database-specific logic within the database does enhance performance and increases the overall deployment scalability. Most of the other problems with two-tiered approaches listed previously, however, still apply. And while the development of stored procedures is an important step forward, they do not solve every problem. In fact, they add some problems of their own. For example, stored procedure languages are often proprietary, which ties clients to a particular database. The whole point of having a database bridge such as ODBC or JDBC, allowing any database to plug in, was defeated. Now it becomes even harder to plug in a database of a different genre, such as an object database. It

Go back to the first page for a quick link to buy this book online!

should be noted, however, that Java is being used more and more as a stored procedure language, which enhances the portability of stored procedures.

N-Tier Architectures

An N-tier architecture adds one or more tiers to the two-tier model. In N-tier deployments, your presentation layer, business logic layer, and data layer are separated into respective physical tiers. With four or more tiers, you decompose each of these layers even further to allow various parts of your system to scale independently.

A concrete example of an N-tier architecture is a three-tiered Web-based deployment, as illustrated in Figure 1.8. A three-tiered Web-based deployment is typically broken down as follows:

Your presentation tier runs within the address space of one or more Web servers. It consists of Java servlets, scripts to customize look-and-feel (such as Active Server Pages, Java Server Pages, etc.), and workflow logic that ties things together.

Your business logic tier runs within the address space of one or more *application servers*. Application servers are necessary to provide a suitable containment environment for the business logic components to run in. The application server also manages these components efficiently and provides a number of services to the components. For example, the application server provides a database access layer for the business components, allowing the business components to persist data to and load data from the data tier. The application server is also responsible for making the business components available to be used, instantiating them as necessary.

Your data tier consists of one or more databases and may contain data-related logic in the form of stored procedures.

Note that a single-company Web deployment is only one example of a viable N-tier deployment. Another example is a banking system that has many tiers, with each tier representing a department of a bank. Another example is a control system that does not have a graphical user interface, yet also requires several tiers. Yet another example is a distributed deployment that spans company boundaries, where several companies work together to provide enhanced value (such as an online purchasing store that involves UPS for shipping logic and VISA for credit card logic).

N-tier architectures have the following characteristics:

Deployment costs are low. Database drivers are installed and configured on the server-side, rather than on client machines. It is much cheaper to deploy and configure software in a controlled server-side environment than to deploy

Go back to the first page for a quick link to buy this book online!

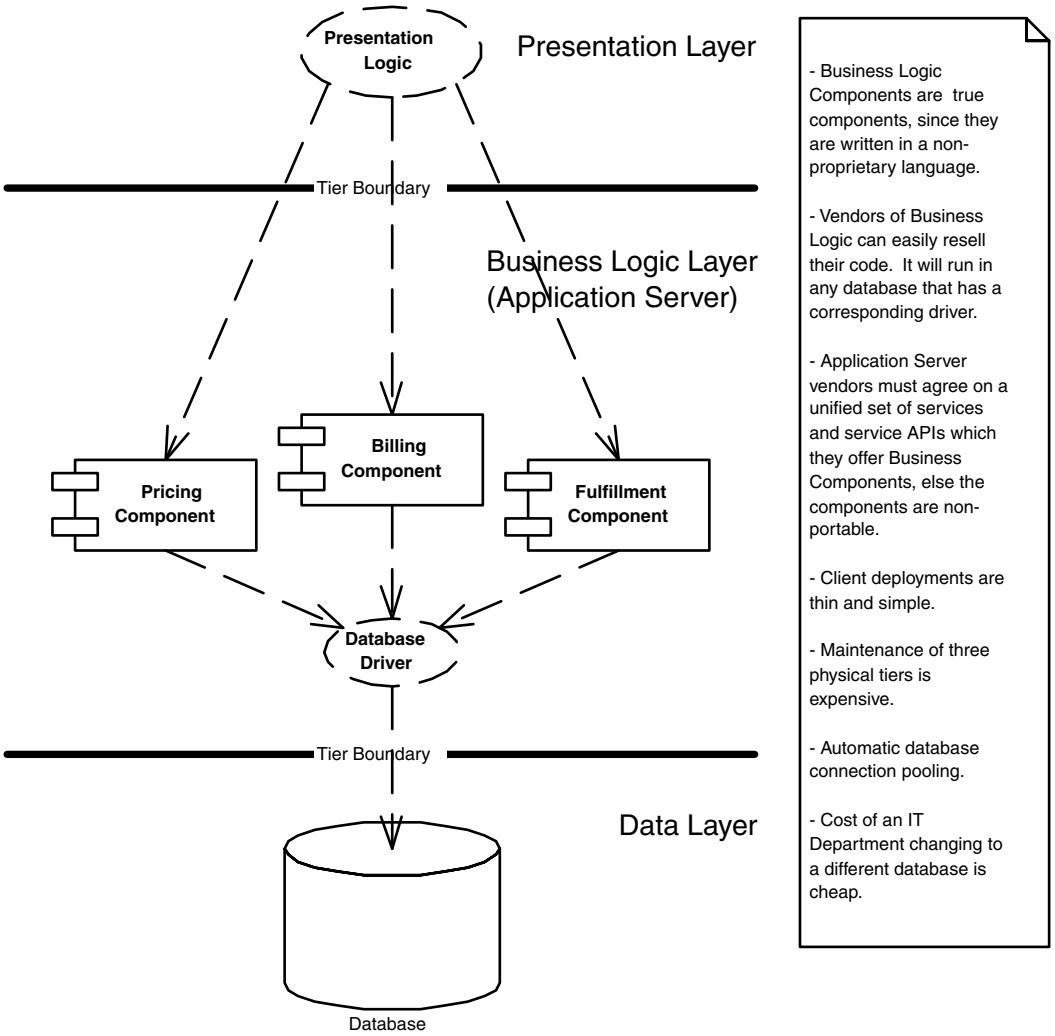

Figure 1.8 An N-tier architecture.

software on (for example) thousands of salesmen's laptops, or on thousands of end user terminals.

Database switching costs are low. Clients no longer access the database directly, but rather go through the middle tier for data access. This enables you to migrate database schemas, change to different database drivers, or even change your persistent storage type without re-deploying clients.

Business logic migration costs are low. Changing the business logic layer may not necessitate recompiling and redeploying the client tier.

Go back to the first page for a quick link to buy this book online!

You can secure parts of your deployment with firewalls. Many businesses are sensitive about protecting their data, yet they do not want to hamper a deployed application. For example, in a Web-based deployment, businesses may not want to expose their business layer directly to outside users. Yet the deployment must expose the presentation layer to outside users so that they can hit the Web site. A solution is to place a firewall in between the presentation and business logic tiers, as shown in Figure 1.9

Resources can be efficiently pooled and re-used. With an N-tier architecture, connections to external resources can be managed very efficiently. *Resource pooling* exploits the fact that clients are often doing other things

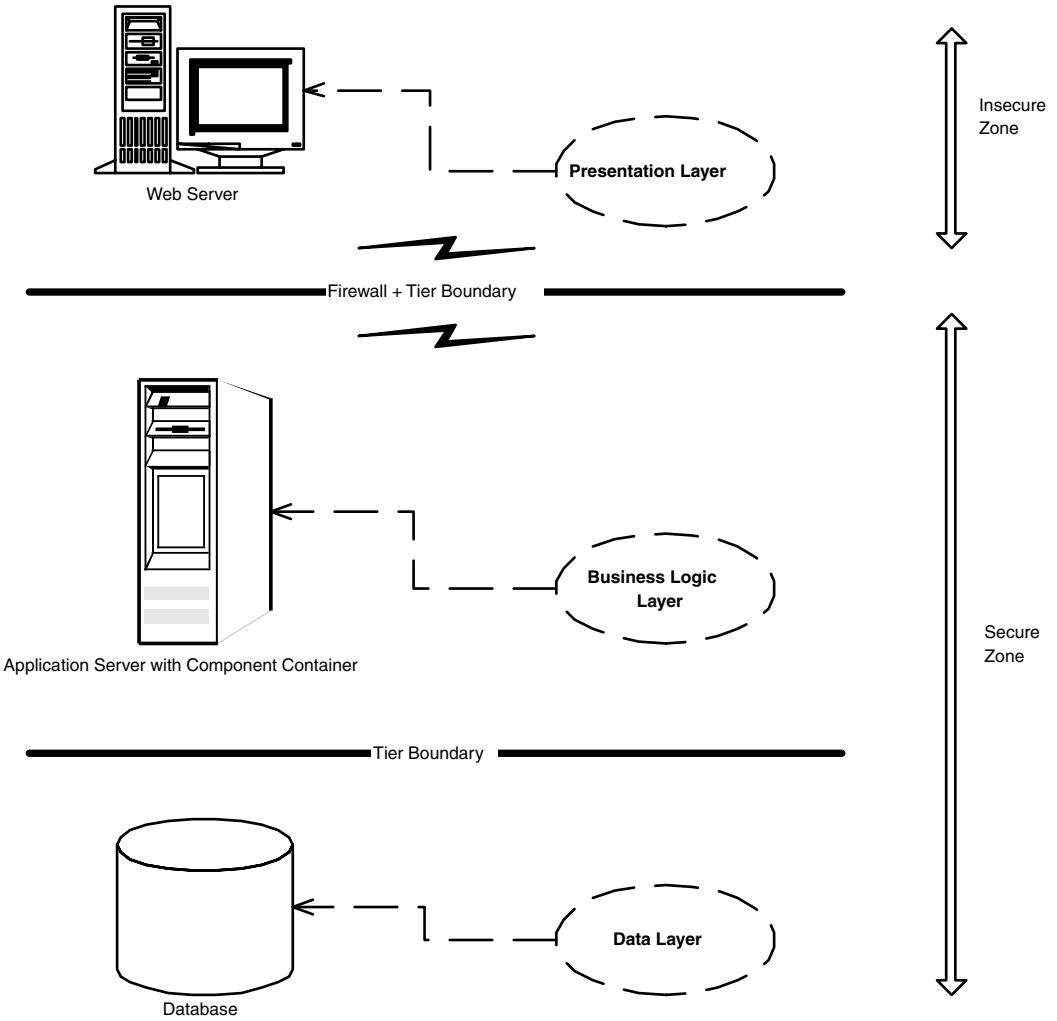

Figure 1.9 Security reasons often necessitate three-tier divisions.

Go back to the first page for a quick link to buy this book online!

besides using resources, such as rendering a graphical user interface. Rather than your business components acquiring and releasing connections to resources (such as databases), the resources can be pooled and re-used for different client requests. The resulting set of database connections required is often far less than the total number of components in a deployed system. Because database connections are very expensive, this paradigm increases the overall scalability of the deployment. Furthermore, connections to resources do not need to be re-established continuously, improving application performance. Resource pooling can be applied to other resources as well, such as socket connections and threads. In fact, with an N-tier architecture, the business components themselves can be pooled and reused by multiple clients. Pooling of components means you don't need a dedicated component for each client, as is true with two-tier thick clients. The pooling of resources is illustrated in Figure 1.10.

Each tier can vary independently. For example, you can add database images while minimizing changes and recompilations of the other tiers.

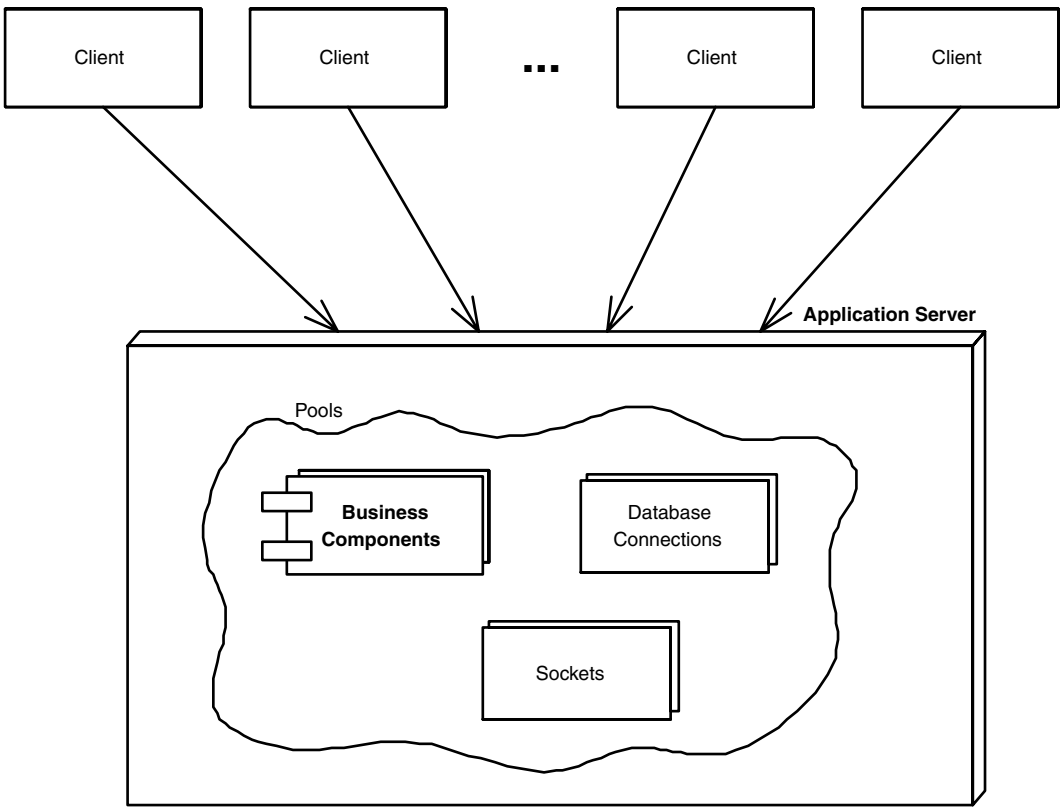

Figure 1.10 Pooling of resources in an N-tier deployment.

Go back to the first page for a quick link to buy this book online!

Performance slowdown is localized. If one tier is overloaded, the other tiers can still function properly. In a Web deployment, users may able to view your front page even though the application server is overburdened.

Errors are localized. If a critical error occurs, it is localized to a single tier. The other tiers can still function properly, gracefully dealing with the situation. For instance, if your application server crashes, your Web server could report a "site down" page to client browsers.

Communication performance suffers. Since the tiers are physically separate, they must communicate across process boundaries, across machine boundaries, or across enterprise domain boundaries. This results in high communications overhead. Only by designing your distributed object application properly can you have an efficient deployment. Unfortunately, this often necessitates being very aware of tier boundaries and reduces the location transparency of your application code.

Maintenance costs are high. You are deploying in three or more physically separate tiers. Software installation costs, software upgrade costs, redeployment costs, and administration costs increase significantly.

Making Multi-tier a Reality

The benefits of multi-tier deployments do not come for free. Someone, at some point in time, needs to write the code that will:

- Broker method requests
- Perform resource pooling
- Handle the lifecycle of your components
- Handle the logic to load-balance communications between each tier
- Deal with the ramifications of two or more clients concurrently accessing the same component
- Reroute client requests to other machines in case of failure
- Provide a secure environment so that unauthorized clients cannot sabotage the system state
- Provide monitoring software that might page a system administrator in case of catastrophic problems
- Authorize users to perform secure operations
- And much, much, *much* more.

Each of these issues is a separate service that needs to be addressed for serious server-side computing. And each of these services requires a lot of thought and a lot of network middleware plumbing to resolve. Often times, companies

Go back to the first page for a quick link to buy this book online!

build in-house proprietary frameworks that give some of these services. Those companies are setting themselves up for failure, because high-end application servers are hideously complicated to build and maintain, require expert-level knowledge of middleware, gain support of industry-standard tools, and are completely orthogonal to most companies' core business. Why not build instead of buy?

The *application server* was born to let you buy these middleware services, rather than build them yourself. Application servers provide a runtime environment for server-side components. They provide component developers 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.

Server-Side Component Architecture Solutions

It has been a number of years now since the idea of multi-tier server-side deployments surfaced. Since then, multitudes of application servers have begun to appear on the market. These application servers provide a usable runtime environment in which components can execute and provide the needed middleware services (such as resource pooling and security) for reliability and scalability. But unfortunately, up until now there has not been a definition of what a middle-tier component really is. Because of this, each application server has been providing component services in a non-uniform, proprietary way. This means that once you bet on an application server, your code is locked into that vendor's solution. This greatly reduces portability and is an especially tough pill to swallow in the Java world, which promotes portability. It also hampers the commerce of components because a customer cannot combine a component written to one application server with another component written to a different application server.

From this, the need for a standard architecture for server-side components has arisen. This architecture needs to craft a well-formed interface between the application server, which contains the components, and the components themselves. These interfaces allow the components to be managed in a portable way, rather than a proprietary one. The component vendors should be able to focus on the business logic of the problems being solved and not worry about external overhead such as resource pooling, networking, security, and so on. These necessary elements of enterprise-class deployments should be externalized to application server vendors, which should provide these common services to all component developers. The goal here is rapid application development of server-side deployments, allowing developers to leverage preexisting network middleware while still writing portable server-side components. This will allow components

Go back to the first page for a quick link to buy this book online!

to be switched in and out of various application servers, without having to change code or potentially even recompile the components themselves.

To address the need, technology advances have been made to compensate for the lack of server-side component architectures. The most popular standards have emerged from Microsoft, Sun Microsystems, and the Object Management Group (OMG). We now outline each of these architectures.

Microsoft's Distributed interNet Applications Architecture

Microsoft has recently consolidated its multi-tier vision into the Windows Distributed interNet Applications Architecture (DNA). This architecture combines the following:

Windows NT, the underlying operating system that provides a runtime environment for all Microsoft technology

Distributed COM, a core technology that promotes interface/implementation separation and language independence and that allows for distributed components

Microsoft Message Queue (MSMQ), a message-queuing product for asynchronous communications between components

Microsoft Transaction Server (MTS), an application server product that manages components

Microsoft Wolfpack, software for server clustering

Microsoft SQL Server, a relational database store

Microsoft Internet Information Server, its Web server, which includes *Active Server Pages (ASP)* scripts, which can be used to coordinate interactions with client browsers

Microsoft Management Console, a tool for deployment and administration

Windows DNA is a server-side development platform that incorporates all these products and more. Note that these technologies are also evolving as we speak—specifically, Distributed COM, MTS, and parts of MSMQ are evolving into a new, combined technology called *COM+*. COM+, along with the ancillary Windows DNA services, allows vendors to build and deploy server-side components with Microsoft technology. Needs such as resource pooling, database transactions, asynchronous communications, and security are all handled implicitly by the underlying distributed object infrastructure.

Microsoft's N-tier vision is quite compelling because it's a one-vendor distributed object framework. Rather than assembling products from competing vendors, developers can work with tools written by a single source. But unfortunately,

Go back to the first page for a quick link to buy this book online!

developers who integrate to Microsoft's N-tier vision must tie themselves to a particular vendor (Microsoft), which allows only the Microsoft platform as a deployment scenario. This makes it difficult if not impossible to perform deployments on mainframes or high-end UNIX machines, which have a significant amount of services available for high-reliability deployments, as well as much higher processor and resource scalability. And while the number of server-side Win32-based workstations is climbing incredibly fast, Windows NT is scalable to only four processors (or reportedly 16 processors with NT 5.0). This means a Windows-based deployment has a very high machine maintenance cost due to the sheer number of machines required. Hopefully, this will change in the future.

Sun Microsystems's Java 2 Platform, Enterprise Edition (J2EE)

Sun Microsystems has also realized the need for a server-side component architecture. Many Java component vendors have been clamoring for a full server-side N-tier story that is Java-based; the Java language itself is well suited for the server. Client-side Java has many problems, such as inconsistency with user interfaces between platforms, slow speed of these user interfaces, and wrong versions of Java Virtual Machines running on client machines. But for the server side, Java is the ideal language. Server-side deployments run in a controlled environment, which means that the correct version of the Java programming language will always be used. The speed of Java is also a less significant issue on the server because often typically 80 percent or more of an N-tier application's time is spent at the database or networking level.

Java is also a very convenient language to write server-side components in, if for one reason alone: The server-side market is dominated by UNIX machines and mainframes. This means a cross-platform language for writing server-side components adds huge value because a developer can write a component once and deploy it on any customer's existing server-side environment. This means that customers with legacy applications (such as programs written in COBOL on mainframe systems) have a well-defined migration path to e-commerce and other modern business processes.

Thus, to enable server-side computing in Java, Sun has produced a complete development platform 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. The cornerstone of J2EE is Enterprise JavaBeans (EJB), a standard for building server-side components in Java.

J2EE simplifies many of the complexities surrounding the construction of a scalable, component-based server-side application, and it is very analogous to Windows DNA. The most notable exception is that J2EE is a specification,

Go back to the first page for a quick link to buy this book online!

whereas Windows DNA is 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. As we will see in this book, though, incompatibilities between vendor products will arise—some problems due to ambiguities with specifications, other problems due to the human nature of competition.

The Object Management Group's CORBA Standard

The Object Management Group (OMG) is an open standards body whose mission is to standardize on industry guidelines and build object management standards to provide a common framework for application development and to promote a software component marketplace. The OMG's two most notable achievements have been the Common Object Request Broker Architecture (CORBA), as well as the Internet Inter-ORB Protocol (IIOP). We will cover these technologies in Chapter 11. For now, you should know that CORBA/IIOP are standards that promote portable distributed objects. If you write your objects to use CORBA/IIOP, you can leverage prewritten middleware supplied by vendors. For example, a vendor could provide a security service, a transaction service, or a persistence service, and you can use those services in your applications without writing that middleware yourself.

Like J2EE, CORBA is a specification, not a product. Vendors (such as Inprise, Iona, and IBM) implement the CORBA specification with their products. The big distinction between CORBA and J2EE is that (until now) CORBA has had only the notion of an object and had no notion of a deployable server-side component that is automatically managed by an application server. The OMG, however, has recently addressed this need by bolstering CORBA with a *CORBA Components* proposal. This proposal defines the notion of a component in CORBA—an entity that is deployable within an application server, is managed by the application server, and is sellable in a component marketplace, resulting in rapid application development through the divide-and-conquer strategy.

CORBA Components is still in its infancy at the time of this writing. In fact, there is great debate within the OMG over whether CORBA Components should even be standardized at all. The main reason behind this is that J2EE contains built-in support for CORBA (as we will see). The real danger is if the OMG creates yet a third server-side component standard beyond Windows DNA and J2EE. This is definitely a specification to keep an eye on in the future.

Go back to the first page for a quick link to buy this book online!

The Java 2 Platform, Enterprise Edition

Now that you have seen the three major server-side architectures out there, let's turn our attention to the Java 2 Platform, Enterprise Edition (J2EE), of which EJB is an integral part.

Why J2EE?

To understand why J2EE is necessary, we need to take a look back in time and chart the evolution of the Java platform.

JDK Efforts

Sun Microsystems first focused on building a robust Java Development Kit (JDK). The JDK became the de facto reference implementation of the Java platform. Not much exciting development was happening here for server-side middleware support.

Enterprise API Efforts

Sun Microsystems then recognized the power of Java on the server and began to develop several Enterprise APIs that provided enterprise-level services for server-side deployments. These services include naming and lookup services, transaction services, and the Enterprise JavaBeans (EJB) 1.0 API. Sun also began to develop elements of Consumer Java, which provide functionality for consumer devices to interoperate.

Several months after EJB 1.0 was finalized, the first EJB-based application servers began to arrive on the market (BEA's *WebLogic* was the first). These application servers took advantage of the other Enterprise APIs as well, such as the ones for naming and lookup services, for transactions, and for database operations. The early application servers served as a great feedback device to Sun because they highlighted many problems with the Enterprise APIs. These include the following:

Ambiguities. The unspecified sections of the Enterprise API specifications (particularly EJB 1.0) hampered portability of components. This is unacceptable due to the "Write Once, Run Anywhere" paradigm of Java.

Poor synchrony between Enterprise APIs. Each of Sun's Enterprise APIs was related, yet each was being developed independently. Sun was not communicating the typical purpose of the enterprise APIs together as a suite, and the specifications themselves were not very intertwined.

Go back to the first page for a quick link to buy this book online!

A moving target. Each of Sun's Enterprise APIs was evolving separately and had new version numbers coming out all the time. This made it somewhat difficult to program using EJB because EJB depends on those Enterprise APIs. What versions of each API should be used? This was unspecified, leading to nonportable code between application server vendors.

No way to test application server compatibility. When a vendor wrote a product to the EJB 1.0 standard, there was no way to test whether this product was compatible with Sun's specification. Similarly, a consumer had no information about whether a vendor was compliant with EJB 1.0.

No reference implementation. The Enterprise APIs were simply specifications; Sun did not provide a default reference implementation. Thus, application developers had no low-end enterprise platform against which to test code. By way of comparison, programmers who did not leverage the Enterprise APIs could simply use Sun's JDK reference implementation. Thus, the Enterprise APIs needed an analog to the JDK.

Platform Separation

Realizing the problems with the Enterprise APIs, Sun Microsystems has recently taken a major leap forward in solving the problems we alluded to previously by issuing *three different* Java platforms. Each platform is a 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 very 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 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 addresses all the problems raised when Sun developed the Enterprise APIs, including the following:

Fewer ambiguities. Sun has addressed a great deal of the incompatibilities in EJB 1.0 with the EJB 1.1 specification (covered in Appendix D). This specification corrects bugs and other ambiguities that had been restricting portability.

Enterprise API synchrony. Each of the Enterprise APIs has a clear role in J2EE, as defined by Sun's *J2EE Application Programming Model* document.

Go back to the first page for a quick link to buy this book online!

This should clear up any confusion about how you can use the Enterprise APIs together to enable a server-side deployment. Similarly, the Enterprise APIs are evolving in harmony, as the specifications leverage each other in an intertwined fashion.

Locked-down revisions. Enterprise Java is no longer a moving target, as Sun has locked down versions of each Enterprise API specification and bundled 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.

Test suite. As we mentioned above, there is no way for a vendor to know if he or she is implementing EJB 1.0 properly. Sun has fixed this with J2EE, as it provides a test suite for vendors to test their products against. If a product passes the tests, Sun will issue a compliance brand, alerting customers that the vendor's product is indeed J2EE-compliant.

Reference implementation. To enable developers to write code against J2EE as they have been doing with the JDK, Sun is providing 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.

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. The technologies included with J2EE are shown in Figure 1.11; they include the following:

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 promotes the spawning of a component marketplace, where vendors can sell reusable components that can be purchased to help solve business problems. EJB is the cornerstone for J2EE and is covered throughout this book.

Java Remote Method Invocation (RMI) and RMI-IIOP. RMI allows for interprocess communication and provides other communication-related services. RMI-IIOP is a portable extension of RMI that can use the Internet Inter-ORB Protocol (IIOP) and can be used for CORBA integration. RMI is covered in Appendix A. RMI-IIOP is covered in Chapter 11.

Java Naming and Directory Interface (JNDI). JNDI identifies the locations of components or other resources across the network. JNDI is covered in Appendix B.

Java Database Connectivity (JDBC). JDBC is a relational database bridge that allows for relatively portable database operations. JDBC is used in Chapter 8.

Go back to the first page for a quick link to buy this book online!

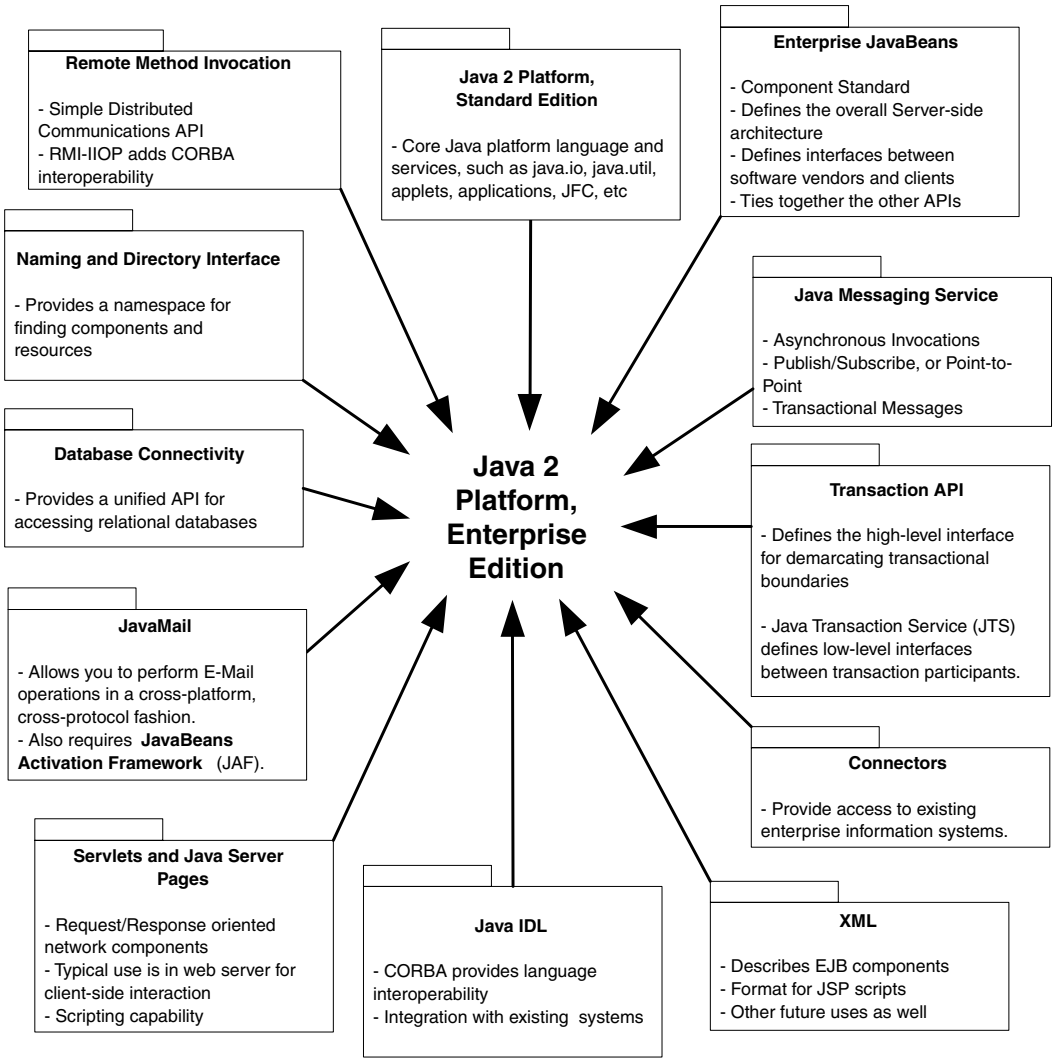

Figure 1.11 The Java 2 Platform, Enterprise Edition.

Java Transaction API (JTA) and 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 asynchronous distributed object communications. We do not cover JMS in this book, as Sun Microsystems has not integrated JMS with EJB yet.

Java Servlets and Java Server Pages (JSPs). Servlets and JSPs are networked components which are ideally suited for request/response oriented computing, such as interacting with clients over HTTP. We illustrate using servlets with EJB in Chapters 12 and 15.

Go back to the first page for a quick link to buy this book online!

Java IDL. Java IDL is Sun Microsystem's 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. Thus, J2EE is fully compatible with CORBA, rounding out the Java 2 Platform, Enterprise Edition completely. Java IDL is explained in Chapter 11.

JavaMail. The JavaMail service allows you to send e-mail 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 e-mail 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.

Connectors. Connectors make J2EE well suited to integrate with mainframe systems running high-end transactions (such as those deployed with IBM's CICS), as well as Enterprise Resource Planning (ERP) systems. Connectors will be included in a future release of J2EE.

The Extensible Markup Language (XML). Several J2EE technologies (such as EJB 1.1 and JSP) depend on XML as a meta-markup language for describing content. We cover XML in Appendix C, and we cover how XML and EJB are related in Appendix D.

The Java 2 Platform, Enterprise Edition (J2EE) builds on the existing technologies in the Java 2 Platform, Standard Edition (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 will likely result in significant industry consolidation in the Enterprise Java space, with a few players emerging from the pack as leaders.

At the time of this writing (May 1999), Sun Microsystems had not finalized the Java 2 Platform, Enterprise Edition (J2EE) specifications. Sun expects to complete them by Fall 1999. This means you are likely to see J2EE products emerging in the marketplace sometime in the year 2000.

To understand more about the real value of J2EE, we now explore each API in more detail.

Enterprise JavaBeans (EJB)

EJB is a component architecture that allows developers to quickly create scalable enterprise applications. It provides complex middleware enterprise features at no cost to application developers. With EJB, you can now focus your energies

Go back to the first page for a quick link to buy this book online!

on writing the applications that solve real-world problems, rather than on all the overhead that goes with distributed server-side systems. You can think of EJB almost as a suite of common functionality that most applications need. Through the EJB model, you won't need to reinvent those wheels.

To this, EJB adds legacy integration flexibility. You can rewrite your components from scratch to be EJB-compliant, or you can wrap an existing component; there's no need to build your enterprise system from the ground up. Many vendors of EJB products (IBM and BEA come to mind) provide well-defined migration paths for existing legacy system customers to jump aboard the EJB bandwagon without abandoning their existing enterprise information system. This is critical for companies that may have their entire business running in millions of lines of COBOL running off CICS systems.

EJB also offers "plug-and-play" enterprise features. With EJB, you barely need to know anything at all about middleware to construct components designed to run in a scalable, multi-tier architecture. Rather than writing to middleware APIs (which is the old CORBA style of distributed object computing), your components gain middleware services *implicitly and transparently* from the EJB server, without writing one line of middleware code yourself. The application server implicitly handles transactions, persistence, security, state management, component lifecycle, threading, and more.

The cross-platform, cross-vendor nature of the EJB standard is another extremely important benefit that EJB brings to the table. It means that there's now a common standard for distributed component architectures to which all vendors can integrate. It also means that there's going to be intense competition in the EJB market. EJB levels the playing field by defining one standard component API. This means you can expect the new EJB-compliant products that emerge in the market to be both reliable and high-performing because middleware vendors can be compared in the same light. There is one standard, allowing the vendors to focus on the quality of their EJB products. The standard also allows application developers to focus on writing best-of-breed applications, rather than supporting every brand of middleware out there. This is the promise of EJB.

Remote Method Invocation (RMI) and RMI-IIOP

The EJB standard depends on several other APIs in J2EE; the most visible of these is Java Remote Method Invocation (RMI). RMI is a mechanism for invoking methods remotely on other machines. It is tightly integrated with the Java language itself; if you know Java, it won't take much to learn RMI. EJB relies on RMI as a communications API between components and their clients.

RMI is seamless—you almost don't even know you're using it when you program your distributed application. RMI allows Java programmers to communicate in

Go back to the first page for a quick link to buy this book online!

a distributed object fashion using an almost identical programming style to coding a standalone Java applet or application. That is the beauty in RMI—it abstracts networking issues away from you, such as marshalling parameters, handling machine byte-order, and so on, which are all necessary for network communications. RMI also contains other niceties such as dynamic class downloading, automatic activation of remote objects, and a distributed garbage collector to clean up unused remote objects. Because many Java programmers already know RMI, we cover it in Appendix A.

RMI-IIOP

Sun Microsystems (in a joint venture with IBM and others) has more recently developed a more portable version of RMI as well, which can use the Object Management Group's (OMG's) Internet Inter-ORB Protocol (IIOP) as a communications protocol. IIOP is a very robust protocol, and it adds several qualities of service than RMI can support in its native protocol. IIOP is also necessary for J2EE deployments to integrate with CORBA systems. We explain RMI-IIOP in Chapter 11.

Java Naming and Directory Interface (JNDI)

The Java Naming and Directory Interface (JNDI) is a standard for *naming and directory services*. Enterprise JavaBeans relies on JNDI for looking up distributed components across the network. JNDI is a key technology required for client code to connect to an EJB component.

JNDI hinges on the notion of a *directory service*. A directory service stores information about where resources (such as components) reside, as well as other information such as username and passwords. In EJB, when client code requests access to a component, directory services are used to locate and retrieve a component to service that client (see Figure 1.12). You can think of the directory service as a matchmaker that connects clients to components.

Historically, there are many types of directory services, as well as protocols to access them. Some examples include Novell's NDS and the Internet standard LDAP. These are all competing standards, so each type of directory service is accessed differently. And each directory service stores information in a proprietary way. If you write an application that accesses a particular directory service, it becomes a real mess if you want to switch to a different directory service.

The Java Naming and Directory Interface (JNDI) solves this problem by bridging the gap between different directory services. With JNDI, you can write portable naming and directory service code, rather than writing nonportable code that works only with a particular directory service or standard. You can do this because JNDI abstracts your code from any particular directory service, enabling

Go back to the first page for a quick link to buy this book online!

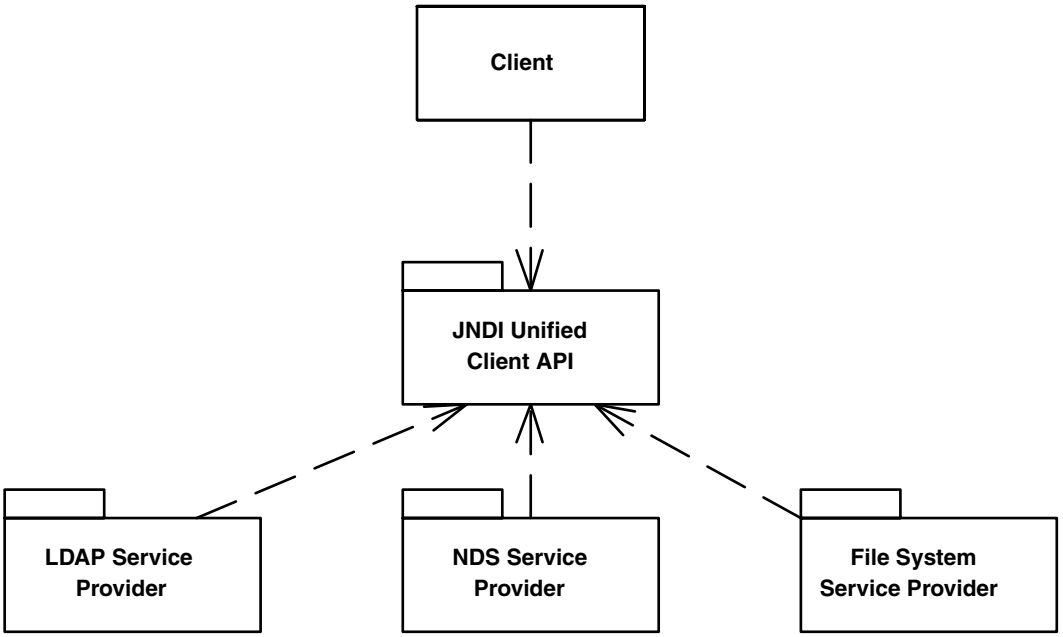

Figure 1.12 The Java Naming and Directory Interface.

you to simply plug in a different kind of directory service without changing your code. In the future, if new directory service standards emerge, JNDI will most probably be updated to reflect those changes.

EJB strictly depends on JNDI, so we've devoted a section to it, but we've left it as Appendix B because some of you may already be familiar with JNDI.

Java Database Connectivity (JDBC)

The Java Database Connectivity (JDBC) 2.0 package is a standard Java extension for data access that allows Java programmers to code to a unified relational database API. By using JDBC, Java programmers can represent database connections, issue SQL statements, process database results, and more in a relatively portable way. Clients program to the unified JDBC API, which is implemented by a *JDBC Driver*, an adapter that knows how to talk to a particular database in a proprietary way (see Figure 1.13). JDBC is similar to the Open Database Connectivity (ODBC) standard, and the two are quite interoperable through JDBC-ODBC bridges. JDBC 2.0 contains built-in support for database connection pooling, further enhancing the database independence of your application code.

Because EJB does not strictly depend on JDBC, we do not cover JDBC explicitly in this book. We do show how to use JDBC in the EJB world; see Chapter 8 for that information.

Go back to the first page for a quick link to buy this book online!

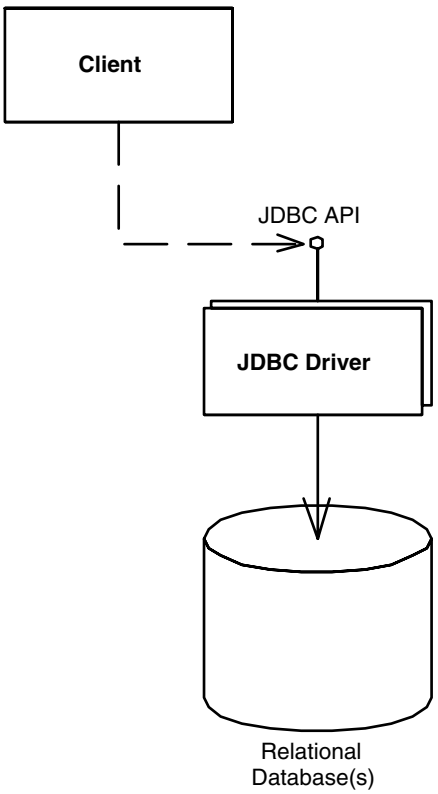

Figure 1.13 Java Database Connectivity.

Java Transaction API (JTA) and Java Transaction Service (JTS)

A *transaction* is a unit of work that makes a set of guarantees about its execution. For example, one guarantee that transactions give you is that any code executing within the scope of a transaction is guaranteed either to all be executed or to not execute at all. Transactions are one of the most useful constructs EJB brings to the table, and they are required to maintain a consistent system state. Transactions allow for multiple users to be modifying the same data, yet each to be isolated from one another—in essence, a very sophisticated form of synchronization.

To facilitate transactions, Sun Microsystems has produced two APIs: the *Java Transaction API* (*JTA*) and the *Java Transaction Service* (*JTS*). These two products embody how transactions can be performed in Java (see Figure 1.14).

JTA is a high-level transaction interface that your applications use to control transactions. You need to understand how the JTA works in order to perform transactions in Java.

Go back to the first page for a quick link to buy this book online!

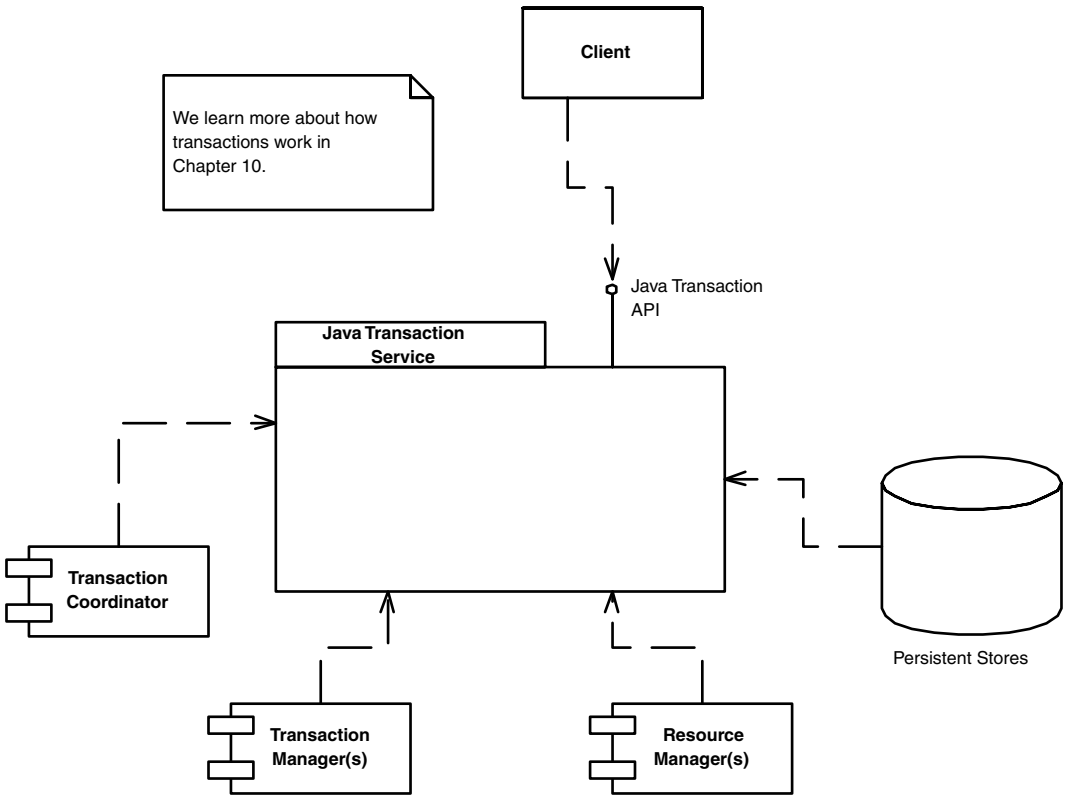

Figure 1.14 Java Transaction API and Java Transaction Service.

JTS is a set of low-level transaction interfaces that EJB uses behind the scenes (your client code doesn't directly interact with JTS). JTS makes it possible for multiple vendors to collaborate when performing transactions in a distributed, heterogeneous environment. JTS is based on the *Object Transaction Service* (*OTS*), which is part of CORBA.

Enterprise JavaBeans strictly depends on JTA, but it does not depend on JTS. As an application programmer, you may need to learn the JTA for performing advanced transaction control. We'll go into more detail about the JTA, as well as touch on the JTS, in Chapter 10.

Java Messaging Service (JMS)

A *messaging service* allows for distributed objects to communicate in an asynchronous, reliable manner. By passing messages asynchronously rather than synchronously, the overall system scalability is increased. Processes can respond

Go back to the first page for a quick link to buy this book online!

to messages at their leisure, and they could potentially be offline when a message is initially sent.

You can perform asynchronous messaging by using an *Enterprise Messaging Product* (these are grouped together under the title *Message Oriented Middleware*, or *MOM*). These messaging systems are very proprietary. As with directory services, choosing a MOM vendor often entails binding yourself to a particular vendor.

Sun Microsystems has released an API called the *Java Messaging Service* (*JMS*) that can be used as a portable messaging service. By using a common API, distributed objects can communicate in a scalable, transactional, fault-tolerant, asynchronous, and, most important, vendor-independent manner. The JMS is illustrated in Figure 1.15.

Unfortunately, the EJB 1.1 specification does not define the integration points necessary to fully leverage JMS in EJB components, so we do not explicitly cover JMS in this book. The EJB 2.0 specification is expected to address this outstanding issue.

Java Servlets and Java Server Pages (JSPs)

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

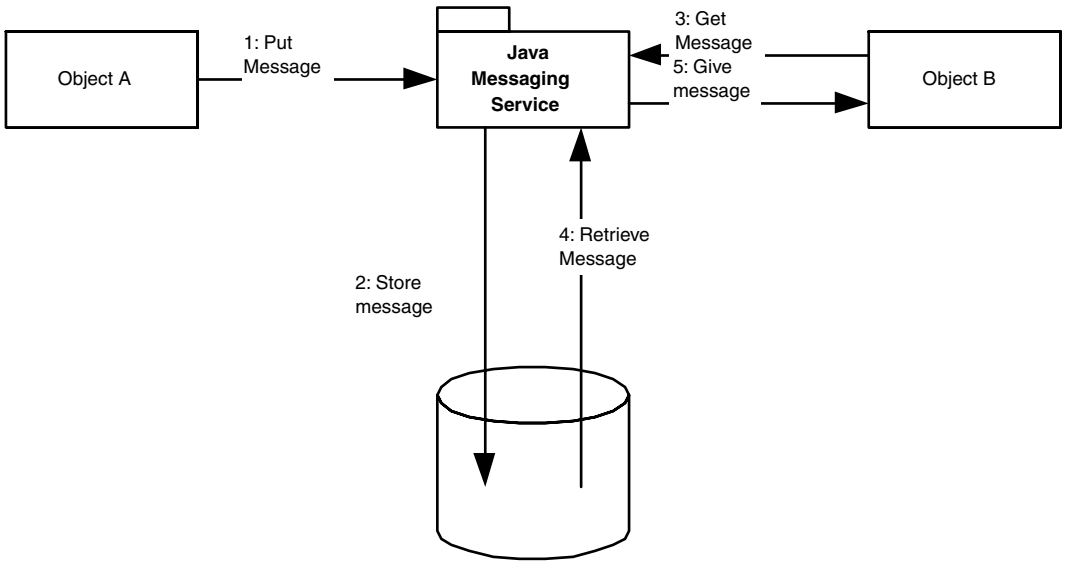

Figure 1.15 Java Messaging Service.

Go back to the first page for a quick link to buy this book online!

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 to an e-commerce catalog. Note, however, that servlets are not necessarily tied to Web servers, and they can be used as generic request/response-oriented components. 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.

Java Server Pages (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, but 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 connecting servlets with EJB components in Chapters 12 and 15.

Java IDL

As we've mentioned, CORBA is a massive middleware effort developed by the Object Management Group (OMG). The OMG represents hundreds of companies that have invested in this cross-platform architecture. CORBA is language independent; it doesn't matter what you code your programs in with CORBA, so long as CORBA supports the language in which you're developing.

Java IDL is an implementation of the CORBA specification in Java, and it allows for connectivity and interoperability with heterogeneous objects. Java IDL is one specific implementation of CORBA; there are many such implementations. And while Java IDL does not implement everything that is specified in CORBA, Java IDL ships free with the Java 2 platform. We cover Java IDL briefly in Chapter 11.

JavaMail

Sun's *JavaMail API* allows your applications to use e-mail capabilities. The JavaMail object model is quite simple to use, and it is easy to pick up if you've been programming with Java (it follows similar methodologies, such as exception handling and event handling). Like most of the other J2EE APIs (JNDI comes to mind), JavaMail defines a set of interfaces to which you write your application code, and those interfaces shield your code from the specific protocols or mail service implementations used. Your Internet mail code effectively becomes portable across platforms, as well as across mail protocols. JavaMail also ships

Go back to the first page for a quick link to buy this book online!

CORBA and EJB

EJB and CORBA share much functionality—many of the qualities of service that EJB offers are also in CORBA. In fact, you can think of EJB as CORBA plus standards for how your components should be written and managed, increasing productivity.

A massive amount of development effort has gone into CORBA. It would be a shame if Enterprise JavaBeans did not integrate with CORBA. Fortunately, Sun Microsystems and the OMG are both supporting EJB/CORBA interoperability, and they have produced standards that will allow that to happen.

The EJB/CORBA mapping specification, along with RMI-IIOP, lifts the restriction that EJB must be solely Java-based. You can expose EJB components as CORBA objects, which generalizes EJB and makes it well suited for cross-language interoperability. This is a very powerful idea, which we'll examine further in Chapter 11.

with a set of *convenience classes* to simplify application development, and it also ships with a few *service providers* that implement the most popular mail protocols. JavaMail depends on the *JavaBeans Activation Framework* (*JAF*) to encapsulate message data and to handle interactions with that data. This makes J2EE depend on JAF, meaning that a vendor that provides a J2EE product must provide JAF as well.

Connectors

J2EE is being enhanced to include integration with existing information systems via *connectors*. A connector is a vendor-specific bridge that links an existing system to the Java 2 Platform, Enterprise Edition. Using connectors, you can write cross-platform, cross-vendor J2EE code that leverages existing investments. These existing investments could be Enterprise Resource Planning (ERP) systems, message-oriented middleware (MOM) systems, mainframe systems such as IBM's CICS and IMS, BEA's Tuxedo, or existing legacy databases.

Connectors are useful because they will automatically manage the hairy details of middleware navigation to existing systems, such as handling transaction and security concerns. Sun's connector specification should also provide standard interfaces that vendors can write to, enabling you to plug any connector into any J2EE product, which means your legacy access code should be portable across application servers. Note that connectors are different from the other J2EE APIs because connectors are very vendor-specific (the API for an IBM CICS connector would most certainly be different from an SAP R/3 connector).

When Sun Microsystems completes the connector specification (most likely not until well into the year 2000), you should see a *connector marketplace* emerge,

Go back to the first page for a quick link to buy this book online!

where *connector providers* such as IBM, BEA, SAP, Baan, Peoplesoft, or other companies provide J2EE connectors to specific existing systems. Until then, you have several choices for connecting to existing information systems, such as using a proprietary connector solution (SAP will likely ship one before J2EE connectors are finalized), using CORBA to integrate with existing systems, or growing an in-house solution.

The Extensible Markup Language (XML)

The Extensible Markup Language (*XML*) is a universal standard for structuring content in electronic documents. XML is extensible in that businesses can use XML to craft new structure to their documents as needed. The XML standard does not suffer the version control problems of other markup languages such as HTML because there are no predefined tags in XML—rather, you define your own tags for your business needs. This makes XML the ideal document format for transferring business data electronically, and it has a wide variety of other applications as well. We cover XML in Appendix C.

J2EE uses XML in several ways. Java Server Pages (JSPs) use XML as a data document format for authoring Web script. EJB 1.1 uses XML to describe components, which we cover in Appendix D. Sun may also mandate that J2EE products make APIs available for applications to parse and interpret XML documents as well.

Summary

By now, you should have a solid idea about the issues involved in server-side computing. You should also understand why a server-side component architecture is needed and why it provides the ability to rapidly develop applications without having to write complex middleware services. You've also seen why the J2EE was invented, what the J2EE value proposition is, and the technologies behind J2EE.

So far, we've taken a snapshot of the various APIs that J2EE encompasses. We're just getting started, so hang in there! There are many more interesting concepts to learn ahead.

Go back to the first page for a quick link to buy this book online!

CHAPTER 2

Enterprise JavaBeans Overview

Chapter 1 introduced the motivation behind creating the EJB component architecture and the J2EE development platform for multi-tier deployments. In this chapter, we'll dive into EJB in detail, including the specific benefits EJB products offer to server-side components. This chapter will remain at a fairly high level, reserving the programming details of EJB for Part II.

As we alluded to in Chapter 1, Enterprise JavaBeans takes a *divide-and-conquer* approach to server-side computing. In fact, the EJB standard contracts allow for a collaboration of *six* different parties. Each of these parties is an expert in its own field and is responsible for a key part of a successful Enterprise JavaBeans deployment. Because each party is a specialist, the total time required to build an enterprise-class deployment is significantly reduced. This chapter begins with a discussion of the distinct role that each of these parties undertakes.

You'll then learn about the physical parts of an EJB deployment, including the following:

- What is an enterprise bean? You'll see the types of beans available, and you'll learn about the appropriate times to use them.
- What are the responsibilities of an EJB server and container? You'll learn about the specific middleware services the EJB server and container provide to EJB components.

This is a perfect chapter for people in a hurry who want to know some details about how EJB works. Whether you're a developer, an IT manager, or simply someone who's curious about EJB, this chapter is applicable. It provides a solid background on EJB, without getting into the programming details.

Go back to the first page for a quick link to buy this book online!

Because EJB is such a huge subject, you can expect to encounter many new concepts here. You may need to read this chapter twice to fully grasp the material. And, of course, everything we cover here will be explained in further detail as the book unfolds.

Who's Who in Enterprise JavaBeans

In the world of Enterprise JavaBeans, business solutions are developed in four stages:

1. Vendors of business logic, such as sales force automation vendors, enterprise resource planning vendors, financial services vendors, and e-commerce vendors, can modularize their products into reusable EJB components. Each component has a well-known but limited duty. These components adhere to the EJB specification, which means anyone with knowledge of EJB and Java 2 Enterprise Edition programming should be able to use the components.
2. The business components need to run in a distributed multi-tier fashion for a scalable, enterprise-class deployment. To achieve this, the components need a variety of tricky middleware, such as connection pooling, security, and life-cycle management. Rather than explicitly writing these middleware services over and over again, EJB enables middleware vendors to provide these system-level services in an application server. How do the components leverage these middleware services? In traditional server-side programming, you as a component developer would need to explicitly write to middleware APIs, such as persistence APIs, security APIs, or transaction APIs. In the EJB world, however, things are much easier. An application server provides middleware services for you in a *transparent* way, without your explicitly coding to middleware APIs. This is possible because the EJB specification lays out the rules governing middleware that both components and application servers must follow. For example, your components can get automatic persistence by setting *attributes* on your component that describe the component's need for persistence. The application server understands these attributes because the EJB specification defines them, and the application server complies with the specification. As a component developer, this means your life is simplified greatly because you can rapidly develop persistent components without coding to, debugging, or even understanding persistence APIs. Transparent middleware is a significant achievement because it allows developers to concentrate on application logic, rather than system-level middleware issues.
3. With application servers and reusable components for sale in a marketplace, a customer can purchase these and combine them to solve a business

Go back to the first page for a quick link to buy this book online!

problem in a server-side deployment. Purchased components, though, won't magically work together—especially if different vendors write the components. To address the customer's business needs, a third party (such as a systems integrator) combines these components into a workflow environment. The party makes recommendations for which components need to be purchased and hooks the components together, adding some custom code to the equation. During this process, coding efforts focus on integrating the business logic of the purchased components, rather than dealing with the overhead of the proprietary middleware that each component may use. Each of the components has adhered to the same EJB contracts and so can make use of the same underlying middleware.

4. The entire system is finally deployed, distributed among multiple machines across a network in a multi-tier environment. The system must be maintained and upgraded as it evolves over time.

As EJB matures, this scenario will become more and more of a reality. Assuming vendors stick to the EJB contracts, a world of true plug-and-play Enterprise features can be actualized. And even though it may not be entirely true today, the vision of portable EJB components deployable across heterogeneous application servers will solidify over the next few years.

The Six Parties

The foundations for EJB portability are the contracts and roles that each party must follow. Going with the divide-and-conquer theme, EJB partitions the responsibility of an EJB deployment to up to six different parties. Here is a brief overview of those parties:

The bean provider provides reusable business components that companies can purchase and use to help solve business problems. Beans are not complete applications, but rather are deployable components that can be assembled into complete solutions. An example of a bean provider that ships reusable components today is BEA, which provides the WebLogic Commerce Server 1.7.1 for building e-business applications. In the future, traditional enterprise software vendors (such as SAP and Trilogy) will offer their software as enterprise beans or will provide connectors to their current technology.

The container provider supplies the low-level runtime execution environment needed to run EJB applications.

The server provider supplies the application server logic to contain, manage, and deploy components. Currently there is no distinction between EJB container providers and EJB server providers. Examples of EJB container/server products are BEA's *WebLogic*, Sun Microsystems' *NetDynamics*, IBM's *WebSphere*, Oracle's *Oracle 8i*, Persistence Software's *PowerTier*, and Inprise's *Inprise Application Server*.

Go back to the first page for a quick link to buy this book online!

The application assembler is the overall application architect, perhaps for a specific deployment. He or she is responsible for understanding how various components fit together and writes the applications that use components. He or she may even author a few components along the way. An example of an application assembler is a systems integrator, a consulting firm, or an in-house programmer.

The deployer takes prewritten components, usually chosen by the application assembler, and applies his or her deployment expertise to install the components in one or more application servers.

The system administrator oversees the well-being of a deployed system.

These parties interact as shown in Figure 2.1.

You may be wondering why so many different participants are needed to provide an EJB deployment. The answer to this is that EJB enables companies or individuals to become experts in roles and that division of labor leads to best-of-breed deployments. For example, an EJB deployer does not need to be an expert at *developing* (designing and implementing) enterprise software but does need to be an expert at *deploying* an already developed EJB solution into a particular domain. This empowers EJB deployers to master deploying solutions, and it enables other parties (such as bean providers and application assemblers) to worry about developing solutions.

The EJB specification makes each role very clear and distinct, enabling experts in different areas to participate in a deployment without loss of interoperability. Note that there is also room for some of these roles to be combined as well. For example, the EJB server and EJB container may indeed come from the same vendor. For some of the parties, EJB merely suggests the possible duties that

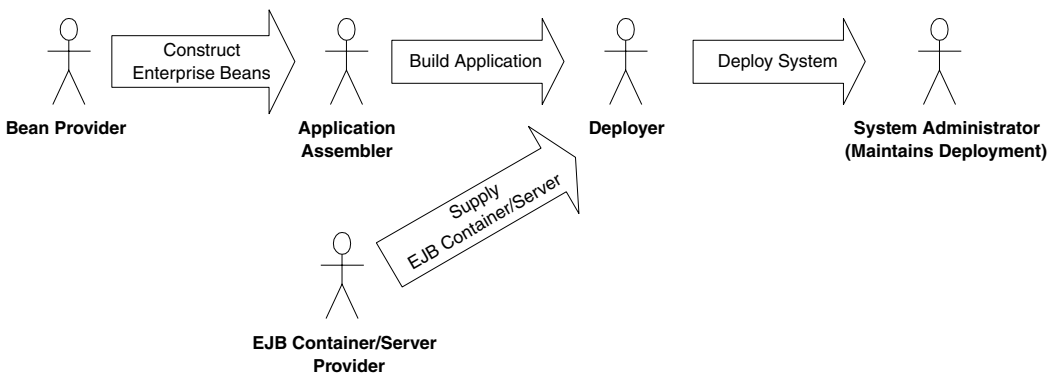

Figure 2.1 The six parties of EJB.

Go back to the first page for a quick link to buy this book online!

the party might assume, 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 strict interfaces and guidelines that must be followed or the entire model 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.

We will now examine the responsibilities of each of the players in the Enterprise JavaBeans realm in more detail.

The Bean Provider

The bean provider is the party who supplies *enterprise beans*—components that expose methods for performing application logic. The bean provider might be a vendor of components that can be resold over and over again on the market—possibly through an indirect sales channel. The bean provider might be an in-house developer as well.

Enterprise beans are distributed, server-side components. They provide useful functionality that can be assembled to form larger applications. Enterprise beans *may* also be reusable components, but this is not guaranteed. Don't believe anyone who tells you that enterprise beans are reusable by definition because that is *false*. You need to design your beans correctly if you want them to be reusable. You need to consider the different applications, domains, and users of your enterprise beans, and you need to develop your beans with as much flexibility as possible. Developing a truly reusable set of beans will likely require many iterations of feedback from customers using your beans in real-world situations.

Roughly speaking, bean reusability can fall into three different levels:

Reuse as given. The application assembler uses the acquired bean as it is to build an application. The bean functionality cannot be tailored to fit the application. This is typically what bean providers are offering in the market.

Reuse by customization. The application assembler configures the acquired bean by modifying the bean properties to fit the specific needs of the application. Bean customization typically occurs during development time. To allow for a more flexible maintenance environment, some bean providers allow runtime bean customization.

Reuse by extension (subclass). The application assembler creates custom application-specific beans by subclassing the prebuilt acquired beans. The behavior of the resulting bean is tailored for the application. This level of reusability is generally more powerful but difficult to achieve. Reuse by extension is made available by only a few bean providers.

Go back to the first page for a quick link to buy this book online!

The more reusability levels that a bean provides, the more useful a bean is. By leveraging prebuilt beans, organizations can potentially lower the development time of building enterprise applications.

Enterprise beans can also range in size and scope. Smaller-grained enterprise beans typically have very concrete, but limited, scoped duties. Larger-grained, fuller-featured enterprise beans have a wider business scope, and they typically interact with other smaller-grained enterprise beans.

For example, imagine you go to a music store to purchase a compact disc. The cashier takes your credit card and runs it through a scanner. The scanner has a small Java Virtual Machine running within it, which acts as a client of enterprise beans. It contacts American Express, which has an EJB-compliant application server containing a number of beans. The beans are responsible for conducting the credit card transaction on behalf of that client.

Once the scanner has a reference to a credit card transaction bean, the bean must first verify that your credit is good before billing your card. At that point, the bean itself acts as a client and contacts another bean—a verifier bean—to verify your credit rating. Once your credit is verified, the original bean can complete the transaction. So beans can indeed be clients of other beans. This is a very powerful, flexible model, and it allows for large-grained components to be composed of smaller ones in a hierarchical fashion.

The EJB Server and EJB Container Providers

If you'll recall, an application server provides middleware services to your applications, such as transaction services, security services, and others. These services are needed for your application to be scalable, robust, and secure for multiple concurrent users. EJB takes the notion of application servers and partitions them into two distinct parts:

The EJB container. The EJB container provides a playground where your enterprise beans can run. There can be many beans running within a container. Bean containers are responsible for managing the beans running within them. They interact with the beans by calling a few required methods that the bean must expose. Containers may also provide access to a legacy system.

The EJB server. The EJB server provides a runtime environment for one or more containers. EJB servers manage low-level system resources, allocating resources to containers as they are needed. The relationship between the EJB server and container is depicted in Figure 2.2.

Go back to the first page for a quick link to buy this book online!

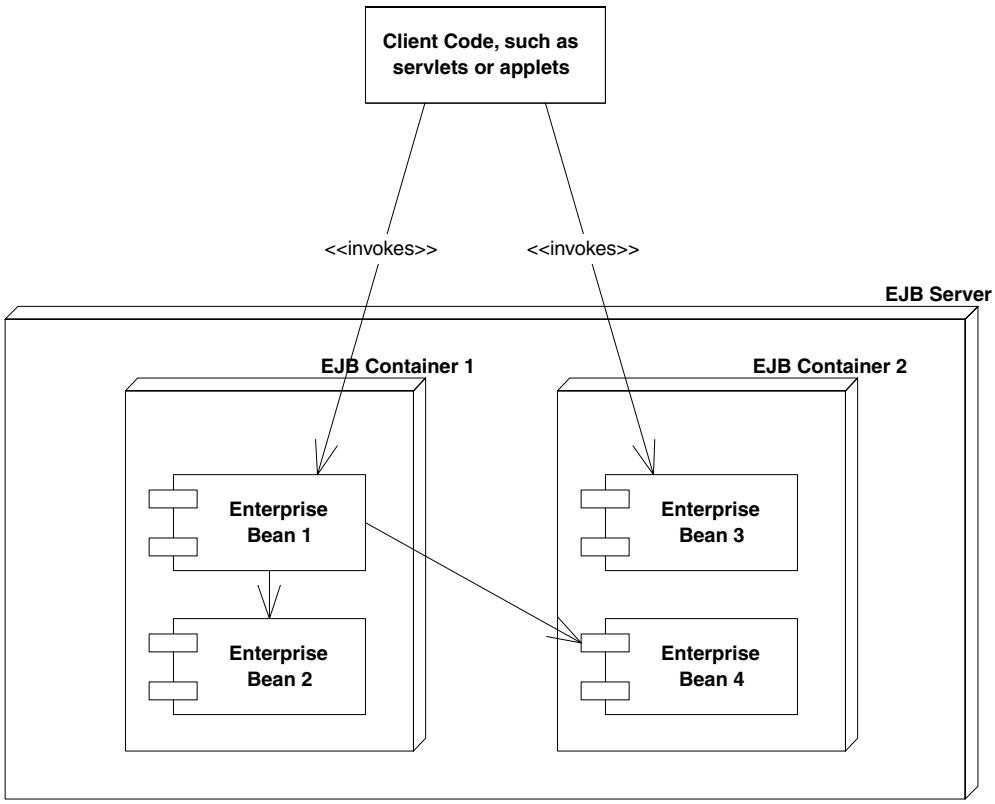

Figure 2.2 The relationship between EJB servers and EJB containers.

No Clear Separation Between Container and Server

Unfortunately, the EJB specification does not explicitly define the separation of roles between EJB servers and EJB containers yet. There is no concrete interface between the two entities. Until EJB addresses this issue, one vendor's EJB container will be not be installable within another vendor's EJB server. For now, you may be tied to one vendor for both the container and server.

To deal with the lack of a good EJB server/container contract, EJB server vendors are publishing proprietary APIs for custom EJB containers to run within their servers, such as BEA's WebLogic EJB server.

Go back to the first page for a quick link to buy this book online!

Highlights of Server and Container Responsibilities

In traditional server-side programming (such as with CORBA), you needed to write to complex middleware APIs to gain an application server's middleware services. But in the EJB world, you can harness complex middleware in your enterprise applications without writing to middleware APIs—instead, you can simply *declare* the middleware services that your application needs, and the application server will provide that middleware *transparently* to your application code. You can focus away from the middleware and concentrate on your application's business code.

Here are just some of the services that containers/servers must provide for you.

Implicit distributed transaction management. Transactions allow for you to perform robust, deterministic operations in a distributed environment by setting attributes on your enterprise beans. We'll get into the details of transactions and how you can use them effectively in Chapter 10. For now, know that the EJB server provides a *transaction service*—a low-level implementation of transaction management and coordination. The transaction service must be exposed through the *Java Transaction API* (*JTA*). The JTA is a high-level interface that you can use to control transactions, which we also cover in Chapter 10.

Implicit security. Security is a major consideration for multi-tier deployments. The Java 2 Platform, Standard Edition yields a robust security service that can authorize and authenticate users, securing deployments from unwanted visitors. EJB adds to this the notion of transparent security, allowing components to reap the benefits of a secure deployment without necessarily coding to a security API.

Implicit resource management and component life cycle. The EJB server implicitly manages resources for your components, such as threads, sockets, and database connections. The life cycle of the components themselves are also managed, allowing for components to be reused by the EJB server as necessary.

Implicit persistence. Persistence is a natural requirement of any deployment that requires permanent storage. EJB offers assistance here by automatically saving persistent object data to an underlying storage and retrieving that data at a later time.

Implicit remote accessibility. EJB products will automatically convert your stand-alone, network-less components into distributed, network-aware beings.

Implicit multiclient support. EJB servers automatically route concurrent requests from clients. EJB servers provide built-in thread support, instantiating multiple copies of your component as necessary and channeling client requests to those instances.

Go back to the first page for a quick link to buy this book online!

Implicit component location transparency. Clients of components are decoupled from the specific whereabouts of the component being used.

In addition to these implicit services, the EJB server provides a number of explicit services. For example, the EJB server must provide a *naming and directory service*, allowing components to be located across the network. The EJB server also ships with a set of *deployment tools* that allow the EJB deployer to deploy components into the EJB server and customize those components as needed.

Finally, EJB servers may go above-and-beyond the bare requirements, providing additional value-adds that are not required by the EJB specification. This could be intelligent load balancing, transparent fail-over, server clustering, and connectors for integration to legacy systems (such as BEA Tuxedo, IBM TXSeries, SAP R/3, and so on).

The Application Assembler

As we've mentioned, the bean provider supplies reusable, deployable server-side components. But when these components are purchased by a customer and put to actual use, who actually assembles them to solve a business problem? The answer is the *application assembler*.

The application assembler is the person, whether on staff or as an outside consultant, who understands the complete application system and understands how the various components fit together. He or she is the application architect—the person who understands what the components do. His or her job is to build an application from those components—an application that can be deployed in a number of settings.

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

1. Write the code that calls on components purchased from vendors.
2. Provide a workflow solution between a number of disparate components, mapping between them.
3. Supply a user interface (perhaps using JFC, JSP, or servlets).
4. Write new enterprise beans to solve domain-specific problems. For example, you might need to model a business process or business entity that is specific to your business. This task will often fall into the hands of the application assembler.

Note that EJB allows for distributed components (versus the traditionally rigid client/server paradigm). Therefore, the application assembler could be involved in the design and implementation of logic residing in several tiers in a multi-tier architecture. Perhaps the code the application assembler writes is local to the

Go back to the first page for a quick link to buy this book online!

components, perhaps it is remote. EJB does not dictate the physical placement of things.

The EJB Deployer

We've described how the container/server provider supplies the runtime management of the components and how the application assembler builds the specific applications. That is not enough, though, for a successful deployment. The applications must still be *deployed* in a running operational environment. And unfortunately, the application assembler (who is usually a developer or systems analyst) may not be familiar with the specific operational environment that the application must run in.

This is where the EJB deployer comes into play. EJB deployers are aware of specific operational environments. They understand how to deploy beans within servers and how to customize the beans for a specific environment (such as a multi-tier deployment involving a firewall). The EJB deployer has the freedom to adapt the beans, as well as the containers and servers, to the environment in which the beans are to be deployed. He or she also has knowledge of a customer's existing naming and directory services and understands how to customize enterprise beans for that scenario.

Another concrete role of the deployer is mapping security settings. Most businesses store lists of their employees and their security levels in some directory service structure, and the EJB deployer may be required to adapt the access level of the beans to fit that particular environment. That way, the multi-tier application developed by the application assembler is usable in the specific deployment scenario.

To facilitate this process, EJB deployers must be aware of the differences between the various beans, servers, and containers on the market. An EJB deployer can be a staff person or an outside consultant.

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 and containers provide.

For example, a sophisticated deployment might page a system administrator if a serious error occurs that requires immediate attention. Note that EJB servers and containers may not provide this explicit monitoring (and it is not required by the EJB specification). Some EJB products, however, are developing hooks into professional monitoring products, such as Tivoli and Computer Associates,

Go back to the first page for a quick link to buy this book online!

to aid with this. The Java 2 Platform, Enterprise Edition may in the future contain specification rules that govern management as well.

Now that you've seen the EJB players, let's move on to the business components themselves—the *enterprise beans*.

Enterprise Beans

An enterprise bean is a server-side software component that can be deployed in a distributed multi-tier environment. An enterprise bean can comprise one or more Java objects because a component may be more than just a simple object. Regardless of an enterprise bean's composition, the clients of the bean deal with a single exposed component interface. This interface, as well as the enterprise bean itself, must conform to the Enterprise JavaBeans specification. The specification requires that your beans expose a few required methods; these required methods allow the EJB container to manage beans uniformly, regardless of which container your bean is running in.

Note that the "client" of an enterprise bean could be anything—perhaps a servlet, an applet, or even another enterprise bean. In the latter case, a client request to a bean can result in a whole chain of beans being called. This is a very powerful idea because you can subdivide a complex bean task, allowing one bean to call on a variety of prewritten beans to handle the subtasks. This hierarchical concept is quite extensible.

Types of Beans

EJB 1.0 and 1.1 defines two different kinds of enterprise beans: *session beans* and *entity beans*. Let's take a look at how they compare.

Session Beans

A session bean represents work being performed for client code that is calling it. Session beans are business process objects. They implement business logic, business rules, and workflow. For example, a session bean could perform price quoting, order entry, video compression, banking transactions, stock trades, database operations, complex calculations, and more. They are reusable components that contain logic for business processes.

Session beans are called session beans because they live for about as long as the *session* (or lifetime) of the client code that's calling the session bean. For example, if client code contacted a session bean to perform order entry logic, the application server (that is, the EJB container/server) is responsible for creating

Go back to the first page for a quick link to buy this book online!

an instance of that session bean component. When the client later disconnects, the application server may destroy the session bean instance.

Session beans are usable by one client at a time—that is, they are not shared between clients. When a client is using a session bean, that client is the only client dealing with that session bean. This is in stark contrast to entity beans, whose state is shared among many clients.

The EJB server is responsible for managing the lifetime of beans. That is, the client does *not* directly instantiate beans—the EJB container does this automatically. The EJB container similarly destroys session beans at the appropriate times. This allows beans to be pooled and reused for multiple clients.

There are two subtypes of session beans—*stateful session beans* and *stateless session beans*.

Stateful Session Beans

As we have said, session beans represent business processes. Some business processes can be performed in a single method request, such as computing the price of goods or verifying a credit card account. Other business processes are more drawn out and can last across multiple method requests and transactions.

One example of a business process that lasts for multiple method calls is an e-commerce Web store. As the user peruses an online e-commerce Web site, he or she can add products to the online shopping cart. This implies a business process that spans multiple method requests. The consequence of such a business process is that the components must track the user's state (such as a shopping cart state) from request to request.

Another example of a drawn-out business process is a banking application. In a bank, you may have code representing a bank teller that deals with a particular client for a long amount of time. That teller may perform a number of banking transactions on behalf of that client, such as checking the account balance, depositing funds, and making a withdrawal.

A *stateful session bean* is a bean that's designed to service business processes that span multiple method requests or transactions. To accomplish this, stateful session beans *retain state* on behalf of an individual client. If a stateful session bean's state is changed during a method invocation, that same state will be available to that same client upon the following invocation.

Stateless Session Beans

Some business processes naturally lend themselves to a single request paradigm. A single request business process is one that does not require state to be maintained across method invocations. Stateless session beans are components that can accommodate these types of single request business processes. They are

Go back to the first page for a quick link to buy this book online!

anonymous method providers—anonymous because they are not aware of any client history.

For instance, a stateless session bean could be a high-performance engine that solves complex mathematical operations on a given input, such as compression of audio or video data. The client could pass in a buffer of uncompressed data, as well as a compression factor. The bean would return a compressed buffer and would then be available to service a different client. The business process has spanned one method request. The bean does not retain any state from previous requests.

Another example of a stateless session bean is a credit card verification component. The verifier bean would take a credit card number, an expiration date, a cardholder's name, and a dollar amount as input. The verified would then return a yes or no answer depending on whether the card holder's credit is valid.

Stateful or Stateless?

When deciding to use stateful session beans, you must first ask yourself whether the business process you're modeling inherently requires a stateful model. If it does, a stateful session bean may be the ideal component to use. When using stateful session beans, however, your inherent statefulness may limit your fault tolerance. For example, what happens if an unexpected system-level error occurs, such as a bean crashing, the network dying, or a machine rebooting? In a stateless model, the request could be transparently rerouted to a different component because any component can service the client's needs. In stateful models, there is little that can be done to reroute the client's request because the client's state is lost when the failure occurs (the state was kept within the lost bean). Note, however, that some high-end EJB container implementations are adding on stateful recovery services as an optional value feature. These services allow for even stateful components to be transparently recovered, by continually persisting the bean's active state and recovering from permanent storage in case of failure.

If you have a drawn-out business process, there is another alternative to using stateful session beans. You can go with a stateless model and pass the entire client state as parameters to the stateless bean during method invocations. Passing of state in such a way could lead to severe performance degradation. This is especially true if the client is remotely located from the bean and if the state passed over the network is large. Thus, you may achieve lightweight fault tolerance, but the overall scalability of your system may be compromised by the added network latency expense.

We'll explore the trade-offs between stateful and stateless models in more detail in Chapter 6.

Go back to the first page for a quick link to buy this book online!

Once the bean completes this task, it is available to service a different client and retains no past knowledge from the original client.

Entity Beans

Another fundamental part of a business is the permanent data that the business processes use. This is illustrated in the following examples:

- A bank teller component performs the business process of banking operations. But the data used by the teller is the bank account data.
- An order-entry component performs the business process of submitting new orders for products, such as submitting an order for a new computer to be delivered to a customer. But the data generated by the order-entry component is the order itself, which contains a number of order line-items describing each part ordered.
- A stock portfolio manager component performs the business process of updating a stock portfolio, such as buying and selling shares of stock. But the data manipulated by the portfolio manager is the portfolio itself, which might contain other data such as account and stock information.

In each of these scenarios, business process components are manipulating data in some underlying data storage, such as a relational database. An *entity bean* is a component that represents such persistent data. Entity beans model bank accounts, orders, order line items, stock portfolios, and so on. Entity beans represent real data objects, such as customers, products, or employees.

Entity beans do not contain business process logic—they model data. Session beans handle the business processes. Session beans might use entity beans to represent the data they use, similar to how a bank teller uses a bank account.

An EJB 1.0 container/server is not required to support entity beans. An EJB 1.1 container/server is required to support them fully.

The value that entity beans provide is an object-oriented in-memory view of data in an underlying data store. The traditional way for applications to deal with data is to work with relational tables in a database, reading and writing that data as needed. Entity beans, on the other hand, are object representations of this underlying data. You can treat data in a relational store as real objects. You can read an entire set of data out of a database at once into an in-memory entity bean component. You can then manipulate this entity bean in memory by calling methods on it. For example, you could call a *withdraw()* method on a bank account entity bean, which would subtract money from a bank account by reducing the value of a private member variable called *balance*. Once that bank account entity bean object is persisted, the database will contain the new bank

Go back to the first page for a quick link to buy this book online!

account balance. Thus, entity beans allow you to combine the functionality of persistent data with the convenience of object encapsulation. In essence, an entity bean implements the data access logic layer in multi-tier architectures. This is shown in Figure 2.3.

Because they model permanent data, entity beans are long lasting. They survive critical failures, such as application servers crashing, because entity beans are just representations of data in a permanent, fault-tolerant underlying storage. If a machine crashes, the entity bean can be reconstructed in memory again by simply reading the data back in from the permanent database. Because the database survives crashes, the components that represent them do as well. This is a huge difference between session and entity beans—entity beans have a life cycle much longer than a client's session, perhaps years long, depending on how long the data sits in the database.

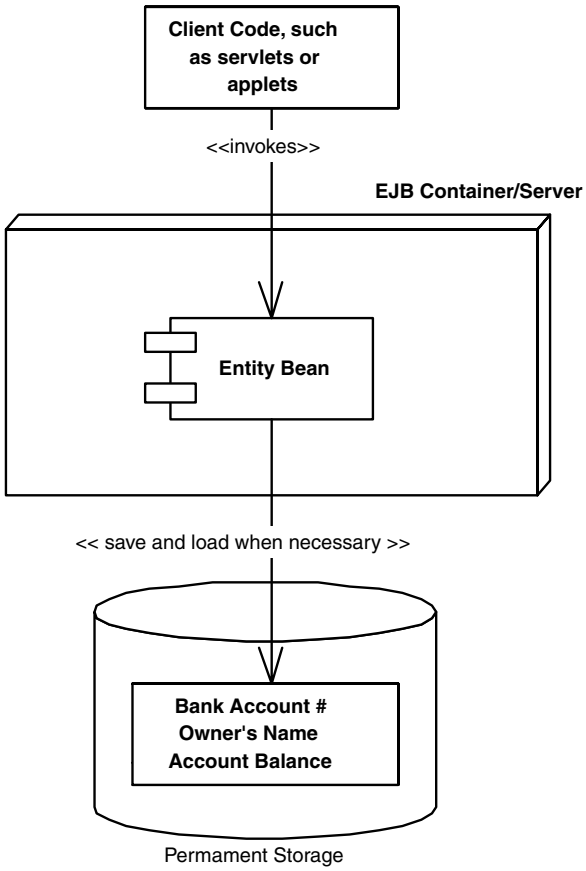

Figure 2.3 Entity beans are a view into an underlying data store.

Go back to the first page for a quick link to buy this book online!

Session Beans versus Entity Beans

Now that you've seen the different types of enterprise beans, how do you decide what logic to put in a session bean and what to put in an entity bean? The key differences between the two are that session beans represent business *processes* and typically contain business logic, and entity beans, on the other hand, embody permanent business *entities*, such as business data, and typically contain data-related logic.

As a concrete example of this, consider an application that is responsible for generating prices of products for customers. If a customer wants to order 15 workstations, with a certain bundle of memory and processors, the pricer would display a price detailing the products and listing the price of each item.

A pricer component is well represented by a session bean: The pricer is having a business process conversation with the client. The client code interacts with the pricer, asking for prices of products that the customer has indicated, and then the client code disconnects from the pricer component. After the conversation is over, the pricer is free to service another client.

Later, when the customer wants to place an order, an order entity bean could represent the customer's permanent request for goods to be delivered. The order bean would represent database data that detailed the customer's purchase. The order can then be fulfilled by another application—perhaps another session bean that contains the logic to do so. This is shown in Figure 2.4.

Notice the theme here: Session beans are performing application logic, which uses persistent entity beans behind the scenes as the data that they're manipulating. This is very similar to the *façade* design pattern. A façade is a high-level interface that masks lower-level subsystems. In the EJB distributed object architecture, session beans can be used to provide a high-level interface façade to business processes, masking the lower-level entity bean subsystems used behind the scenes.

Because entity beans model permanent business entities, entity beans typically achieve a higher level of reuse than session beans. For instance, consider our banking example, with a session bean that acts as a bank teller. This session bean knows how to withdraw and deposit by calling methods on a bank account entity bean. One day, you may decide to replace your session bean teller with a different teller bean. But you'd still want all your customers' bank accounts to remain the same.

Now consider our order-entry example. Here, your company's product line, as well as the purchase orders themselves, are data. Therefore, they are well represented as entity beans—objects that are saved over time and are forever part of the database. The components that generate and manipulate the products and the purchase orders are very well modeled as session beans. You'd probably want to fine-tune and change these session beans over time, as user requirements change.

Thus, in practice you can expect the reuse of session beans to be a bit lower than that of entity beans. Session beans model a current business process, which can be

Go back to the first page for a quick link to buy this book online!

tweaked and tuned with different algorithms and approaches. Entity beans, on the other hand, define your core business. Data such as purchase orders, customers, and bank accounts do not change very much over time, and in practice entity beans achieve a higher level of reuse.

Note that what we've presented here are merely guidelines, not hard-and-fast rules. Indeed, a session bean can contain data-related logic as well, such as a session bean performing a bulk database read via JDBC or SQL/J. The key to remember is that session beans never *embody* permanent data, but merely provide access to data. Session beans are not persistable; they represent business processes. By comparison, entity beans embody data and are persistable. If you have a business process that you'd like to run in a transactional, distributed, and secure environment, you can use session beans to model that business process. If you have a permanent business entity, you can access that entity as a distributed component by representing it as an entity bean.

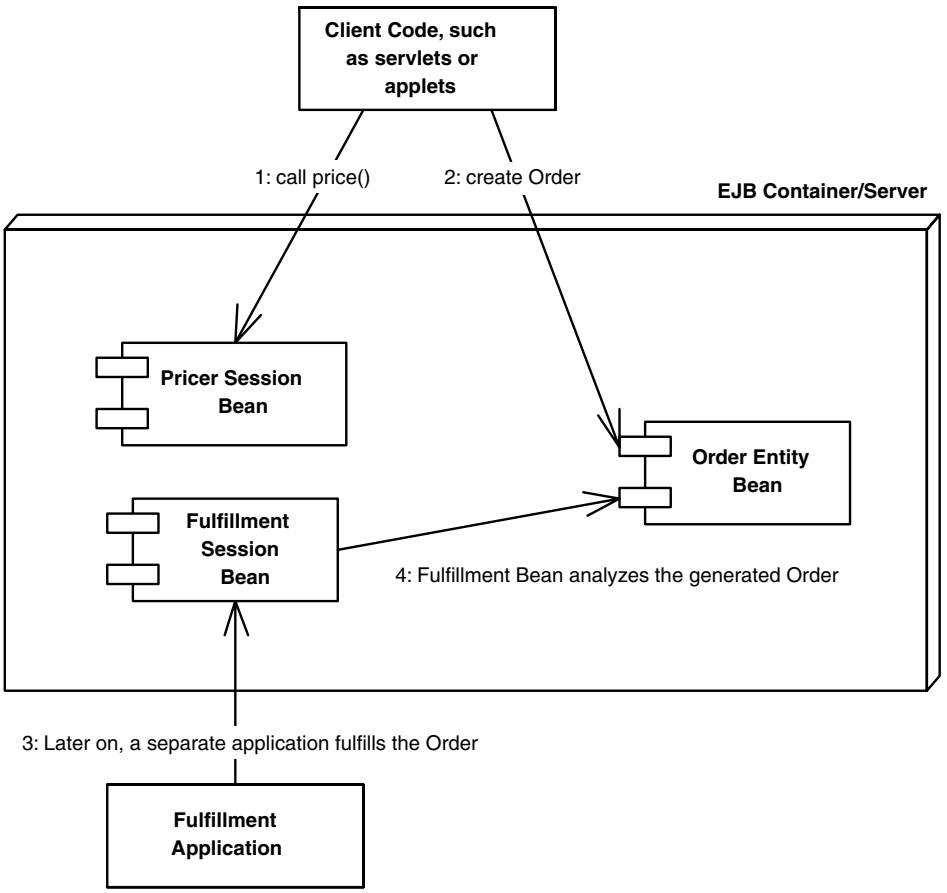

Figure 2.4 Combinations of beans in a pricing application.

Go back to the first page for a quick link to buy this book online!

Entity beans are distinct from session beans in another respect as well—multiple clients can use entity beans simultaneously. Think of this as multiple clients manipulating the same data in a database simultaneously. You can isolate these clients from one another by using *transactions*. When used properly, transactions guarantee that multiple clients who perform persistent operations act completely independently of one another. We'll find out more about transactions in Chapter 10.

Entity beans are very useful when you have a legacy database with data already inside. They are great for providing access to existing persistent data. In this manner, the data that entity beans model could exist in a database before a company decided to employ Enterprise JavaBeans.

In fact, the database records representing an object could have existed before the company even decided to go with a Java-based solution because a database structure can be language-independent. Database records can be read in and interpreted as objects in almost any language. EJB takes advantage of this and allows for the transformation of a database's data into a Java object. The burden of this transformation can fall on the bean itself or the EJB container can perform the transformation automatically.

This brings us to the next topic. There are two subtypes of entity beans available: *bean-managed persistent* entity beans and *container-managed persistent* entity beans.

Bean-Managed Persistent Entity Beans

As we have seen, entity beans are persistent components because their state is saved to a secondary storage such as a relational database. For example, by using *object-relational mapping* technology, you can take an in-memory *object* and *map* that object into a persistent series of *relational* database records. You can then retrieve those records at a later time to reconstruct the object in-memory and use it again. Another scheme is to use an *object database* as your persistent store, which stores actual objects rather than relational records.

A bean-managed persistent entity bean is an entity bean that must be persisted by hand. In other words, you as the component developer must write code to translate your in-memory fields into an underlying data store, such as a relational database or an object database. You handle the persistent operations yourself—including saving, loading, and finding data—within the entity bean. Therefore, you must write to a persistence API, such as JDBC or SQL/J.

Container-Managed Persistent Entity Beans

The good news is that EJB allows entity bean developers to not worry about coding persistence logic. One service that EJB 1.0 containers *may* provide, and EJB 1.1 containers *must* provide, is automatic persistence for your entity beans.

Go back to the first page for a quick link to buy this book online!

The container/server performs every function of your component's data access layer for you, including saving, loading, and finding component data. You do not have to hard-code to a relational database API or an object database API, saving much coding time. Rather, you simply describe what you want persisted upfront to the container, and it persists it for you, using whatever storage it happens to have. This gives you theoretical database independence, allowing you to switch one data store for another, since you don't write any code to a database API. The EJB container/servers on the market today perform a wide variety of mapping functionality and tools—some simple, some complex. For example, BEA's *WebLogic* server performs fairly simple object/relational mapping. But BEA also supports the Object People's *TOPLink*, an enhanced persistence module that allows for complex mapping.

When choosing between container-managed persistence and bean-managed persistence, many issues are at stake. We'll contrast the promises and realities of automatic persistence fully in Part II.

Motivation for Multiple Bean Types

You may be wondering why the EJB paradigm is so robust in offering the various kinds of beans. Why couldn't Sun come up with a simpler model? Microsoft's N-tier vision, for example, does not include the equivalent of entity beans—components that represent data in a permanent storage.

The answer is that Sun is not the only company involved in constructing the Enterprise JavaBeans standard. In fact, many companies have been involved, each with customers that have different kinds of distributed systems. To accommodate the needs of different enterprise applications, Sun decided to allow users the flexibility of each kind of bean.

Admittedly, this increases the ramp-up time to learn EJB. But it also pays off in the long run with increased functionality. By including session beans, Sun has provided a mechanism to model business processes without writing middleware in a distributed multi-tier environment. By including entity beans in the EJB specification, Sun has taken the first steps toward persistent, distributed objects usable by those business processes.

Overview of EJB Container and EJB Server Responsibilities

Earlier in this chapter, we mentioned that EJB containers provide the implicit services to your EJB components and that containers live within the runtime environment of an EJB server. Because the EJB specification has not drawn the

Go back to the first page for a quick link to buy this book online!

Single-Threaded versus Multithreaded Session Beans

One great benefit of EJB is you don't need to write thread-safe code. You design your enterprise beans as single-threaded components, and you never need to worry about thread synchronization when concurrent clients access your component. Your EJB container will automatically instantiate multiple instances of your component to service concurrent client requests.

The container's thread services can be both a benefit and a restriction. The benefit is that you don't need to worry about race conditions or deadlock in your application code. The restriction is that some problems lend themselves very well to multithreaded programming, and that class of problems cannot be easily solved in an EJB environment.

So why doesn't the EJB specification allow for multithreaded beans? The answer is that EJB is intended to relieve component developers from worrying about threads or thread synchronization. The EJB container handles those issues for you by load-balancing client requests to multiple instances of a single-threaded component. An EJB server provides a highly scalable environment for single-threaded components, and adding the abillity for beans to control threads opens up a Pandora's box of problems. For example, the ability for an EJB container to control a transaction (discussed in Chapter 10) becomes a *very* complicated problem if threads are being started and stopped randomly by beans.

One alternative to threading is to use a transactional messaging API such as the Java Messaging Service (JMS) that allows for asynchronous actions to occur in a distributed object environment. JMS enables you to safely and reliably achieve multitasking, without the beans themselves messing around with threads. JMS support is expected to be in the 2.0 EJB specification, due out in late 2000.

The bottom line here is that EJB was not meant be a swiss-army knife, solving every problem in existence. It was designed to assist with server-side *business problems*, which are largely single-threaded. For applications that absolutely must be multithreaded, EJB may not be the correct choice of distributed object architectures.

line between a container and a server, we will use the words interchangeably in this book.

EJB containers are responsible for managing your beans. Containers can interact with your beans by calling your beans' required management methods as necessary. These management methods are your bean's callback methods that the container, and only the container, invokes. The management methods allow the container to alert your bean when middleware events take place, such as when an entity bean is about to be persisted to storage.

Go back to the first page for a quick link to buy this book online!

The most important responsibility of an EJB container is to provide an environment in which enterprise beans can run. EJB containers house the enterprise beans and make them available for clients to invoke remotely. In essence, EJB containers act as invisible middlemen between the client and the beans. They are responsible for connecting clients to beans, performing transaction coordination, providing persistence, managing a bean's life cycle, and other tasks. The EJB container-bean relationship is depicted in Figure 2.5.

The key to understanding EJB containers is to realize that they are *abstract entities*. Neither the beans nor the clients that call beans ever explicitly code to the API of an EJB container. Rather, the container implicitly manages the overhead of a distributed component architecture. The container is analogous to a 'behind the scenes' stage manager in a theatre, providing the lighting and backdrop necessary for a successful stage performance by the actors on stage. But neither the actors nor the audience directly interact with the stage manager. The same is true for EJB containers. Neither the beans nor the clients that call the beans ever code directly to an EJB container API.

EJB containers are a huge challenge to write because they are so complex and perform so many tasks. As of the time of this writing, there are 27 EJB containers in development, varying widely in implementation. For example, BEA makes

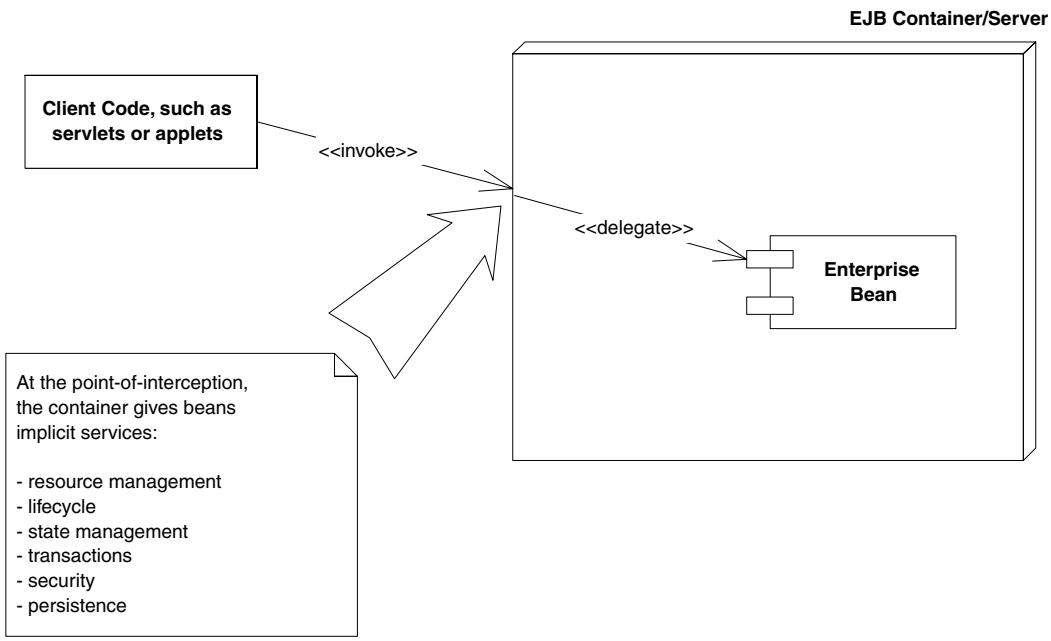

Figure 2.5 An EJB container housing a bean.

Go back to the first page for a quick link to buy this book online!

an EJB container that is written entirely in Java, managing your Java-based enterprise beans. Sybase, on the other hand, makes an EJB container written in C++, which interacts with beans via the Java Native Interface. Oracle provides a container that physically runs within the context of a database. As you can see, a container can be manifested in a number of functional environments.

We now present an architectural overview of the features an EJB container provides.

Resource Management and Bean Life Cycle Management

As we've mentioned in Chapter 1, a multi-tier architecture's overall scalability is enhanced when an application server intelligently manages needed resources across a variety of deployed components. The resources could be threads, socket connections, database connections, and more. For example, database connections could be pooled by application servers and reused across heterogeneous components. In the EJB realm, the container is responsible for providing all resource management services behind the scenes.

In addition to resource management, the EJB container is responsible for controlling the life cycle of the deployed enterprise bean components. As bean client requests arrive, the EJB container dynamically instantiates, destroys, and reuses beans as appropriate. For example, if a client requests a certain type of bean that does not exist in memory yet, the EJB container may instantiate a new in-memory instance on behalf of the client. On the other hand, if a bean already exists in memory, it may not be appropriate to instantiate a new bean—especially if the system is low on memory. Instead, it might make more sense to reassign a bean from one client to another. It might also make sense to destroy some beans that are not being used anymore. This is called *instance pooling*.

The take-away point here is that the EJB container is responsible for coordinating the entire effort of resource management as well as managing the deployed beans' life cycle. Note that the exact scheme used is EJB container-specific.

State Management

State management is another value that containers bring to the table. To explain the need for state management, we first must observe that users, as well as client code, often take a lot of time to "think" in between method calls to a component. The classic example of this is an HTML (Web) client interacting with a human being. Web users often will click on a button that executes some business logic in a component, but then they wait around and read text before initiating another action. While the user is doing this, the application server could reuse that component to service other clients.

Go back to the first page for a quick link to buy this book online!

This is exactly what happens in EJB. If a bean is stateless, it can be reassigned to other clients dynamically by the EJB container. This is possible because there is no state lost from the primary client. This reuse of beans results in incredible resource gains—often only a few beans need to be instantiated to handle a multitude of clients. If a client times out (for example, because of a crash), the EJB container can destroy the bean or perhaps reuse it. This is all possible because the EJB container is constantly monitoring bean invocation activity.

On the other hand, if the bean is stateful, things get a little more complicated. The EJB container must provide transparent *state management* for stateful components. State management is necessary when you want to reuse a stateful component to service multiple clients.

Consider the scenario where a client hasn't used a stateful bean for a long time. This stateful bean could be a stateful session bean or an entity bean (entity beans are inherently stateful because they represent data). When a new client connects and requests a component, the container may have reached its limit of instantiated components. In this case, the container can take a component that hasn't been used in a while, then *serialize* (convert to a binary stream—see Appendix A) the bean's conversational state, and write the state out to disk. Now that the original client's state has been preserved, the bean can be reassigned to a different client, and it can retain state for that new client exclusively. Later on, if the original client makes a request, the original client's bean state can be read back in from disk and used again, perhaps in a different in-memory bean object.

The management of state is a responsibility of the EJB container, and it is mostly masked from component vendors. It is an implicit service provided by the EJB architecture.

Transactions

Transactions are a safe way to have multiple components participate in distributed object operations. A *transaction* is a series of operations that appear to execute as one large, atomic operation. Transactions allow multiple users to share the same data, and they guarantee that any set of data they update will be completely and wholly written, with no interleaving of updates from other clients. In a sense, transactions are a very sophisticated form of concurrency control.

When properly used, transactions ensure that database data is kept consistent. Transactions also ensure that two components' database operations are isolated from one another. Transactions prevent disaster from striking if your database crashes, too. Without transactions, your database could easily become corrupt, which is unnacceptable for such mission-critical applications as banking applications.

Go back to the first page for a quick link to buy this book online!

The EJB server/container handles the underlying transaction operations, coordinating efforts behind the scenes between transaction participants. The value-add of EJB here is that transactions can be performed implicitly and automatically. This allows beans to leverage transactions in deployments without writing to an explicit transaction API. We cover transactions more thoroughly in Chapter 10.

Security

In any critical deployment, security is always going to be an issue. The role of EJB containers in security is to handle the validation of users for tasks they wish to accomplish. This is done via *Access Control Lists* (*ACLs*). An ACL is a list of users and their rights. If the user has the correct rights, he or she can perform the desired operation. The Java Development Kit 1.2 provides a robust security model that allows for authentication (identifying that the user is who he or she claims to be) and authorization (identifying that the user is of the correct role to perform the desired operation).

EJB containers add transparent security to this. Rather than programmatically accessing a security API, enterprise beans can automatically run as a certain security identity. Alternatively, enterprise beans can programmatically ensure that clients are authorized to perform desired operations.

Persistence

As we've mentioned earlier in the chapter, entity beans are persistent objects that represent data in an underlying storage. EJB containers can provide the transparent persistence of container-managed persistent entity beans. Note that while the EJB 1.0 specification does not require that containers manage persistence for beans, it is required in the 1.1 specification. We'll see more about how EJB persistence works in Part II.

Remote Accessibility and Location Transparency

Remote accessibility is the conversion of a network-naive component into a fully networked component that can be invoked remotely. Enterprise JavaBeans insulates the bean provider from all networking issues. Beans are written as standalone, nonnetworked components. But once they are deployed in the EJB realm, they become distributed components, deployable across multiple tiers.

EJB containers use the Java Remote Method Invocation (RMI) interfaces to specify remote accessibility, which you can learn about by reading Appendix A. The benefit of distributed communication technologies such as RMI is that your client code is unaware of the physical location of the component it is calling. The component could be located across the world, on a local area network

Go back to the first page for a quick link to buy this book online!

right next door to you, or on the client's machine itself. It could even reside in the client code's address space (such as an application server that supports Java servlets and enterprise beans in a single JVM). Whatever is the case, the client should be totally unaware of where the component really is—whether the component is local or remote should be *transparent* to the client. This is known as *location transparency*.

Why is location transparency beneficial? Well, for one thing, you aren't writing your bean's client code to take advantage of a particular deployment configuration because you're not hard-coding machine locations. This is an essential part of reusable components that can be deployed in a wide variety of multitier situations.

Location transparency also enables container vendors to provide additional value-adds, such as the ability to take a machine on the network down temporarily to perform system maintenance, to install new software, or to upgrade the components on that machine. During maintenance, location transparency allows for another machine on the network to serve up components for a component's client because that client is not dependent on the hard locations of any components. If a machine that has components on it crashes due to hardware or software error, you may be able to reroute client invocations to other machines without the client even knowing about the crash, allowing for an enhanced level of fault-tolerance.

Glue-Code and Bean Installation Tools

Each EJB container ships with a suite of *glue-code tools*. These tools are meant to integrate beans into the EJB container's environment. The tools generate helper Java code, such as stubs, skeletons, data access classes, and other classes that this specific container requires. Bean providers do not have to think about the specifics of how each EJB container works because the container's tools generate its own proprietary Java code automatically.

The container's glue-code tools are responsible for transforming an enterprise bean into a fully managed, distributed server-side component. This involves logic to handle resource management, life cycle, state management, transactions, security, persistence, and remote accessibility—every service we've mentioned so far. The automatic code generated handles these services in the container's proprietary way.

Specialized Container Features

Beyond the normal duties of a container, *specialized containers* can provide additional qualities of service that are not required by the EJB 1.0 specification. Most vendors have several different versions of their containers for sale, allowing

Go back to the first page for a quick link to buy this book online!

customers to pay for what they get. These services help differentiate EJB container vendors, allow for innovation, and foster best-of-breed products. They also add an element of danger. If a bean depends on particular qualities of service, it may not run in other containers.

For example, let's take the case of a load-balancing service. Load-balancing is the fair selection of components on behalf of their clients. These components may reside in many containers, housed in multiple EJB servers, distributed across the network. Given the fact that N-tier deployments can vary widely in the actual physical locations of components, the actual load-balancing algorithm is implementation dependent. This allows a lot of creativity for the EJB container provider. For example, some containers may provide a way to perform customized load balancing between distributed components. The customized load-balancer might be tunable for particular deployments and might make the overall system more scalable.

A high-end EJB container may also provide dynamic resizing of managed resources and components. For example, if clients are less active at night than during the day, perhaps a smaller pool of resources is necessary at night. Some EJB containers may be intelligent enough to allow for this dynamic resizing of pools, yielding precious machine resources to be used for other purposes.

Other examples of specialized container functions could include the following:

- Integration to mainframe systems
- COM+ integration
- Transparent fail-over
- Stateful recovery
- Web server, servlet, and JSP support within the container
- Server clustering
- Dynamic redeployment of components in a running system
- Support for a variety of persistence mechanisms, including relational and object databases
- Shared object caches to improve the number of times a database needs to be accessed
- Sophisticated monitoring support
- Distributed transactions
- Resolution of the transactional distributed diamond problem
- Complex database persistence services
- Visual development environment integration

Go back to the first page for a quick link to buy this book online!

- Integrated XML facilities
- CORBA integration and support

These are merely examples of the variety of services that a vendor could choose to add. The key idea to keep in mind when choosing to use services such as these is whether the services lock you into a particular vendor. If so, you need to ask yourself if that's a safe bet and if you want your components to be deployable across a variety of middleware vendor products.

Summary

In this chapter, we've taken a whirlwind tour of Enterprise JavaBeans. We started by looking at how EJB promotes a divide-and-conquer approach to server-side computing by defining the roles of six parties in a successful EJB deployment. We then went into what an enterprise bean is, and we looked at the different kinds of beans, including session and entity beans, and their subtypes. We analyzed the trade-offs of when it was appropriate to use each kind of bean.

Finally, we took a look at the responsibilities of the EJB container and server. We examined the services containers provide to distributed server-side components, such as resource management, state management, and location transparency.

And congratulations—you've made it to the end of Part I! In these two chapters, you've taken the first steps necessary to understanding and mastering Enterprise JavaBeans. Now that you understand the high-level concepts, let's move on and begin programming with EJB.

Go back to the first page for a quick link to buy this book online!