CHAPTER 1

Overview

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

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

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

Let's kick things off with a brainstorming session.

The Motivation for Enterprise JavaBeans

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

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

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

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

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

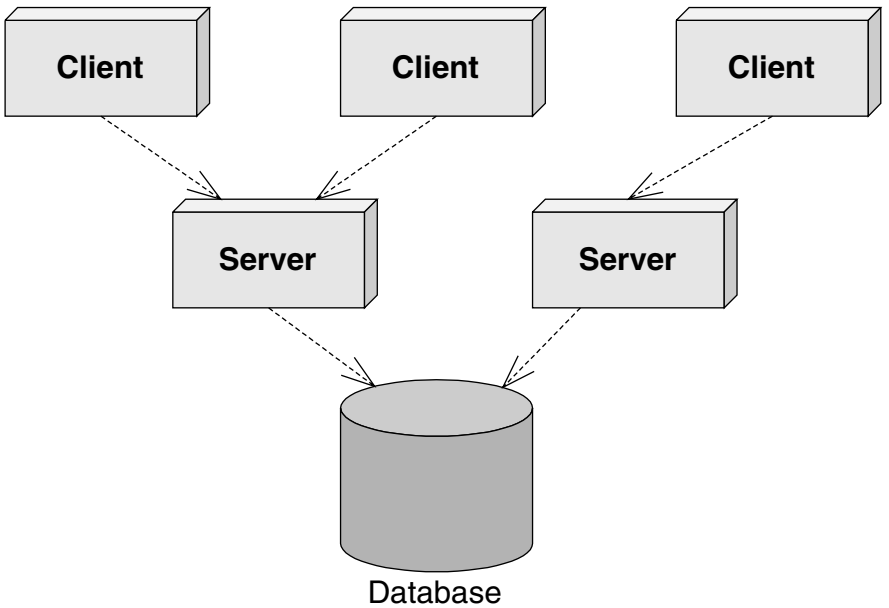

Figure 1.1 Standard multitier-only deployment.

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

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

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

THINGS TO CONSIDER WHEN BUILDING LARGE BUSINESS SYSTEMS

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

- **Remote method invocations. We need logic that connects a client and server via a network connection. This includes dispatching method requests, brokering parameters, and more.**
- **Load balancing. Clients must be directed to the server with the lightest load. If a server is overloaded, a different server should be chosen.**
- **Transparent fail-over. If a server crashes, or if the network crashes, can clients be rerouted to other servers without interruption of service? If so, how fast does fail-over happen? Seconds? Minutes? What is acceptable for your business problem?**
- **Back-end integration. Code needs to be written to persist business data into databases as well as integrate with legacy systems that may already exist.**
- **Transactions. What if two clients access the same row of the database simultaneously? Or what if the database crashes? Transactions protect you from these issues.**
- **Clustering. What if the server contains state when it crashes? Is that state replicated across all servers, so that clients can use a different server?**
- **Dynamic redeployment. How do you perform software upgrades while the site is running? Do you need to take a machine down, or can you keep it running?**

(continued)

THINGS TO CONSIDER WHEN BUILDING LARGE BUSINESS SYSTEMS *(continued)*

- **Clean shutdown. If you need to shut down a server, can you do it in a smooth, clean manner so that you don't interrupt service to clients who are currently using the server?**
- **Logging and auditing. If something goes wrong, is there a log that you can consult to determine the cause of the problem? A log would help you debug the problem so it doesn't happen again.**
- **Systems management. In the event of a catastrophic failure, who is monitoring your system? You want monitoring software that paged a system administrator if a catastrophe occurred.**
- **Threading. Now that you have many clients connecting to a server, that server is going to need the capability of processing multiple client requests simultaneously. This means the server must be coded to be multithreaded.**
- **Message-oriented middleware. Certain types of requests should be *message-based* where the clients and servers are very loosely coupled. You need infrastructure to accommodate messaging.**
- **Object life cycle. The objects that live within the server need to be created or destroyed when client traffic increases or decreases, respectively.**
- **Resource pooling. If a client is not currently using a server, that server's precious resources can be returned to a *pool* to be reused when other clients connect. This includes sockets (such as database connections) as well as objects that live within the server.**
- **Security. The servers and databases need to be shielded from saboteurs. Known users must be allowed to perform only operations that they have rights to perform.**
- **Caching. Let's assume there is some database data that all clients share and make use of, such as a common product catalog. Why should your servers retrieve that same catalog data from the database over and over again? You could keep that data around in the servers' memory and avoid costly network roundtrips and database hits.**
- **And much, much, *much* more.**

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

Component Architectures

It has been a number of years since the idea of multitier server-side deployments surfaced. Since then, more than 50 application servers have appeared on the market. At first, each application server provided component services in a nonstandard, proprietary way. This occurred because there was no agreed definition of what a component should be or how it should be provided with services or how should it interact with the application server. The result? Once you bet on an application server, your code was locked into that vendor's solution. This greatly reduced portability and was an especially tough pill to swallow in the Java world, which promotes openness and portability.

What we need is an *agreement*, or set of interfaces, between application servers and components. This agreement will enable any component to run within any application server. This will allow components to be switched in and out of various application servers without having to change code or potentially even recompile the components themselves. Such an agreement is called *component architecture* and is shown in Figure 1.2.

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

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

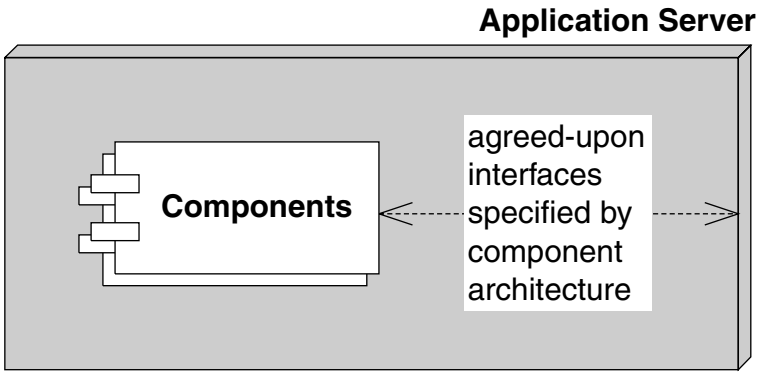

Figure 1.2 A component architecture.

Service-Oriented Architectures

At the core of a service-oriented architecture lies the concept of *service*. A simplistic definition of service is a group of related components that carry out a given business process function, for example transferring funds between banks or booking an itinerary. A *service-oriented architecture* (SOA) thus is a paradigm focusing on development of services rather than piecemeal components such that these services provide a higher level of abstraction from a functional standpoint. Of course, there are more properties to SOA than mere coarse-granularity. One such characteristic property of SOA is that they are autonomous in nature. These independent entities can interact with others in spite of differences in the way they have been implemented or the platform they have been deployed on. The notion of putting together (integrating) such autonomous and loosely coupled services to address the changing business needs has a huge value proposition and it is well on its way to realization with the emergence of various choreography, orchestration and collaboration technologies such as WS-BPEL, EbXML BPSS, and WS Choreography.

SOA and Web Services

The terms Web Services and SOA are often used interchangeably and wrongly so. SOA is a paradigm. There are many possible ways of building software so that it implements salient features of SOA, mainly coarse granularity and loose coupling. One such way is Web services. Web Services are a group of XML technologies, which can be used for implementing SOA. Core Web service technologies—mainly SOAP and WSDL—form the basis of most of these Web service implementations today.

Simple Object Access Protocol (SOAP) is an XML-based application-level protocol intended for exchanging information in a distributed network. SOAP supports both the models of distributed computing: RPC as well as document-style messaging. RPC style SOAP allows remote invocation of operations. Parameters and return in/out values of these operations are serialized in XML. Whereas, in document-style SOAP because an operation's input and output are XML, serialization of parameters and return value to XML is not needed. Although most of the Web service applications use SOAP over HTTP today, the standard does not preclude using SOAP over other IP protocols, such as SMTP. SOAP 1.2 is a W3C recommendation at the time of this writing.

Web Service Description Language (WSDL) is an XML-based metadata standard that is used to describe the service interface—in terms of the operations it supports, the parameters that the operations accept, and their return values in case of SOAP RPC, the XML schema that the input and output messages to the operations in case of document-style SOAP—as well as service binding information—in terms of the communication protocols, ports, service URL, and so

on. At the time of this writing, WSDL 2.0 is well on its way to becoming a W3C standard.

Thus, Web Services present a powerful solution for distributed but loosely coupled, coarse-grained SOA wherein services are described using WSDL and accessed via SOAP. In fact, one of the main reasons for using Web Services to realize SOA is the ubiquitous support for XML, SOAP, and WSDL technologies on disparate platforms, ranging from mainframes to mobile devices. This is the main reason why Web Services provide a true solution for interoperability between applications deployed on these disparate platforms.

We will spend some more time explaining fundamental concepts in Chapter 5; however, explaining Web Services and related technologies in their entirety is outside the scope of this book. If you are new to Web Services, there are many books and online papers that you can refer to get started with Web Services conceptually. Given the solid adoption of this stack by the industry, we suggest that you familiarize yourself properly with Web services.

SOA and Component Architectures

SOA is *not* a replacement for component architecture; rather it neatly complements the component architecture. While component architectures enhance reusability at a finer grain level, SOA can enhance reusability at a coarser grained level. Hence, from an implementation standpoint, a given service might very well be developed using well-defined component frameworks such as EJB. The latest EJB standard, therefore, has in-built support for Web Services, the most popular stack for building SOA. So EJB is still very much in demand!

Chapter 5 covers Web Services support in EJB framework in detail.

Divide and Conquer to the Extreme with Reusable Services

We have been seeing a slow but steady shift in the "build-from-scratch" trend, for years now. More and more businesses want CIOs to stretch their IT dollars to the maximum. Naturally, this has led the IT departments to think of reuse; reuse in terms of systems as well as software. What better candidate than highly functional and autonomous services to fulfill this promise of reuse? SOA offers maximum reuse, especially when implemented using ubiquitous protocols such as those supported by Web services. Architects want to design their software as a composition of services such that these services can be used from any platform through well-defined service interfaces.

Why just stop at corporate ITs? Even ISVs are thinking of providing their software as services. Prime examples of "software as a service" include Salesforce.com and Siebel. Both these companies have made their enterprise software available to customers as hosted services. Many other businesses such as Amazon.com and Google provide their core business services, E-commerce, and Web searching, as reusable services to customers and end-users.

Reusable services are a very powerful concept, because:

- **Businesses can focus on strategic software development.** In cases where software functionality is horizontal and cuts across multiple business domains, it should be treated as commodity and hence procured from a specialized ISV in the form of services. For example, each business requires a corporate treasury management and cash management system. For such a commodity business need, it is best to acquire software from an outside vendor than to build it. This will relieve the IT staff from having to deal with complex treasury functions involving millions of regulations; it anyway does not have direct relevance to the business's core function.
- **The business processes can be assembled faster.** The autonomous and loosely coupled nature of services makes it easy to assemble them into business processes. This strength makes services the chosen paradigm for encapsulating business logic.
- **There is a lower total cost of ownership.** Businesses that build their software as services end up with a lower total cost of ownership in the long term because they are building software such that it can be easily reusable and assembled into business processes. This is a definite plus when businesses are required to adapt business processes to address the changing market demands or when they are required to support new customers and their IT systems. Businesses that sell software as services, on the other hand, can benefit their customers by offering flexible software licensing options, such as per-month billing or per-year billing, thereby enabling their customers to lower total cost of ownership.

Remember that these services can and should be built using components. Therefore, the component architectures are very much here to stay. Figure 1.3 depicts such a Treasury management service built using EJB components.

With this introduction to SOA and their relevance to EJB, let us further explore the EJB technology.

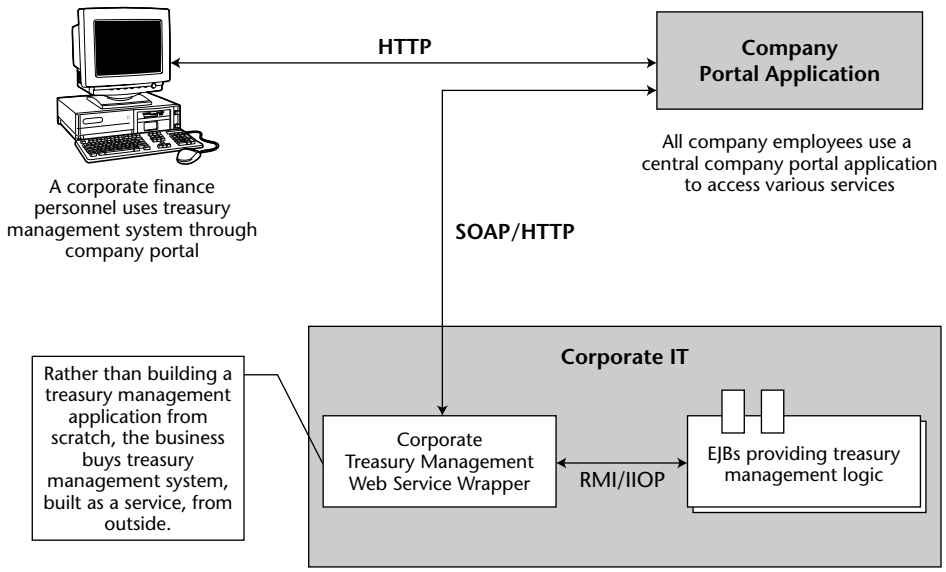

Figure 1.3 Reusable services built using EJB.

Introducing Enterprise JavaBeans

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

The top three propositions of EJB are as follows:

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

- **Rapid application development.** Your application can be constructed faster because you get middleware infrastructure services such as transactions, pooling, security, and so on from the application server. There's also less of a mess to maintain.

Note that while EJB does have these virtues, there are also scenarios in which EJB is overkill. See Chapters 11 and 16 for best practices and discussion surrounding the issue of when to (and when not to) use EJB.

Physically, EJB is actually two things in one:

- ***A specification.*** **This is a 640-plus-page Adobe Acrobat PDF file, freely downloadable from http://java.sun.com/products/ejb/docs.html. This specification lays out the rules of engagement between components and application servers. It constricts how you code enterprise beans to enable "write once, run anywhere" behavior for your EJB application.**
- ***A set of Java interfaces.*** **Components and application servers must conform to these interfaces. Since all components are written to the same interfaces, they all look the same to the application server. The application server therefore can manage anyone's components.**

Why Java?

EJB architecture has supported only the Java language thus far. Though this sounds a bit restrictive, the good news is that Java is one of the best-suited languages for building components for the following reasons.

- **Interface/implementation separation.** We need a language that supports clean separation between the interface and implementation mainly to keep the component upgrades and maintenance to minimum. Java supports this separation at a syntactic level through the *interface* and *class* keywords.
- **Safe and secure.** The Java architecture is much safer than traditional programming languages. In Java, if a thread dies, the application stays up. Pointers are no longer an issue. Memory leaks occur much less often. Java also has a rich library set, so that Java is not just the syntax of a language but a whole set of prewritten, debugged libraries that enable developers to avoid reinventing the wheel in a buggy way. This safety is extremely important for mission-critical applications. Sure, the overhead required to achieve this level of safety might make your application slower, but 90 percent of all business programs are glorified graphical user interfaces (GUIs) to databases. That database is going to be your number one bottleneck, not Java.

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

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

- **Lightweight open source Java frameworks such as Spring. In Chapter 11 we discuss when to use EJB versus such non-standard frameworks.**
- **Microsoft .NET–managed components, part of the Microsoft .NET platform**

EJB as a Business Tier Component

The real difference between presentation tier components such as thick clients, dynamically generated Web pages, or Web Service clients and enterprise beans is the domain in which they operate. Presentation components are well suited to handle *client-side* operations, such as rendering GUIs, executing client-side validations, constructing appropriate SOAP messages to send them to Web Service, and so on. They deal directly with the end user or business partner.

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

Specifically, EJB is used to help write logic that solves *business problems*. Typically, EJB components (enterprise beans) can perform any of the following tasks:

- **Perform business logic.** Examples include computing the taxes on the shopping cart, ensuring that the manager has authority to approve the purchase order, or sending an order confirmation e-mail using the *JavaMail API*.
- **Access a database.** Examples include submitting an order for books, transferring money between two bank accounts, or calling a stored procedure to retrieve a trouble ticket in a customer support system. Enterprise beans can achieve database access using the *Java Database Connectivity* (JDBC) *API*.

- **Access another system.** Examples include calling a high-performing *CICS* legacy system written in COBOL that computes the risk factor for a new insurance account, calling a legacy *VSAM* data store, or calling *SAP R/3*. Enterprise beans can integrate with an existing application through the *J2EE Connector Architecture* (JCA), which we will talk about in detail in Chapter 17.

Thus, EJB components are not presentation tier components; rather, they sit behind the presentation tier components (or clients) and do all the hard work. Examples of the clients that can connect to enterprise beans include the following:

- **Thick clients.** Thick clients execute on a user's desktop. They could connect through the network with EJB components that live on a server. These EJB components may perform any of the tasks listed previously (business logic, database logic, or accessing other systems). Thick clients in Java include applets and applications.
- **Dynamically generated Web pages.** Web sites that are transactional and personalized in nature need their Web pages generated specifically for each request. For example, the home page for Amazon.com is completely different for each user, depending on the user's profile. Core technologies such as Java servlets and JavaServer Pages (JSP) are used to dynamically generate such specific pages. Both servlets and JSPs live within a Web server and can connect to EJB components, generating pages differently based upon the values returned from the EJB layer.
- **Web Service clients.** Some business applications require no user interface at all. They exist to interconnect with other business partners' applications that may provide their own user interface. For example, consider a scenario where Dell Computer Corporation needs to procure Intel chips to assemble and distribute desktop computers. Here, Intel could expose an *Order Parts* Web Service that enables the Dell Web Service client to order chips. In this case, the Intel system does not provide a graphical user interface per se, but rather provides a Web Service interface. This scenario is shown in Figure 1.4.

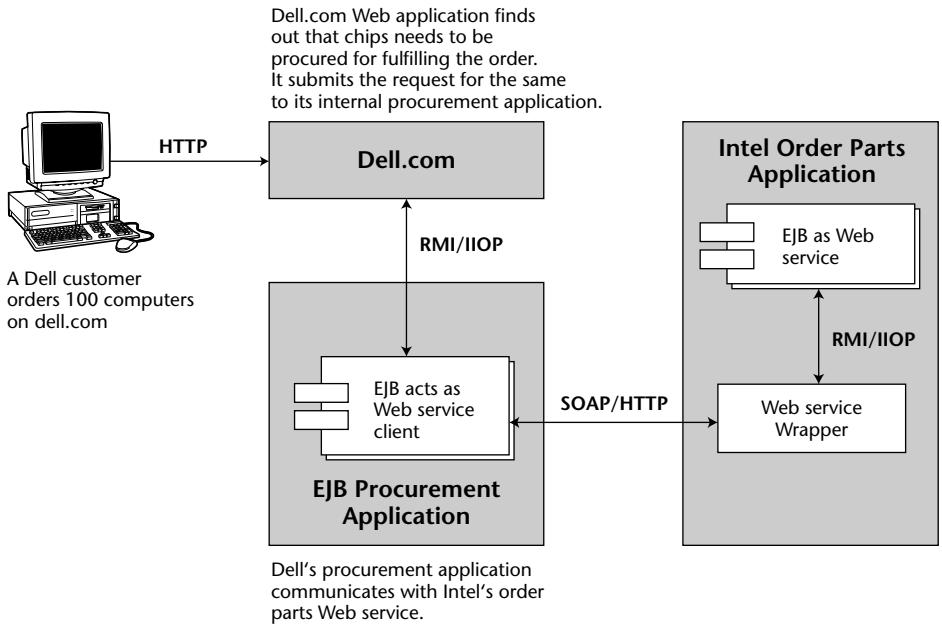

Figure 1.4 EJBs as Web Service clients.

The EJB Ecosystem

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

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

The EJB Ecosystem is not for everyone. At my company, we've heard ghastly stories of businesses choosing EJB because everyone else is using it, or because it is new and exciting. Those are the wrong reasons to use EJB and can result in disappointing results. For best practices and a discussion surrounding the issue of when and when not to use EJB, see Chapters 11 and 16.

JAVABEANS VERSUS ENTERPRISE JAVABEANS

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

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

The JavaBeans framework is lightweight compared to Enterprise JavaBeans. You can use JavaBeans to assemble larger components or to build entire applications. JavaBeans, however, are development components and are not deployable components. You typically do not deploy a JavaBean; rather, JavaBeans help you construct larger software that is deployable. And because they cannot be deployed, JavaBeans do not need to live in a runtime environment and hence, in a container. Since JavaBeans are just Java classes, they do not need an application server to instantiate them, to destroy them, and to provide other services to them. An EJB application can use JavaBeans, especially when marshalling data from EJB layer to another, say to components belonging to a presentation tier or to a non-J2EE application written in Java.

The Bean Provider

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

The Application Assembler

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

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

- From knowledge of the business problem, decide which combination of existing components and new enterprise beans are needed to provide an effective solution; in essence, plan the application assembly.
- Supply a user interface (perhaps Swing, servlet or JSP, application or applet) or a Web Service.
- Write new enterprise beans to solve some problems specific to your business problem.

- Write the code that calls on components supplied by bean providers.
- Write integration code that maps data between components supplied by different bean providers. After all, components won't magically work together to solve a business problem, especially if different parties write the components.

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

The EJB Deployer

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

- Securing the deployment with a hardware or software firewall and other protective measures.
- Integrating with enterprise security and policy repositories, which oftentimes is an LDAP server such as Sun Java System Directory Server (formerly Netscape Directory Server), Novell Directory Server, or Microsoft Active Directory.
- Choosing hardware that provides the required level of quality of service.
- Providing redundant hardware and other resources for reliability and fault tolerance.
- Performance-tuning the system.

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

An EJB deployer can be a staff person, an outside consultant, or a vendor.

The System Administrator

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

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

The Container and Server Provider

The container provider supplies an *EJB container* (the application server). This is the runtime environment in which beans live. The container supplies middleware services to the beans and manages them. There are about 20 Sun Microsystems–certified J2EE application servers. Although a complete list can be obtained from http://java.sun.com/j2ee/licensees.html, some of the popular J2EE application servers include BEA WebLogic, Sun Java System Application Server (formerly, Sun ONE Application Server), IBM WebSphere, Oracle Application Server, and of course JBoss open source application server.

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

The Tool Vendors

To facilitate the component development process, there should be a standardized way to build, manage, and maintain components. In the EJB Ecosystem, there are several *integrated development environments* (IDEs) that assist you in rapidly building and debugging components. Some of the popular closed

QUALITIES OF SERVICE IN EJB

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

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

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

source and open source EJB development IDEs include Borland JBuilder, Oracle JDeveloper, BEA WebLogic Workshop, IBM WebSphere Studio Application Developer, Sun Microsystems Java Studio (formerly Forte for Java), NetBeans, and last but not least, Eclipse.

Most of these tools enable you to model components using unified modeling language (UML), which is the diagram style used in this book. You can also generate EJB code from these UML models. Some of the examples of specialized closed source products in this space include Borland Together and IBM Rational line of products. Also there are a bunch of open source code utilities and tools, which we discuss in Chapter 11, that can be used for UML modeling and code generation.

There are other tools as well, which you will need to develop your EJB applications rapidly and successfully. The categories mainly include testing tools (JUnit), build tools (Ant/XDoclet), and profilers (Borland OptimizeIt or Quest Software JProbe).

Summary of Roles

Figure 1.5 summarizes the interaction of the different parties in EJB.

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

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

For some of the parties EJB merely suggests possible duties, such as the system administrator overseeing the well being of a deployed system. For other parties, such as the bean provider and container provider, EJB defines a set of strict interfaces and guidelines that must be followed or the entire ecosystem will break down. By clearly defining the roles of each party, EJB lays a foundation for a distributed, scalable component architecture where multiple vendors' products can interoperate.

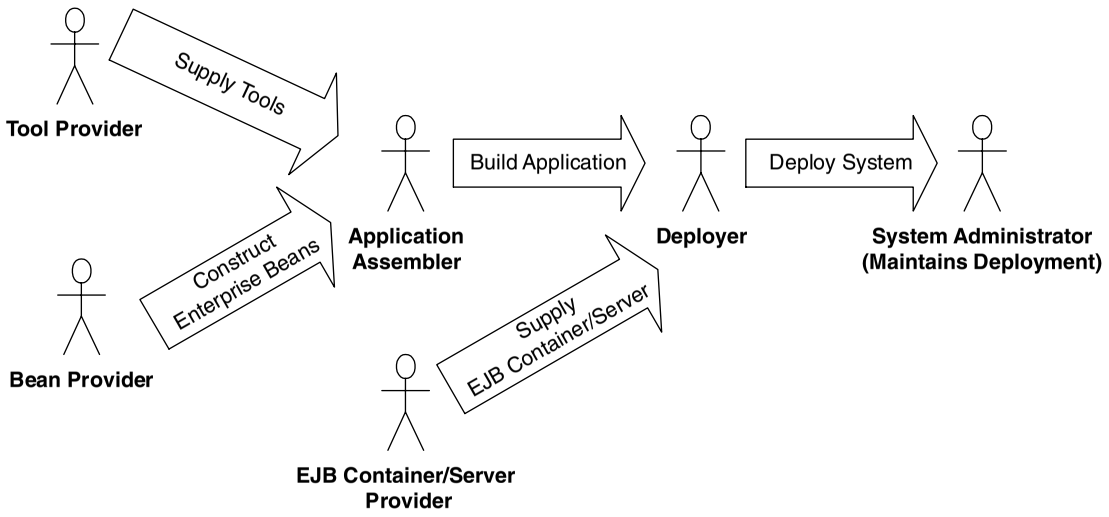

Figure 1.5 The parties of EJB.

The Java 2 Platform, Enterprise Edition (J2EE)

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

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

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

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

- **The Java 2 Platform, Micro Edition (J2ME)** is a development platform for applications running on mobile Java-enabled devices, such as Phones, Palm Pilots, Pagers, set-top TV boxes, and so on. This is a restricted form of the Java language due to the inherent performance and capacity limitations of small-form-factor wireless devices.
- **The Java 2 Platform, Standard Edition (J2SE)** defines a standard for core libraries that can be used by applets, applications, J2EE applications, mobile applications, and such. These core libraries span a much wider spectrum including input/output, graphical user interface facilities, networking, and so on. This platform contains what most people use in standard Java programming.
- **The Java 2 Platform, Enterprise Edition (J2EE)** is an umbrella standard for Java's enterprise computing facilities. It basically bundles together technologies for a complete enterprise-class server-side development and deployment platform in Java.

J2EE is significant because it creates a unified platform for server-side Java development. The J2EE stack consists of the following:

- **Specifications.** Each enterprise API within J2EE has its own specification, which is a PDF file downloadable from www.jcp.org. Each time there is a new version of J2EE, the J2EE Expert Group at JCP locks down the versions of each Enterprise API specification and bundles them together as the de facto versions to use when developing with J2EE. This increases code portability across vendors' products, because

each vendor supports exactly the same API revision. This is analogous to a company such as Microsoft releasing a new version of Windows every few years: Every time a new version of Windows comes out, Microsoft locks down the versions of the technologies bundled with Windows and releases them together.

- **Test suite.** Sun provides a test suite (a.k.a. Test Compatibility Kit or TCK) for J2EE server vendors to test their implementations against. If a server passes the tests, Sun issues a J2EE compliance brand, alerting customers that the vendor's product is indeed J2EE-compliant. There are numerous J2EE-certified vendors, and you can read reviews of their products for free on TheServerSide.com.
- **Reference implementation.** To enable developers to write code against J2EE Sun provides its own free reference implementation for each version of J2EE. Sun is positioning it as a low-end reference platform, because it is not intended for commercial use. You can download the reference implementation for J2EE 1.4, the latest version of J2EE platform that includes EJB 2.1, the technology of focus in this book, from http://java.sun.com/j2ee/download.html.

The J2EE Technologies

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

In this book, we will discuss EJB 2.1, an integral part of J2EE 1.4. Some of the major J2EE technologies are shown working together in Figure 1.6.

To understand more about the real value of J2EE, here are some of the important technologies and APIs that a J2EE 1.4-compliant implementation will support for you.

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

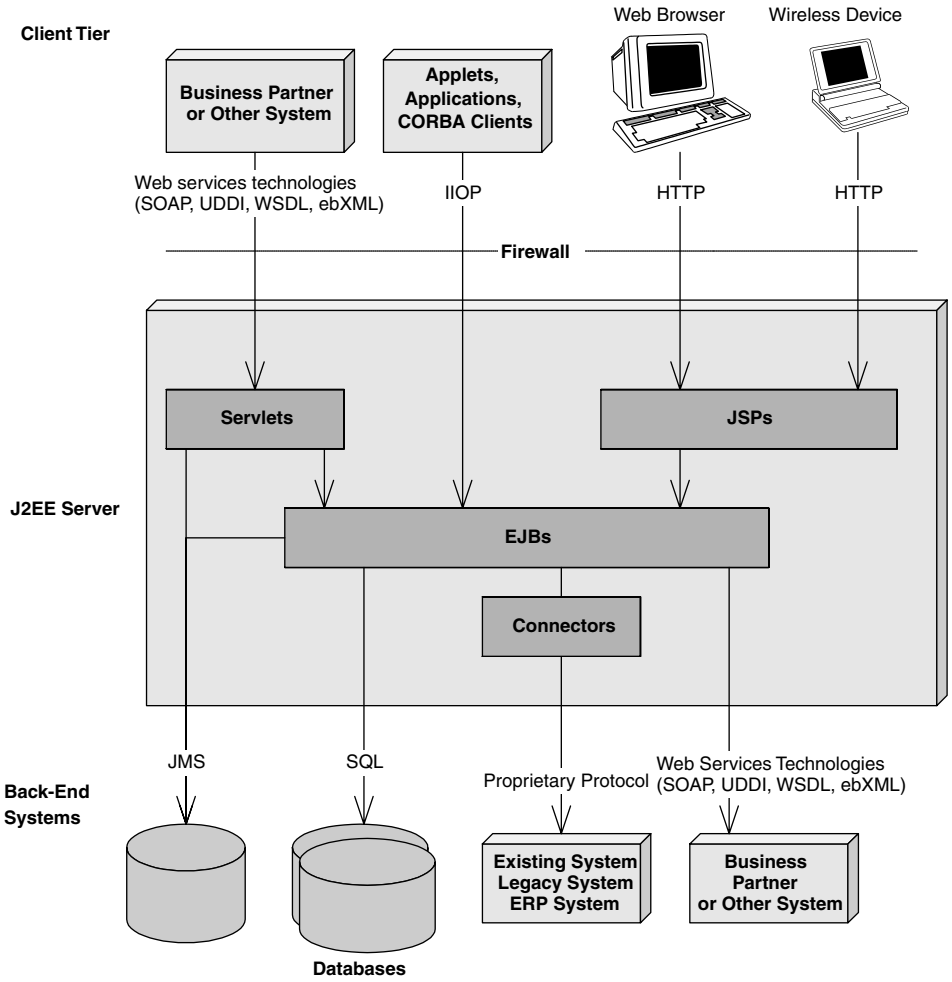

Figure 1.6 A J2EE deployment.

- **Java API for XML RPC (JAX-RPC).** JAX-RPC is the main technology that provides support for developing Web Services on the J2EE platform. It defines two Web Service endpoint models—one based on servlet technology and another based on EJB. It also specifies a lot of runtime requirements regarding the way Web Services should be supported in a J2EE runtime. Another specification called Web Services for J2EE defines deployment requirements for Web Services and uses the JAX-RPC programming model. Chapter 5 discusses support of Web Services provided by both these specifications for EJB applications.
- **Java Remote Method Invocation (RMI) and RMI-IIOP.** RMI is the Java language's native way to communicate between distributed objects, such as two different objects running on different machines. RMI-IIOP

is an extension of RMI that can be used for CORBA integration. RMI-IIOP is the official API that we use in J2EE (not RMI). We cover RMI-IIOP in Appendix A.

- **Java Naming and Directory Interface (JNDI).** JNDI is used to access naming and directory systems. You use JNDI from your application code for a variety of purposes, such as connecting to EJB components or other resources across the network, or accessing user data stored in a naming service such as Microsoft Exchange or Lotus Notes. JNDI is covered in Appendix A.
- **Java Database Connectivity (JDBC).** JDBC is an API for accessing relational databases. The value of JDBC is that you can access any relational database using the same API. JDBC is used in Chapter 7.
- **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 12.
- **Java Messaging Service (JMS).** JMS allows for your J2EE deployment to communicate using messaging. You can use messaging to communicate within your J2EE system as well as outside your J2EE system. For example, you can connect to existing message-oriented middleware (MOM) systems such as IBM MQSeries or Microsoft Message Queue (MSMQ). Messaging is an alternative paradigm to RMI-IIOP, and has its advantages and disadvantages. We explain JMS in Chapter 9.
- **Java servlets.** Servlets are networked components that you can use to extend the functionality of a Web server. Servlets are request/response oriented in that they take requests from some client host (such as a Web browser) and issue a response back to that host. This makes servlets ideal for performing Web tasks, such as rendering an HTML interface. Servlets differ from EJB components in that the breadth of server-side component features that EJB offers, such as declarative transactions, is not readily available to servlets. Servlets are much better suited to handling simple request/response needs, and they do not require sophisticated management by an application server. We illustrate using servlets with EJB in Chapter 22.
- **JavaServer Pages (JSP).** JSP technology is very similar to servlets. In fact, JSP scripts are compiled into servlets. The largest difference between JSP scripts and servlets is that JSP scripts are not pure Java code; they are much more centered on 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. JSP technology is perfect for this, and it can be easily written and maintained by non–Java-savvy staff members (JSP technology

does not require a Java compiler). We illustrate using JSP with EJB in Chapter 22.

- **Java IDL.** Java IDL is the Sun Microsystems Java-based implementation of CORBA. Java IDL allows for integration with other languages. Java IDL also allows for distributed objects to leverage the full range of CORBA services. J2EE is thus fully compatible with CORBA, completing the Java 2 Platform, Enterprise Edition. We discuss CORBA interoperability in Appendix B.
- **JavaMail.** The JavaMail service enables 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.
- **J2EE Connector Architecture (JCA).** Connectors enable you to access existing enterprise information systems from a J2EE application. This could include *any* existing system, such as a mainframe system running high-end transactions (such as those deployed with IBM CICS, or BEA TUXEDO), Enterprise Resource Planning (ERP) systems, or your own proprietary systems. Connectors are useful because they automatically manage the details of middleware integration to existing systems, such as handling transactions and security concerns, life-cycle management, thread management, and so on. Another value of this architecture is that you can write a connector to access an existing system once, and then deploy it into any J2EE-compliant server. This is important because you only need to learn how to access an existing system once. Furthermore, the connector needs to be developed only once and can be reused in any J2EE server. This is extremely useful for independent software vendors (ISVs) such as SAP, Siebel, Peoplesoft and others who want their software to be accessible from within J2EE application servers. Rather than write a custom connector for each application server, these ISVs can write a standard J2EE connector. We discuss legacy integration in more details in Chapter 17.
- **The Java API for XML Parsing (JAXP).** There are many applications of XML in a J2EE deployment. For example, you might need to parse XML if you are performing B2B interactions (such as through Web Services), if you are accessing legacy systems and mapping data to and from XML, or if you are persisting XML documents to a database. JAXP is the de facto API for parsing XML documents in a J2EE application and is an implementation-neutral interface to XML parsing technologies such as DOM and SAX. You typically use the JAXP API from within servlets, JSP, or EJB components.

- **The Java Authentication and Authorization Service (JAAS).** JAAS is a standard API for performing security-related operations in J2EE. Conceptually, JAAS also enables you to plug in an authentication mechanism into a J2EE application server. See Chapter 13 for more details on security pertaining to EJB applications.

Summary

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

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