# Chapter 1

# Introduction to Java EE for IBM WebSphere Application Server, Version 7.0

## *Chapter Topics*

- What is WebSphere?
- A brief history of Java EE
- Java EE architecture
- Introducing WebSphere

Business and machines have been partners since the Industrial Revolution. As with any other business machine, businesses incorporated computers as a means to extend the productivity of workers, complete long-running tedious tasks without complaint, and maintain business records—the lifeblood of the business—with the net result being reduced operational costs. Computers were thus used only within the business—they were not intended for the customer.

The innovation IBM termed *e-business* in the 1990s added to the tasks of business computers that of interfacing with the customer. Since then, as companies the world over struggle to remain competitive, e-business has emerged as a way to reach a potential global marketplace, while drastically reducing costs.

Take a moment to consider the significance of this: When customers think of a business, they most likely think of the services that business offers, the reliability of these services, how readily accessible they are, and how quickly the business renders the services. Taken

together, these aspects of a business represent, in the minds of its customers, the *quality* of service the business offers.

Inevitably, customers derive their perception of this quality through the "face" of the business. In the beginning, that face was human: The carrier delivered the newspaper; a banker granted the loan; and at the better grocery stores, courteous grocery staff carried the groceries to the car.

The substitution of the human face with the Web interface is perhaps the most delicate and most critical of all business transitions. Customers certainly will not expect fewer services or reduced quality from e-business, but, rather, more services and higher quality. Furthermore, the success of a business is now measured in mouse-clicks, not miles to the mall. Indeed, it is becoming very difficult to imagine a business that is not an e-business, or at least not substantially enabled by the Internet.

Change is inevitable and rapid. This is a given in the world of information technology (IT). The enterprise developer is now at the center of the most significant change in IT since the advent of the enterprise application: the widespread adoption of service-oriented architecture (SOA) as the model for e-business.

While enterprise developers still develop and deploy distributed enterprise-level applications that model an organization's processes and practices, they must also incorporate multiple applications into a higher-level architecture. In other words, the application is no longer the overarching architecture that models the business; rather, the application itself has become (perhaps) just another component.

Enterprise developers work more closely than ever before with business analysts, modelers, application architects, application assemblers, and systems administrators to apply sound object-oriented analysis and design techniques to a business model, without altering that model in any way. Developers understand application assembly within the business domain across a multitier architecture, and they must assess carefully the need for new components, the reuse of existing components, and the integration of existing computing resources that are not Java related as services. Finally, developers require system administration skills to tune the application at the code/design level to meet performance requirements.

The systems we design and build should be limited only by our imagination, not by lack of interoperability between proprietary technologies, or, worse, our ignorance of the very existence of an enabling technology. Most importantly, the technologies we use must be extremely flexible, to match the flexibility required of modern business. IBM has recognized that the open-source and open-standards movements in the IT industry best meet the needs of business, and that the merger of open-standards computing with

creative business analysis is the correct formula to make businesses grow. The WebSphere Software group has evolved to enable this merger.

## What Is WebSphere?

WebSphere is a software platform for e-business and a comprehensive suite of software offerings to support this platform. The focus of this book is the creation, using the IBM Rational Software Delivery Platform, of enterprise applications and services that will run on any server product in the WebSphere suite. In addition to Web Services creation and enterprise application development and deployment to WebSphere Application Server, the Rational Software Delivery Platform supports the most comprehensive range of tooling in the industry—a list too long to include here. Because this book addresses the role of the enterprise developer, most of the focus will be on the composite set of tooling known as IBM Rational Application Developer (hereafter referred to as Application Developer).

Beginning with Version 4 and continuing through Version 7.0, WebSphere Application Server and Application Developer are fully compliant with the Java Enterprise Edition (Java EE) specification (formerly known as J2EE), thus delivering both open-standard interoperability beyond the Java programming language and the platform independence of Java. Furthermore, Java EE allows developers to focus on the integration of IT with business processes, instead of on low-level implementation details. A brief history of Java EE can help us understand how WebSphere supports modern enterprise computing.

## A Brief History of Java EE

Java EE is the convergence of the Java Programming Language, the Common Object Request Broker Architecture (CORBA), and X/Open transaction specifications. Thus, work on Java EE really began with Sun, IBM, and others in the Object Management Group in 1989. However, the specification called J2EE did not emerge until the late 1990s in an effort to make distributed enterprise computing more agile by simplifying the tasks of development and deployment. At the heart of the roadmap to enterprise computing, Java EE promised to separate the programming logic from the technology plumbing. Developers could focus on the design aspects of computing as they developed Java components, and the infrastructure would provide the necessary transport and transaction services in an open, platform-independent way.

Major challenges of e-business include security, dynamic processing, and growth. These are real business requirements, rather than computing concepts.

First, to conduct e-business, organizations must make information usable, available, and secure. A traditional storefront business achieves security in physical ways, such as with locks and keys. To do e-business, our computing model must emulate those physical security mechanisms.

Second, data that just sits there is not business: The data needs to be manipulated—manipulated in such a way as to resemble actual, intuitively satisfying business processes (e.g., transactions). A transaction models the concept of an indivisible unit of work that supports the business.

Once we have secure transactions involving business data, we still need one ingredient to complete an e-business application: growth. Businesses that do not grow do not survive. Because growth implies that each new customer receives the same, if not improved, quality of service, a business often must acquire new physical property and hire new employees to support that growth. In computing terms, this means that throughput must be scalable, to accommodate increased demand.

CORBA led the way to vendor-independent distributed computing. Many of the good ideas of Java EE, such as services, which we will be discussing shortly, actually appeared in CORBA first.

The Java solution for the incorporation of CORBA functionality was Enterprise JavaBeans (EJBs), a way to incorporate the robustness of the CORBA Object Request Broker (ORB)'s transactional and secure services into a standard architecture. Existing CORBA implementations, although based on the same, robust CORBA standards, were not "standard" (i.e., portable and interoperable). But EJB went beyond CORBA: The designer of Java wanted the remote procedure-calling architecture to be more intuitive than the complicated and nongenial CORBA code. Incorporating the intuitiveness and ease of use of JavaBeans technology into CORBA-compliant components was the deciding factor in the industry acceptance of EJB over CORBA as the e-business solution of choice.

With EJB, Java provided a technological basis for implementing robust e-business solutions, addressing and going beyond the technologies CORBA provided. However, the picture was not complete. As programmers began to incorporate the new Java technologies into their solutions, it became apparent that there was no true architectural standard to which programmers could apply the new technologies. In other words, Java needed to evolve from a set of technologies into a framework that would standardize not only the technologies used but also the way in which they were used in architecture. This standard would become the J2EE specification, which is the basis for this entire book.

## Java EE Architecture

When the Java EE specification mentions architecture, it clearly refers to runtime architecture. (Note: Henceforth, although not always stated explicitly, this discussion is limited to the Java EE 5 framework specification.) Every discussion of this architecture must begin with the figure from the specification itself that you see in Figure 1-1.

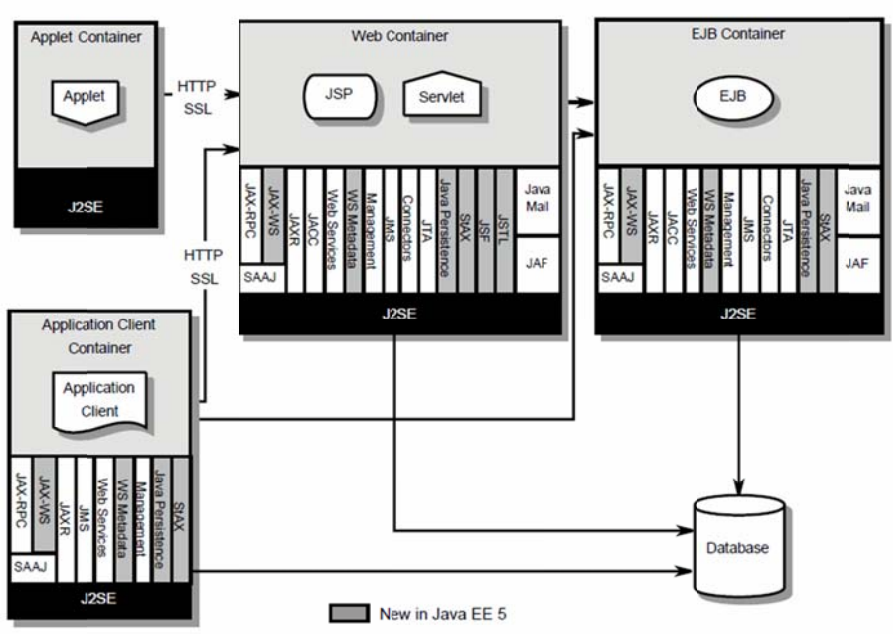


*Figure 1-1: Java EE 5 Architecture*

Although the diagram in the figure may imply a programming architecture, at some level, its main purpose is to outline two things:

1. That there are specific components such as servlets, JSPs, and EJBs.
2. That those components need specific runtime support, which containers such as the Web container and the EJB container provide.

Also implied in the figure is a connectivity specification (those arrows that connect the containers/components to one another). Although they are obviously blank in this diagram—in fact, absent from any diagram in the actual specification—the transports are "filled in" in the text of the Java EE specification.

We will use this diagram to provide an outline for most of the rest of this chapter: We will characterize the Java EE platform as being divided (implicitly) into application components, system components, and standard services (yet to come).We will begin this discussion of these three architectural elements with the application components, because their definition really implies, and was ultimately responsible for, the services that the containers, as well as the transport protocols, provide. We will treat the notion of application roles as part of the platform architecture—once again, a kind of pedagogical choice in this book, although certainly implied within the Java EE specification.

## *Java EE Architecture: Application Components*

This might be a good time to bring out a salient point in the definition of Java EE application components: The Java Platform, Enterprise Edition specification, v5 (Java EE 5 specification) says, "Application components can be divided into three categories according to their dependence on a Java EE server":

- Components that are deployed, managed, and executed on a Java EE server. These components include Web components and Enterprise JavaBeans components.
- Components that are deployed and managed on a Java EE server, but are loaded to and executed on a client machine. These components include HTML pages and applets embedded in HTML pages.
- Components whose deployment and management is not completely defined by this specification. Application Clients fall into this category. (Java EE 5 specification)

### The Application Client

Instead of starting in the upper left-hand corner of the Figure 1-1 diagram, as is typical in these reviews, we'll begin with the component labeled Application Client. Historically, this is what we've always known, simply, as a computer program. Before Java EE was released, the name for this component in Java terms was just *application*: a set of related methods, linked together by a runable method; namely, the **main()** method. At other times, this component was referred to as *standalone,* which implied that it required only the Java Virtual Machine (JVM)—no container—to execute: The JVM was given the name of the class, as an argument, which contained the **main()** method, and the JVM executed that method. At some point, the Java EE architecture designers realized that the fact that the application was standalone was inconsistent with the component model, which already had been applied to servlets and EJBs. In other words, everything, including the application, should be a component, which by definition, was a software element that required a container. Thus was born the client container, which was to run application client components.

In the Java EE 5 architecture, the application client is a largely graphical program, which gets its data from EJBs, running in an application server, via access to the Java Naming and Directory Interface (JNDI) namespace. So, put another way, the application that formerly could stand alone will now be a client to a Java EE server, with the client container providing access to both the JNDI namespace of that server and the necessary libraries (e.g., **javax.naming**).

Application clients also require a JVM to execute in—not the JVM of the application server variety, but a basic JVM that resides on the client side. We will try to flesh out the nature of these components in both the container section to follow and in the packaging/deployment discussion of chapter 16. (If, on the whole, you find the treatment

of application clients "vague," it is. The Java EE 5 specification says, "Future versions of this specification may more fully define deployment and management of Application Clients.")

### The Applet

The applet (the name implies "little application") is often compared to the application (the application client, which we have just discussed above) in early literature. In fact, many texts contained exercises that asked the developer to convert an application into an applet, and vice versa.

### The Servlet

Servlets are among the "Web components" in the first category of components mentioned at the beginning of this section. We will cover servlets in detail in chapter 3. The strength of servlets lies in their ability to generate and deliver dynamic content in response to requests.

### The JavaServer Pages (JSP)

JavaServer Pages (JSP) were a response to the problems inherent in embedding HTML in servlet code that was ultimately to be compiled by an ordinary Java compiler. Escape characters need to be included to differentiate between similar notions the compiler recognizes. Also, the compiler, or the Java editor in Rational Application Developer, cannot check the validity of the HTML. JSP technology, in contrast, enables an enterprise developer to write the functionality of a servlet in purely HTML form; the servlet's syntax is protected by the system's HTML editor, such as the Page Designer in IBM Rational Application Developer.

Moreover, a best-practice idea has been developing that recognizes a design pattern from SmallTalk called Model-View-Controller (MVC). Simply put, MVC requires that the code in each element of an application restrict itself solely to the role of that element in the overall architecture of the application. In other words, code that processes business data (fulfilling the role of model) should remain separate from code that displays the results of that processing (fulfilling the role of view). Code that delegates and coordinates between functions in the view and model—i.e., code that controls the flow between other functions—falls into the role of controller. The JSP fulfills the role of view. When used as part of MVC, the JSP page is commonly referred to as a *display page*. We include a more complete discussion of JSPs in the MVC pattern in chapter 4.

### The Filter

Often, servlets and JSP pages require preprocessing, such as logging and security checking. The filter is used to perform any preprocessing that a servlet or JSP requires.

### JavaServer Faces (JSF)

JavaServer Faces (JSF) technologies have been added to the Java EE specification to facilitate Web application development. JSF is a framework that provides developers

with components to develop Web interfaces and gives tool providers a specification in Java EE that they can build tools around.

### The Web Event Notifier

The Java Standard Edition (JSE) provides an event-notification system, which allows components to be notified of events fired by other components. For a time, this mechanism was absent from Java EE. As of J2EE 1.3, however, at the Web component level, an event-notification system, which allows Web components to trigger functions in other Web components via event notification, was added to the Web container's functions.

### The Enterprise JavaBean (EJB)

In the earliest stages of development of the EJB specification, one could say that an EJB was an inherently remote, network-aware, distributed transactional component. Certainly, the EJB was conceived as Java's response to the remote procedure call (RPC), of which CORBA and, now, Web Services are also attempts at a solution. The notion of "bean" in the name Enterprise JavaBean stems from the design intent of making EJBs reusable components similar to JavaBeans, which are intended to be self-describing Java software components. Also, EJBs advertise their properties via getters and setters, and they use Java Reflection in a comparable way to JavaBeans. But it is important to make a clear distinction between JavaBeans and Enterprise JavaBeans.

With the advent of EJB 2.0, one can no longer define an EJB as being inherently remote, since local EJBs are inherently not remote but are still EJBs. Insofar as local EJBs are not remote, they also are not network aware and, likewise, not truly distributed.

Whereas the Java EE 5 specification states that the EJBs "typically contain the business logic of a Java EE application," the entity EJB is probably used more for persistence than for execution of business logic. The session EJB could be used to execute business logic, but there are contending design patterns, most notably the EJB Command pattern, that would place the business logic in JavaBeans and use the stateless session EJBs as "transaction listeners"—presenting an interface to coarse-grained business processes.

Because of inherent problems in the concept of entity EJBs, and the requirements put on the shoulders of developers who were responsible for their design, implementation, and deployment, the Java Persistence API (JPA) and annotations in Java EE are becoming very popular as an alternative to the traditional entity bean mapped to a backend data source.

## *Java EE Architecture: System Components*

We continue to follow the organization implied in Figure 1-1, in which the Java EE platform is divided (implicitly) into application components (which we have just discussed), system components (the current section), and standard services (yet to come).

When we refer to system components, we are talking about those features in the architecture that provide the runtime environment for the application components. At one point historically, system components were loosely referred to as *engines*—for example, there was reference to the "servlet engine." This notion would have been useful to carry forward, but it seems to have gotten lost. However, *engine* would not have allowed the easy incorporation of a resource adapter, and the resource adapter is indeed a system-level runtime component.

Let's characterize system components as those components that provide the runtime layer between the application components and the external operating system, and any other external runtimes (e.g., enterprise information system (EIS).

## Resource Adapters

The resource adapter is somewhat ambiguous in that, while it is indeed a component, it does not need to be a part of a Java EE Enterprise application, nor does it execute in a Java EE container. It is used to provide a bridge between Java applications and non-Java applications. For example, if a Java EE application needs to call and retrieve data from the Customer Information Control System (CICS), a CICS resource adapter is installed in the Java EE server. Interestingly, since the relational database is technically a non-Java application (true, DB2 is implemented in Java), WebSphere wraps database access in a relational resource adapter.

## Containers

A container's basic function is to manage and provide the runtime support for the components it contains. This support can be broken down into four separate functions:

1. To provide access to all the necessary Java EE and JSE APIs (essentially, class libraries) that the components will call.
2. To isolate the contained components from the other components in the Java EE server.
3. To take care of the communication between components running in different containers.
4. To manage the life cycles of objects, creating them so that they are available when requested and destroying them when they are no longer needed. The final function includes providing a way for the objects to persist themselves.

The necessity of the first function is clear: Precompiled class libraries have long been used as a means of deferring many of the mechanical complexities of the runtime from the programmer. This makes programming more intuitive, closer to resembling human thought. The second function, that of isolation, provides the essence of this programming model: It allows the runtime to be uniquely configured and managed through the use of deployment descriptors, which the containers read and implement at deployment time. As stated, the third function is really a logical follow-on of the first two: The complexities of intercomponent communication are deferred to the containers, thus freeing the

programmer from the details, while at the same time allowing for isolation or grouping of varied processes.

With regard to the first function of providing necessary APIs, we can see from Figure 1-1 that all containers must provide the Java 2 Platform Standard Edition 5 (J2SE 5) APIs (many people believe all package names in J2SE 5 begin with the topmost package name of java—e.g., **java.sql**; but there are plenty of **javax** and **org** packages). The Applet container, uniquely, can provide the JSE through the Java plug-in product; most of us have seen the dialog box concerning this plug-in after we have installed a new browser.

The diagram also shows that containers must provide access to additional APIs from the Java EE, although not uniformly but on a per-container basis. (These packages all have **javax** for the topmost package name, indicating they are standard extensions to Java.) These APIs often work in conjunction with the J2SE 5 packages to support the services the containers provide, which we discuss in the following section.

Because containers must perform the functionality outlined in the deployment descriptors, it goes without saying that the containers must understand the packaging conventions and instructions contained in the deployment descriptors.

### Java EE Server

Although JVMs are all based on the standard J2SE technology, they can be highly specialized to provide server-side functionality. Such a JVM is referred to as a *Java EE server* or, variously, an *application server* (the server being simply a set of threads that run through instances of classes running on a standard JVM). The Java EE server is minimally a JVM that is capable of managing the containers described above, with some additional standard transaction-processing infrastructure. Also, there must be some mechanism for interaction with application clients. The Java EE server must, ultimately, fully implement the Java EE specification.

### Drivers

The functionality of Java EE servers will be extended through their coupling (network connectivity) with external resource managers, by implementing Java EE Service Provider Interfaces (SPIs). The SPI guarantees that the implementation classes will work with any and all Java EE products. These implementation classes are packaged software collections, referred to as *resources manager drivers,* or, most commonly, just *drivers.* Many external resource applications have corresponding Java APIs, such as Java Database Connectivity (JDBC). Other external applications using non-Java APIs can be connected to Java drivers through connectors.

### Database

As stated in the Java EE specification 5, the Java EE platform requires a database that is accessible by all components, with the optional exception of applets, through the JDBC

API. It follows then that all such database vendors and/or third-party vendors would have to provide JDBC drivers.

### JPA

The JPA is a standard for mapping objects to relational databases that Java EE requires. This provides a standard management API for both Java EE and JSE components.

### Management

For a time, there was no unified way to manage a Java EE server's runtime: A plethora of administration clients was developed, each with different capabilities. The Java EE management specification provides an API to manage Java EE servers. There are additional APIs under the heading Java Management Extensions (JMX)—obviously extensions to the core APIs. Although the two are often interchanged, they are not the same thing.

### Web Services

While Web services evolved independently of Java EE, Java EE incorporated a number of APIs in support of Web services. These include the Java API for XML-based RPC (JAX-RPC) and its successor, the Java API for XML Web Services (JAX-WS), which provides support for Web service calls using the SOAP/HTTP protocol. The SOAP with Attachments API for Java (SAAJ) provides support for manipulating low-level SOAP messages; the Java Architecture for XML Binding (JAXB) defines the mapping between Java classes and XML; the Web Services for Java EE specification fully defines the deployment of Web service clients and Web service endpoints in Java EE, as well as the implementation of Web service endpoints using enterprise beans; and the Java API for XML Registries (JAXR) provides client access to XML registry servers.

## *Java EE Architecture: Standard Services*

In its effort to standardize the distributed computing environment, the Java EE Platform requires a set of services. The term *services* is rather all-encompassing and so might benefit from a bit of delineation at this point. The following paragraphs outline elements of the specification that are considered services.

### Naming and Directory

The Java Naming and Directory Interface service delivers a critical piece of the puzzle with regard to distributed enterprise computing by providing the ability to organize and locate components. Seen from the outside, this is simply the ability to bind a name to an object so that that same object can later be located by only its name. A "name" object is simply a string identifier bound to a specific location in memory or on a network. A directory is a more robust "name" object that has attributes. For example, a Microsoft Windows shortcut is an example of an icon that is bound, somehow internally within the operating system (OS), to the location of the executable that is launched when a user clicks the icon. Likewise, the directory `C:\WINDOWS` is the string identifier `"C:\WINDOWS"`

bound to the location in storage at which **"files"** (another type of directory) for the OS are located. This **WINDOWS** directory has attributes, such as size and access permissions.

## Resource Processing and Compilers

Extensible Markup Language (XML) has become an integral format for structured data transfer. Its powerful structured data processing is reliant on a strict parsing mechanism that can load highly configurable parsing grammars. Two such parsing engines are referred to as Document Object Model (DOM) and Simple API for XML (SAX). DOM parses documents by creating a tree structure from the grammar and storing that tree along with the parsed data, so that it can be reconstructed. The SAX parser parses without the tree structure—that is, in a linear, event style, thus saving on memory space. The Java API for XML Parsing (JAXP) has APIs for both these parsing paradigms, as well as APIs for XML Stylesheet Language for Transformations (XSLT) transform engines. The Streaming API for XML (StAX) provides parsing capabilities for XML in applications that require the capability to pull XML elements (in contrast to XML being pushed at an application). Thus, Java EE components are guaranteed interoperability with XML-centric applications.

## EIS Interoperability Enablement

The Java EE Connector Architecture (JCA) is perhaps one of the most powerful services added to the Java EE specification. The goal was to enable existing Enterprise Information Systems (EIS) that had over the years proven their robustness, but that were neither necessarily object-oriented nor CORBA-enabled, to be incorporated into the Java EE architecture. Examples of such systems include the industry-tested CICS, from IBM; in-house applications, such as those employed by JD Edwards; and third-party workflow systems, such as PeopleSoft. It was clear that such systems could, on an individual basis, connect to Java objects in much the same way the messaging and database connectivity had been achieved—through a type of adapter.

## Security

Clearly, the most difficult standardization effort would be security. Security mechanisms, even physical ones, can be characterized as *nonstandard*—that is, we wouldn't want just one key that would allow entry to any car! In the J2EE 1.2 specification, inroads to such a standardization were attempted with a broad set of application requirements, including authentication; application-internal (thus, portable) authorization; and a delegation mechanism for EJB, referred to as "run-as" mode. At the 1.3 level, rather than relying on generalities (which would, of necessity, be implemented in a wide variety of proprietary ways), the specification brought to the fore a set of Java APIs referred to collectively as Java Authentication and Authorization Service(s) (JAAS). JAAS is actually an extension of the pluggable authentication module (PAM) framework. The Java Authorization Service Provider Contract for Containers (JACC) defines a standard contract between Java EE servers and authorization service providers to allow pluggable authorization providers for Java EE, further expanding security options.

### Asynchronous Messaging

Often in an e-business application, processes must be *synchronous*—in this sense, we mean that the pieces of a transaction must be executed serially, each piece waiting for completion before continuing to the next. For example, the output of one action must be the input of another action. The subsequent action cannot proceed before the prior action has completed successfully.

While this is desirable for transactions whose flow must be serial, it has serious performance implications. The locking behavior in databases, connections, EJBs, and so on can slow the processing of many such transactions to an unacceptable level—even to the point of halting the processing of the entire set of transactions (deadlock). Yet, given the critical nature of the transactional integrity of today's e-business applications, this is a necessary evil.

The Java EE architecture requires that its components have access to such systems in a standardized way. The Java Message Service (JMS) offers to Java EE components both point-to-point and publish-subscribe asynchronous messaging, and it requires an external service that can offer these options. Point-to-point messaging is characterized by the notion that there must be a unique receiver for a given message, and that that receiver must be connected at the time of delivery. The message delivery is considered successful when the specific receiver acknowledges receipt of the message. Publish-subscribe implies that messages are sent to many interested receivers, but the messages are retained in a queue and delivered when a subscriber connects. Message delivery is said to be successful when all subscribers have received a given message.

The message provider is configured administratively, and the service is offered through a JNDI lookup. We will cover JMS in the context of message-driven beans in chapter 9.

### Transactions

The backbone of any enterprise application is the ability to provide transactional integrity for all the components and services mentioned in the Java EE architecture. This integrity is made possible through two sets of Java interfaces: the Java Transaction API (JTA) and the Java Transaction Service (JTS). Chapter 11 provides a thorough discussion of transactions.

## *Java EE Architecture: Roles*

At this point, most of our discussion is complete. We have defined a programming model, the component style, delineating thoroughly the types of software components that are required. In turn, the component design model implies a highly stylized runtime environment, and we have delineated all of the services required of this runtime, even breaking the runtime down into tiers of runtime support. Thus, the mechanics are in place and ready to go. Because of the complexities of full Java EE implementations, certain responsibilities must be assigned to persons and firms having specific roles, so that all the

tasks necessary for the development and deployment of enterprise applications are accomplished.

## Java EE Product Provider

IBM WebSphere Application Server is a Java EE product that provides the containers and services, as well as other related features. These products are generally referred to as *servers* in the architecture. They provide the runtime infrastructure. They do not refer to development tools, which have their own role distinction. Thus, vendors such as IBM provide Java EE-compliant application servers, database servers, and Web servers. Because many of the services required of these servers are at the system level, OS vendors such as IBM are well suited as product providers. The product provider must provide the system-level integration with all the services the containers and server require, but not necessarily the services themselves; vendors of messaging services, transaction services, email services, and the like are not necessarily Java EE product providers.

It is important to note that Java EE describes the minimum requirements for product providers, specifically outlining the places where proprietary value-adds and enhancements may be offered and where they may not.

The product provider also is responsible for the mapping to services tools, deployment tools, and systems administration tools.

## Tool Provider

Tool providers make tools for packaging and development of components, for deployment into Java EE-compliant containers. IBM is such a provider. IBM Rational Application Developer for WebSphere Software, Rational Software Architect for WebSphere Software, and WebSphere Integration Developer are prime examples of IBM's contributions as a Java EE tool provider.

The tool specification draws a distinction at the point of deployment. Up to deployment, these tools can be platform independent. Deployment and post-deployment tools may be platform dependent; that is, tools for deployment, monitoring, and management of applications do not have to be interoperable.

## Application Component Provider

Component providers are developers of Java EE components. The specification implies subroles, such as HTML designers and EJB programmers, but ultimately leaves the sublist open. An enterprise developer using IBM Rational Application Developer for WebSphere Software Version 7.5 is a prime example of an application component provider.

Interestingly, application component providers do not assemble their components into complete enterprise applications. Also, they do not necessarily implement security in

their EJB/Web methods. These functions are deferred to the application assembler and the system administrator, respectively, per the strict definition in the Java EE specification.

### System Component Provider

This role was added to Java EE 1.4 to separate those components that were solely Java-centric from those that incorporated EIS technologies—specifically, resource adapters.

### Application Assembler

The application assembler puts it all together, optionally assembling components and modules into enterprise applications. Because of the advanced tool-to-tool integration between IBM Rational Application Developer and WebSphere Application Server, an application assembler is not always required. The complete enterprise application is ultimately packaged into an enterprise archive, known as an Enterprise Archive (EAR) file. In chapter 16, we will explore packaging, as well as deployment and installation.

### Deployer

The deployer is responsible for installation and configuration of an enterprise application, which can include customization for the operating environment. In Java EE, deployment can entail generation and compilation of the Java source in order to generate the stubs, ties, and implementation classes specific to the operating environment. During the installation of the application, the deployer maps the logical security roles onto real principals (users and groups) defined in the authentication server. Thus, a deployer may need to understand the application, as well as the security infrastructure of the enterprise, in order to assign the roles to users correctly. The deployer does not establish security policy but does, in a sense, implement security policy. Also, it is at this point that JNDI names are configured and placed in the Java namespace. Finally, the deployer runs the application.

### System Administrator

The system administrator's role doesn't really begin until the application is installed. Of course, it will probably fall to the administrator to install WebSphere Application Server in the first place, create additional clones of that application server, and federate nodes on a multiserver environment. Clearly, the administrator is solely responsible for the configuration of networking and the general computing environment. Finally, it is the administrator's job to keep the application running, and running as quickly as possible. To this end, the administrator must collect performance data, using the Performance Monitoring Infrastructure (PMI) and Java Virtual Machine Profiler Interface (JVMPI) — special interfaces implemented by the JVM to produce performance data, which we will discuss in detail in chapter 15—and understand the application's baseline performance. Eventually, the administrator will tune the Java EE server.

## The WebSphere Java EE Toolset

As stated at the beginning of this chapter, WebSphere is a software platform for e-business. Let us now introduce in further detail the two products we will be concerned with for the remainder of this book.

### *WebSphere Application Server*

The specification describes a minimum set of facilities that all Java EE products must provide. Most Java EE products will provide facilities beyond the minimum required by this specification. This specification includes only a few limits to a product's ability to provide extensions. In particular, it includes the same restrictions as JSE on extensions to Java APIs. A Java EE product may not add classes to the Java programming language packages included in this specification, and it may not add methods or otherwise alter the signatures of the specified classes.

However, many other extensions are allowed. A Java EE product may provide additional Java APIs, either other Java optional packages or other (appropriately named) packages. A Java EE product may include support for additional protocols or services not specified here. A Java EE product may support applications written in other languages, or it may support connectivity to other platforms or applications.

WebSphere Application Server Version 7.0 is IBM's latest implementation of the Java EE application server. While it is compliant with the JEE 5 specification we have examined, WebSphere Application Server Version 7.0 includes added functionality, as allowed in the Java EE specification, including IBM's high-performance Web services runtime and delivery system, as well as Programming Model Extensions (PMEs), formerly only available exclusively in the Enterprise edition of WebSphere Application Server. The packaging set comprises an Express edition, a Network Deployment edition, and an Extended Deployment edition. The administrative user interface (UI) of this edition is almost identical to that of Version 6—that is, Web browser-based. The server configuration is maintained entirely in an XML repository (a *profile*). All editions of the product comply with all Java EE server requirements, including EJB.

The Agent Controller is provided with this version, which allows for remote and local debugging, performance profiling, and publishing integration with the Rational Application Developer set of development products.

Finally, WebSphere Application Server Version 7.0 comes with IBM HTTP Server Version 7.0, the latest build of IBM's adaptation of the Apache open-source Web server.

### *IBM Rational Application Developer*

As a Java EE tool provider, IBM offers a selection of development tools to fit a wide variety of developer needs. IBM Rational Application Developer is based on the open-source Eclipse platform (*www.eclipse.org*). Varying degrees and types of functionality are added to the Eclipse platform to suit individual development needs. IBM has sought to provide a broad set of test configurations of WebSphere Application Server, which support these varying degrees of functionality. WebSphere Integration Developer (WID) adds the Eclipse Workbench, Process Choreographer, a toolset for creating SOA solutions, custom JCA, and wizards to help deploy those adapters, EJB, and JavaBeans as Web services. WID is beyond the scope of this book.

## Summary

We have now introduced WebSphere as IBM's foundation server product that supports enterprise computing as it relates to the open-standard platform of Java EE, which was a major influence on the evolution of WebSphere. In the next chapter, we will introduce the industry-standard development environment for component-based enterprise applications and SOA solutions: IBM Rational Application Developer Version 7.5.