
Overview

TODAY, more and more developers want to write distributed transactional applications for the enterprise and thereby leverage the speed, security, and reliability of server-side technology. If you are already working in this area, you know that in the fast-moving and demanding world of e-commerce and information technology, enterprise applications must be designed, built, and produced for less money, with greater speed, and with fewer resources than ever before.

To reduce costs and fast-track application design and development, the Java™ 2 Platform, Enterprise Edition (J2EE™) provides a component-based approach to the design, development, assembly, and deployment of enterprise applications. The J2EE platform offers a multitiered distributed application model, reusable components, a unified security model, flexible transaction control, and web services support through integrated data interchange on Extensible Markup Language (XML)-based open standards and protocols.

Not only can you deliver innovative business solutions to market faster than ever, but also your platform-independent J2EE component-based solutions are not tied to the products and application programming interfaces (APIs) of any one vendor. Vendors and customers enjoy the freedom to choose the products and components that best meet their business and technological requirements.

This tutorial uses examples to describe the features and functionalities available in the J2EE platform version 1.4 for developing enterprise applications. Whether you are a new or an experienced developer, you should find the examples and accompanying text a valuable and accessible knowledge base for creating your own solutions.

If you are new to J2EE enterprise application development, this chapter is a good place to start. Here you will review development basics, learn about the J2EE architecture and APIs, become acquainted with important terms and concepts, and find out how to approach J2EE application programming, assembly, and deployment.

Distributed Multitiered Applications

The J2EE platform uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the various application components that make up a J2EE application are installed on different machines depending on the tier in the multitiered J2EE environment to which the application component belongs. Figure 1–1 shows two multitiered J2EE applications divided into the tiers described in the following list. The J2EE application parts shown in Figure 1–1 are presented in J2EE Components (page 3).

- Client-tier components run on the client machine.
- Web-tier components run on the J2EE server.
- Business-tier components run on the J2EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Although a J2EE application can consist of the three or four tiers shown in Figure 1–1, J2EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the J2EE server machine, and the database or legacy machines at the back end. Three-tiered applications that run in this way extend the standard two-tiered client and server model by placing a multithreaded application server between the client application and back-end storage.

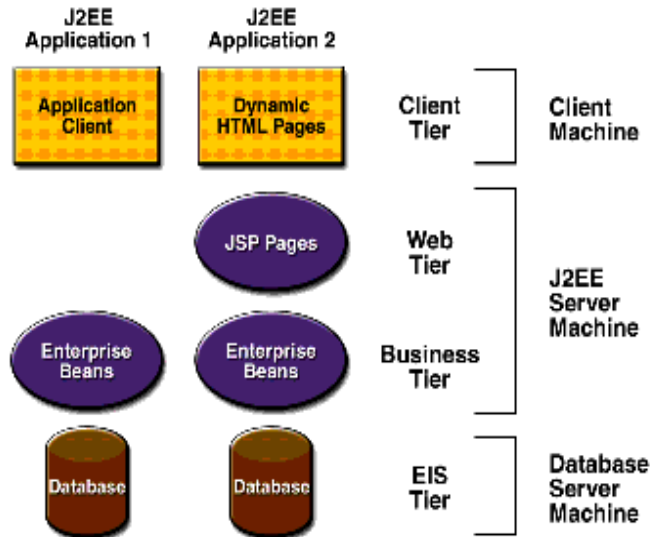


Figure 1-1 Multitiered Applications

J2EE Components

J2EE applications are made up of components. A *J2EE component* is a self-contained functional software unit that is assembled into a J2EE application with its related classes and files and that communicates with other components. The J2EE specification defines the following J2EE components:

- Application clients and applets are components that run on the client.
- Java Servlet and JavaServer Pages™ (JSP™) technology components are web components that run on the server.
- Enterprise JavaBeans™ (EJB™) components (enterprise beans) are business components that run on the server.

J2EE components are written in the Java programming language and are compiled in the same way as any program in the language. The difference between J2EE components and “standard” Java classes is that J2EE components are assembled into a J2EE application, are verified to be well formed and in compliance with the J2EE specification, and are deployed to production, where they are run and managed by the J2EE server.

J2EE Clients

A J2EE client can be a web client or an application client.

Web Clients

A *web client* consists of two parts: (1) dynamic web pages containing various types of markup language (HTML, XML, and so on), which are generated by web components running in the web tier, and (2) a web browser, which renders the pages received from the server.

A web client is sometimes called a *thin client*. Thin clients usually do not query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, such heavyweight operations are off-loaded to enterprise beans executing on the J2EE server, where they can leverage the security, speed, services, and reliability of J2EE server-side technologies.

Applets

A web page received from the web tier can include an embedded applet. An *applet* is a small client application written in the Java programming language that executes in the Java virtual machine installed in the web browser. However, client systems will likely need the Java Plug-in and possibly a security policy file in order for the applet to successfully execute in the web browser.

Web components are the preferred API for creating a web client program because no plug-ins or security policy files are needed on the client systems. Also, web components enable cleaner and more modular application design because they provide a way to separate applications programming from web page design. Personnel involved in web page design thus do not need to understand Java programming language syntax to do their jobs.

Application Clients

An *application client* runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from the Swing or the Abstract Window Toolkit (AWT) API, but a command-line interface is certainly possible.

Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a servlet running in the web tier.

The JavaBeans™ Component Architecture

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans components) to manage the data flow between an application client or applet and components running on the J2EE server, or between server components and a database. JavaBeans components are not considered J2EE components by the J2EE specification.

JavaBeans components have properties and have get and set methods for accessing the properties. JavaBeans components used in this way are typically simple in design and implementation but should conform to the naming and design conventions outlined in the JavaBeans component architecture.

J2EE Server Communications

Figure 1–2 shows the various elements that can make up the client tier. The client communicates with the business tier running on the J2EE server either directly or, as in the case of a client running in a browser, by going through JSP pages or servlets running in the web tier.

Your J2EE application uses a thin browser-based client or thick application client. In deciding which one to use, you should be aware of the trade-offs between keeping functionality on the client and close to the user (thick client) and off-loading as much functionality as possible to the server (thin client). The more functionality you off-load to the server, the easier it is to distribute, deploy, and manage the application; however, keeping more functionality on the client can make for a better perceived user experience.

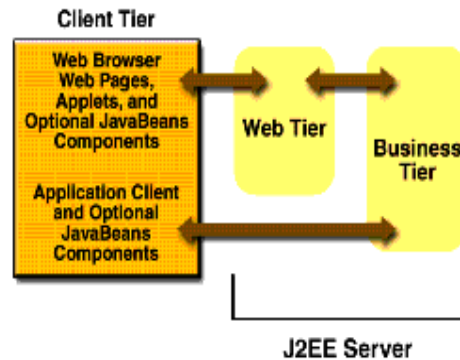


Figure 1–2 Server Communications

Web Components

J2EE web components are either servlets or pages created using JSP technology (JSP pages). *Servlets* are Java programming language classes that dynamically process requests and construct responses. *JSP pages* are text-based documents that execute as servlets but allow a more natural approach to creating static content.

Static HTML pages and applets are bundled with web components during application assembly but are not considered web components by the J2EE specification. Server-side utility classes can also be bundled with web components and, like HTML pages, are not considered web components.

As shown in Figure 1–3, the web tier, like the client tier, might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.

Business Components

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in the business tier. Figure 1–4 shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enter-

prise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.

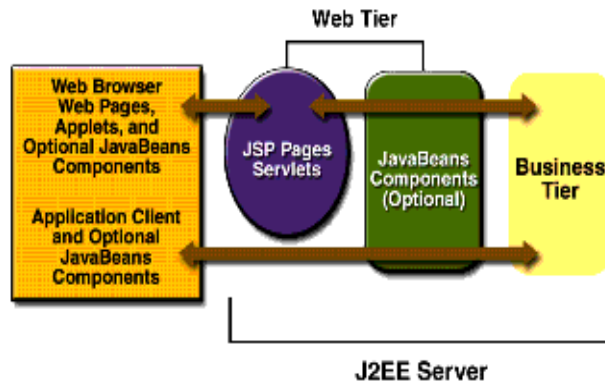


Figure 1-3 Web Tier and J2EE Applications

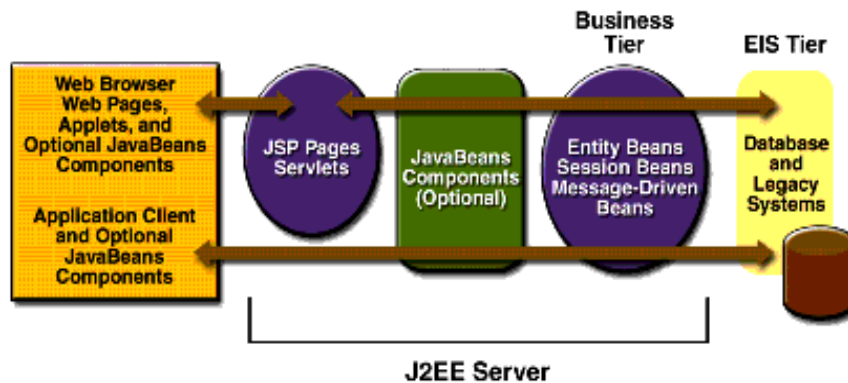


Figure 1-4 Business and EIS Tiers

There are three kinds of enterprise beans: session beans, entity beans, and message-driven beans. A *session bean* represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone. In contrast, an *entity bean* represents persistent data stored in one row of a database table. If the client terminates or if the server shuts down, the underlying services ensure that the entity bean data is saved. A *message-driven bean* combines fea-

tures of a session bean and a Java Message Service (JMS) message listener, allowing a business component to receive JMS messages asynchronously.

Enterprise Information System Tier

The enterprise information system tier handles EIS software and includes enterprise infrastructure systems such as enterprise resource planning (ERP), main-frame transaction processing, database systems, and other legacy information systems. For example, J2EE application components might need access to enterprise information systems for database connectivity.

J2EE Containers

Normally, thin-client multitiered applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details. The component-based and platform-independent J2EE architecture makes J2EE applications easy to write because business logic is organized into reusable components. In addition, the J2EE server provides underlying services in the form of a container for every component type. Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.

Container Services

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Before a web component, enterprise bean, or application client component can be executed, it must be assembled into a J2EE module and deployed into its container.

The assembly process involves specifying container settings for each component in the J2EE application and for the J2EE application itself. Container settings customize the underlying support provided by the J2EE server, including services such as security, transaction management, Java Naming and Directory

Interface™ (JNDI) lookups, and remote connectivity. Here are some of the highlights:

- The J2EE security model lets you configure a web component or enterprise bean so that system resources are accessed only by authorized users.
- The J2EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access naming and directory services.
- The J2EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

Because the J2EE architecture provides configurable services, application components within the same J2EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages nonconfigurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the J2EE platform APIs described in section J2EE 1.4 APIs (page 15). Although data persistence is a nonconfigurable service, the J2EE architecture lets you override container-managed persistence by including the appropriate code in your enterprise bean implementation when you want more control than the default container-managed persistence provides. For example, you might use bean-managed persistence to implement your own finder (search) methods or to create a customized database cache.

Container Types

The deployment process installs J2EE application components in the J2EE containers illustrated in Figure 1–5.

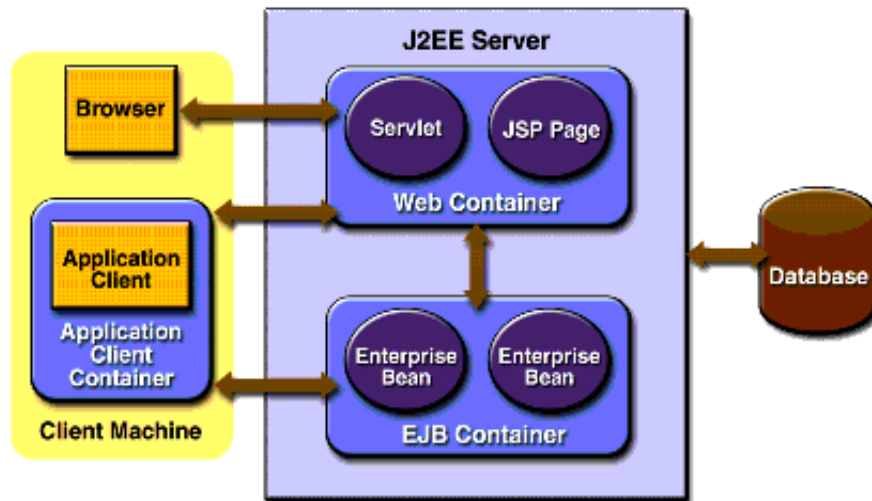


Figure 1–5 J2EE Server and Containers

J2EE server

The runtime portion of a J2EE product. A J2EE server provides EJB and web containers.

Enterprise JavaBeans (EJB) container

Manages the execution of enterprise beans for J2EE applications. Enterprise beans and their container run on the J2EE server.

Web container

Manages the execution of JSP page and servlet components for J2EE applications. web components and their container run on the J2EE server.

Application client container

Manages the execution of application client components. Application clients and their container run on the client.

Applet container

Manages the execution of applets. Consists of a web browser and Java Plug-in running on the client together.

Web Services Support

Web services are web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with calling clients. The J2EE

platform provides the XML APIs and tools you need to quickly design, develop, test, and deploy web services and clients that fully interoperate with other web services and clients running on Java-based or non-Java-based platforms.

To write web services and clients with the J2EE XML APIs, all you do is pass parameter data to the method calls and process the data returned; or for document-oriented web services, you send documents containing the service data back and forth. No low-level programming is needed because the XML API implementations do the work of translating the application data to and from an XML-based data stream that is sent over the standardized XML-based transport protocols. These XML-based standards and protocols are introduced in the following sections.

The translation of data to a standardized XML-based data stream is what makes web services and clients written with the J2EE XML APIs fully interoperable. This does not necessarily mean that the data being transported includes XML tags because the transported data can itself be plain text, XML data, or any kind of binary data such as audio, video, maps, program files, computer-aided design (CAD) documents and the like. The next section introduces XML and explains how parties doing business can use XML tags and schemas to exchange data in a meaningful way.

XML

XML is a cross-platform, extensible, text-based standard for representing data. When XML data is exchanged between parties, the parties are free to create their own tags to describe the data, set up schemas to specify which tags can be used in a particular kind of XML document, and use XML stylesheets to manage the display and handling of the data.

For example, a web service can use XML and a schema to produce price lists, and companies that receive the price lists and schema can have their own stylesheets to handle the data in a way that best suits their needs. Here are examples:

- One company might put XML pricing information through a program to translate the XML to HTML so that it can post the price lists to its intranet.
- A partner company might put the XML pricing information through a tool to create a marketing presentation.
- Another company might read the XML pricing information into an application for processing.

SOAP Transport Protocol

Client requests and web service responses are transmitted as Simple Object Access Protocol (SOAP) messages over HTTP to enable a completely interoperable exchange between clients and web services, all running on different platforms and at various locations on the Internet. HTTP is a familiar request-and-response standard for sending messages over the Internet, and SOAP is an XML-based protocol that follows the HTTP request-and-response model.

The SOAP portion of a transported message handles the following:

- Defines an XML-based envelope to describe what is in the message and how to process the message
- Includes XML-based encoding rules to express instances of application-defined data types within the message
- Defines an XML-based convention for representing the request to the remote service and the resulting response

WSDL Standard Format

The Web Services Description Language (WSDL) is a standardized XML format for describing network services. The description includes the name of the service, the location of the service, and ways to communicate with the service. WSDL service descriptions can be stored in UDDI registries or published on the web (or both). The Sun Java System Application Server Platform Edition 8 provides a tool for generating the WSDL specification of a web service that uses remote procedure calls to communicate with clients.

UDDI and ebXML Standard Formats

Other XML-based standards, such as Universal Description, Discovery and Integration (UDDI) and ebXML, make it possible for businesses to publish information on the Internet about their products and web services, where the information can be readily and globally accessed by clients who want to do business.

Packaging Applications

A J2EE application is delivered in an Enterprise Archive (EAR) file, a standard Java Archive (JAR) file with an `.ear` extension. Using EAR files and modules makes it possible to assemble a number of different J2EE applications using some of the same components. No extra coding is needed; it is only a matter of assembling (or packaging) various J2EE modules into J2EE EAR files.

An EAR file (see Figure 1–6) contains J2EE modules and deployment descriptors. A *deployment descriptor* is an XML document with an `.xml` extension that describes the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the J2EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

There are two types of deployment descriptors: J2EE and runtime. A *J2EE deployment descriptor* is defined by a J2EE specification and can be used to configure deployment settings on any J2EE-compliant implementation. A *runtime deployment descriptor* is used to configure J2EE implementation-specific parameters. For example, the Sun Java System Application Server Platform Edition 8 runtime deployment descriptor contains information such as the context root of a web application, the mapping of portable names of an application's resources to the server's resources, and Application Server implementation-specific parameters, such as caching directives. The Application Server runtime deployment descriptors are named `sun-moduleType.xml` and are located in the same directory as the J2EE deployment descriptor.

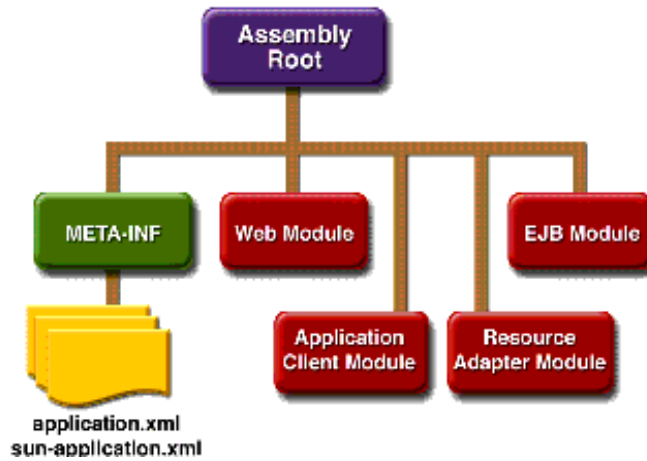


Figure 1–6 EAR File Structure

A *J2EE module* consists of one or more J2EE components for the same container type and one component deployment descriptor of that type. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. A J2EE module without an application deployment descriptor can be deployed as a *stand-alone* module. The four types of J2EE modules are as follows:

- EJB modules, which contain class files for enterprise beans and an EJB deployment descriptor. EJB modules are packaged as JAR files with a `.jar` extension.
- Web modules, which contain servlet class files, JSP files, supporting class files, GIF and HTML files, and a web application deployment descriptor. Web modules are packaged as JAR files with a `.war` (web archive) extension.
- Application client modules, which contain class files and an application client deployment descriptor. Application client modules are packaged as JAR files with a `.jar` extension.
- Resource adapter modules, which contain all Java interfaces, classes, native libraries, and other documentation, along with the resource adapter deployment descriptor. Together, these implement the Connector architecture (see J2EE Connector Architecture, page 19) for a particular EIS. Resource adapter modules are packaged as JAR files with an `.rar` (resource adapter archive) extension.

J2EE 1.4 APIs

Figure 1–7 illustrates the availability of the J2EE 1.4 platform APIs in each J2EE container type. The following sections give a brief summary of the technologies required by the J2EE platform and the J2SE enterprise APIs that would be used in J2EE applications.

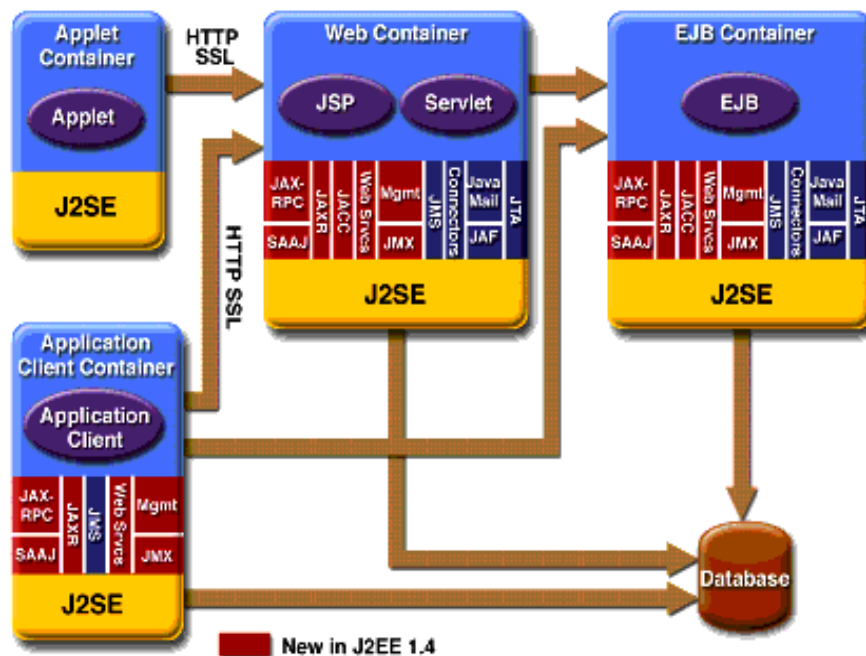


Figure 1–7 J2EE Platform APIs

Enterprise JavaBeans Technology

An Enterprise JavaBeans™ (EJB™) component, or *enterprise bean*, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the J2EE server.

As mentioned earlier, there are three kinds of enterprise beans: session beans, entity beans, and message-driven beans. Enterprise beans often interact with databases. One of the benefits of entity beans is that you do not have to write any SQL code or use the JDBC™ API (see JDBC API, page 19) directly to perform

database access operations; the EJB container handles this for you. However, if you override the default container-managed persistence for any reason, you will need to use the JDBC API. Also, if you choose to have a session bean access the database, you must use the JDBC API.

Java Servlet Technology

Java servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications that are accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

JavaServer Pages Technology

JavaServer Pages™ (JSP™) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text: static data (which can be expressed in any text-based format such as HTML, WML, and XML) and JSP elements, which determine how the page constructs dynamic content.

Java Message Service API

The Java Message Service (JMS) API is a messaging standard that allows J2EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.

Java Transaction API

The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The J2EE architecture provides a default auto commit to handle transaction commits and rollbacks. An *auto commit* means that any other applications that are viewing data will see the updated data after each database read or write operation. However, if your application performs two separate database access operations that depend on each other, you will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

JavaMail API

J2EE applications use the JavaMail™ API to send email notifications. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface. The J2EE platform includes JavaMail with a service provider that allows application components to send Internet mail.

JavaBeans Activation Framework

The JavaBeans Activation Framework (JAF) is included because JavaMail uses it. JAF provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate JavaBeans component to perform those operations.

Java API for XML Processing

The Java API for XML Processing (JAXP) supports the processing of XML documents using Document Object Model (DOM), Simple API for XML (SAX), and Extensible Stylesheet Language Transformations (XSLT). JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation.

JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts. Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the W3C schema. You can find information on the W3C schema at this URL: <http://www.w3.org/XML/Schema>.

Java API for XML-Based RPC

The Java API for XML-based RPC (JAX-RPC) uses the SOAP standard and HTTP, so client programs can make XML-based remote procedure calls (RPCs) over the Internet. JAX-RPC also supports WSDL, so you can import and export WSDL documents. With JAX-RPC and a WSDL, you can easily interoperate with clients and services running on Java-based or non-Java-based platforms such as .NET. For example, based on the WSDL document, a Visual Basic .NET client can be configured to use a web service implemented in Java technology, or a web service can be configured to recognize a Visual Basic .NET client.

JAX-RPC relies on the HTTP transport protocol. Taking that a step further, JAX-RPC lets you create service applications that combine HTTP with a Java technology version of the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols to establish basic or mutual authentication. SSL and TLS ensure message integrity by providing data encryption with client and server authentication capabilities.

Authentication is a measured way to verify whether a party is eligible and able to access certain information as a way to protect against the fraudulent use of a system or the fraudulent transmission of information. Information transported across the Internet is especially vulnerable to being intercepted and misused, so it's very important to configure a JAX-RPC web service to protect data in transit.

SOAP with Attachments API for Java

The SOAP with Attachments API for Java (SAAJ) is a low-level API on which JAX-RPC depends. SAAJ enables the production and consumption of messages that conform to the SOAP 1.1 specification and SOAP with Attachments note. Most developers do not use the SAAJ API, instead using the higher-level JAX-RPC API.

Java API for XML Registries

The Java API for XML Registries (JAXR) lets you access business and general-purpose registries over the web. JAXR supports the ebXML Registry and Repository standards and the emerging UDDI specifications. By using JAXR, developers can learn a single API and gain access to both of these important registry technologies.

Additionally, businesses can submit material to be shared and search for material that others have submitted. Standards groups have developed schemas for particular kinds of XML documents; two businesses might, for example, agree to use the schema for their industry's standard purchase order form. Because the schema is stored in a standard business registry, both parties can use JAXR to access it.

J2EE Connector Architecture

The J2EE Connector architecture is used by J2EE tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged in to any J2EE product. A *resource adapter* is a software component that allows J2EE application components to access and interact with the underlying resource manager of the EIS. Because a resource adapter is specific to its resource manager, typically there is a different resource adapter for each type of database or enterprise information system.

The J2EE Connector architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of J2EE-based web services with existing EISs that can be either synchronous or asynchronous. Existing applications and EISs integrated through the J2EE Connector architecture into the J2EE platform can be exposed as XML-based web services by using JAX-RPC and J2EE component models. Thus JAX-RPC and the J2EE Connector architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.

JDBC API

The JDBC API lets you invoke SQL commands from Java programming language methods. You use the JDBC API in an enterprise bean when you override the default container-managed persistence or have a session bean access the database. With container-managed persistence, database access operations are handled by the container, and your enterprise bean implementation contains no JDBC code or SQL commands. You can also use the JDBC API from a servlet or a JSP page to access the database directly without going through an enterprise bean.

The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the J2EE platform.

Java Naming and Directory Interface

The Java Naming and Directory Interface™ (JNDI) provides naming and directory functionality. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for

objects using their attributes. Using JNDI, a J2EE application can store and retrieve any type of named Java object.

J2EE naming services provide application clients, enterprise beans, and web components with access to a JNDI naming environment. A *naming environment* allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI *naming context*.

A J2EE component locates its environment naming context using JNDI interfaces. A component creates a `javax.naming.InitialContext` object and looks up the environment naming context in `InitialContext` under the name `java:comp/env`. A component's naming environment is stored directly in the environment naming context or in any of its direct or indirect subcontexts.

A J2EE component can access named system-provided and user-defined objects. The names of system-provided objects, such as JTA `UserTransaction` objects, are stored in the environment naming context, `java:comp/env`. The J2EE platform allows a component to name user-defined objects, such as enterprise beans, environment entries, JDBC `DataSource` objects, and message connections. An object should be named within a subcontext of the naming environment according to the type of the object. For example, enterprise beans are named within the subcontext `java:comp/env/ejb`, and JDBC `DataSource` references in the subcontext `java:comp/env/jdbc`.

Because JNDI is independent of any specific implementation, applications can use JNDI to access multiple naming and directory services, including existing naming and directory services such as LDAP, NDS, DNS, and NIS. This allows J2EE applications to coexist with legacy applications and systems. For more information on JNDI, see *The JNDI Tutorial*:

<http://java.sun.com/products/jndi/tutorial/index.html>

Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) provides a way for a J2EE application to authenticate and authorize a specific user or group of users to run it.

JAAS is a Java programming language version of the standard Pluggable Authentication Module (PAM) framework, which extends the Java 2 Platform security architecture to support user-based authorization.

Simplified Systems Integration

The J2EE platform is a platform-independent, full systems integration solution that creates an open marketplace in which every vendor can sell to every customer. Such a marketplace encourages vendors to compete, not by trying to lock customers into their technologies but instead by trying to outdo each other in providing products and services that benefit customers, such as better performance, better tools, or better customer support.

The J2EE APIs enable systems and applications integration through the following:

- Unified application model across tiers with enterprise beans
- Simplified request-and-response mechanism with JSP pages and servlets
- Reliable security model with JAAS
- XML-based data interchange integration with JAXP, SAAJ, and JAX-RPC
- Simplified interoperability with the J2EE Connector architecture
- Easy database connectivity with the JDBC API
- Enterprise application integration with message-driven beans and JMS, JTA, and JNDI

You can learn more about using the J2EE platform to build integrated business systems by reading *J2EE Technology in Practice*, by Rick Cattell and Jim Inscore (Addison-Wesley, 2001):

<http://java.sun.com/j2ee/inpractice/aboutthebook.html>

Sun Java System Application Server Platform Edition 8.1

The Sun Java System Application Server Platform Edition 8.1 is a fully compliant implementation of the J2EE 1.4 platform. In addition to supporting all the APIs described in the previous sections, the Application Server includes a num-

ber of J2EE technologies and tools that are not part of the J2EE 1.4 platform but are provided as a convenience to the developer.

This section briefly summarizes the technologies and tools that make up the Application Server, and instructions for starting and stopping the Application Server, starting the Admin Console, starting `deploytool`, and starting and stopping the PointBase database server. Other chapters explain how to use the remaining tools.

Technologies

The Application Server includes two user interface technologies—JavaServer Pages Standard Tag Library and JavaServer™ Faces—that are built on and used in conjunction with the J2EE 1.4 platform technologies Java servlet and JavaServer Pages.

JavaServer Pages Standard Tag Library

The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in your JSP applications, you employ a single, standard set of tags. This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of the tags is optimized.

JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

JavaServer Faces

JavaServer Faces technology is a user interface framework for building web applications. The main components of JavaServer Faces technology are as follows:

- A GUI component framework.
- A flexible model for rendering components in different kinds of HTML or different markup languages and technologies. A `Renderer` object generates the markup to render the component and converts the data stored in a model object to types that can be represented in a view.

- A standard RenderKit for generating HTML/4.01 markup.

The following features support the GUI components:

- Input validation
- Event handling
- Data conversion between model objects and components
- Managed model object creation
- Page navigation configuration

All this functionality is available via standard Java APIs and XML-based configuration files.

Tools

The Application Server contains the tools listed in Table 1–1. All can be used from the IDE. Basic usage information for many of the tools appears throughout the tutorial. For detailed information, see the online help in the GUI tools and the man pages at <http://docs.sun.com/db/doc/817-6092> for the command-line tools.

Table 1–1 Application Server Tools

Component	Description
Admin Console	A web-based GUI Application Server administration utility. Used to stop the Application Server and manage users, resources, and applications.
PointBase database	An evaluation copy of the PointBase database server.
verifier	A command-line tool to validate J2EE deployment descriptors.
wscompile	A command-line tool to generate stubs, ties, serializers, and WSDL files used in JAX-RPC clients and services.
wsdeploy	A command-line tool to generate implementation-specific, ready-to-deploy WAR files for web service applications that use JAX-RPC.

Registering the Application Server

Before you can begin working with J2EE project types in the IDE, you have to register the Application Server's installation location and a target domain. Note that if you downloaded and installed the version of the IDE that comes bundled with the Application Server, you do not have to perform this step. The IDE knows the location of the bundled Application Server.

1. In the IDE, choose Tools→Server Manager from the main window.
2. Click Add Server. Select Sun Java Systems Application Server 8.1 and give a name to the instance. Then click Next.
3. Specify the installation directory of the application server (for example, C:\Sun\Appserver) and click Next.
4. Select the location of a local instance of the application server from the Location combo box.
5. Optionally, enter your administrator username and password. If you do not want to store the username and password in your IDE user directory, you can leave these fields blank. The IDE will prompt you every time it needs the information. Note that the default admin password is adminadmin.

Starting and Stopping the Application Server

To start the Application Server, open the IDE, go to the Runtime window (Ctrl-5), expand the Servers node, right-click the Application Server's node, and choose Start/Stop Server. In the Server Status dialog box, click Start Server.

You can also start the Application Server from a terminal window or command prompt by executing the following:

```
asadmin start-domain --verbose domain1
```

A *domain* is a set of one or more Application Server instances managed by one administration server. Associated with a domain are the following:

- The Application Server's port number. The default is 8080.
- The administration server's port number. The default is 4848.
- An administration user name and password.

You specify these values when you install the Application Server. The examples in this tutorial assume that you choose the default ports.

With no arguments, the IDE initiates the default domain, which is `domain1`. The `--verbose` flag causes all logging and debugging output to appear in the Output window (it will also go into the server log, which is located in `<J2EE_HOME>/domains/domain1/logs/server.log`).

After the server has completed its startup sequence, you will see the following output:

```
Domain domain1 started.
```

To stop the Application Server, click Stop Server in the Server Status dialog box. You can also stop the Application Server from a terminal window or command prompt by executing the following:

```
asadmin stop-domain domain1
```

When the server has stopped you will see the following output:

```
Domain domain1 stopped.
```

Starting the Admin Console

To administer the Application Server and manage users, resources, and J2EE applications, you use the Admin Console tool. The Application Server must be running before you invoke the Admin Console. To start the Admin Console, open the IDE, go to the Runtime window (Ctrl-5), expand the Servers node, right-click the node for the Application Server, and choose View Admin Console.

Alternatively, you can open a web browser and go to the following URL:

```
http://localhost:4848/asadmin/
```

Starting and Stopping the PointBase Database Server

The Application Server includes an evaluation copy of the PointBase database.

To start the PointBase database server, open the IDE and choose Tools→Pointbase Database→Start Local Pointbase Database from the main menu.

For information about the PointBase database included with the Application Server see the PointBase web site at www.pointbase.com.

Debugging J2EE Applications

This section describes how to determine what is causing an error in your application deployment or execution.

Using the Server Log

One way to debug applications is to look at the server log in `<J2EE_HOME>/domains/domain1/logs/server.log`. The log contains output from the Application Server and your applications. You can log messages from any Java class in your application with `System.out.println` and the Java Logging APIs (documented at <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/index.html>) and from web components with the `ServletContext.log` method.

You can also view the Server Log in the IDE. Open the IDE, go to the Runtime window (Ctrl-5), expand the Servers node, right-click the node for the Application Server, and choose View Server Log.

If you start the Application Server with the `--verbose` flag, all logging and debugging output will appear on the terminal window or command prompt and the server log. If you start the Application Server in the background, debugging information is only available in the log. You can view the server log with a text editor or with the Admin Console log viewer. To use the log viewer:

1. Select the Application Server node.
2. Select the Logging tab.
3. Click the Open Log Viewer button. The log viewer will open and display the last 40 entries.

If you wish to display other entries:

1. Click the Modify Search button.
2. Specify any constraints on the entries you want to see.
3. Click the Search button at the bottom of the log viewer.

Using the NetBeans Debugger

The IDE uses the Sun Microsystems JPDA debugger to debug your programs. When you start a debugging session, all of the relevant debugger windows appear automatically at the bottom of your screen. You can debug an entire project, any executable class, and any JUnit tests. The IDE also lets you debug applications that are running on a remote machine by attaching the debugger to the application process.

When you run or debug web applications, JSP pages, or servlets, you can also use the HTTP Monitor to monitor data flow. The HTTP Monitor appears by default. The HTTP Monitor gathers data about HTTP requests that the servlet engine processes. For each HTTP request that the engine processes, the monitor records data about the incoming request, the data states maintained on the server, and the servlet context. You can view data, store data for future sessions, and replay and edit previous requests.

When you start a debugging session in the IDE, the IDE compiles the files that you are debugging, runs them in debug mode, and displays debugger output in the Debugger windows. To start a debugging session, select the file that you want to debug and choose one of the following commands from the Run menu:

- **Debug Main Project (F5).** Runs the main project until the first breakpoint is encountered.
- **Step Into (F7).** Starts running the main project's main class and stops at the first executable statement.
- **Run to Cursor (F4).** Starts a debugging session, runs the application to the cursor location in the Source Editor, and pauses the application.

