The J2EE™ 1.4 Tutorial

Eric Armstrong
Jennifer Ball
Stephanie Bodoff
Debbie Carson
Ian Evans
Maydene Fisher
Dale Green
Kim Haase
Eric Jendrock

November 16, 2003

1 Overview

TODAY, more and more developers want to write distributed transactional applications for the enterprise and leverage the speed, security, and reliability of server-side technology. If you are already working in this area, you know that in today's fast-moving and demanding world of e-commerce and information technology, enterprise applications have to 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, 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 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 takes an examples-based approach to describing the features and functionalities available in J2EE 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 learn development basics, be introduced to 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 multitiered distributed 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 different 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.

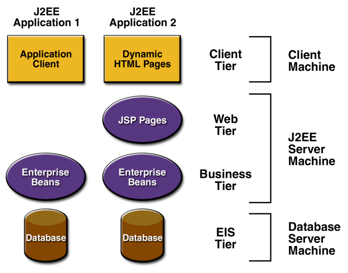

Figure 1–1 Multitiered Applications

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, verified to be well formed and in compliance with the J2EE specification, and 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: 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 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 do things like query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, heavyweight operations like these 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

A J2EE 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 Swing or Abstract Window Toolkit (AWT) APIs, 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, a J2EE application client can open an HTTP connection to establish communication with a servlet running in the Web tier.

JavaBeans™ Component Architecture

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans component) 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 instance variables and `get` and `set` methods for accessing the data in the instance variables. 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.

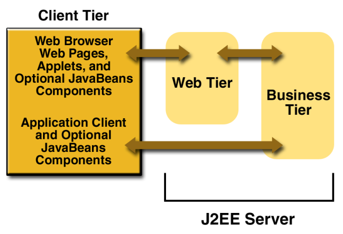
Client Tier
Web Browser Web Pages, Applets, and Optional JavaBeans Components
Application Client and Optional JavaBeans Components
Web Tier
Business Tier
J2EE Server

Figure 1–2 Server Communications

Web Components

J2EE Web components can be either servlets or 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.

Like the client tier and as shown in Figure 1–3, the Web 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 enterprise 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.

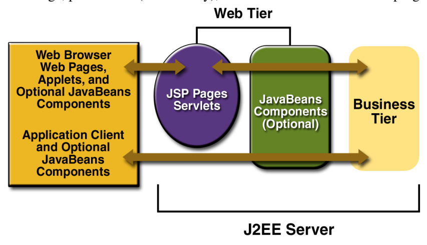
Web Tier
Web Browser Web Pages, Applets, and Optional JavaBeans Components
Application Client and Optional JavaBeans Components
JSP Pages Servlets
JavaBeans Components (Optional)
Business Tier
J2EE Server

Figure 1–3 Web Tier and J2EE Applications

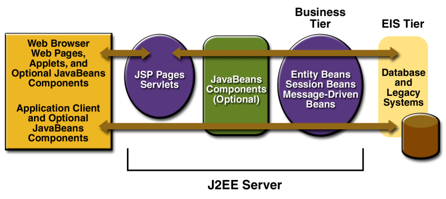
Business Tier
EIS Tier
Web Browser Web Pages, Applets, and Optional JavaBeans Components
Application Client and Optional JavaBeans Components
JSP Pages Servlets
JavaBeans Components (Optional)
Entity Beans Session Beans Message-Driven Beans
Database and Legacy Systems
J2EE Server

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 features 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 enterprise information system software and includes enterprise infrastructure systems such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. J2EE application components might need access to enterprise information systems for database connectivity, for example.

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, enterprise bean, or application client component can be executed, it must be assembled into a J2EE application 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, which includes 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.

The fact that the J2EE architecture provides configurable services means that 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 non-configurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the J2EE platform APIs described in the section J2EE APIs (page 17). Although data persistence is a non-configurable 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.

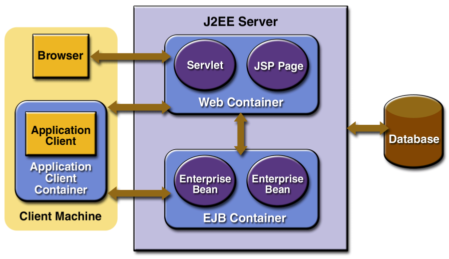

Figure 1–5 J2EE Server and Containers

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.

Packaging

A J2EE application is delivered in an Enterprise Archive (EAR) file. An EAR file is a standard Java Archive (JAR) file with an `.ear` extension. The EAR file contains J2EE modules. 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 just a matter of assembling various J2EE modules into J2EE EAR files.

A *J2EE module* consists of one or more J2EE components for the same container type and one component deployment descriptor of that type. A *deployment descriptor* is an XML document with an `.xml` extension that describes a component's deployment settings. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. Because deployment descriptor information is declarative, it can be changed without modifying the bean source code. At run time, the J2EE server reads the deployment descriptor and acts upon the component accordingly. A J2EE module without an application deployment descriptor can be deployed as a *stand-alone* module. The four types of J2EE modules are:

- Enterprise JavaBeans modules contain class files for enterprise beans and an EJB deployment descriptor. EJB modules are packaged as JAR files with a `.jar` extension.
- Web modules contain JSP files, class files for servlets, GIF and HTML files, and a Web deployment descriptor. Web modules are packaged as JAR files with a `.war` (Web ARchive) extension.
- Resource adapter modules 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 21) for a particular EIS. Resource adapter modules are packages as JAR files with a `.rar` (Resource adapter ARchive) extension.
- Application client modules contain class files and an application client deployment descriptor. Application client modules are packaged as JAR files with a `.jar` extension.

Web Services Support

Web services are Web-based enterprise applications that use open, Extensible Markup Language (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.

It is easy 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, 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 next 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 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, CAD documents or 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.

Extensible Markup Language

Extensible Markup Language is a cross-platform, extensible, and 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 style sheets 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 style sheets to handle the data in a way that best suits their needs.

- One company might put the XML pricing information through a program to translate the XML to HTML so 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.

HTTP-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 how to communicate with the service. WSDLs can be stored in UDDI registries and/or published on the Web. The J2EE platform provides a tool for generating the WSDL for 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.

Development Roles

Reusable modules make it possible to divide the application development and deployment process into distinct roles so that different people or companies can perform different parts of the process.

The first two roles involve purchasing and installing the J2EE product and tools. Once software is purchased and installed, J2EE components can be developed by application component providers, assembled by application assemblers, and deployed by application deployers. In a large organization, each of these roles might be executed by different individuals or teams. This division of labor works because each of the earlier roles outputs a portable file that is the input for a subsequent role. For example, in the application component development phase, an enterprise bean software developer delivers EJB JAR files. In the application assembly role, another developer combines these EJB JAR files into a J2EE application and saves it in an EAR file. In the application deployment role, a system administrator at the customer site uses the EAR file to install the J2EE application into a J2EE server.

The different roles are not always executed by different people. If you work for a small company, for example, or if you are prototyping a sample application, you might perform the tasks in every phase.

J2EE Product Provider

The J2EE product provider is the company that designs and makes available for purchase the J2EE platform, APIs, and other features defined in the J2EE specification. Product providers are typically operating system, database system, application server, or Web server vendors who implement the J2EE platform according to the Java 2 Platform, Enterprise Edition Specification.

Tool Provider

The tool provider is the company or person who creates development, assembly, and packaging tools used by component providers, assemblers, and deployers.

Application Component Provider

The application component provider is the company or person who creates Web components, enterprise beans, applets, or application clients for use in J2EE applications.

Enterprise Bean Developer

An enterprise bean developer performs the following tasks to deliver an EJB JAR file that contains the enterprise bean:

- Writes and compiles the source code
- Specifies the deployment descriptor
- Bundles the `.class` files and deployment descriptor into an EJB JAR file

Web Component Developer

A Web component developer performs the following tasks to deliver a WAR file containing the Web component:

- Writes and compiles servlet source code
- Writes JSP and HTML files
- Specifies the deployment descriptor for the Web component
- Bundles the `.class`, `.jsp`, `.html`, and deployment descriptor files in the WAR file

J2EE Application Client Developer

An application client developer performs the following tasks to deliver a JAR file containing the J2EE application client:

- Writes and compiles the source code
- Specifies the deployment descriptor for the client
- Bundles the `.class` files and deployment descriptor into the JAR file

Application Assembler

The application assembler is the company or person who receives application component JAR files from component providers and assembles them into a J2EE application EAR file. The assembler or deployer can edit the deployment descriptor directly or use tools that correctly add XML tags according to interactive selections. A software developer performs the following tasks to deliver an EAR file containing the J2EE application:

- Assembles EJB JAR and WAR files created in the previous phases into a J2EE application (EAR) file
- Specifies the deployment descriptor for the J2EE application
- Verifies that the contents of the EAR file are well formed and comply with the J2EE specification

Application Deployer and Administrator

The application deployer and administrator is the company or person who configures and deploys the J2EE application, administers the computing and networking infrastructure where J2EE applications run, and oversees the runtime environment. Duties include such things as setting transaction controls and security attributes and specifying connections to databases.

During configuration, the deployer follows instructions supplied by the application component provider to resolve external dependencies, specify security settings, and assign transaction attributes. During installation, the deployer moves the application components to the server and generates the container-specific classes and interfaces.

A deployer/system administrator performs the following tasks to install and configure a J2EE application:

- Adds the J2EE application (EAR) file created in the preceding phase to the J2EE server
- Configures the J2EE application for the operational environment by modifying the deployment descriptor of the J2EE application
- Verifies that the contents of the EAR file are well formed and comply with the J2EE specification
- Deploys (installs) the J2EE application EAR file into the J2EE server

J2EE APIs

Enterprise JavaBeans Technology

An Enterprise JavaBeans™ (EJB™) component or *enterprise bean* is a body of code with 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.

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 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 have to use the JDBC API.

JDBC API

The JDBC™ API lets you invoke SQL commands from Java programing 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 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 Servlet Technology

Java Servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications 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 template 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

The Java Message Service (JMS) 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 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 implementations, 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 online JNDI Tutorial:

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

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 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 can use the JavaMail™ API to send e-mail 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. It 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 Parsing (SAX), and XML Stylesheet Language Transformation (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 of 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 Registries

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

Additionally, businesses submit material to be shared and search for material that others have submitted. Standards groups have developed schemas for particular kinds of XML documents, and 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.

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 and/or the fraudulent transmission of information. Information transported across the Internet is especially vulnerable to being intercepted and misused, so configuring a JAX-RPC Web service to protect data in transit is very important.

SOAP with Attachments API for Java

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

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 into any J2EE product. A *resource adapter* is a software component that allows J2EE application components to access and interact with the underlying resource manager. Because a resource adapter is specific to its resource manager, there is typically a different resource adapter for each type of database or enterprise information system.

JAX-RPC and the J2EE Connector Architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.

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 using JAX-RPC and J2EE component models.

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 programing language version of the standard Pluggable Authentication Module (PAM) framework that 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 by trying to outdo each other by 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 response and request mechanism with JSP pages and servlets
- Reliable security model with JAAS
- XML-based data interchange integration with JAXP
- 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*:

```
http://java.sun.com/j2ee/inpractice/aboutthebook.html
```