Java™ 2 Platform Enterprise Edition Specification, v1.3

Please send technical comments to: j2ee-spec-technical@eng.sun.com
Please send business comments to: j2ee-spec-business@eng.sun.com

Proposed Final Draft - 10/20/00

Bill Shannon

We're the dot in .com™

901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax: 650 969-9131

Sun Microsystems, Inc.

Java™ 2 Platform, Enterprise Edition Specification ("Specification")
Version: 1.3
Status: Proposed FInal Draft
Release: 10/20/00

RESTRICTED RIGHTS LEGEND.

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT.
You may wish to report any ambiguities, inconsistencies, or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

Please
Recycle

Contents

CHAPTER 1

Introduction

Enterprises today need to extend their reach, reduce their costs, and lower their response times by providing easy-to-access services to their customers, employees, and suppliers.

Typically, applications that provide these services must combine existing Enterprise Information Systems (EISs) with new business functions that deliver services to a broad range of users. These services need to be:

- *Highly available*, to meet the needs of today's global business environment.
- *Secure*, to protect the privacy of users and the integrity of the enterprise.
- *Reliable and scalable*, to insure that business transactions are accurately and promptly processed.

In most cases, these services are architected as multi-tier applications. A middle-tier that implements the new services needs to integrate existing EISs with the business functions and data of the new service. The service middle-tier shields first-tier clients from the complexity of the enterprise and takes advantage of rapidly maturing web technologies to eliminate or drastically reduce user administration and training while leveraging existing enterprise assets.

The Java™ 2 Platform, Enterprise Edition (J2EE™) reduces the cost and complexity of developing these multi-tier services, resulting in services that can be rapidly deployed and easily enhanced as the enterprise responds to competitive pressures.

J2EE achieves these benefits by defining a standard architecture that is delivered as the following elements:

- **J2EE BluePrints** - A standard application model for developing multi-tier, thin-client services.
- **J2EE Platform** - A standard platform for hosting J2EE applications.
- **J2EE Compatibility Test Suite** - A suite of compatibility tests for verifying that a J2EE platform product complies with the J2EE platform standard.

J2EE Reference Implementation - A reference implementation for demonstrating the capabilities of J2EE and for providing an operational definition of the J2EE platform.

This document provides the specification of the J2EE platform and describes the requirements that a J2EE platform product must meet.

Acknowledgements

This specification is the work of many people. Vlada Matena wrote the first draft as well as the Transaction Management and Naming chapters. Sekhar Vajjhala, Kevin Osborn, and Ron Monzillo wrote the Security chapter. Hans Hrasna wrote the Application Assembly and Deployment chapter. Seth White wrote the JDBC API requirements. Jim Inscore, Eric Jendrock, and Beth Stearns provided editorial assistance. Shel Finkelstein, Mark Hapner, Danny Coward, Tom Kincaid, and Tony Ng provided feedback on many drafts. And of course this specification was formed and molded based on conversations with and review feedback from our many industry partners.

Acknowledgements for version 1.3

Version 1.3 of this specification grew out of discussions with our partners during the creation of version 1.2, as well as meetings with those partners subsequent to the final release of version 1.2. Version 1.3 was created under the Java Community Process as JSR-058. The JSR-058 Expert Group included representatives from the following companies and organizations: Allaire, BEA Systems, Bluestone Software, Bull S.A., Exoffice, Fujitsu Limited, GemStone Systems, Inc., IBM, Inline Software, Inprise, IONA Technologies, iPlanet, jGuru.com, Orion Application Server, Persistence, POET Software, SilverStream, Sun, and Sybase. In addition, most of the people who helped with the previous version continued to help with this version, along with Jon Ellis and Ram Jeyaraman.

CHAPTER 2

Platform Overview

This chapter provides an overview of the Java™ 2 Platform, Enterprise Edition (J2EE™).

2.1 Architecture

The J2EE runtime environment consists of the following parts:

- **Application components**. The J2EE programming model defines four application component types that a J2EE product must support:
  - Application clients are Java programming language programs that are typically GUI programs that execute on a desktop computer. Application clients offer a user experience similar to that of native applications, and have access to all of the facilities of the J2EE middle tier.
  - Applets are GUI components that typically execute in a web browser, but can execute in a variety of other applications or devices that support the applet programming model. Applets can be used to provide a powerful user interface for J2EE applications. (Simple HTML pages can also be used to provide a more limited user interface for J2EE applications.)
  - Servlets and JSP pages typically execute in a web server and respond to HTTP requests from web clients. Servlets and JSP pages may be used to generate HTML pages that are an application's user interface. They may also be used to generate XML or other format data that is consumed by other application components. Servlets, and pages created with the JavaServer Pages™ technology, are often referred to collectively in this specification as "web components." Web applications are composed of web components and other data such as HTML pages.

– Enterprise JavaBeans™ (EJB) components execute in a managed environment that supports transactions. Enterprise beans typically contain the business logic for a J2EE application.

These application components can be divided into three categories:

– Components that are deployed, managed, and executed on a J2EE server. These components include JavaServer Pages, Servlets, and Enterprise JavaBeans.

– Components that are deployed and managed on a J2EE server, but are loaded to and executed on a client machine. These components include HTML pages and applets embedded in the HTML pages.

– Components whose deployment and management is not completely defined by this specification. Application clients fall into this category. Future versions of this specification may more fully define deployment and management of application clients.

■ **Containers.** Containers provide the runtime support for the application components. A container provides a federated view of the underlying J2EE APIs to the application components. Interposing a container between the application components and the J2EE services allows the container to transparently inject services defined by the components' deployment descriptors, such as declarative transaction management, security checks, resource pooling, and state management. A typical J2EE product will provide a container for each application component type: application client container, applet container, web component container, and enterprise bean container.

This specification requires that these containers provide a Java Compatible™ runtime environment, as defined by the Java 2 Platform, Standard Edition, v1.3 specification (J2SE). The applet container may use the Java Plugin product to provide this environment, or it may provide it natively. The use of applet containers providing only the JDK™ 1.1 APIs is outside the scope of this specification.

The container tools also understand the file formats for packaging of the application components for deployment. The containers are implemented by a J2EE Product Provider.

This specification defines a set of standard services that each J2EE product must support. These standard services are described below. The J2EE containers provide the APIs to access these services to application components. This specification also describes standard ways to extend J2EE services with connectors to other non-J2EE application systems, such as mainframe systems and ERP systems.

Underlying the J2EE containers is the J2EE core. A J2EE Product Provider typically implements the J2EE server core using an existing transaction processing infrastructure in combination with Java 2 technology. The J2EE client core is typically built on Java 2 Platform, Standard Edition technology.

FIGURE 2-1 illustrates the relationship of these components of the J2EE platform. Note that this figure shows the logical relationships of the components; it is *not* meant to imply a physical partitioning of the components into separate machines, processes, address spaces, or virtual machines.

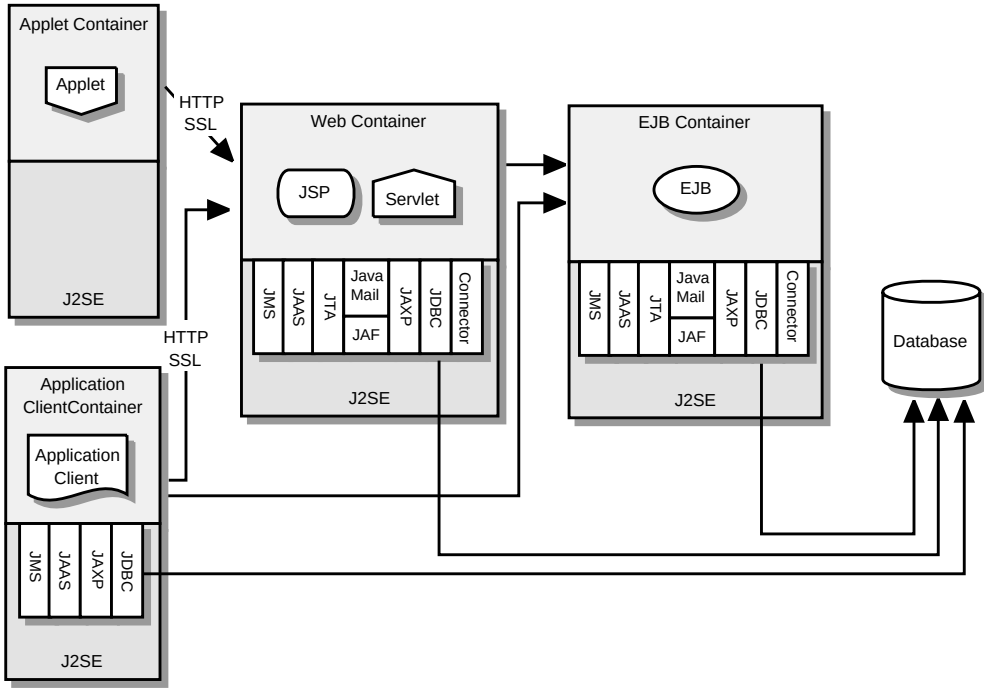

FIGURE 2-1 J2EE Architecture Diagram

- **Resource manager drivers**. A resource manager driver (driver for short) is a system-level software component that implements network connectivity to an external resource manager. A driver can extend the functionality of the J2EE platform either by implementing one of the J2EE standard service APIs (such as a JDBC™ driver), or by defining and implementing a resource manager driver for a connector to an external application system. Drivers interface with the J2EE platform through the J2EE service provider interfaces (J2EE SPI). A driver that uses the J2EE SPIs to attach to the J2EE platform will be able to work with all J2EE products.

- **Database.** The J2EE platform includes a database, accessible through the JDBC API, for the storage of business data. The database is accessible from web components, enterprise beans, and application client components. The database need not be accessible from applets.

The J2EE standard services include the following (specified in more detail later in this document). Some of these standard services are actually provided by J2SE.

- **HTTP**. The HTTP client-side API is defined by the `java.net` package. The HTTP server-side API is defined by the servlet and JSP interfaces.
- **HTTPS**. Use of the HTTP protocol over the SSL protocol is supported by the same client and server APIs as HTTP.
- **Java™ Transaction API (JTA)**. The Java Transaction API consists of two parts:
  - An application-level demarcation interface that is used by the container and application components to demarcate transaction boundaries.
  - An interface between the transaction manager and a resource manager used at the J2EE SPI level (in a future release).
- **RMI-IIOP**. The RMI-IIOP subsystem is composed of APIs that allow for the use of RMI-style programming that is independent of the underlying protocol, as well as an implementation of these APIs that supports both the J2SE native RMI protocol (JRMP) and the CORBA IIOP protocol. J2EE applications can use RMI-IIOP, with the IIOP protocol support, to access CORBA services that are compatible with the RMI programming restrictions (see the RMI-IIOP spec for details). Such CORBA services would typically be defined by components that live outside of a J2EE product, usually in a legacy system. Only J2EE application clients are required to be able to define their own CORBA services directly, using the RMI-IIOP APIs. Typically such CORBA objects would be used for callbacks when accessing other CORBA objects.

  J2EE applications are required to use the RMI-IIOP APIs (specifically the `narrow` method of `javax.rmi.PortableRemoteObject`) when accessing Enterprise JavaBeans components, as described in the EJB specification. This allows enterprise beans to be protocol independent. In addition, J2EE products must be capable of exporting enterprise beans using the IIOP protocol, and accessing enterprise beans using the IIOP protocol, as specified in the EJB 2.0 specification. The ability to use the IIOP protocol is required to enable interoperability between J2EE products, however a J2EE product may also use other protocols.
- **JavaIDL**. JavaIDL allows J2EE application components to invoke external CORBA objects using the IIOP protocol. These CORBA objects may be written in any language and typically live outside a J2EE product. J2EE applications

may use JavaIDL to act as clients of CORBA services, but only J2EE application clients are required to be allowed to use JavaIDL directly to present CORBA services themselves.

- **JDBC™**. The JDBC API is the API for connectivity with relational database systems. 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™ Message Service (JMS)**. The Java Messaging Service is a standard API for messaging that supports reliable point-to-point messaging as well as the publish-subscribe model. This specification requires a JMS provider that implements both point-to-point messaging as well as publish-subscribe messaging.
- **Java Naming and Directory Interface™ (JNDI)**. The JNDI API is the standard API for naming and directory access. The JNDI API has two parts: an application-level interface used by the application components to access naming and directory services and a service provider interface to attach a provider of a naming and directory service.
- **JavaMail™**. Many Internet applications require the ability to send email notifications, so the J2EE platform includes the JavaMail API along with a JavaMail service provider that allows an application component to send Internet mail. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface used at the J2EE SPI level.
- **JavaBeans™ Activation Framework (JAF)**. The JavaMail API makes use of the JAF API, so it must be included as well.
- **Java™ API for XML Parsing (JAXP).** JAXP provides support for the industry standard SAX and DOM APIs for parsing XML documents.
- **J2EE™ Connector Architecture.** The Connector architecture is a J2EE SPI that allows resource adapters that support access to Enterprise Information Systems to be plugged in to any J2EE product. The Connector architecture defines a standard set of system-level contracts between a J2EE server and a resource adapter. The standard contracts include:
  - A connection management contract that lets a J2EE server pool connections to an underlying EIS, and lets application components connect to an EIS. This leads to a scalable application environment that can support a large number of clients requiring access to EIS systems.
  - A transaction management contract between the transaction manager and an EIS that supports transactional access to EIS resource managers. This contract lets a J2EE server use a transaction manager to manage transactions across multiple resource managers. This contract also supports transactions that are managed internal to an EIS resource manager without the necessity of involving an external transaction manager.

– A security contract that enables secure access to an EIS. This contract provides support for a secure application environment, which reduces security threats to the EIS and protects valuable information resources managed by the EIS.

- **Java™ Authentication and Authorization Service (JAAS).** JAAS enables services to authenticate and enforce access controls upon users. It implements a Java technology version of the standard Pluggable Authentication Module (PAM) framework, and extends the access control architecture of the Java 2 Platform in a compatible fashion to support user-based authorization.

Many of the APIs described above provide interoperability with components that are not a part of the J2EE platform, such as external web or CORBA services. FIGURE 2-2 illustrates the interoperability facilities of the J2EE platform. (The directions of the arrows indicate the client/server relationships of the components.)

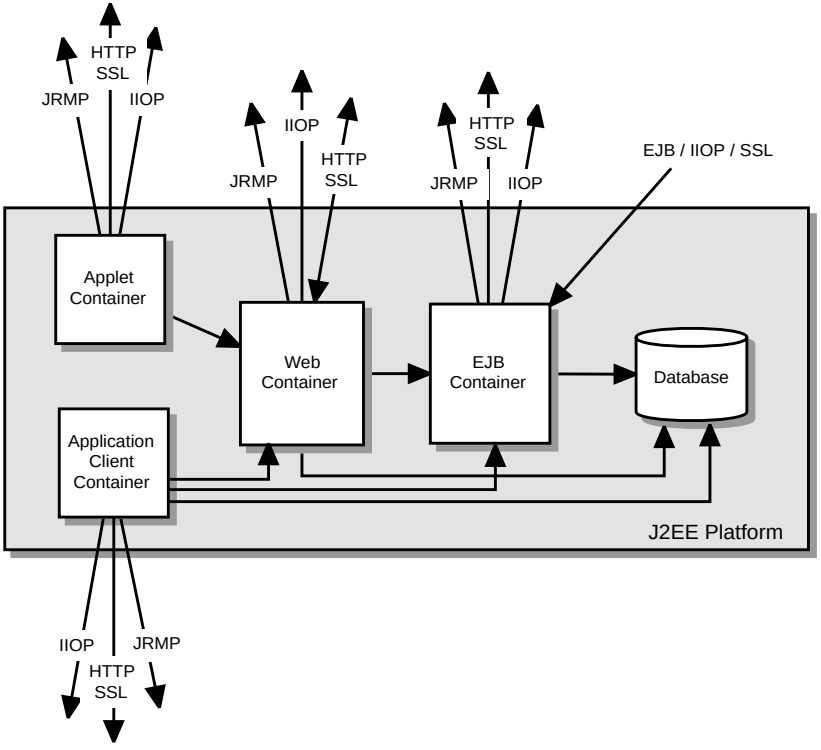

FIGURE 2-2 J2EE Interoperability

2.2 Product Requirements

This specification doesn't require that a J2EE product be implemented by a single program, a single server, or even a single machine. In general, this specification doesn't describe the partitioning of services or functions between machines, servers, processes, etc. As long as the requirements in this specification are met, J2EE Product Providers can partition the functionality however they see fit. A J2EE product must be able to deploy application components that execute with the semantics described by this specification.

A very simple J2EE product might be provided as a single Java virtual machine that supports applets, web components, and enterprise beans simultaneously in one container (although this would be an extreme, and probably rare, case), and application clients each in their own container. A typical low end J2EE product will support applets in one of the popular browsers, application clients each in their own Java virtual machine, and will provide a single server that supports both web components and enterprise beans. A high end J2EE product might split the server components into multiple servers, each of which can be distributed and load-balanced across a collection of machines. This specification does not prescribe or preclude any of these configurations.

A wide variety of J2EE product configurations and implementations, all of which meet the requirements of this specification, are possible. A portable J2EE application will function correctly when successfully deployed in any of these products.

2.3 Product Extensions

This specification describes a minimum set of facilities that all J2EE products must provide. Most J2EE products will provide facilities beyond the minimum required by this specification. This specification includes only a few limits to the ability of a product to provide extensions. In particular, it includes the same restrictions as J2SE on extensions to Java APIs. A J2EE product may not add classes to the Java programming language packages included in this specification, and may not add methods or otherwise alter the signatures of the specified classes.

However, many other extensions are possible. A J2EE product may provide additional Java APIs, either other Java optional packages or other (appropriately named) packages. A J2EE product may include support for additional protocols

or services not specified here. A J2EE product may support applications written in other languages, or may support connectivity to other platforms or applications.

Of course, portable applications will not make use of any platform extensions. Applications that do make use of facilities not required by this specification will be less portable. Depending on the facility used, the loss of portability may be minor or it may be significant. The document *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition* will help application developers construct portable applications, and will contain advice on how best to manage the use of non-portable code when the use of such facilities is necessary.

In addition, we expect J2EE products to vary widely, and in fact compete, on various quality of service aspects. Different products will provide different levels of performance, scalability, robustness, availably, security, etc. In some cases this specification describes minimum required levels of service. Future versions of this specification may allow applications to describe their requirements in these areas.

2.4 Platform Roles

This section describes typical Java 2 Platform, Enterprise Edition roles. Although these roles are considered to be typical, an organization could use slightly different roles to match that organization's actual application development and deployment workflow.

The following sections describe the roles in greater detail. Subsets of some of these roles are defined in the EJB, JSP, and Servlet specifications.

2.4.1 J2EE Product Provider

A J2EE Product Provider, typically an operating system vendor, database system vendor, application server vendor, or a web server vendor, implements a J2EE product providing the component containers, J2EE platform APIs, and other features defined in this specification. A J2EE Product Provider must provide the J2EE APIs to the application components through the containers. A Product Provider frequently bases their implementation on an existing infrastructure.

A J2EE Product Provider must provide the mapping of the application components to the network protocols as specified by this specification. A J2EE product is free to implement the interfaces that are not specified by this specification in an implementation-specific way.

A J2EE Product Provider must provide application deployment and management tools. Deployment tools enable a Deployer (see Section 2.4.4, “Deployer”) to deploy application components on the J2EE product. Management tools allow a System Administrator (see Section 2.4.5, “System Administrator”) to manage the J2EE product and the applications deployed on the J2EE product. The form of these tools is not prescribed by this specification.

2.4.2 Application Component Provider

There are multiple roles for Application Component Providers, including HTML document designers, document programmers, enterprise bean developers, etc. These roles use tools to produce J2EE applications and components.

2.4.3 Application Assembler

The Application Assembler takes a set of components developed by Application Component Providers and assembles them into a complete J2EE application delivered in the form of a Enterprise Archive (.ear) file. The Application Assembler will generally use GUI tools provided by either a Platform Provider or Tool Provider. The Application Assembler is responsible for providing assembly instructions describing external dependencies of the application that the Deployer must resolve in the deployment process.

2.4.4 Deployer

The Deployer, an expert in a specific operational environment, is responsible for deploying web applications and Enterprise JavaBeans components into that environment. The Deployer uses tools supplied by the J2EE Product Provider to perform the deployment tasks. The deployment process is typically a three-stage process:

1. **Installation:** moves the media to the server, generates the additional container-specific classes and interfaces that enable the container to manage the application components at runtime, and installs the application components and additional classes and interfaces into the J2EE containers.

2. **Configuration:** resolves all the external dependencies declared by the Application Component Provider and follows the application assembly instructions defined by the Application Assembler. For example, the Deployer is responsible for mapping the security roles defined by the Application Assembler to the user groups and accounts that exist in the operational environment into which the application components are deployed.

3. **Execution:** starts up the newly installed and configured application.

In some cases, a qualified Deployer may customize the business logic of the application's components at deployment time by using tools provided with a J2EE product to write relatively simple application code that wraps an enterprise bean's business methods, or to customize the appearance of a JSP page, for example.

The Deployer's output is web applications, enterprise beans, applets, and application clients that have been customized for the target operational environment and are deployed in a specific J2EE container.

2.4.5 System Administrator

The System Administrator is responsible for the configuration and administration of the enterprise's computing and networking infrastructure. The System Administrator is also responsible for overseeing the runtime well-being of the deployed J2EE applications. The System Administrator typically uses runtime monitoring and management tools provided by the J2EE Product Provider to accomplish these tasks.

2.4.6 Tool Provider

A Tool Provider provides tools used for the development and packaging of application components. A variety of tools are anticipated, corresponding to the many application component types supported by the J2EE platform. Platform independent tools can be used for all phases of development up to the deployment of an application. Platform dependent tools are used for deployment, management, and monitoring of applications. Future versions of this specification may define more interfaces that allow such tools to be platform independent.

2.5 Platform Contracts

This section describes the Java 2 Platform, Enterprise Edition contracts that must be fulfilled by the J2EE Product Provider.

2.5.1 J2EE API

The J2EE API defines the contract between the J2EE application components and the J2EE platform. The contract specifies both the runtime and deployment interfaces.

The J2EE Product Provider must implement the J2EE APIs in a way that supports the semantics and policies described in this specification. The Application Component Provider should provide components that conform to these APIs and policies.

2.5.2 J2EE SPI

The J2EE SPI defines the contract between the J2EE platform and service providers that may be plugged in to a J2EE product. The Connector APIs define service provider interfaces for integrating resource adapters with a J2EE application server. These resource adapter components are called Connectors.

The J2EE Product Provider must implement the J2EE SPIs in a way that supports the semantics and policies described in this specification. A provider of Service Provider components (for example, a Connector Provider) should provide components that conform to these SPIs and policies.

2.5.3 Network Protocols

This specification defines the mapping of the application components to industry-standard network protocols. The mapping allows client access to the application components from systems that have not installed J2EE product specific technology. See Chapter 7, “Interoperability” for details on the network protocol support required for interoperability.

The J2EE Product Provider is required to publish the installed application components on the industry-standard protocols. This specification defines the mapping of servlets and JSP pages to the HTTP and HTTPS protocols, and the mapping of EJB to IIOP.

2.5.4 Deployment Descriptors

Deployment descriptors are used to communicate the needs of application components to the Deployer. The deployment descriptor is a contract between the Application Component Provider or Assembler and the Deployer. The Application Component Provider or Assembler is required to specify the application component's external resource requirements, security requirements, environment parameters, etc. in the component's deployment descriptor. The J2EE Product Provider is required to provide a deployment tool that interprets the J2EE deployment descriptors and allows the Deployer to map the application component's requirements to the capabilities of a specific J2EE product and environment.

CHAPTER 3

Security

This chapter describes the security requirements for the Java™ 2 Platform, Enterprise Edition (J2EE).

The J2EE Product Provider is responsible for determining the level of security and security assurances afforded by their implementation. However, a J2EE Product Provider is required to satisfy the requirements specified in this chapter.

3.1 Introduction

An enterprise contains many resources that can be accessed by many users. Sensitive information often traverses unprotected open networks (such as the Internet). In such an environment, almost every enterprise has security requirements and specific mechanisms and infrastructure to meet them. Although the quality assurances and implementation details may vary, they all share some of the following characteristics:

- **Authentication:** The means by which communicating entities prove to one another that they are acting on behalf of specific identities (e.g., client to server and/or server to client).
- **Access control for resources**: The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.
- **Data integrity:** The means used to prove that information could not have been modified by a third party (some entity other than the source of the information). For example, a recipient of data sent over an open network must be able to detect and discard messages that were modified after they were sent.

- **Confidentiality or Data Privacy:** The means used to ensure that information is only made available to users who are authorized to access it.
- **Non-repudiation:** The means used to prove that a user performed some action such that the user cannot reasonably deny having done so.
- **Auditing:** The means used to capture a tamper-resistant record of security related events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms.

This chapter specifies how the J2EE platform addresses some of these security requirements, and identifies those requirements left to be addressed by J2EE Product Providers. Issues being considered for future versions of this specification are briefly mentioned in Section 3.7, "Future Directions."

3.2 A Simple Example

The security behavior of a J2EE environment may be better understood by examining what happens in a simple application with a web client, JSP page user interface, and enterprise bean business logic. We include here a descriptive example; this example is not meant to specify requirements. In this example, the web client relies on the web server to act as its authentication proxy by collecting user authentication data from the client and using it to establish an authenticated session.

Step 1: Initial Request

The web client requests the main application URL, shown in FIGURE 3-1.

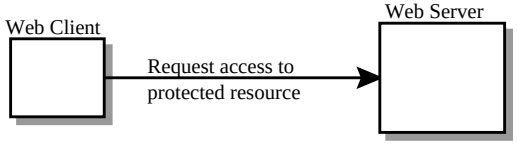

FIGURE 3-1 Initial Request

Since this client has not yet authenticated itself to the application environment, the server responsible for delivering the web portion of the application (hereafter referred to as "web server") detects this and invokes the appropriate authentication mechanism for this resource.

Step 2: Initial Authentication

The web server returns a form that the web client uses to collect authentication data (e.g., username and password) from the user. The web client forwards the authentication data to the web server, where it is validated by the web server, as shown in FIGURE 3-2.

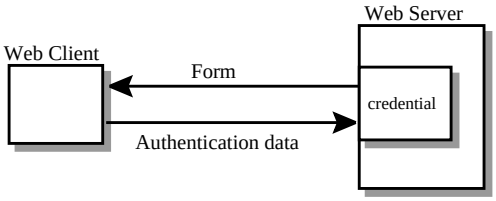

FIGURE 3-2 Initial Authentication

This validation mechanism could be local to the server, or it could leverage the underlying security services. The web server then sets a credential for the user.

Step 3: URL Authorization

The web server determines if the user whose identity is captured in the credential is authorized to access the resource represented by the URL. The web server performs the authorization decision by consulting the security policy (derived from the deployment descriptor) associated with the web resource to determine the security roles that are permitted access to the resource. The web container then tests the user's credentials against each role to determine if it can map the user to the role. FIGURE 3-3 shows this process.

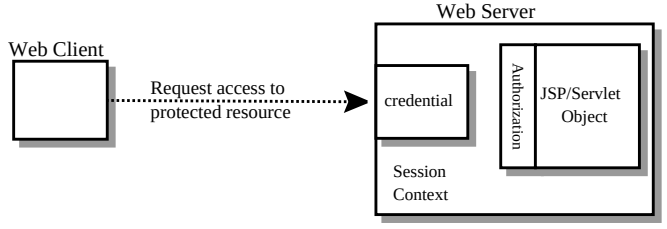

FIGURE 3-3 URL Authorization

The evaluation stops with an "is authorized" outcome on the first role that the web container is able to map the user to. A "not authorized" outcome is reached if the web container is unable to map the user to any of the permitted roles.

Step 4: Fulfilling the Original Request

If the user is authorized, the web server returns the result of the original URL request, as shown in FIGURE 3-4.

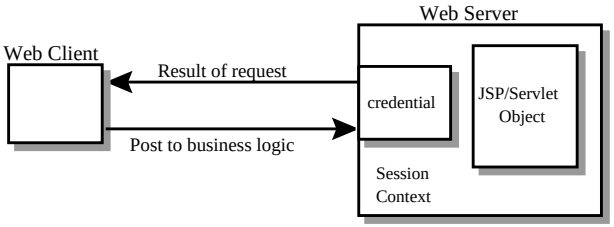

FIGURE 3-4 Fulfilling the Original Request

In this case, the response of a JSP page is returned. Next, the user performs some action (perhaps posting form data) that needs to be handled by the business logic component of the application.

Step 5. Invoking Enterprise Bean Business Methods

When the JSP page performs the remote method call to the enterprise bean, the user's credential is used to establish (as shown in FIGURE 3-5) a secure association between the JSP page and the enterprise bean. The association is implemented as two related security contexts, one in the web server and one in the EJB container.

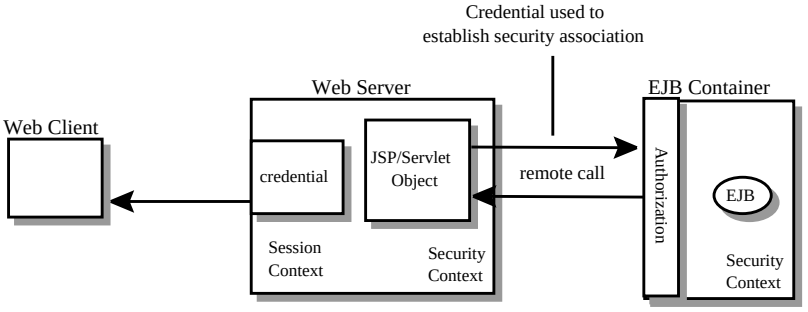

FIGURE 3-5 Invoking an Enterprise Bean Business Method

The EJB container is responsible for enforcing access control on the enterprise bean method; it does so by consulting the security policy (derived from the deployment descriptor) associated with the enterprise bean to determine the security roles that are permitted access to the method. Then for each role, the EJB container will use the security context associated with the call to determine if it can map the caller to the role. The evaluation stops with an "is authorized" outcome on the first role that the EJB container is able to map the caller to. A

"not authorized" outcome is reached if the container is unable to map the caller to any of the permitted roles and results in an exception being thrown by the container, and propagated back to the caller (in this case the JSP page). If the call "is authorized", the container dispatches control to the enterprise bean method, which then returns a result to the caller.

The platform provides two sets of methods for use by security aware applications: the `EJBContext` methods `isCallerInRole` and `getCallerPrincipal` available to enterprise beans through the EJB container, and the `HttpServletRequest` methods `isUserInRole` and `getUserPrincipal` available to servlets and JSP pages through the web container. When an enterprise bean calls the `isCallerInRole` method, the enterprise bean container determines if the caller (as represented by the security context) is in the specified role. When an enterprise bean calls the `getCallerPrincipal` method, the enterprise bean container returns the principal associated with the security context. The web container APIs behave similarly.

3.3 Security Architecture

This section describes the J2EE security architecture on which the security requirements defined by this specification are based.

3.3.1 Goals

1. Portability: The J2EE security architecture must support the Write Once, Run Anywhere™ application property.
2. Transparency: Application Component Providers should not have to know anything about security to write an application.
3. Isolation: The J2EE platform will perform authentication and access control, and its ability to do so will be established by the Deployer and managed by the System Administrator.

   By divorcing responsibility for security from the application, this specification ensures greater portability of J2EE applications.
4. Extensibility: The use of platform services by security aware applications must not compromise application portability. For applications that need access to information available in the security environment, this specification

provides APIs in the component programming model for the purpose of interacting with container/server security information. Applications that restrict their interactions to the provided APIs should retain portability.

5. Flexibility: Mechanisms and declarations of security properties of applications should not impose a particular security policy, but facilitate the implementation of security policies specific to the particular J2EE installation.

6. Abstraction: A component's security requirements are logically specified using deployment descriptors. Security roles and access requirements are mapped into environment specific security roles, users, and policies. A Deployer may choose to modify the security properties to be consistent with the deployment environment. The deployment descriptor should document which parameters can be modified and which should not.

7. Independence: Required security behaviors and deployment contracts should be implementable using a variety of popular security technologies.

8. Compatibility testing: The J2EE security requirements architecture must be expressed in a manner that allows for an unambiguous determination of whether or not an implementation is compatible.

9. Secure interoperability: Components executing in one J2EE product must be able to securely invoke services provided by another J2EE product from a different vendor. Those services may be provided by either web components or enterprise beans.

3.3.2 Non Goals

1. This specification does not dictate a specific security policy. Security policy for applications and for enterprise information systems vary for many reasons. This specification allows Product Providers to provide people the technology to implement and administer the policies they require.

2. This specification does not mandate a specific security technology, such as Kerberos, PK, NIS+, NTLM, etc.

3. This specification does not require that the J2EE security behaviors be universally implementable (i.e., using any or all security technologies).

4. This specification does not afford any warranty or assurance of the effective security of a J2EE product.

3.3.3 Terminology

This section introduces the terminology that is used to describe the security requirements of the J2EE platform.

Principal

A *principal* is an entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise. A principal is identified using a *principal name* and authenticated using *authentication data.* The content and format of the principal name and the authentication data depend upon the authentication protocol.

Security Policy Domain

A *security policy domain* (referred to as *security domain)* is a scope over which security policies are defined and enforced by a security administrator of the security service. A security policy domain is also sometimes referred to as a *realm.* This specification uses the term security policy domain or security domain.

Security Technology Domain

A *security technology domain* is the scope over which the same security mechanism (e.g., Kerberos) is used to enforce a security policy. A single security technology domain may include multiple security policy domains, for example.

Security Attributes

A set of *security attributes* is associated with every principal. The security attributes have many uses (e.g., access to protected resources, auditing of users, etc.). Security attributes can be associated with a principal by an authentication protocol and/or by the J2EE Product Provider.

The J2EE platform does not specify the security attributes that can be associated with a principal.

Credential

A *credential* might contain or reference information (security attributes) that can authenticate a principal to additional services. A principal acquires a credential upon authentication or from another principal that allows its credential to be used (*delegation*).

This specification does not specify the contents or the format of a credential, because both can vary widely.

3.3.4 Container Based Security

To achieve the goals for security in a J2EE environment, security for components is provided by their containers. A container provides security in two forms:

- Declarative security.
- Programmatic security.

3.3.5 Declarative Security

Declarative security refers to the means of expressing an application's security structure, including security roles, access control, and authentication requirements in a form external to the application. The deployment descriptor is the primary vehicle for declarative security in the J2EE platform.

A deployment descriptor is a contract between an Application Component Provider and a Deployer or Application Assembler. In the context of J2EE security, it can be used by an application programmer to represent an application's security related environmental requirements. Groups of components are associated with a deployment descriptor.

The application's logical security requirements are mapped by a Deployer to a representation of the security policy that is specific to the environment at deployment time. A Deployer uses a deployment tool to process the deployment descriptor.

At runtime, the container uses the security policy that was derived from the deployment descriptor and configured by the Deployer to enforce authorization (see Section 3.3.8, "Authorization Model").

3.3.6 Programmatic Security

Programmatic security is used by security aware applications. Programmatic security is useful when declarative security alone is not sufficient to express the security model of the application. Programmatic security consists of two methods of the EJB `EJBContext` interface and two methods of the servlet `HttpServletRequest` interface:

- `isCallerInRole (EJBContext)`
- `getCallerPrincipal (EJBContext)`
- `isUserInRole (HttpServletRequest)`
- `getUserPrincipal (HttpServletRequest)`

These methods allow components to make business logic decisions based on the security role of the caller or remote user. They also allow the component to determine the principal name of the caller or remote user to use as a database key, for example. (Note that the form and content of principal names will vary widely between products and enterprises, and portable components will not depend on the actual contents of a principal name.)

3.3.7 Distributed Security

Some Product Providers may produce J2EE products in which the containers for various component types are distributed. In a distributed environment, communication between J2EE components can be subject to security attacks (for example, data modification and replay attacks).

Such threats can be countered by using a *secure association* to secure communications. A secure association is shared security state information which permits secure communication between two components. Establishing a secure association could involve several steps such as:

1. Authenticating the target principal to the client and/or authenticating the client to the target principal.
2. Negotiating a quality of protection, such as confidentiality or integrity.
3. Establishing a security context between the components.

Since a container provides security in J2EE, secure associations for a component are typically established by a container. Secure associations for web access are specified here. Secure associations for access to enterprise beans are described in the EJB specification..

A J2EE Product Provider may allow control over the quality of protection or other aspects of secure association at deployment time. These aspects depend upon the application and are essentially application requirements. Applications can specify their quality of protection requirements for access to web resources using elements in their deployment descriptor. This specification does not define any mechanisms that an Application Component Provider can use to communicate an enterprise bean's requirements for secure associations.

3.3.8 Authorization Model

The J2EE authorization model is based on the concept of security roles. A security role is a logical grouping of users that is defined by an Application Component Provider or Assembler. It is then mapped by a Deployer to security identities (e.g., principals, groups, etc.) in the operational environment. A security role can be used either with declarative security or with programmatic security.

Declarative authorization can be used to control access to an enterprise bean method and is specified in the deployment descriptor. An enterprise bean method can be associated with a `method-permission` element in the deployment descriptor. The `method-permission` element contains a list of methods that can be accessed by a given security role. If the calling principal is in one of the security roles allowed access to a method, the principal is allowed to execute the method. Conversely, if the calling principal is in none of the roles, the caller is not allowed to execute the method. Access to web resources can be protected in a similar manner.

A security role can be used in the `EJBContext` method `isCallerInRole` and the `HttpServletRequest` method `isUserInRole`. Each method returns `true` if the calling principal is in the specified security role.

3.3.9 Role Mapping

Enforcement of either programmatic or declarative security depends upon determining if the principal associated with an incoming request of an enterprise bean or web resource is in a given security role or not. A container makes this determination based on the security attributes of the calling principal. For example,

1. A Deployer could have mapped a security role to a user group in the operational environment. In this case, the user group to which the calling principal belongs is retrieved from its security attributes. If the principal's user group matches the user group in the operational environment that the security role has been mapped to, the principal is in the security role.

2. A Deployer could have mapped a security role to a principal name in a security policy domain. In this case, the principal name of the calling principal is retrieved from its security attributes. If this principal is the same as the principal name to which the security role was mapped, the calling principal is in the security role.

The source of security attributes may vary across implementations of the J2EE platform. Security attributes could have been transmitted in the calling principal's credential or in the security context. If security attributes are not transmitted, they may be retrieved from a trusted third party such as a directory service or security service.

3.3.10 HTTP Login Gateways

Secure interoperability between enterprise beans in different security policy domains is not addressed in this specification. A future version will specify an interoperability protocol (based on industry standards) that will allow EJB containers in different domains to interoperate securely.

To gain access to another J2EE product that may be incompatible (i.e., in terms of communications protocols, security technology, or policy domain), a component may choose to log in to a foreign server via HTTP. HTTP over SSL can be used to provide secure, interoperable communication between J2EE products from different providers. An application component can be configured to use SSL mutual authentication when accessing a remote resource using HTTP. Applications using HTTP in this way may want to exchange data using XML or some other structured format, rather than HTML.

We call this use of HTTP with SSL mutual authentication to access a remote service an *HTTP Login Gateway*. Requirements in this area are specified in Section 3.4.1, "Web Clients."

3.3.11 User Authentication

User authentication is the process by which a user proves his or her identity to the system. This authenticated identity is then used to perform authorization decisions for accessing J2EE application components. An end user can authenticate using either of the two supported client types:

- Web client
- Application client

3.3.11.1 Web Client

A web client can authenticate a user to a web server using one of the following mechanisms:

- HTTP Basic Authentication

- HTTPS Client Authentication
- Form Based Authentication

HTTP Digest Authentication is not widely supported by web browsers and hence is not required.

The Deployer or System Administrator determines which method to apply to an application or groups of applications. A web client can employ a web server as its authentication proxy. In this case, the client's credentials are established for the client in the server, where they may be used by the server to perform authorization decisions, to act as the client in calls to enterprise beans, or to negotiate secure associations with resources.

Current web browsers commonly rely on proxy authentication.

HTTP Basic Authentication

HTTP Basic Authentication is the authentication mechanism supported by the HTTP protocol. This mechanism is based on a username and password. A web server requests a web client to authenticate the user. As part of the request, the web server passes the *realm* in which the user is to be authenticated. The web client obtains the username and the password from the user and transmits them to the web server. The web server then authenticates the user in the specified realm (referred to as *HTTP Realm* in this document).

HTTP Basic Authentication is not secure. Passwords are sent with a simple base64 encoding. The target server is not authenticated. Additional protection can be applied to overcome these weaknesses. For example, the password may be protected by applying security at the transport layer (e.g., HTTPS) or at the network layer (e.g., IPSEC or VPN).

Despite its limitations, the HTTP Basic Authentication mechanism is included in this specification because it is widely used in many form based applications.

HTTPS Authentication

End user authentication using HTTPS (HTTP over SSL) is a strong authentication mechanism. This mechanism requires the user to possess a Public Key Certificate (PKC). Currently, PKC is rarely used by end users on the Internet. However, it is useful in e-commerce applications and also for single-signon from within the browser. For these reasons, it is a required feature of the J2EE platform.

Form Based Authentication

The look and feel of the “login screen” cannot be controlled with the web browser’s built-in authentication mechanisms. This specification introduces the ability to package a standard HTML or servlet/JSP based form for logging in. The login form allows customization of the user interface. The form based authentication mechanism is described in the Servlet specification.

Web Single Signon

HTTP is a stateless protocol. However, many web applications need support for sessions that can maintain state across multiple requests from a client. Therefore, it is desirable to:

1. Make login mechanisms and policies a property of the environment the web application is deployed in.
2. Be able to use the same login session to represent a user to all the applications that they access.
3. Require the user to re-authenticate only when crossing a security policy domain.

Credentials that are acquired through a web login process are associated with the session. The container uses these credentials to establish a security context for the session. The container uses the security context to determine authorization for access to web resources and for the establishment of secure associations with other components or with enterprise beans.

Login Session

In the J2EE platform, login session support is provided by a Servlet container. When a user successfully authenticates with a web server, the container establishes a login session context for the user. The login session contains the credentials associated with the user.[1]

1.This is true where the client is stateless with respect to authentication, and as such, requires that the server act as its proxy and maintain its login context. In this case login session state is made available for use by the client by giving the client a reference (to use with its requests) to its authentication state stored on the server. Cookies or URL re-writing are used to carry such references. If SSL mutual authentication is used as the authentication protocol, the client can manage its own authentication context, and need not depend on references.

3.3.11.2 Application Client

Application clients (described in detail in Chapter 9) are client programs that may directly (i.e., without the help of a web browser and without traversing a web server) interact with enterprise beans. Of course, application clients may also access web resources.

Application clients, like the other J2EE application component types, execute in a managed environment that is provided by an appropriate container. Application clients are expected to have access to a graphical display and input device and can expect to communicate with a human user.

Application clients are used to authenticate the end user to the J2EE platform, for instance when accessing protected web resources or enterprise beans.

3.3.11.3 Lazy Authentication

There is a cost associated with authentication. For example, an authentication process may require exchanging multiple messages across the network. Therefore, it is desirable to perform authentication only when necessary (i.e., *lazy authentication*). With lazy authentication, an end user is not required to authenticate until the user tries to access a protected resource.

Lazy authentication can be used by first tier clients (applets, application clients) when they access protected resources that require authentication. When a user tries to access such a resource, the user can be asked to provide the needed authentication data. If a user is successfully authenticated, the user is allowed to access the resource.

3.4 User Authentication Requirements

3.4.1 Web Clients

The J2EE Product Provider must support several methods of web based user authentication.

3.4.1.1 Web Single Signon

All J2EE compatible web servers must support single signon by maintaining a login session for each web user. This allows applications to remain independent of the details of implementing and securing login information. This also provides the J2EE Product Provider with the flexibility to choose authentication mechanisms independent of the applications secured by these mechanisms. It must be possible for one login session to span more than one application, allowing a user to log in once and access multiple applications.

3.4.1.2 Login Sessions

All J2EE products must support login sessions as described in the Servlet specification.

Lazy authentication must be supported by web servers for protected web resources. If authentication is required, then one of the three required login mechanisms listed in the next section may be used.

3.4.1.3 Required Login Mechanisms

All J2EE products are required to support three login mechanisms: HTTP basic authentication, SSL mutual authentication, and form-based login. An application is not required to use any of these mechanisms, but they are required to be available for any application's use.

HTTP Basic Authentication

All J2EE products are required to support HTTP basic authentication (RFC2068). Platform Providers are also required to support basic authentication over SSL.

SSL Mutual Authentication

SSL 3.0[1] and the means to perform mutual (client and server) certificate based authentication are required by this specification.

All J2EE products must support the following cipher suites to ensure interoperable authentication with clients:

- `SSL_RSA_EXPORT_WITH_RC4_40_MD5`
- `SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`

These cipher suites are supported by the major web browsers and meet the U.S. government export restrictions.

Form Based Login

The web application deployment descriptor contains an element that causes a J2EE product to associate an HTML form resource (perhaps dynamically generated) with the web application. If the Deployer chooses this form of authentication (over HTTP basic, or SSL certificate based authentication), this form must be used as the user interface for login to the application.

The form based login mechanism and web application deployment descriptors are described in the Servlet 2.2 specification.

3.4.1.4 Unauthenticated Users

Web containers are required to support access to web resources by clients that have not authenticated themselves to the container. This is the common mode of access to web resources on the Internet. A web container reports that no user has been authenticated by returning `null` from the `HttpServletRequest` method `getUserPrincipal`.

The EJB specification requires that the `EJBContext` method `getCallerPrincipal` always return a valid `Principal` object. It can never return `null`. However, it's important that components running in a web container be able to call enterprise beans, even when no user has been authenticated in the web container. When a call is made in such a case from a component in a web container to an enterprise bean, a J2EE product must provide a principal for use in the call.

A J2EE product may provide a principal for use by unauthenticated callers using many approaches, including, but not limited to:

- Always use a single distinguished principal.
- Use a different distinguished principal per server, or per session, or per application.

1.The SSL 3.0 specification is available at: `http://home.netscape.com/eng/ssl3`

- Allow the deployer or system administrator to choose which principal to use.

This specification does not specify how a J2EE product should choose a principal to represent unauthenticated users, although future versions of this specification may add requirements in this area.

3.4.2 Application Clients

To satisfy the authentication and authorization constraints enforced by the enterprise bean containers and web containers, the application client container must authenticate the application user. The techniques used may vary based on the implementation of the application client container and are beyond the control of the application. The application client container may integrate with a J2EE product's authentication system, to provide a single signon capability, or the container may authenticate the user when the application is started. The container may delay authentication until it is necessary to access a protected resource or enterprise bean.

If the container needs to interact with the user to gather authentication data, the container must provide an appropriate user interface. In addition, an application client may provide a class that implements the `javax.security.auth.callback.CallbackHandler` interface and specify the class name in its deployment descriptor (see Section 9.7, "J2EE:application-client XML DTD" for details). The Deployer may override the callback handler specified by the application and require use of the container's default authentication user interface instead.

If use of a callback handler has been configured by the Deployer, the application client container must instantiate an object of this class and use it for all authentication interactions with the user. The application's callback handler must support all the `Callback` objects specified in the `javax.security.auth.callback` package.

Application clients execute in an environment controlled by a J2SE security manager and are subject to the security permissions defined in TABLE 6-2, "J2EE Security Permissions Set". Although this specification does not define the relationship between the operating system identity associated with a running application client and the authenticated user identity, a J2EE product's ability to relate these identities is a fundamental aspect of single signon.

Additional application client requirements are described in Chapter 9 of this specification.

3.4.3 Resource Authentication Requirements

Resources within an enterprise are often deployed in security policy domains that are different from the security policy domain to which the component belongs. Because the authentication mechanisms used to authenticate the caller to resources can vary widely, a J2EE product must provide the ability to authenticate in the security policy domain of the resource.

A Product Provider must support both of the following:

1. **Configured Identity.** Specification of the principal and authentication data for a resource by the Deployer at deployment time. A J2EE container must authenticate to the resource using the specified principal and authentication data; the application component must not need to provide this data. If the authentication data is stored by a J2EE container, then its confidentiality is the responsibility of the Product Provider.

2. **Programmatic Authentication.** Specification of the principal and authentication data for a resource by the application component at runtime using APIs appropriate to the resource. The application may obtain the principal and authentication data through a variety of mechanisms, including receiving them as parameters, obtaining them from the component's environment, etc.

In addition, the following techniques are recommended but not required by this specification:

3. **Principal Mapping.** A resource principal is determined by mapping from the identity and/or security attributes of the initiating/caller principal. In this case, a resource principal does not inherit identity or security attributes of a principal that it has been mapped from; the resource principal gets its identity and security attributes based on the mapping.

4. **Caller Impersonation.** A resource principal acts on behalf of an initiating/caller principal. Acting on behalf of a caller principal requires that the caller's identity and credentials be delegated to the underlying resource manager. The mechanism by which this is accomplished is specific to a security mechanism and an application server implementation.

   In some scenarios, a caller principal can be a delegate of an initiating principal. In this case, a resource principal transitively impersonates an initiating principal.

The support for principal delegation is typically specific to a security mechanism. For example, Kerberos supports a mechanism for the delegation of authentication. (Refer to the Kerberos v5 specification for more details.) The security technology specific details are out of the scope of this specification.

5. **Credentials Mapping.** This technique may be used when an application server and EIS support different authentication domains. For example, the initiating principal has been authenticated and has public key certificate-based credentials. The security environment for the resource manager is configured with the Kerberos authentication service. The application server is configured to map the public key certificate-based credentials associated with the initiating principal to the Kerberos credentials.

Additional information on resource authentication requirements can be found in the Connector specification.

3.5 Authorization Requirements

To support the authorization models described in this chapter, the following requirements are imposed on J2EE products.

3.5.1 Code Authorization

A J2EE product may restrict the use of certain J2SE classes and methods to secure and insure proper operation of the system. The minimum set of permissions that are required for a J2EE product are defined in Section 6.2, “Java 2 Platform, Standard Edition (J2SE) Requirements.” All J2EE products must be capable of deploying application components with exactly these permissions.

A J2EE Product Provider may choose to enable selective access to resources using the Java 2 protection model. The mechanism used is J2EE product dependent.

A future version of the J2EE deployment descriptor definition (see Chapter 8) may make it possible to express any additional permissions that a component needs.

3.5.2 Caller Authorization

A J2EE product must enforce the access control rules specified at deployment time (see Section 3.6, “Deployment Requirements”) and more fully described in the EJB and Servlet specifications.

It must be possible to configure a J2EE product so that the propagated caller identity is used in authorization decisions. This is, for all calls to all enterprise beans from a single application within a single J2EE product, the principal name returned by the `EJBContext` method `getCallerPrincipal` must be the same as that returned by the first enterprise bean in the call chain. If the first enterprise bean in the call chain is called by a servlet or JSP page, the principal name must be the same as that returned by the `HttpServletRequest` method `getUserPrincipal` in the calling servlet or JSP page. (However, if the `HttpServletRequest` method `getUserPrincipal` returns `null`, the principal used in calls to enterprise beans is not specified by this specification, although it must still be possible to configure enterprise beans to be callable by such components.) Note that this does not require delegation of credentials, only identification of the caller. This principal must be the principal used in authorization decisions for access to all enterprise beans in the call chain. The requirements in this paragraph apply only when a J2EE product has been configured to propagate caller identity.

J2EE products must also support the Run As capability that allows the Application Component Provider and the Deployer to specify an identity under which an enterprise bean or web component must run. In this case the original caller identity is not propagated to subsequent components in the call chain; instead the Run As identity is propagated. Note that this specification doesn’t specify any relationship between the Run As identity and any underlying operating system identity that may be used to access system resources such as files.

3.6 Deployment Requirements

The deployment descriptor describes the contract between the Application Component Provider or Assembler and the Deployer. All J2EE products must implement the access control semantics described in the EJB, JSP, and Servlet specifications, and provide a means of mapping the deployment descriptor security roles to the actual roles exposed by a J2EE product.

While most J2EE products will allow the Deployer to customize the role mappings and change the assignment of roles to methods, all J2EE products must support the ability to deploy applications and components using exactly the mappings and assignments specified in their deployment descriptors.

As described in the EJB specification and the Servlet specification, a J2EE product must provide a deployment tool or tools capable of assigning the security roles in deployment descriptors to the entities that are used to determine role membership at authorization time.

Application developers will need to specify (in the application's deployment descriptors) the security requirements of an application in which some components may be accessed by unauthenticated users as well as authenticated users (as described above in Section 3.4.1.4, "Unauthenticated Users"). Applications express their security requirements in terms of security roles, which the Deployer maps to users (principals) in the operational environment at deployment time. An application might define a role representing all authenticated and unauthenticated users and configure some enterprise bean methods to be accessible by this role.

To support such usage, this specification requires that it be possible to map an application defined security role to the universal set of application principals independent of authentication.

3.7 Future Directions

3.7.1 Auditing

This specification does not specify requirements for the auditing of security relevant events, nor APIs for application components to generate audit records. A future version of this specification may include such a specification for products that choose to provide auditing.

CHAPTER 4

Transaction Management

This chapter describes the transaction management and runtime environment that must be supported by Product Providers on the Java™ 2 Platform, Enterprise Edition (J2EE).

Product Providers must transparently support transactions that span multiple components and transactional resources within a single J2EE product, as described in this chapter. These requirements must be met regardless of whether the J2EE product is implemented as a single process, multiple processes on the same node, or multiple processes on multiple network nodes.

The following components are considered transactional resources and must behave as specified here:

- JDBC connections
- JMS sessions
- Resource adapter connections for resource adapters specifying the `XA_TRANSACTION` transaction level

4.1 Overview

A J2EE Product Provider must support a transactional application that is comprised of a combination of Servlets or JSP pages accessing multiple enterprise beans within a single transaction. Each component may also acquire one or more connections to access one or more transactional resource managers.

For example, in FIGURE 4-1, the call tree starts from a Servlet or JSP page accessing multiple enterprise beans, which in turn may access other enterprise beans. The components access resource managers via connections.

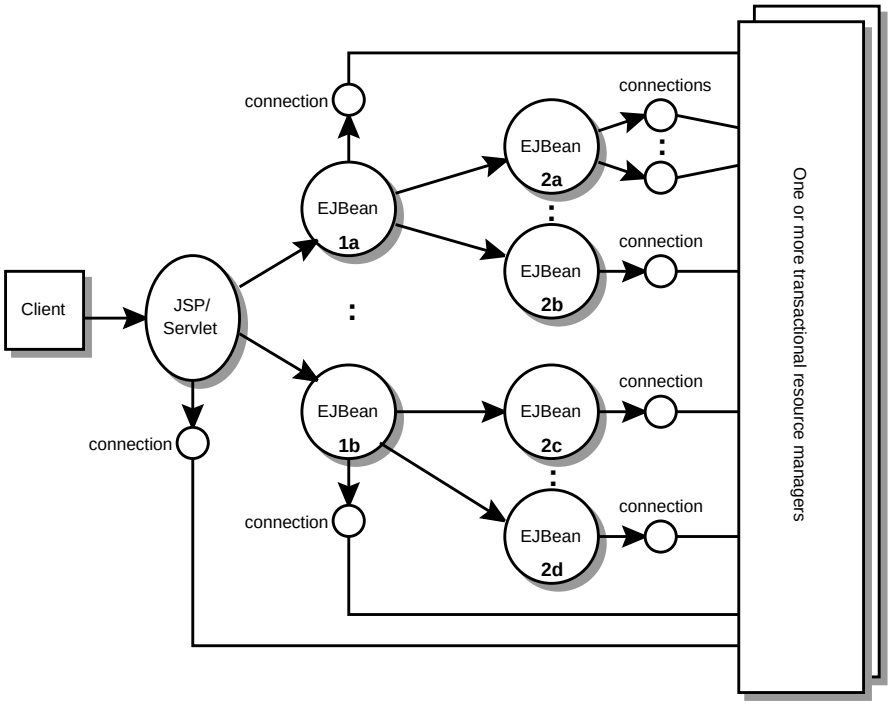

FIGURE 4-1 Servlets / JSP pages accessing enterprise Beans

The Application Component Provider specifies, using a combination of programmatic and declarative transaction demarcation APIs, how the platform must manage transactions on behalf of the application.

For example, the application may require that all the components in FIGURE 4-1 access resources as part of a single transaction. The Platform Provider must provide the transaction capabilities to support such a scenario.

This specification does not define how the components and the resources are partitioned or distributed within a single J2EE product. In order to achieve the transactional semantics required by the application, the J2EE Product Provider is free to execute the application components sharing a transaction in the same Java virtual machine, or distribute them across multiple virtual machines.

The rest of this chapter describes the transactional requirements for a J2EE product in more detail.

4.2 Requirements

This section defines the transaction support requirements for the J2EE Product Provider.

4.2.1 Web Components

Servlets and JSP pages are allowed to demarcate transactions using the `javax.transaction.UserTransaction` interface, defined in the JTA specification. These web components are allowed to access multiple resource managers and invoke multiple enterprise beans within a single transaction. The web component assumes that the transaction is automatically propagated to the enterprise beans (subject to the enterprise bean transaction attributes, e.g., such beans must use Container Managed Transactions) and transactional resource managers.

The J2EE platform must meet the following requirements:

- The J2EE platform must provide an object implementing the `javax.transaction.UserTransaction` interface to both servlets and JSP pages. The platform must publish the `UserTransaction` object in the Java™ Naming and Directory Interface (JNDI) name space available to web components under the name `java:comp/UserTransaction`.
- If a web component invokes an enterprise bean from a thread associated with a JTA transaction, the J2EE platform must propagate the transaction context with the enterprise bean invocation. Whether the target enterprise bean will be invoked in this transaction context or not is determined by the rules defined in the EJB specification.

  Note that this transaction propagation requirement applies only to invocations of enterprise beans in the same J2EE product instance as the invoking component. Invocations of enterprise beans in another J2EE product instance (e.g., using the EJB interoperability protocol) need not propagate the transaction context; see the EJB specification for details.
- If a web component accesses a transactional resource manager from a thread associated with a JTA transaction, the J2EE platform must ensure that the resource access is included as part of the JTA transaction.
- If a web component creates a thread, the J2EE platform must ensure that the newly created thread is not associated with any JTA transaction.

The Product Provider is not required to support import of a transaction context by a web component from its client.

The Product Provider is not required to support transaction context propagation across multiple web components accessed via an HTTP request. If a web component associated with a transaction makes an HTTP request to another web component, the transaction context is not propagated to the target servlet or page. (The HTTP protocol does not support such transaction context propagation.)

However, when another web component is invoked through the `RequestDispatcher` interface, any active transaction context must be propagated to the called web component.

A web component may only start a transaction in its `service` method. A transaction that is started by a servlet or JSP page must be completed before the `service` method returns. That is, transactions may not span web requests from a client. Returning from the `service` method with an active transaction context is an error. The web container is required to detect this error and abort the transaction.

There are many subtle and complex interactions between the use of transactional resources and threads. To ensure correct operation, web components should obey the following guidelines, and the web container must support at least these usages.

- JTA transactions should be started and completed only from the thread in which the `service` method is called. If the web component creates additional threads for any purpose, these threads should not attempt to start JTA transactions.
- Transactional resources may be acquired and released by a thread other than the `service` method thread, but should not be shared between threads.
- Transactional resource objects (e.g., JDBC `Connection` objects) should not be stored in static fields.
- For web components implementing `SingleThreadModel`, transactional resource objects may be stored in class instance fields.
- For web components not implementing `SingleThreadModel`, transactional resource objects should not be stored in class instance fields, and should be acquired and released within the same invocation of the `service` method.
- Enterprise beans may be invoked from any thread used by a web component. Transaction context propagation requirements are described above and in the EJB specification.

4.2.2 Enterprise JavaBeans™ Components

The J2EE Product Provider must implement support for transactions as defined in the EJB specification.

4.2.3 Application Clients

The J2EE Product Provider is not required to provide any transaction management support for application clients.

4.2.4 Applet Clients

The J2EE Product Provider is not required to provide any transaction management support for applets.

4.2.5 Transactional JDBC™ Technology Support

A J2EE product must support a JDBC technology database as a transactional resource manager. The platform must enable transactional JDBC API access from servlets, JSP pages, and enterprise beans.

It must be possible to access the JDBC technology database from multiple application components within a single transaction. For example, a servlet may wish to start a transaction, access a database, invoke an enterprise bean that accesses the same database as part of the same transaction, and finally commit the transaction.

4.2.6 Transactional JMS Support

A J2EE product must support a JMS provider as a transactional resource manager. The platform must enable transactional JMS access from servlets, JSP pages, and enterprise beans.

It must be possible to access the JMS provider from multiple application components within a single transaction. For example, a servlet may wish to start a transaction, send a JMS message, invoke an enterprise bean that also sends a JMS message as part of the same transaction, and finally commit the transaction.

4.2.7 Transactional Resource Adapter Support

A J2EE product must support resource adapters that use `XA_TRANSACTION` mode as transactional resource managers. The platform must enable transactional access to the resource adapter from servlets, JSP pages, and enterprise beans.

It must be possible to access the resource adapter from multiple application components within a single transaction. For example, a servlet may wish to start a transaction, access the resource adapter, invoke an enterprise bean that also accesses the resource adapter as part of the same transaction, and finally commit the transaction.

4.3 Transaction Interoperability

4.3.1 Multiple J2EE Platform Interoperability

This specification does not require the Product Provider to implement any particular protocol for transaction interoperability across multiple J2EE products. J2EE compatibility requires neither interoperability among identical J2EE products from the same Product Provider, nor among heterogeneous J2EE products from multiple Product Providers.

Note – We recommend that J2EE Product Providers use the IIOP transaction propagation protocol defined by OMG and described in the OTS specification (and implemented by the Java Transaction Service), for transaction interoperability when using the EJB interoperability protocol based on RMI-IIOP. We plan to require the IIOP transaction propagation protocol as the EJB server transaction interoperability protocol in a future release of this specification.

4.3.2 Support for Transactional Resource Managers

This specification requires all J2EE products to support the `javax.transaction.xa.XAResource` interface, as specified in the Connector specification. This specification does not require that JDBC drivers or

JMS providers use the `javax.transaction.xa.XAResource` interface, although they must meet the requirements of a transactional resource manager described in this chapter. In particular, it must be possible to combine operations on one or more JDBC databases, one or more JMS sessions, one or more enterprise beans, and all resource adapters supporting the `XA_TRANSACTION` mode in a single JTA transaction.

4.4 Local Transaction Optimization

4.4.1 Requirements

Using a resource manager specific local transaction mechanism instead of an externally coordinated global transaction is a useful optimization containers might provide, if all the work done as part of a transaction uses a single resource manager. A resource manager local transaction is typically more efficient than a global transaction and results in better performance.

Containers may choose to provide local transaction optimization, but are not required to do so. Local transaction optimization must be transparent to a J2EE application. The following section describes a possible mechanism for containers to accomplish local transaction optimization.

Containers may attempt local transaction optimization only for transactions initiated within the same container. Local transaction optimization must not be attempted for transactions that are imported from a different container.

4.4.2 A Possible Design

This section does not describe any additional requirements for a J2EE product, but is meant only as an illustration of how to implement previously described requirements.

Whenever a transaction is initiated, a global transaction is not started. Instead the container remembers the fact that a transaction has been started. When the first connection to a resource manager is established as part of the transaction, a resource manager specific local transaction is started on the connection.

Any subsequent connection acquired as part of the transaction that can share the local transaction on the first connection is allowed to share the local transaction.

A global transaction is started lazily under the following conditions:

- When a subsequent connection acquired as part of the transaction cannot share the resource manager local transaction on the first connection or if it uses a different resource manager.
- When a transaction is exported to a different container.

After the lazy start of a global transaction, any subsequent connection acquired may either share the local transaction on the first connection, or be part of the global transaction, depending on the resource manager it accesses.

When a transaction completion (commit or rollback) is attempted, there are two possibilities:

- A global transaction had never been started; only a single resource manager had been accessed as part of the transaction. In such a case, the transaction is completed using the resource manager specific local transaction mechanism.
- A global transaction had been started. In such a case, the transaction is completed treating the resource manager local transaction as a last resource in the global 2-phase commit protocol, that is, using the last resource 2-phase commit optimization.

4.5 Connection Sharing

When multiple connections acquired by a J2EE application use the same resource manager, containers may attempt to share connections within the same transaction scope. Sharing connections typically results in efficient usage of resources and better performance. Containers may choose to provide connection sharing, but are not required to do so.

Connections to resource managers acquired by J2EE applications are considered potentially shared or shareable. However, a J2EE application component that intends to use a connection in an unshareable way must leave a deployment hint to that effect, which will prevent the connection from being shared by the container. Examples of unshareable usage of a connection include changing the security attributes, isolation levels, character settings, localization configuration, etc. Containers must not attempt to share connections that are marked unshareable. If a connection is marked shareable, it must be transparent to the application whether the connection is actually shared or not, provided that the application is properly using the connection in a shareable manner.

J2EE application components may use the optional deployment descriptor element `res-sharing-scope` to indicate whether a connection to a resource manager is shareable or unshareable. Containers should assume connections to be shareable if no deployment hint is provided. Refer to Chapter 9, "J2EE:application-client XML DTD," the EJB specification, and the Servlet specification for a description of the deployment descriptor element.

J2EE application components may cache connection objects and reuse them across multiple transactions. Containers that provide connection sharing should transparently switch such cached connection objects (at dispatch time) to point to an appropriate shared connection with the correct transaction scope. Refer to the Connector specification for a detailed description of connection sharing.

4.6 System Administration Tools

Although there are no compatibility requirements for system administration capabilities, the J2EE Product Provider will typically include tools that allow the System Administrator to perform the following tasks:

- Integrate transactional resource managers with the platform.
- Configure the transaction management parts of the platform.
- Monitor transactions at runtime.
- Receive notifications of abnormal transaction processing conditions (such as abnormally high number of transaction rollbacks).

CHAPTER 5

Naming

This chapter describes the naming system requirements for the Java™ 2 Platform, Enterprise Edition (J2EE). These requirements are based on features defined in the JNDI specification.

Note – This chapter is largely derived from the EJB specification chapter, “Enterprise bean environment.”

5.1 Overview

The Application Assembler and Deployer should be able to customize an application’s business logic without accessing the application’s source code.

In addition, ISVs typically develop applications that are, to a large degree, independent from the operational environment in which the application will be deployed. Most applications must access resources and external information. The key issue is how applications can locate the external information without knowledge of how the external information is named and organized in the target operational environment.

This specification defines naming requirements for the J2EE platform that address both of the above issues.

This chapter is organized as follows.

- Section 5.2 defines the interfaces that specify and access the application component’s naming environment. The section illustrates the use of the application component’s naming environment for generic customization of the application component’s business logic.

- Section 5.3 defines the interfaces for obtaining the home interface of an enterprise bean using an EJB reference. An EJB reference is a special entry in the application component's environment.
- Section 5.4 defines the interfaces for obtaining a resource manager connection factory using a resource manager connection factory reference. A resource manager connection factory reference is a special entry in the application component's environment.

Only J2EE application clients, enterprise beans, and web components are required to have access to a JNDI naming environment. Only the containers for these application component types must provide the naming environment support described here.

The deployment descriptor entries described here are present in identical form in the deployment descriptor DTDs for each of these application component types. See the corresponding specification of each application component type for the details.

5.2 Java Naming and Directory Interface™ (JNDI) Naming Context

The application component's naming environment is a mechanism that allows customization of the application component's business logic during deployment or assembly. The application component's environment allows the application component to be customized without the need to access or change the application component's source code.

The container implements the application component's environment, and provides it to the application component instance as a JNDI naming context. The application component's environment is used as follows:

1. The application component's business methods access the environment using the JNDI interfaces. The Application Component Provider declares in the deployment descriptor all the environment entries that the application component expects to be provided in its environment at runtime.
2. The container provides an implementation of the JNDI naming context that stores the application component environment. The container also provides the tools that allow the Deployer to create and manage the environment of each application component.

3. The Deployer uses the tools provided by the container to initialize the environment entries that are declared in the application component's deployment descriptor. The Deployer can set and modify the values of the environment entries.

4. The container makes the environment naming context available to the application component instances at runtime. The application component's instances use the JNDI interfaces to obtain the values of the environment entries.

Each application component defines its own set of environment entries. All instances of an application component within the same container share the same environment entries. Application component instances are not allowed to modify the environment at runtime.

Note – Terminology warning: The application component's "environment" should not be confused with the "environment properties" defined in the JNDI documentation. The JNDI environment properties are used to initialize and configure the JNDI naming context itself. The application component's environment is accessed through a JNDI naming context for direct use by the application component.

The following subsections describe the responsibilities of each J2EE Role.

5.2.1 Application Component Provider's Responsibilities

This section describes the Application Component Provider's view of the application component's environment, and defines his or her responsibilities. It does so in two sections, the first describing API for accessing environment entries, and the second describing syntax for declaring the environment entries.

5.2.1.1 Access to application component's environment

An application component instance locates the environment naming context using the JNDI interfaces. An instance creates a `javax.naming.InitialContext` object by using the constructor with no arguments, and looks up the naming environment via the `InitialContext` under the name `java:comp/env`. The application component's environment entries are stored directly in the environment naming context, or in any of its direct or indirect subcontexts.

The value of an environment entry is of the Java programming language type declared by the Application Component Provider in the deployment descriptor.

The following code example illustrates how an application component accesses its environment entries.

```
public void setTaxInfo(int numberOfExemptions, ...)
      throws InvalidNumberOfExemptionsException {
   ...

   // Obtain the application component's environment naming context.
   Context initCtx = new InitialContext();
   Context myEnv = (Context)initCtx.lookup("java:comp/env");

   // Obtain the maximum number of tax exemptions
   // configured by the Deployer.
   Integer max = (Integer)myEnv.lookup("maxExemptions");

   // Obtain the minimum number of tax exemptions
   // configured by the Deployer.
   Integer min = (Integer)myEnv.lookup("minExemptions");

   // Use the environment entries to customize business logic.
   if (numberOfExeptions > max.intValue() ||
       numberOfExemptions < min.intValue())
          throw new InvalidNumberOfExemptionsException();

   // Get some more environment entries. These environment
   // entries are stored in subcontexts.
   String val1 = (String)myEnv.lookup("foo/name1");
   Boolean val2 = (Boolean)myEnv.lookup("foo/bar/name2");

   // The application component can also lookup using full pathnames.
   Integer val3 = (Integer)
      initCtx.lookup("java:comp/env/name3");
   Integer val4 = (Integer)
      initCtx.lookup("java:comp/env/foo/name4");
   ...
}
```

5.2.1.2 Declaration of environment entries

The Application Component Provider must declare all the environment entries accessed from the application component's code. The environment entries are declared using the `env-entry` elements in the deployment descriptor. Each `env-entry` element describes a single environment entry. The `env-entry` element consists of an optional description of the environment entry, the environment entry name relative to the `java:comp/env` context, the expected Java programming language type of the environment entry value (i.e., the type of the object returned from the JNDI `lookup` method), and an optional environment entry value.

The environment entry values may be one of the following Java types: `String`, `Byte`, `Short`, `Integer`, `Long`, `Boolean`, `Double`, and `Float`.

If the Application Component Provider provides a value for an environment entry, the value can be changed later by the Application Assembler or Deployer. The value must be a string that is valid for the constructor of the specified type that takes a single `String` parameter.

The following example is the declaration of environment entries used by the application component whose code was illustrated in the previous subsection.

```
...
<env-entry>
   <description>
         The maximum number of tax exemptions
         allowed to be set.
   </description>
   <env-entry-name>maxExemptions</env-entry-name>
   <env-entry-type>java.lang.Integer</env-entry-type>
   <env-entry-value>15</env-entry-value>
</env-entry>
<env-entry>
   <description>
         The minimum number of tax exemptions
         allowed to be set.
   </description>
   <env-entry-name>minExemptions</env-entry-name>
   <env-entry-type>java.lang.Integer</env-entry-type>
   <env-entry-value>1</env-entry-value>
</env-entry>
<env-entry>
   <env-entry-name>foo/name1</env-entry-name>
   <env-entry-type>java.lang.String</env-entry-type>
```

```
   <env-entry-value>value1</env-entry-value>
</env-entry>
<env-entry>
   <env-entry-name>foo/bar/name2</env-entry-name>
   <env-entry-type>java.lang.Boolean</env-entry-type>
   <env-entry-value>true</env-entry-value>
</env-entry>
<env-entry>
   <description>Some description.</description>
   <env-entry-name>name3</env-entry-name>
   <env-entry-type>java.lang.Integer</env-entry-type>
</env-entry>
<env-entry>
   <env-entry-name>foo/name4</env-entry-name>
   <env-entry-type>java.lang.Integer</env-entry-type>
   <env-entry-value>10</env-entry-value>
</env-entry>
...
```

5.2.2 Application Assembler's Responsibilities

The Application Assembler is allowed to modify the values of the environment entries set by the Bean Provider, and is allowed to set the values of those environment entries for which the Bean Provider has not specified any initial values.

5.2.3 Deployer's Responsibilities

The Deployer must ensure that the values of all the environment entries declared by an application component are set to meaningful values.

The Deployer can modify the values of the environment entries that have been previously set by the Application Component Provider and/or Application Assembler, and must set the values of those environment entries for which no value has been specified.

5.2.4 J2EE Product Provider's Responsibilities

The J2EE Product Provider has the following responsibilities:

- Provide a deployment tool that allows the Deployer to set and modify the values of the application component's environment entries.
- Implement the `java:comp/env` environment naming context, and provide it to the application component instances at runtime. The naming context must include all the environment entries declared by the Application Component Provider, with their values supplied in the deployment descriptor or set by the Deployer. The environment naming context must allow the Deployer to create subcontexts if they are needed by an application component.
- The container must ensure that the application component instances have only read access to their environment variables. The container must throw the `javax.naming.OperationNotSupportedException` from all the methods of the `javax.naming.Context` interface that modify the environment naming context and its subcontexts.

5.3 Enterprise JavaBeans™ (EJB) References

This section describes the programming and deployment descriptor interfaces that allow the Application Component Provider to refer to the homes of enterprise beans using "logical" names called EJB references. The EJB references are special entries in the application component's naming environment. The Deployer binds the EJB references to the enterprise bean's homes in the target operational environment.

The deployment descriptor also allows the Application Assembler to *link* an EJB reference declared in one application component to an enterprise bean contained in an ejb-jar file in the same J2EE application. The link is an instruction to the tools used by the Deployer describing the binding of the EJB reference to the home of the specified target enterprise bean.

5.3.1 Application Component Provider's Responsibilities

This subsection describes the Application Component Provider's view and responsibilities with respect to EJB references.It does so in two sections, the first describing the API for accessing EJB references, and the second describing the syntax for declaring the EJB references.

5.3.1.1 Programming interfaces for EJB references

The Application Component Provider must use EJB references to locate the home interfaces of enterprise bean as follows.

- Assign an entry in the application component's environment to the reference. (See subsection 5.3.1.2 for information on how EJB references are declared in the deployment descriptor.)
- This specification recommends, but does not require, that all references to enterprise beans be organized in the `ejb` subcontext of the application component's environment (i.e., in the `java:comp/env/ejb` JNDI context).
- Look up the home interface of the referenced enterprise bean in the application component's environment using JNDI.

The following example illustrates how an application component uses an EJB reference to locate the home interface of an enterprise bean.

```
public void changePhoneNumber(...) {
   ...

   // Obtain the default initial JNDI context.
   Context initCtx = new InitialContext();

   // Look up the home interface of the EmployeeRecord
   // enterprise bean in the environment.
   Object result = initCtx.lookup(
      "java:comp/env/ejb/EmplRecord");

   // Convert the result to the proper type.
   EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)
      javax.rmi.PortableRemoteObject.narrow(result,
               EmployeeRecordHome.class);
   ...
}
```

In the example, the Application Component Provider assigned the environment entry `ejb/EmplRecord` as the EJB reference name to refer to the home of an enterprise bean.

5.3.1.2 Declaration of EJB references

Although the EJB reference is an entry in the application component's environment, the Application Component Provider must not use a `env-entry` element to declare it. Instead, the Application Component Provider must declare all the EJB references using the `ejb-ref` elements of the deployment descriptor. This allows the consumer of the application component's jar file (i.e., the Application Assembler or Deployer) to discover all the EJB references used by the application component.

Each `ejb-ref` element describes the interface requirements that the referencing application component has for the referenced enterprise bean. The `ejb-ref` element contains an optional `description` element; and the mandatory `ejb-ref-name`, `ejb-ref-type`, `home`, and `remote` elements.

The `ejb-ref-name` element specifies the EJB reference name; its value is the environment entry name used in the application component code. The `ejb-ref-type` element specifies the expected type of the enterprise bean; its value must be either `Entity` or `Session`. The `home` and `remote` elements specify the expected Java types of the referenced enterprise bean's home and remote interfaces.

The following example illustrates the declaration of EJB references in the deployment descriptor.

```
...
<ejb-ref>
   <description>
         This is a reference to the entity bean that
         encapsulates access to employee records.
   </description>
   <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <home>com.wombat.empl.EmployeeRecordHome</home>
   <remote>com.wombat.empl.EmployeeRecord</remote>
</ejb-ref>

<ejb-ref>
   <ejb-ref-name>ejb/Payroll</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <home>com.aardvark.payroll.PayrollHome</home>
   <remote>com.aardvark.payroll.Payroll</remote>
</ejb-ref>

<ejb-ref>
```

```
    <ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.wombat.empl.PensionPlanHome</home>
    <remote>com.wombat.empl.PensionPlan</remote>
</ejb-ref>
...
```

5.3.2 Application Assembler's Responsibilities

The Application Assembler can use the `ejb-link` element in the deployment descriptor to link an EJB reference to a target enterprise bean. The link will be observed by the deployment tools.

The Application Assembler specifies the link to an enterprise bean as follows:

- The Application Assembler uses the optional `ejb-link` element of the `ejb-ref` element of the referencing application component. The value of the `ejb-link` element is the name of the target enterprise bean. (It is the name defined in the `ejb-name` element of the target enterprise bean.) The target enterprise bean can be in any ejb-jar file in the same J2EE application as the referencing application component.
- Alternatively, to avoid the need to rename enterprise beans to have unique names within an entire J2EE application, the Application Assembler may use the following syntax in the `ejb-link` element of the referencing application component. The Application Assembler specifies the path name of the ejb-jar file containing the referenced enterprise bean and appends the `ejb-name` of the target bean separated from the path name by "#". The path name is relative to the referencing application component jar file.
- The Application Assembler must ensure that the target enterprise bean is type-compatible with the declared EJB reference. This means that the target enterprise bean must be of the type indicated in the `ejb-ref-type` element, and that the home and remote interfaces of the target enterprise bean must be Java type-compatible with the interfaces declared in the EJB reference.

The following example illustrates the use of the `ejb-link` element in the deployment descriptor. The enterprise bean reference should be satisfied by the bean named `EmployeeRecord`. The `EmployeeRecord` enterprise bean may be packaged in the same module as the component making this reference, or it may be packaged in another module within the same J2EE application as the component making this reference.

```
...
<ejb-ref>
```

```
   <description>
         This is a reference to the entity bean that
         encapsulates access to employee records. It
         has been linked to the entity bean named
         EmployeeRecord in this application.
   </description>
   <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <home>com.wombat.empl.EmployeeRecordHome</home>
   <remote>com.wombat.empl.EmployeeRecord</remote>
   <ejb-link>EmployeeRecord</ejb-link>
</ejb-ref>
...
```

The following example illustrates using the `ejb-link` element to indicate an enterprise bean reference to the `ProductEJB` enterprise bean that is in the same J2EE application unit but in a different ejb-jar file.

```
...
<ejb-ref>
   <description>
         This is a reference to the entity bean that
         encapsulates access to a product. It
         has been linked to the entity bean named
         ProductEJB in the product.jar file in this application.
   </description>
   <ejb-ref-name>ejb/Product</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <home>com.acme.products.ProductHome</home>
   <remote>com.acme.products.Product</remote>
   <ejb-link>../products/product.jar#ProductEJB</ejb-link>
</ejb-ref>
...
```

5.3.3 Deployer's Responsibilities

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared EJB references are bound to the homes of enterprise beans that exist in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the target enterprise bean's home.

- The Deployer must ensure that the target enterprise bean is type-compatible with the types declared for the EJB reference. This means that the target enterprise bean must be of the type indicated in the `ejb-ref-type` element, and that the home and remote interfaces of the target enterprise bean must be Java type-compatible with the home and remote interfaces declared in the EJB reference.
- If an EJB reference declaration includes the `ejb-link` element, the Deployer must bind the enterprise bean reference to the home of the enterprise bean specified as the link's target.

5.3.4 J2EE Product Provider's Responsibilities

The J2EE Product Provider must provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection. The deployment tools provided by the J2EE Product Provider must be able to process the information supplied in the `ejb-ref` elements in the deployment descriptor.

At the minimum, the tools must be able to:

- Preserve the application assembly information in the `ejb-link` elements by binding an EJB reference to the home interface of the specified target enterprise bean.
- Inform the Deployer of any unresolved EJB references, and allow him or her to resolve an EJB reference by binding it to a specified compatible target enterprise bean.

5.4 Resource Manager Connection Factory References

A resource manager connection factory is an object that is used to create connections to a resource manager. For example, an object that implements the `javax.sql.DataSource` interface is a resource manager connection factory for `java.sql.Connection` objects that implement connections to a database management system.

This section describes the application component programming and deployment descriptor interfaces that allow the application component code to refer to resource factories using logical names called resource manager connection factory references. The resource manager connection factory references are

special entries in the application component's environment. The Deployer binds the resource manager connection factory references to the actual resource manager connection factories that exist in the target operational environment. Because these resource manager connection factories allow the Container to affect resource management, the connections acquired through the resource manager connection factory references are called managed resources (e.g. these resource manager connection factories allow the Container to implement connection pooling and automatic enlistment of the connection with a transaction).

Resource manager connection factory objects accessed through the naming environment are only valid within the component instance that performed the lookup. See the individual component specifications for additional restrictions that may apply.

5.4.1 Application Component Provider's Responsibilities

This subsection describes the Application Component Provider's view of locating resource factories and defines his or her responsibilities. It does so in two sections, the first describing the API for accessing resource manager connection factory references, and the second describing the syntax for declaring the factory references.

5.4.1.1 Programming interfaces for resource manager connection factory references

The Application Component Provider must use resource manager connection factory references to obtain connections to resources as follows.

- Assign an entry in the application component's naming environment to the resource manager connection factory reference. (See subsection 5.4.1.2 for information on how resource manager connection factory references are declared in the deployment descriptor.)
- This specification recommends, but does not require, that all resource manager connection factory references be organized in the subcontexts of the application component's environment, using a different subcontext for each resource manager type. For example, all JDBC™ DataSource references should be declared in the `java:comp/env/jdbc` subcontext, all JMS connection factories in the `java:comp/env/jms` subcontext, all JavaMail connection factories in the `java:comp/env/mail` subcontext, and all URL connection factories in the `java:comp/env/url` subcontext.

- Lookup the resource manager connection factory object in the application component's environment using the JNDI interface.
- Invoke the appropriate method on the resource manager connection factory method to obtain a connection to the resource. The factory method is specific to the resource type. It is possible to obtain multiple connections by calling the factory object multiple times.

The Application Component Provider has two choices with respect to dealing with associating a principal with the resource manager access:

- Allow the Deployer to set up principal mapping or resource manager signon information. In this case, the application component code invokes a resource manager connection factory method that has no security-related parameters.
- Sign on to the resource from the application component code. In this case, the application component invokes the appropriate resource manager connection factory method that takes the signon information as method parameters.

The Application Component Provider uses the `res-auth` deployment descriptor element to indicate which of the two resource authentication approaches is used.

We expect that the first form (i.e., letting the Deployer set up the resource signon information) will be the approach used by most application components.

The following code sample illustrates obtaining a JDBC connection.

```
public void changePhoneNumber(...) {
   ...

   // obtain the initial JNDI context
   Context initCtx = new InitialContext();

   // perform JNDI lookup to obtain resource manager connection factory
   javax.sql.DataSource ds = (javax.sql.DataSource)
      initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");

   // Invoke factory to obtain a resource. The security
   // principal for the resource is not given, and therefore
   // it will be configured by the Deployer.
   java.sql.Connection con = ds.getConnection();
   ...
}
```

5.4.1.2 Declaration of resource manager connection factory references in deployment descriptor

Although a resource manager connection factory reference is an entry in the application component's environment, the Application Component Provider must not use an `env-entry` element to declare it.

Instead, the Application Component Provider must declare all the resource manager connection factory references in the deployment descriptor using the `resource-ref` elements. This allows the consumer of the application component's jar file (i.e., the Application Assembler or Deployer) to discover all the resource manager connection factory references used by an application component.

Each `resource-ref` element describes a single resource manager connection factory reference. The `resource-ref` element consists of the `description` element; and the mandatory `res-ref-name`, `res-type`, and `res-auth` elements. The `res-ref-name` element contains the name of the environment entry used in the application component's code. The name of the environment entry is relative to the `java:comp/env` context (e.g., the name should be `jdbc/EmployeeAppDB` rather than `java:comp/env/jdbc/EmployeeAppDB`). The `res-type` element contains the Java type of the resource manager connection factory that the application component code expects. The `res-auth` element indicates whether the application component code performs resource signon programmatically, or whether the container signs on to the resource based on the principal mapping information supplied by the Deployer. The Application Component Provider indicates the signon responsibility by setting the value of the `res-auth` element to `Application` or `Container`.

A resource manager connection factory reference is scoped to the application component whose declaration contains the `resource-ref` element. This means that the resource manager connection factory reference is not accessible from other application components at runtime, and that other application components may define `resource-ref` elements with the same `res-ref-name` without causing a name conflict.

The type declaration allows the Deployer to identify the type of the resource manager connection factory.

Note that the indicated type is the Java programming language type of the resource manager connection factory, not the type of the connection.

The following example is the declaration of resource references used by the application component illustrated in the previous subsection.

```
...
```

```
<resource-ref>
   <description>
         A data source for the database in which
         the EmployeeService enterprise bean will
         record a log of all transactions.
   </description>
   <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
</resource-ref>
```

5.4.1.3 Standard resource manager connection factory types

The Application Component Provider must use the `javax.sql.DataSource` resource manager connection factory type for obtaining JDBC API connections.

The Application Component Provider must use the `javax.jms.QueueConnectionFactory` or the `javax.jms.TopicConnectionFactory` for obtaining JMS connections.

The Application Component Provider must use the `javax.mail.Session` resource manager connection factory type for obtaining JavaMail connections.

The Application Component Provider must use the `java.net.URL` resource manager connection factory type for obtaining URL connections.

It is recommended that the Application Component Provider name JDBC API data sources in the `java:comp/env/jdbc` subcontext, all JMS connection factories in the `java:comp/env/jms` subcontext, all JavaMail API connection factories in the `java:comp/env/mail` subcontext, and all URL connection factories in the `java:comp/env/url` subcontext.

The J2EE Connector Extension allows an application component to use the API described in this section to obtain resource objects that provide access to additional back-end systems.

5.4.2 Deployer's Responsibilities

The Deployer uses deployment tools to bind the resource manager connection factory references to the actual resource factories configured in the target operational environment.

The Deployer must perform the following tasks for each resource manager connection factory reference declared in the deployment descriptor:

- Bind the resource manager connection factory reference to a resource manager connection factory that exists in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the resource manager connection factory. The resource manager connection factory type must be compatible with the type declared in the `res-type` element.
- Provide any additional configuration information that the resource manager needs for opening and managing the resource. The configuration mechanism is resource manager specific, and is beyond the scope of this specification.
- If the value of the `res-auth` element is `Container`, the Deployer is responsible for configuring the signon information for the resource manager . This is performed in a manner specific to the container and resource manager; it is beyond the scope of this specification.

For example, if principals must be mapped from the security domain and principal realm used at the application component level to the security domain and principal realm of the resource manager, the Deployer or System Administrator must define the mapping. The mapping is performed in a manner specific to the container and resource manager; it is beyond the scope of this specification.

5.4.3 J2EE Product Provider's Responsibilities

The J2EE Product Provider is responsible for the following:

- Provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection.
- Provide the implementation of the resource manager connection factory classes that are required by this specification.
- If the Application Component Provider set the `res-auth` of a resource reference to `Application`, the container must allow the application component to perform explicit programmatic signon using the resource manager's API.
- The container must provide tools that allow the Deployer to set up resource signon information for the resource manager references whose `res-auth` element is set to `Container`. The minimum requirement is that the Deployer must be able to specify the user / password information for each resource manager connection factory reference declared by the application component, and the container must be able to use the user / password combination for user authentication when obtaining a connection by invoking the resource manager connection factory.

Although not required by this specification, we expect that containers will support some form of a single signon mechanism that spans the application server and the resource managers. The container will allow the Deployer to set up the resources such that the principal can be propagated (directly or through principal mapping) to a resource manager, if required by the application.

While not required by this specification, most J2EE products will provide the following features:

- A tool to allow the System Administrator to add, remove, and configure a resource manager for the J2EE Server.
- A mechanism to pool resources for the application components and otherwise manage the use of resources by the container. The pooling must be transparent to the application components.

5.4.4 System Administrator's Responsibilities

The System Administrator is typically responsible for the following:

- Add, remove, and configure resource managers in the J2EE Server environment.

In some scenarios, these tasks can be performed by the Deployer.

5.5 Resource Environment References

This section describes the programming and deployment descriptor interfaces that allow the Application Component Provider to refer to administered objects that are associated with resource (for example, JMS Destinations) by using "logical" names called resource environment references. The resource environment references are special entries in the application component's environment. The Deployer binds the resource environment references to administered objects in the target operational environment.

5.5.1 Application Component Provider's Responsibilities

This subsection describes the Application Component Provider's view and responsibilities with respect to resource environment references.

5.5.1.1 Resource environment reference programming interfaces

The Application Component Provider must use resource environment references to locate administered objects, such as JMS Destinations, that are assoicated with resources as follows.

- Assign an entry in the application component's environment to the reference. (See subsection 5.5.1.2 for information on how resource environment references are declared in the deployment descriptor.)
- This specification recommends, but does not require, that all resource environment references be organized in the appropriate subcontext of the component's environment for the resource type (e.g., in the `java:comp/env/jms` JNDI context for JMS Destinations).
- Look up the administered object in the application component's environment using JNDI.

The following example illustrates how an application component uses a resource environment reference to locate a JMS Destination .

```
// Obtain the default initial JNDI context.
Context initCtx = new InitialContext();

// Look up the JMS StockQueue in the environment.
Object result = initCtx.lookup(
   "java:comp/env/jms/StockQueue");

// Convert the result to the proper type.
javax.jms.Queue queue = (javax.jms.Queue)result;
```

In the example, the Application Component Provider assigned the environment entry `jms/StockQueue` as the resource environment reference name to refer to a JMS queue.

5.5.1.2 Declaration of resource environment references in deployment descriptor

Although the resource environment reference is an entry in the application component's environment, the Application Component Provider must not use a `env-entry` element to declare it. Instead, the Application Component Provider must declare all references to administered objects associated with resources using the `resource-env-ref` elements of the deployment descriptor. This allows the application component's jar file consumer to discover all the resource environment references used by the application component.

Each `resource-env-ref` element describes the requirements that the referencing application component has for the referenced administered object. The `resource-env-ref` element contains an optional `description` element; and the mandatory `resource-env-ref-name` and `resource-env-ref-type` elements.

The `resource-env-ref-name` element specifies the resource environment reference name; its value is the environment entry name used in the application component code. The name of the environment entry is relative to the `java:comp/env` context (e.g., the name should be `jms/StockQueue` rather than `java:comp/env/jms/StockQueue`). The `resource-env-ref-type` element specifies the expected type of the referenced object. For example, in the case of a JMS Destination, its value must be either `javax.jms.Queue` or `javax.jms.Topic`.

A resource environment reference is scoped to the application component whose declaration contains the `resource-env-ref` element. This means that the resource environment reference is not accessible to other application components at runtime, and that other application components may define `resource-env-ref` elements with the same `resource-env-ref-name` without causing a name conflict.

The following example illustrates the declaration of resource environment references in the deployment descriptor.

```
...
<resource-env-ref>
   <description>
         This is a reference to a JMS queue used in the
         processing of Stock info
   </description>
   <resource-env-ref-name>
         jms/StockInfo
   </resource-env-ref-name>
   <resource-env-ref-type>
         javax.jms.Queue
   </resource-env-ref-type>
</resource-env-ref>
...
```

5.5.2 Deployer's Responsibilities

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared resource environment references are bound to administered objects that exist in the operational environment. The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the target object.
- The Deployer must ensure that the target object is type-compatible with the type declared for the resource environment reference. This means that the target object must be of the type indicated in the `resource-env-ref-type` element.

5.5.3 J2EE Product Provider's Responsibilities

The J2EE Product Provider must provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection. The deployment tools provided by the J2EE Product Provider must be able to process the information supplied in the `resource-env-ref` elements in the deployment descriptor.

At the minimum, the tools must be able to inform the Deployer of any unresolved resource environment references, and allow him or her to resolve a resource environment reference by binding it to a specified compatible target object in the environment.

5.6 UserTransaction References

Certain J2EE application component types are allowed to use the JTA `UserTransaction` interface to start, commit, and abort transactions. Such application components can find an appropriate object implementing the `UserTransaction` interface by looking up the JNDI name `java:comp/UserTransaction`. The container is only required to provide `java:comp/UserTransaction` for those components that can validly make use of it. Any such `UserTransaction` object is only valid within the component instance that performed the lookup. See the individual component definitions for further information.

The following example illustrates how an application component acquires and uses a `UserTransaction` object.

```
public void updateData(...) {
   ...

   // Obtain the default initial JNDI context.
   Context initCtx = new InitialContext();

   // Look up the UserTransaction object.
   UserTransaction tx = (UserTransaction)initCtx.lookup(
      "java:comp/UserTransaction");

   // Start a transaction.
   tx.begin();
   ...
   // Perform transactional operations on data.
   ...
   // Commit the transaction.
   tx.commit();
   ...
}
```

5.6.1 Application Component Provider's Responsibilities

The Application Component Provider is responsible for using the defined name to lookup the `UserTransaction` object. Only some application component types are required to have access to a `UserTransaction` object; see TABLE 6-1 in this specification and the EJB specification for details.

5.6.2 Deployer's Responsibilities

The Deployer has no specific responsibilities associated with the `UserTransaction` object.

5.6.3 J2EE Product Provider's Responsibilities

The J2EE Product Provider is responsible for providing an appropriate `UserTransaction` object as required by this specification.

5.6.4 System Administrator's Responsibilities

The System Administrator has no specific responsibilities associated with the `UserTransaction` object.

CHAPTER 6

Application Programming Interface

The Java™ 2 Platform, Enterprise Edition (J2EE) provides a number of APIs for use by J2EE applications, starting with the core Java APIs and including several Java optional packages[1]. This chapter describes the requirements for those APIs.

6.1 Required APIs

J2EE application components execute in runtime environments provided by the containers that are a part of the J2EE platform. The J2EE platform supports four separate types of containers, one for each J2EE application component type - application client containers, applet containers, web containers for servlets and JSP pages, and enterprise bean containers. The containers provide all application components with the Java 2 Platform, Standard Edition, v1.3 (J2SE) APIs, which include the following enterprise APIs:

- JavaIDL API
- JDBC Core API
- RMI-IIOP API
- JNDI API

The containers for all application component types must provide Java Compatible™ runtime environments. In particular, a J2EE product must provide an applet execution environment that is J2SE 1.3 compatible. Since typical browsers don't yet provide such support, the J2EE platform may make use of the Java Plugin to provide the required applet execution environment. Use of the Java Plugin is not required, but a J2EE product **is** required to provide a J2SE 1.3 compatible applet execution environment.

1. Note that "optional packages" were previously called "standard extensions"; they are optional relative to J2SE, but the optional packages described in this specification are **required** for J2EE.

The specifications for the J2SE components are available at `http://java.sun.com/products/jdk/1.3/docs/`.

The J2EE platform also includes a number of Java optional packages. TABLE 6-1 indicates which optional packagesare required to be available in each type of container. TABLE 6-1 also indicates the required version of the optional package.

TABLE 6-1 J2EE-Required Java Optional Packages

Optional Package	app client	applet	web	EJB
JDBC 2.0 Extension	Y	N	Y	Y
EJB 2.0	Y[1]	N	Y[2]	Y
Servlets 2.3	N	N	Y	N
JSP 1.2	N	N	Y	N
JMS 1.0	Y	N	Y	Y
JTA 1.0	N	N	Y	Y
JavaMail 1.2	N	N	Y	Y
JAF 1.0	N	N	Y	Y
JAXP 1.1	Y	N	Y	Y
Connector 1.0	N	N	Y	Y
JAAS 1.0	Y	N	Y	Y

1. Application clients can only make use of the enterprise bean client APIs.
2. Servlets and JSP pages can only make use of the enterprise bean client APIs.

The APIs included in the J2EE platform must be included in their entirety. Definitions for all classes and interfaces required by the specifications must be included. Some of the APIs include interfaces that are intended to be implemented by an application server. In some cases a J2EE product is not required to provide objects that implement such interfaces. Nonetheless, the definitions of such interfaces must be included in the J2EE platform.

6.2 Java 2 Platform, Standard Edition (J2SE) Requirements

6.2.1 Programming Restrictions

The J2EE programming model splits the responsibilities between the Application Component Providers and the J2EE Product Provider. As a result of this split, the Application Component Providers focus on writing business logic and the J2EE Product Providers focus on providing a managed system infrastructure in which the application components can be deployed. This division of responsibilities requires that the application components do not contain functionality that would clash with the functions provided by the J2EE platform. If an application component tried to provide a function that the J2EE platform implements, the J2EE platform could not properly manage the function.

Thread management is one example of functionality that would clash with the J2EE platform's function. If enterprise beans were allowed to manage threads, the J2EE platform could not manage the life cycle of the enterprise beans, and it could not properly manage transactions.

This means that the application components must not use certain Java 2 Platform, Standard Edition (J2SE) functions. Because we do not want to subset the J2SE platform, and we want a J2SE product to be usable without modification by the J2EE Product Providers in the J2EE platform, we use the J2SE security permissions mechanism to express the programming restrictions imposed on Application Component Providers. We specify the J2SE security permissions that the J2EE Product Provider must provide for the execution of each application component type. We call these permissions the J2EE security permissions set. The J2EE security permissions set is part of the J2EE API contract.

Since the exact set of security permissions in use in any installation is a matter of policy, this specification does not define a fixed set of permissions. Instead, the J2EE security permissions set defines the minimum set of permissions that application components should expect. Application components that need permissions not in this minimal set should describe their requirements in their documentation. (A future version of this specification will allow these security requirements to be specified in the deployment descriptor for application components.) All J2EE products must be capable of deploying application

components that require the set of permissions described here. Some J2EE products will allow the set of permissions available to a component to be configurable, providing some components with more or fewer permissions than those described here.

From the Application Component Provider's perspective, the provider must ensure that the application components do not use functions that would conflict with the J2EE security permission set.

The J2SE security permissions are fully described in `http://java.sun.com/products/jdk/1.3/docs/guide/security/permissions.html`.

TABLE 6-2 lists the J2EE security permissions set. This is the typical set of permissions that components of each type should expect to have.

TABLE 6-2 J2EE Security Permissions Set

Security Permissions	Target	Action
Application Clients		
`java.awt.AWTPermission`	accessClipboard	
`java.awt.AWTPermission`	accessEventQueue	
`java.awt.AWTPermission`	showWindowWithout WarningBanner	
`java.lang.RuntimePermission`	exitVM	
`java.lang.RuntimePermission`	loadLibrary	
`java.lang.RuntimePermission`	queuePrintJob	
`java.net.SocketPermission`	*	connect
`java.net.SocketPermission`	localhost:1024-	accept,listen
`java.io.FilePermission`	*	read, write
`java.util.PropertyPermission`	*	read
Applet Clients		
`java.net.SocketPermission`	*codebase*	connect
`java.util.PropertyPermission`	*limited*	read
Servlets/JSPs		
`java.lang.RuntimePermission`	loadLibrary	
`java.lang.RuntimePermission`	queuePrintJob	

TABLE 6-2 J2EE Security Permissions Set

Security Permissions	Target	Action
`java.net.SocketPermission`	*	connect
`java.io.FilePermission`	*	read, write
`java.util.PropertyPermission`	*	read
EJB Components		
`java.lang.RuntimePermission`	queuePrintJob	
`java.net.SocketPermission`	*	connect
`java.util.PropertyPermission`	*	read

Note that an operating system that hosts a J2EE product may impose additional security restrictions of its own that must be taken into account. For instance, the user identity under which a servlet executes is not likely to have permission to read and write all files.

6.2.2 Additional Requirements

6.2.2.1 Networking

The J2SE platform includes a pluggable mechanism for supporting multiple URL protocols through the `java.net.URLStreamHandler` class and `java.net.URLStreamHandlerFactory` interface.

The following URL protocols must be supported:

- file

  Only reading from a file URL need be supported, that is, the corresponding `URLConnection` object's `getOutputStream` method may fail with an `UnknownServiceException`. Of course, file access is restricted according to the permissions described above.

- http

  Only version 1.0 of the HTTP protocol need be supported, although HTTP 1.1 is allowed. An http URL must support both input and output.

- https

  SSL version 3.0 must be supported by https URL objects. Both input and output must be supported.

The J2SE platform also includes a mechanism for converting a URL's byte stream to an appropriate object, using the `java.net.ContentHandler` class and `java.net.ContentHandlerFactory` interface. A `ContentHandler` object converts from a MIME byte stream to an object. `ContentHandler` objects are typically accessed indirectly using the `getContent` method of `URL` and `URLConnection`.

When accessing data of the following MIME types using the `getContent` method, objects of the corresponding Java type listed in TABLE 6-3 must be returned.

TABLE 6-3 Java Type of Objects Returned When Using the `getContent` Method

MIME Type	Java Type
image/gif	java.awt.Image
image/jpeg	java.awt.Image

Many environments will use HTTP proxies rather than connecting directly to HTTP servers. If HTTP proxies are being used in the local environment, the HTTP support in the J2SE platform should be configured to use the proxy appropriately; application components must not be required to configure proxy support in order to use an `http` URL.

Most enterprise environments will include a firewall that limits access from the internal network (intranet) to the public Internet, and vice versa. While it is typical for access using the HTTP protocol to pass through such firewalls, perhaps by using proxy servers, it is not typical that general TCP/IP traffic, including RMI-JRMP, RMI-IIOP, etc., can pass through firewalls. This of course has implications on the use of various protocols to communicate between application components. This specification requires only that, where local policy allows, HTTP access through firewalls be possible. Some J2EE products may provide support for tunneling other communication through firewalls, perhaps using HTTP, but this is neither specified nor required.

6.2.2.2 AWT

AWT provides the ability to read binary image data and convert it into a `java.awt.image` object, using the `createImage` methods in `java.awt.Toolkit`. The AWT Toolkit must support binary data in the GIF and JPEG formats.

6.2.2.3 JDBC™ API

The JDBC API allows for access to a wide range of data storage systems. The J2SE platform does not require that a system meeting the Java Compatible™ quality standards provide a database that is accessible through the JDBC API. To allow for the development of portable applications, this specification does require that such a database be available and accessible from a J2EE product through the JDBC API. Such a database must be accessible from web components, enterprise beans, and application clients, but need not be accessible from applets. In addition, the driver for the database must meet the JDBC Compatible requirements in the JDBC specification.

J2EE applications should not attempt to load JDBC drivers directly. Instead, they should use the technique recommended in the JDBC specification and perform a JNDI lookup to locate a `DataSource` object. The JNDI name of the `DataSource` object should be chosen as described in Section 5.4, “Resource Manager Connection Factory References”. The J2EE platform must be able to supply a `DataSource` that does not require the application to supply any authentication information when obtaining a database connection. Of course, applications may also supply a user name and password explicitly when connecting to the database.

When a JDBC API connection is used in an enterprise bean, the transaction characteristics will typically be controlled by the container. The component should not attempt to change the transaction characteristics of the connection, commit the transaction, rollback the transaction, or set autocommit mode. Attempts to make changes that are incompatible with the current transaction context may result in a `SQLException` being thrown. The EJB specification contains the precise rules for enterprise beans.

Similar restrictions apply when a component creates a transaction using the JTA `UserTransaction` interface. The component should not attempt operations on the JDBC `Connection` object that would conflict with the transaction context.

Drivers supporting the JDBC API used in a J2EE environment must meet a number of additional requirements on their implementation of JDBC APIs, described below.

- Drivers are required to provide accurate and complete metadata through the `Connection.getMetaData` method. J2EE applications should examine the `DatabaseMetaData` object and adapt their behavior to the capabilities of the current database. How this information is used to create portable applications that are independent of the underlying database vendor and driver is beyond the scope of this specification.

- Drivers must support stored procedures (`DatabaseMetaData.supportsStoredProcedures` must return `true`). The driver must also support the full JDBC API escape syntax for calling stored procedures with the following methods on the `Statement`, `PreparedStatement`, and `CallableStatement` classes:
  - `executeUpdate`
  - `executeQuery`

  Support for calling stored procedures using the method `execute` on the `Statement`, `PreparedStatement`, and `CallableStatement` interfaces is not required because some databases don't support returning more than a single `ResultSet` from a stored procedure.
- Drivers must support all of the `CallableStatement` methods that apply to SQL92 types, including the following:
  - `getBigDecimal(int parameterIndex)`
  - `getBoolean`
  - `getByte`
  - `getBytes`
  - `getDate(int parameterIndex)`
  - `getDate(int parameterIndex, Calendar cal)`
  - `getDouble(int parameterIndex)`
  - `getFloat(int parameterIndex)`
  - `getInt`
  - `getLong`
  - `getObject(int parameterIndex)`
  - `getShort`
  - `getString`
  - `getTime(int parameterIndex)`
  - `getTime(int parameterIndex, Calendar cal)`
  - `getTimestamp(int parameterIndex)`
  - `getTimestamp(int parameterIndex, Calendar cal)`
  - `registerOutParameter(int parameterIndex, int sqlType)`
  - `registerOutParameter(int parameterIndex, int sqlType, int scale)`
  - `wasNull()`

  Support for the new `BLOB`, `CLOB`, `ARRAY`, `REF`, `STRUCT` and `JAVA_OBJECT` types is not required. All parameter types (`IN`, `OUT`, and `INOUT`) must be supported.
- Full support for `PreparedStatements` is required. This implies support for the following methods:
  - `setAsciiStream`
  - `setBigDecimal`

- setBinaryStream(int parameterIndex, InputStream x, int length)
- setBoolean
- setByte
- setBytes
- setCharacterStream
- setDate(int parameterIndex, Date x)
- setDate(int parameterIndex, Date x, Calendar cal)
- setDouble
- setFloat
- setInt
- setLong
- setNull
- setObject(int parameterIndex, Object x)
- setObject(int parameterIndex, Object x, int targetSqlType)
- setObject(int parameterIndex, Object x, int targetSqlType, int scale)
- setShort
- setString
- setTime(int parameterIndex, Time x)
- setTime(int parameterIndex, Time x, Calendar cal)
- setTimestamp(int parameterIndex, Timestamp x)
- setTimestamp(int parameterIndex, Timestamp x, Calendar cal)

Support for the new BLOB, CLOB, ARRAY, REF, STRUCT and JAVA_OBJECT types is not required. Support for the PreparedStatement method getMetaData is not required. This method must throw an SQLException if it is not supported.

- Full support for batch updates is required. This implies support for the following methods on the Statement, PreparedStatement, and CallableStatement classes:
  - addBatch
  - clearBatch
  - executeBatch

  Drivers are free to implement these methods any way they choose (including a non-batching implementation) as long as the semantics are correct.
- A driver must provide full support for DatabaseMetaData and ResultSetMetaData. This implies that all of the methods in the DatabaseMetaData interface must be implemented and must behave as

specified in the JDBC 2.1 specification. None of the methods in `DatabaseMetaData` and `ResultSetMetaData` may throw an exception because they are not implemented.

- The JDBC API core specification requires that JDBC compliant drivers provide support for the SQL92, Transitional Level, `DROP TABLE` command, full support for the `CASCADE` and `RESTRICT` options is required. As many popular databases do not support `DROP TABLE` as specified in the SQL92 specification, the following clarification is required.

  A JDBC 2.1 compliant driver is required to support the `DROP TABLE` command as specified by the SQL92, Transitional Level. However, support for the `CASCADE` and `RESTRICT` options of `DROP TABLE` is optional. In addition, the behavior of `DROP TABLE` is implementation defined when there are views or integrity constraints defined that reference the table that is being dropped.

- A driver must support the `Statement` escape syntax for the following functions as specified by the JDBC 2.1 specification:
  - `CONCAT`
  - `SUBSTRING`
  - `LOCATE`
  - `LENGTH`
  - `ABS`
  - `SQRT`

6.2.2.4 Java™IDL

JavaIDL allows applications to access any CORBA object, written in any language, using the standard IIOP protocol. The J2EE security restrictions typically prevent all application component types except application clients from creating and exporting a CORBA object, but all J2EE application component types can be clients of CORBA objects.

A J2EE product must support JavaIDL as defined by chapters 1 - 8, 13, and 15 of the CORBA 2.3.1 specification, available at `http://cgi.omg.org/cgi-bin/doc?formal/99-10-07`, and the IDL To Java Language Mapping Specification, available at `http://cgi.omg.org/cgi-bin/doc?ptc/2000-01-08`.

J2EE applications need to use an instance of `org.omg.CORBA.ORB` to perform many JavaIDL and RMI-IIOP operations. The default ORB returned by a call to `ORB.init(new String[0], null)` must be usable for such purposes; an application need not be aware of the implementation classes used for the ORB and RMI-IIOP support.

In addition, for performance reasons it is often advantageous to share an ORB instance among components in an application. To support such usage, all web and enterprise bean containers are required to provide an ORB instance in the JNDI namespace under the name `java:comp/ORB`. The container is allowed, but not required, to share this instance between components. The container may also use this ORB instance itself. To support isolation between applications, an ORB instance should not be shared between components in different applications. To allow this ORB instance to be safely shared between components, components must restrict their usage of certain ORB APIs and functionality:

- Do not call the ORB `shutdown` method.
- Do not call the `org.omg.CORBA_2_3.ORB` methods `register_value_factory` and `unregister_value_factory` with an `id` used by the container.

A J2EE product must provide a COSNaming service to support the EJB interoperability requirements. It must be possible to access this COSNaming service using the JavaIDL COSNaming APIs. Applications with appropriate privileges must be able to lookup objects in the COSNaming service. COSNaming is defined in the Interoperable Naming Service specification, available at `http://cgi.omg.org/cgi-bin/doc?formal/2000-06-19`.

6.2.2.5 RMI-JRMP

JRMP is the Java technology-specific Remote Method Invocation (RMI) protocol. The J2EE security restrictions typically prevent all application component types except application clients from creating and exporting an RMI object, but all J2EE application component types can be clients of RMI objects.

6.2.2.6 RMI-IIOP

RMI-IIOP allows objects defined using RMI style interfaces to be accessed using the IIOP protocol. It must be possible to make any enterprise bean accessible via RMI-IIOP. Some J2EE products will simply make all enterprise beans always (and only) accessible via RMI-IIOP; other products might control this via an administration or deployment action. These and other approaches are allowed, provided that any enterprise bean (or by extension, all enterprise beans) can be made accessible using RMI-IIOP.

All components accessing enterprise beans must use the `narrow` method of the `javax.rmi.PortableRemoteObject` class, as described in the EJB specification. Because enterprise beans may be deployed using other RMI protocols, portable applications must not depend on the characteristics of RMI-IIOP objects (e.g., the use of the `Stub` and `Tie` base classes) beyond what is specified in the EJB specification.

The J2EE security restrictions typically prevent all application component types, except application clients, from creating and exporting an RMI-IIOP object. All J2EE application component types can be clients of RMI-IIOP objects. J2EE applications should also use JNDI to lookup non-EJB RMI-IIOP objects. The JNDI names used for such non-EJB RMI-IIOP objects should be configured at deployment time using the standard environment entries mechanism (see Section 5.2, "Java Naming and Directory Interface™ (JNDI) Naming Context"). The application should fetch the name from JNDI using an environment entry and then use that name to lookup the RMI-IIOP object. Typically such names will be configured to be names in the COSNaming name service.

This specification does not provide a portable way for applications to bind objects to names in a name service. Some products may support use of JNDI and COSNaming for binding objects, but this is not required. Portable J2EE application clients can create non-EJB RMI-IIOP server objects for use as callback objects or to pass in calls to other RMI-IIOP objects.

Note that while RMI-IIOP alone doesn't specify how to propagate the current security context or transaction context, the EJB interoperability specification does define such context propagation. This specification only requires that the use of RMI-IIOP to access enterprise beans be able to propagate context information (as defined in the EJB specification); uses of RMI-IIOP to access objects other than enterprise beans may or may not propagate context information.

The RMI-IIOP specification describes how portable `Stub` and `Tie` classes can be created. A J2EE application that defines or uses RMI-IIOP objects other than enterprise beans must include such portable `Stub` and `Tie` classes in the application package. `Stub` and `Tie` objects for enterprise beans must not be included with the application; they will be generated, if needed, by the J2EE product at deployment time or at run time.

RMI-IIOP is defined by chapters 5, 6, 13, 15, and section 10.6.2 of the CORBA 2.3.1 specification, available at `http://cgi.omg.org/cgi-bin/doc?formal/99-10-07`, and by the *Java™ Language To IDL Mapping Specification*, available at `http://cgi.omg.org/cgi-bin/doc?ptc/2000-01-06`.

6.2.2.7 JNDI

A J2EE product must make the following types of objects available in the application visible JNDI namespace - `EJBHome` objects, JTA `UserTransaction` objects, JDBC API `DataSource` objects, JMS `ConnectionFactory` and `Destination` objects, JavaMail `Session` objects, and resource manager connection factory objects (as specified in the Connector specification). The JNDI implementation in a J2EE product must be capable of supporting all of these uses in a single application component using a single JNDI `InitialContext`. Application components will generally create a JNDI `InitialContext` using the default constructor with no arguments. The application component may then perform lookups on that `InitialContext` to find objects as specified above.

The names used to perform lookups for J2EE objects are application-dependent; the application component's deployment descriptor lists all of the names and the type of object expected to correspond to each name. The Deployer configures the JNDI namespace to make appropriate components available. The JNDI names used to lookup such objects must be in the JNDI `java:` namespace. See Chapter 5, "Naming" for details.

One particular name is defined by this specification. For all application components that have access to the JTA `UserTransaction` interface, the appropriate `UserTransaction` object can be found using the name `java:comp/UserTransaction`.

The name used to lookup a particular J2EE object may be different in different application components. In general, JNDI names can not be meaningfully passed as arguments in remote calls from one application component to another remote component (for example, in a call to an enterprise bean).

The JNDI `java:` namespace is commonly implemented as *symbolic links* to other naming systems. Different underlying naming services may be used to store different kinds of objects, or even different instances of objects. It is up to a J2EE product to provide the JNDI service providers necessary to access the various J2EE-defined objects.

Different JNDI service providers may provide different capabilities, for instance, some service providers may provide only read-only access to the data in the name service. This specification requires that the J2EE platform provide the ability to perform lookup operations as described above.

All J2EE products must provide a COSNaming name service to meet the EJB interoperability requirements. In addition, a COSNaming JNDI service provider must be available in the web, EJB, and application client containers. It will also typically be available in the applet container, but this is not required. (A COSNaming JNDI service provider is a part of the J2SE 1.3 SDK and JRE from

Sun, but is not a required component of the J2SE specification.) The COSNaming JNDI service provider specification is available at `http://java.sun.com/j2se/1.3/docs/guide/jndi/jndi-cos.html`.

See Chapter 5, “Naming” for the complete naming requirements for the J2EE platform.

The JNDI specification is available at `http://java.sun.com/products/jndi/docs.html`.

6.3 JDBC™ 2.0 Extension Requirements

The JDBC 2.0 extension includes APIs for row sets, connection naming via JNDI, connection pooling, and distributed transaction support. The connection pooling and distributed transaction features are intended for use by JDBC drivers to coordinate with an application server. J2EE products are not required to support the application server facilities described by these APIs, although they may prove useful. The Connector architecture defines an SPI that essentially extends the functionality of the JDBC SPI with additional security functionality as well as providing a full packaging and deployment functionality for resource adapters. A future version of this specification may require support for deploying JDBC drivers as resource adapters using the Connector architecture.

The JDBC 2.0 extension specification is available at `http://java.sun.com/products/jdbc/jdbcse2.html`.

6.4 Enterprise JavaBeans™ (EJB) 2.0 Requirements

This specification requires that a J2EE product provide support for enterprise beans as specified in the EJB 2.0 specification. The EJB specification is available at `http://java.sun.com/products/ejb/docs.html`.

This specification does not impose any additional requirements at this time. Note that the EJB 2.0 specification includes the specification of the EJB interoperability protocol based on RMI-IIOP. All containers that support EJB clients must be capable of using the EJB interoperability protocol to invoke

enterprise beans. All EJB containers must support the invocation of enterprise beans in the container using the EJB interoperability protocol. A J2EE product may also support other protocols for the invocation of enterprise beans.

A J2EE product may support multiple object systems (e.g., RMI-IIOP, RMI-JRMP, etc.). It may not always be possible to pass object references from one object system to objects in another object system. However, when an enterprise bean is using the RMI-IIOP protocol, it must be possible to pass object references for RMI-IIOP or JavaIDL objects as arguments to methods on such an enterprise bean, and to return such object references as the return value of a method on such an enterprise bean. In addition, it must be possible to pass a reference to an RMI-IIOP-based enterprise bean's Home or Remote interface to a method on an RMI-IIOP or JavaIDL object, or to return such an enterprise bean object reference as a return value from such an RMI-IIOP or JavaIDL object.

6.5 Servlet 2.3 Requirements

The Servlet specification defines the packaging and deployment of web applications, standalone and as part of a J2EE application. The Servlet specification also addresses security, both standalone and within the J2EE platform. A J2EE product must support these optional components of the Servlet specification. The Servlet specification includes additional requirements on web containers that are part of a J2EE product; a J2EE product must meet these requirements as well.

The Servlet specification defines *distributable* web applications. To support J2EE applications that are distributable, this specification adds the following requirements.

A distributable application may only place objects of the following types into a `javax.servlet.http.HttpSession` object using the `setAttribute` or `putValue` methods:

- `java.io.Serializable`
- `javax.ejb.EJBObject`
- `javax.ejb.EJBHome`
- `javax.transaction.UserTransaction`
- a `javax.naming.Context` object for the `java:comp/env` context

Web containers may throw an `IllegalArgumentException` if an object that is not one of the above types is passed to the `setAttribute` or `putValue` methods of an `HttpSession` object corresponding to a distributable session.

This exception indicates to the programmer that the web container does not support moving the object between VMs. A web container that supports multi-VM operation must ensure that, when a session is moved from one VM to another, all objects of the above types are accurately recreated on the target VM.

The Servlet specification is available at `http://java.sun.com/products/servlet`.

6.6 JavaServer Pages™ (JSP) 1.2 Requirements

JSP depends on and builds on the Servlet framework. A J2EE product must support the entire JSP specification.

The JSP specification is available at `http://java.sun.com/products/jsp`.

6.7 Java™ Message Service (JMS) 1.0 Requirements

A Java Message Service provider must be included in a J2EE product. The JMS implementation must provide support for both JMS point-to-point and publish/subscribe messaging, and thus must make those facilities available using the `ConnectionFactory` and `Destination` APIs.

The JMS specification defines several interfaces intended for integration with an application server. A J2EE product need not provide objects that implement these interfaces, and portable J2EE applications must not use these interfaces:

- `javax.jms.ServerSession`
- `javax.jms.ServerSessionPool`
- `javax.jms.ConnectionConsumer`
- all `javax.jms` XA interfaces

Note that the JMS API creates threads to deliver messages to message listeners. The use of this message listener facility may be limited by the restrictions on the use of threads in various containers. In EJB containers, for instance, it is

typically not possible to create threads. The following methods must not be used by application components executing in containers that prevent them from creating threads:

- `javax.jms.Session` method `setMessageListener`
- `javax.jms.Session` method `getMessageListener`
- `javax.jms.Session` method `run`
- `javax.jms.QueueConnection` method `createConnectionConsumer`
- `javax.jms.TopicConnection` method `createConnectionConsumer`
- `javax.jms.TopicConnection` method `createDurableConnectionConsumer`

In addition, use of the following methods may interfere with the connection management functions of the container; applications in web and EJB containers must not use these methods on `javax.jms.Connection` objects:

- `setExceptionListener`
- `stop`
- `setClientID`

A J2EE container may throw a `JMSException` if the application component violates these restrictions.

The latest JMS 1.0 specification is version 1.0.2 and is available at `http://java.sun.com/products/jms`.

6.8 Java™ Transaction API (JTA) 1.0 Requirements

JTA defines the `UserTransaction` interface that is used by applications to start and commit or abort transactions. Enterprise beans are expected to get `UserTransaction` objects through the `EJBContext`'s `getUserTransaction` method. Other application components get a `UserTransaction` object through a JNDI lookup using the name `java:comp/UserTransaction`.

JTA also defines a number of interfaces that are used by an application server to communicate with a transaction manager, and for a transaction manager to interact with a resource manager. These interfaces must be supported as described in the Connector specification. In addition, support for other transaction facilities may be provided by a J2EE product, transparently to the application.

The latest JTA 1.0 specification is version 1.0.1 and is available at `http://java.sun.com/products/jta`.

6.9 JavaMail™ 1.2 Requirements

The JavaMail API allows for access to email messages contained in message stores, and for the creation and sending of email messages using a message transport. Specific support is included for Internet standard MIME messages. Access to message stores and transports is through protocol providers supporting specific store and transport protocols. The JavaMail API specification does not require any specific protocol providers, but the JavaMail reference implementation includes an IMAP message store provider and an SMTP message transport provider.

Configuration of the JavaMail API is typically done by setting properties in a `Properties` object that is used to create a `javax.mail.Session` object using a static factory method. To allow the J2EE platform to configure and manage JavaMail API sessions, an application component that uses the JavaMail API should request a `Session` object using JNDI, and should list its need for a `Session` object in its deployment descriptor using a `resource-ref` element. A JavaMail API `Session` object should be considered a resource factory, as described in Section 5.4, "Resource Manager Connection Factory References." This specification requires that the J2EE platform support `javax.mail.Session` objects as resource factories, as described in that section.

The J2EE platform requires that a message transport be provided that is capable of handling addresses of type `javax.mail.internet.InternetAddress` and messages of type `javax.mail.internet.MimeMessage`. The default message transport must be properly configured to send such messages using the `send` method of the `javax.mail.Transport` class. Any authentication needed by the default transport must be handled without need for the application to provide a `javax.mail.Authenticator` or to explicitly connect to the transport and supply authentication information.

This specification does not require that a J2EE product support any message store protocols.

Note that the JavaMail API creates threads to deliver notifications of `Store`, `Folder`, and `Transport` events. The use of these notification facilities may be limited by the restrictions on the use of threads in various containers. In EJB containers, for instance, it is typically not possible to create threads.

The JavaMail API uses the JavaBeans Activation Framework API to support various MIME data types. The JavaMail API must include `javax.activation.DataContentHandlers` for the following MIME data types, corresponding to the Java programming language type indicated in TABLE 6-4.

TABLE 6-4 JavaMail API MIME Data Type to Java Type Mappings

Mime Type	Java Type
text / plain	`java.lang.String`
multipart / *	`javax.mail.internet.MimeMultipart`
message / rfc822	`javax.mail.internet.MimeMessage`

The JavaMail API specification is available at `http://java.sun.com/products/javamail`.

6.10 JavaBeans™ Activation Framework 1.0 Requirements

The JavaBeans Activation Framework integrates support for MIME data types into the Java platform. MIME byte streams can be converted to and from Java programming language objects, using `javax.activation.DataContentHandler` objects. JavaBeans components can be specified for operating on MIME data, such as viewing or editing the data. The JavaBeans Activation Framework also provides a mechanism to map filename extensions to MIME types.

The JavaBeans Activation Framework is used by the JavaMail API to handle the data included in email messages; typical applications will not need to use the JavaBeans Activation Framework directly, although applications making sophisticated use of email may need it.

This specification requires that the J2EE platform need only provide the `DataContentHandlers` specified above for the JavaMail API. This specification requires the J2EE platform to provide a `javax.activation.MimetypesFileTypeMap` that supports the mappings listed in TABLE 6-5.

TABLE 6-5 Filename Extension to MIME Type Mappings

MIME Type	Filename Extensions
text/html	html htm
text/plain	txt text
image/gif	gif GIF
image/jpeg	jpeg jpg jpe JPG

The JavaBeans Activation Framework 1.0 specification is available at `http://java.sun.com/beans/glasgow/jaf.html`.

6.11 Java™ API for XML Parsing (JAXP) 1.1 Requirements

JAXP includes the industry standard SAX and DOM APIs, as well as a pluggability API that allows SAX and DOM parsers and XSLT transform engines to be plugged into the framework, and allows applications to find parsers that support the features needed by the application.

All J2EE products must meet the JAXP conformance requirements and must provide at least one SAX 2 parser, at least one DOM 2 parser, and at least one XSLT transform engine. There must be a SAX parser or parsers that support all combinations of validation modes and namespace support. There must be a DOM parser or parsers that support all combinations of validation modes namespace support.

The JAXP specification is available at `http://java.sun.com/xml`.

6.12 J2EE™ Connector Architecture 1.0 Requirements

All EJB containers and all web containers must support the Connector APIs. All such containers must support Resource Adapters that use any of the specified transaction capabilities. The J2EE deployment tools must support deployment of Resource Adapters, as defined in the Connector specification, and must support the deployment of applications that use Resource Adapters.

The Connector specification is available at `http://java.sun.com/j2ee/connector/`.

6.13 Java™ Authentication and Authorization Service (JAAS) 1.0 Requirements

All EJB containers and all web containers must support the use of the JAAS APIs as specified in the Connector specification. All application client containers must support use of the JAAS APIs as specified in Chapter 9, "Application Clients."

The JAAS specification is available at `http://java.sun.com/products/jaas`.

CHAPTER 7

Interoperability

This chapter describes the interoperability requirements for the Java™ 2 Platform, Enterprise Edition (J2EE).

7.1 Introduction to Interoperability

The J2EE platform will be used by enterprise environments that support clients of many different types. Often, these enterprise environments will add new services to their existing Enterprise Information Systems. These enterprise environments very likely will be using different hardware platforms and various software applications written in different languages.

Enterprise environments may leverage the J2EE platform to bring together applications written in such languages as C++ and Visual Basic. One or more of these existing applications may be running on a personal computer platform, while others may be running on Unix® workstations. In addition, these enterprise environments may also be supporting standalone Java technology-based applications that are not directly supported by the J2EE platform.

The J2EE platform provides indirect support for various types of clients, different hardware platforms, and a multitude of software applications through its interoperability requirements. To an Application Component Provider or System Administrator in an enterprise environment, the interoperability features of the J2EE platform permit the underlying disparate systems to work together seamlessly. In addition, the platform hides much of the complexity required to join these pieces.

The interoperability requirements for the current J2EE platform release allows:

- J2EE applications to connect to legacy systems using CORBA or low-level Socket interfaces.

- J2EE applications to connect to other J2EE applications across multiple J2EE products. The J2EE products can be from multiple Product Providers or they can be from the same Provider.

This specification allows J2EE applications to connect to EIS or legacy systems. At the current time, these connections may be accomplished using CORBA services or low-level Socket interfaces.

In addition, this specification requires that J2EE applications be allowed to connect to other J2EE applications. These other J2EE applications may be running on other J2EE platforms. J2EE applications must also be able to connect and work with J2EE applications written by other Application Providers. In this version of the specification, interoperability between J2EE applications running in different platforms is accomplished through the HTTP protocol, possibly using SSL, or the EJB interoperability protocol based on IIOP.

7.2 Interoperability Protocols

This specification requires that a J2EE product support a standard set of protocols and formats to ensure interoperability. The specification requires support for the following groups of protocols and formats:

- Internet protocols
- OMG protocols
- Java technology protocols
- Data formats

Many of these protocols and formats are supported by J2SE and by the underlying operating system.

7.2.1 Internet Protocols

Internet protocols define the standards by which the different pieces of the platform communicate with each other. The J2EE platform requires support for the following Internet protocols:

- TCP/IP protocol family—This is the core component of Internet communication. TCP/IP and UDP/IP are the standard transport protocols for the Internet.

- HTTP 1.0—This is the core protocol of Web communication. As with TCP/IP, HTTP 1.0 is supported by J2SE and the underlying operating system. A J2EE web container must be capable of advertising its HTTP services on the standard HTTP port, port 80.
- SSL 3.0, TLS 1.0—SSL 3.0 (Secure Socket Layer) represents the security layer for Web communication. It is available indirectly when using the `https` URL as opposed to the `http` URL. A J2EE web container must be capable of advertising its HTTPS service on the standard HTTPS port, port 443. SSL 3.0 and TLS 1.0 are also required as part of the EJB interoperability protocol, as defined in the EJB specification.

7.2.2 OMG Protocols

This specification requires the J2EE platform to support the following Object Management Group (OMG) based protocols:

- IIOP (Internet Inter-ORB Protocol)—Supported by Java IDL and RMI-IIOP in J2SE. Java IDL provides standards-based interoperability and connectivity through the Common Object Request Broker Architecture (CORBA). CORBA specifies the Object Request Broker (ORB) which allows applications to communicate with each other regardless of location. This interoperability is delivered through IIOP, and is typically found in an intranet setting. IIOP can be used as an RMI protocol using the RMI-IIOP technology. IIOP is defined in Chapters 13 and 15 of the CORBA 2.3.1 specification, available at `http://cgi.omg.org/cgi-bin/doc?formal/99-10-07`.
- EJB interoperability protocol—The EJB interoperability protocol is based on IIOP (GIOP 1.2) and the (draft) CSIv2 CORBA Secure Interoperability specification. The EJB interoperability protocol is defined in the EJB specification.
- COSNaming—The COSNaming protocol is an IIOP-based protocol to access a name service. The EJB interoperability protocol requires the use of the COSNaming protocol to lookup EJB objects using the JNDI API. In addition, it must be possible to use the JavaIDL COSNaming API to access the COSNaming name service. All J2EE products must provide a COSNaming name service that meets the requirements of the Interoperable Naming Service specification, available at `http://cgi.omg.org/cgi-bin/doc?formal/2000-06-19`. This name service may be provided as a separate name server or as a protocol bridge or gateway to another name service; this specification does not prescribe or preclude either approach.

7.2.3 Java Technology Protocols

This specification also requires the J2EE platform to support the JRMP protocol, which is the Java technology-specific Remote Method Invocation (RMI) protocol. JRMP is a required component of J2SE and is one of the two required RMI protocols. IIOP, also required by J2SE, is the other required RMI protocol, see above.

JRMP is a distributed object model for the Java programming language. Distributed systems, which run in different address spaces and often on different hosts, must still be able to communicate with each other. JRMP permits program-level objects in different address spaces to invoke remote objects using the semantics of the Java programming language object model.

Complete information on the JRMP specification can be found at `http://java.sun.com/products/jdk/1.2/docs/guide/rmi`.

7.2.4 Data Formats

In addition to the protocols that allow communication between components, this specification also requires J2EE platform support for a number of data formats. These formats provide a definition for data that is exchanged between components.

The following data formats must be supported:

- HTML 3.2—This represents the minimum web standard. It is not directly supported by J2EE APIs. However, it must be able to be displayed by J2EE web clients.
- Image file formats—The J2EE platform must support both GIF and JPEG images. Support for these formats is provided by the `java.awt.image` APIs (see the URL: `http://java.sun.com/products/jdk/1.2/docs/api/java/awt/image/package-summary.html`) and by J2EE web clients.
- JAR files—JAR (Java Archive) files are the standard packaging format for Java technology-based application components, including the ejb-jar specialized format, the Web application archive (war) format, the Resource Adapter archive (rar), and the J2EE enterprise application archive (ear) format. JAR is a platform-independent file format that permits many files to be aggregated into one file. This allows multiple Java components to be bundled into one JAR file and downloaded to a browser in a single HTTP transaction. JAR file formats are supported by the `java.util.jar` and `java.util.zip` packages. For complete information on the JAR specification, see the URL: `http://java.sun.com/products/jdk/1.2/docs/guide/jar`.

- Class file format—The class file format is specified in the Java Virtual Machine specification. Each class file contains one Java programming language type—either a class or an interface—and consists of a stream of 8-bit bytes. For complete information on the class file format, see the URL: `http://java.sun.com/docs/books/vmspec`.

CHAPTER 8

Application Assembly and Deployment

This chapter specifies Java™ 2 Platform, Enterprise Edition (J2EE) requirements for assembling, packaging, and deploying a J2EE application. The main goal of these requirements is to provide scalable modular application assembly and portable deployment of J2EE applications into any J2EE product.

J2EE applications are composed of one or more J2EE components and one J2EE application deployment descriptor. The deployment descriptor lists the application's components as modules. A J2EE module represents the basic unit of composition of a J2EE application. J2EE modules consist of one or more J2EE components and one component level deployment descriptor. The flexibility and extensibility of the J2EE component model facilitates the packaging and deployment of J2EE components as individual components, component libraries, or J2EE applications.

FIGURE 8-1 shows the composition model for J2EE deployment units and includes the optional usage of alternate deployment descriptors by the application package to preserve the signing of the original J2EE modules.

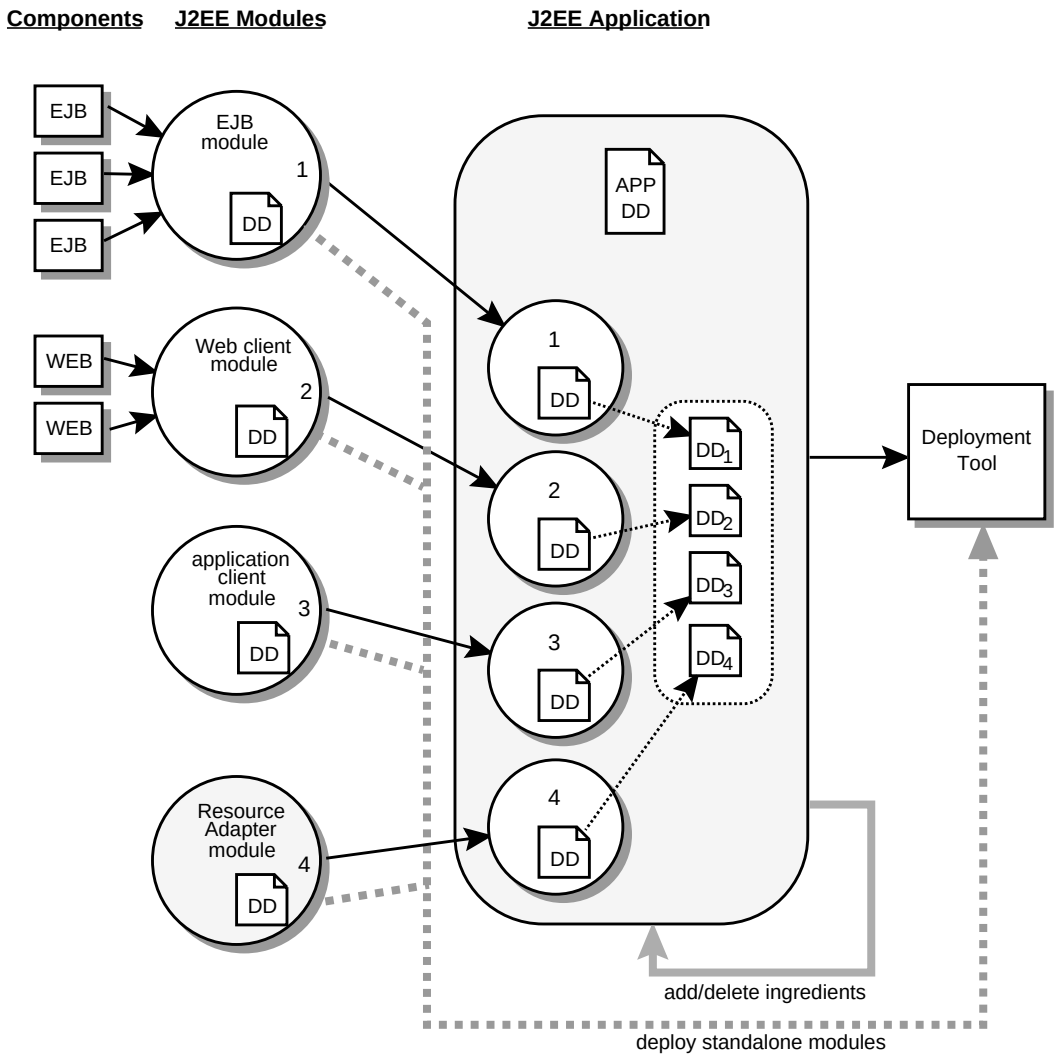

FIGURE 8-1 J2EE Deployment

8.1 Application Development Life Cycle

The development life cycle of a J2EE application begins with the creation of discrete J2EE components. These components are then packaged with a component level deployment descriptor to create a J2EE module. J2EE modules can be deployed as stand-alone units or can be assembled with a J2EE application deployment descriptor and deployed as a J2EE application.

FIGURE 8-2 shows the life cycle of a J2EE application.

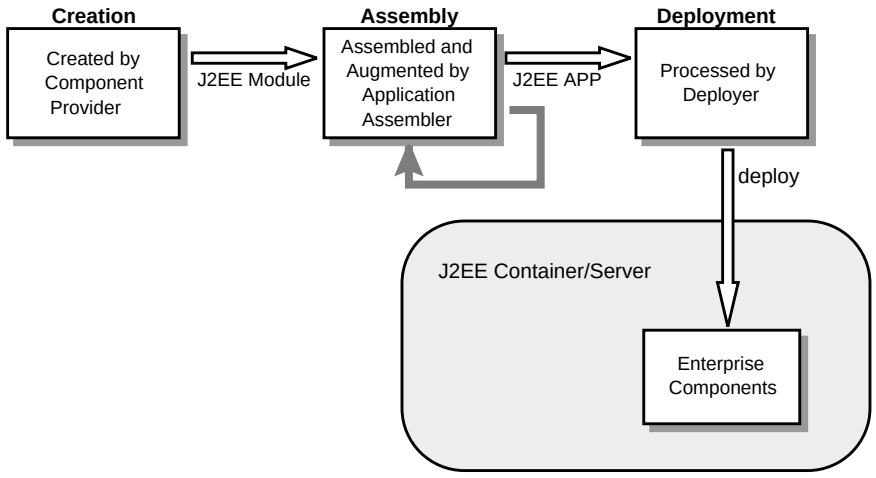

FIGURE 8-2 J2EE Application Life Cycle

8.1.1 Component Creation

The EJB, Servlet, application client, and Connector specifications include the XML document type definition (DTD) of the associated component level deployment descriptors and component packaging architecture required to produce J2EE modules. (The application client specification is found in Chapter 9 of this document.)

8.1.2 Component Packaging: Composing a J2EE module

A J2EE module is a collection of one or more J2EE components of the same component type (web, EJB, application client, or Connector) with one component deployment descriptor of that type. Deployment descriptors for J2EE modules are extensible. Any number of components of the same container type can be packaged together with a single a container-specific J2EE deployment descriptor to produce a J2EE module.

- A J2EE module represents the basic unit of composition of a J2EE application.
- The deployment descriptor for a J2EE module contains all of the declarative data required to deploy the components in the module. The deployment descriptor for a J2EE module also contains assembly instructions that describe how the components are composed into an application.
- An individual J2EE module can be deployed as a stand-alone J2EE module without an application level deployment descriptor.
- The J2EE platform supports the use of bundled optional packages as specified in *Extension Mechanism Architecture* (available at `http://java.sun.com/products/jdk/1.3/docs/guide/extensions/spec.html`). Using this mechanism a J2EE `.jar` file can reference utility classes or other shared classes or resources packaged in a separate `.jar` file and included in the same J2EE application package. (The use of this mechanism to reference classes or resources that are not in `.jar` files need not be supported.)

  A `.jar` file can reference another `.jar` file by naming the referenced `.jar` file in a `Class-Path` header in the referencing `.jar` file's Manifest file. The referenced `.jar` file is named as a relative URL, relative to the URL of the referencing `.jar` file. (This mechanism is only applicable to JAR format files containing class files or resources to be loaded by a `ClassLoader`; such files are always named with a `.jar` extension. This mechanism does not apply to other JAR format files such as `.ear` files, `.war` files, and `.rar` files that are never loaded directly by a `ClassLoader`.)

  The J2EE deployment tools must process all such referenced files when processing a J2EE module that is or contains a `.jar` file. Any deployment descriptors in referenced `.jar` files are ignored when processing the referencing `.jar` file. The deployment tool must install the `.jar` files in a way that preserves the relative references between `.jar` files, typically by installing the `.jar` files into a directory hierarchy that matches the original application directory hierarchy. All referenced `.jar` files must appear in the logical class path of the referencing `.jar` files at runtime.

8.1.3 Application Assembly

A J2EE application consists of one or more J2EE modules and one J2EE application deployment descriptor. A J2EE application is packaged using the Java Archive (JAR) file format into a file with a `.ear` (Enterprise ARchive) filename extension. A minimal J2EE application package will only contain J2EE modules and the application deployment descriptor. A J2EE application package may also include libraries referenced by J2EE modules (using the `Class-Path` mechanism described in the bundled extensions specification referenced above), help files and documentation to aid the deployer, etc.

The deployment of a portable J2EE application should not depend on any entities that may be contained in the package other than those defined by this specification. Deployment of a portable J2EE application must be possible using only the application deployment descriptor and the J2EE modules (and their dependent libraries) and descriptors listed in it.

The J2EE application deployment descriptor represents the top level view of a J2EE application's contents. The J2EE application deployment descriptor is specified by the J2EE:application XML document type definition (DTD) (see Section 8.4, "J2EE:application XML DTD").

8.1.3.1 Customization

In certain cases, a J2EE application will need customization before it can be deployed into the enterprise. New J2EE modules may be added to the application. Existing modules may be removed from the application. Some J2EE modules may need custom content created, changed, or replaced. For example, an application consumer may need to use an html editor to add company graphics to a template login page that was provided with a J2EE web application.

8.1.4 Deployment

During the deployment phase of an application's life cycle, the application is installed on the J2EE platform and then is configured and integrated into the existing infrastructure. Each J2EE module listed in the application deployment descriptor must be deployed according to the requirements of the specification for the respective J2EE module type. Each module listed must be installed in the appropriate container type and the environment properties of each module must be set appropriately in the target container to reflect the values declared by the deployment descriptor element for each component.

8.2 Application Assembly

This section specifies the sequence of steps that are typically followed when composing a J2EE application.

▼ Assembling a J2EE Application

1. Select the J2EE modules that will be used by the application.
2. Create an application directory structure.

   The directory structure of an application is arbitrary. The structure should be designed around the requirements of the contained components.
3. Reconcile J2EE module deployment parameters.

   The deployment descriptors for the J2EE modules must be edited to link internally satisfied dependencies and eliminate any redundant security role names. An optional element `alt-dd` (described in Section 8.4, "J2EE:application XML DTD") may be used when it is desirable to preserve the original deployment descriptor. The element `alt-dd` specifies an alternate deployment descriptor to use at deployment time. The edited copy of the deployment descriptor file may be saved in the application directory tree in a location determined by the Application Assembler. If the `alt-dd` element is not present, the Deployer must read the deployment descriptor directly from the JAR.

   a. Link the internally satisfied dependencies of all components in every module contained in the application. For each component dependency, there must only be one corresponding component that fulfills that dependency in the scope of the application.

      i. For each `ejb-link`, there must be only one matching `ejb-name` in the scope of the entire application (see Section 5.3, "Enterprise JavaBeans™ (EJB) References").

      ii. Dependencies that are not linked to internal components must be handled by the Deployer as external dependencies that must be met by resources previously installed on the platform. External dependencies must be linked to the resources on the platform during deployment.

b. Synchronize security role-names across the application. Rename unique role-names with redundant meaning to a common name. Rename role-names with common names but different meanings to unique names. Descriptions of role-names that are used by many components of the application can be included in the EAR file.

c. Assign a context root for each web module included in the J2EE application. The context root is a relative name in the web namespace for the application. Each web module must be given a distinct name for its context root. The web modules will be assigned a complete name in the namespace of the web server at deployment time. The context root may be the empty string.

d. Make sure that each component in the application properly describes any dependencies it may have on other components in the application. A J2EE application should not assume that all components in the application will be available on the "classpath" of the application at run time. Each component might be loaded into a separate class loader with a separate namespace. If the classes in a JAR file depend on classes in another JAR file, the first JAR file must reference the second JAR file using the `Class-Path` mechanism.

e. There must be only one version of each class in an application. If one component depends on one version of an optional package, and another component depends on another version, it may not be possible to deploy an application containing both components. A J2EE application should not assume that each component is loaded in a separate class loader and has a separate namespace. All components in a single application may be loaded in a single class loader and share a single namespace. (Note that it is not currently possible to reference a single copy of an optional package between multiple web modules, for instance. There may need to be multiple copies of the optional package included in the application package, but all copies should be identical.)

4. Create an XML deployment descriptor for the application.

   The deployment descriptor must be named "`application.xml`" and must reside in the top level of the `META-INF` directory of the application `.ear` file. The deployment descriptor must be a valid XML document according to the document type definition (DTD) for a J2EE:application XML document. The deployment descriptor must include an XML document type definition with a PUBLIC identifier of either "`-//Sun Microsystems//J2EE Application 1.2//EN`" or "`-//Sun Microsystems//J2EE Application 1.3//EN`".

5. Package the application.

a. Place the J2EE modules and the deployment descriptor in the appropriate directories. The deployment descriptor must be located at `META-INF/application.xml`.

b. Package the application directory hierarchy in a file using the Java Archive (JAR) file format. The file should be named with a `.ear` filename extension.

▼ Adding and Removing Modules

After the application is created, J2EE modules may be added or removed before deployment. When adding or removing a module the following steps must be performed:

1. Decide on a location in the application package for the new module. Optionally create new directories in the application package hierarchy to contain any J2EE modules that are being added to the application.
2. Copy the new J2EE modules to the desired location in the application package. The packaged modules are inserted directly in the desired location; the modules are not unpackaged.
3. Edit the deployment descriptors for the J2EE modules to link the dependencies which are internally satisfied by the J2EE modules included in the application.
4. Edit the J2EE application deployment descriptor to meet the content requirements of the J2EE platform and the validity requirements of the J2EE:application XML DTD.

8.3 Deployment

The J2EE platform supports two types of deployment units:

- Stand-alone J2EE modules.
- J2EE applications, consisting of one or more J2EE modules. A J2EE application must include one J2EE application deployment descriptor.

Any J2EE platform must be able to accept a J2EE application delivered as a `.ear` file or a stand-alone J2EE module delivered as a `.jar`,`.war`, or `.rar` file (as appropriate to its type).

8.3.1 Deploying a Stand-Alone J2EE Module

This section specifies the requirements for deployment of a stand-alone J2EE module.

1. The deployment tool must first read the J2EE module deployment descriptor from the package. See the component specifications for the required location and name of the deployment descriptor for each component type.
2. The deployment tool must deploy all of the components listed in the J2EE module deployment descriptor according to the deployment requirements of the respective J2EE component specification. If the module is a type that contains `.jar` files (for example, Web and Connector modules), all classes in `.jar` files within the module referenced from other `.jar` files within the module using the `Class-Path` manifest header must be included in the deployment.
3. The deployment tool must allow the Deployer to configure the container to reflect the values of all the properties declared by the deployment descriptor element for each component.
4. The deployment tool must allow the Deployer to deploy the same module multiple times, as multiple independent applications, possibly with different configurations. For example, the enterprise beans in an ejb-jar file might be deployed multiple times under different JNDI names and with different configurations of their resources.

8.3.2 Deploying a J2EE Application

This section specifies the requirements for deployment of a J2EE application.

1. The deployment tool must first read the J2EE application deployment descriptor from the application `.ear` file (`META-INF/application.xml`).
2. The deployment tool must open each of the J2EE modules listed in the J2EE application deployment descriptor and read the J2EE module deployment descriptor from the package. See the Enterprise JavaBeans, Servlet, J2EE Connector and application client specifications for the required location and name of the deployment descriptor for each component type. (The application client specification is Chapter 9, “Application Clients”.)

3. The deployment tool must install all of the components described by each module deployment descriptor into the appropriate container according to the deployment requirements of the respective J2EE component specification. All classes in `.jar` files referenced from other `.jar` files using the `Class-Path` manifest header must be included in the deployment.

4. The deployment tool must allow the Deployer to configure the container to reflect the values of all the properties declared by the deployment descriptor element for each component.

5. The deployment tool must allow the Deployer to deploy the same J2EE application multiple times, as multiple independent applications, possibly with different configurations. For example, the enterprise beans in an ejb-jar file might be deployed multiple times under different JNDI names and with different configurations of their resources.

6. When presenting security role descriptions to the Deployer, the deployment tool must use the descriptions in the J2EE application deployment descriptor rather than the descriptions in any component deployment descriptors for security roles with the same name.

8.4 J2EE:application XML DTD

This section provides the XML DTD for the J2EE application deployment descriptor. The XML grammar for a J2EE application deployment descriptor is defined by the J2EE:application document type definition. The granularity of composition for J2EE application assembly is the J2EE module. A J2EE:application deployment descriptor contains a name and description for the application and the URI of a UI icon for the application, as well a list of the J2EE modules that comprise the application. The content of the XML elements is in general case sensitive. This means, for example, that `<role-name>Manager</role-name>` is a different role than `<role-name>manager</role-name>`.

All valid J2EE application deployment descriptors must contain the following DOCTYPE declaration:

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD
J2EE Application 1.3//EN" "http://java.sun.com/dtd/
application_1_3.dtd">
```

or the DOCTYPE declaration from a previous version of this specification. (See Appendix A, “Previous Version DTDs.”) The deployment descriptor must be named `META-INF/application.xml` in the `.ear` file.

FIGURE 8-3 shows a graphic representation of the structure of the J2EE:application XML DTD.

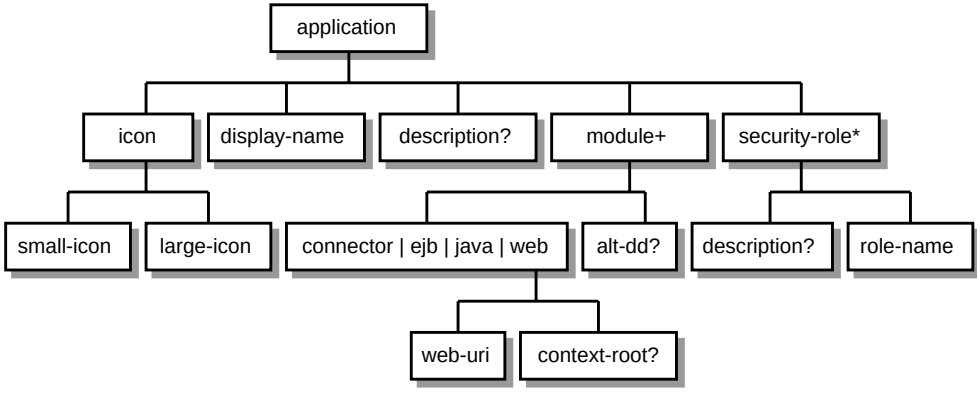

FIGURE 8-3 J2EE:application XML DTD Structure

The DTD that follows defines the XML grammar for a J2EE application deployment descriptor.

```
<!--
The alt-dd element specifies an optional URI to the post-
assembly version of the deployment descriptor file for a
particular J2EE module. The URI must specify the full pathname
of the deployment descriptor file relative to the application's
root directory. If alt-dd is not specified, the deployer must
read the deployment descriptor from the default location and
file name required by the respective component specification.
-->
<!ELEMENT alt-dd (#PCDATA)>

<!--
The application element is the root element of a J2EE
application deployment descriptor.
-->
<!ELEMENT application (icon?, display-name, description?,
module+, security-role*)>

<!--
The connector element specifies the URI of a resource adapter
archive file, relative to the top level of the application
package.
-->
<!ELEMENT connector (#PCDATA)>

<!--
The context-root element specifies the context root of a web
application.
-->
<!ELEMENT context-root (#PCDATA)>

<!--
The description element provides a human readable description
of the application. The description element should include any
information that the application assembler wants to provide the
deployer.
-->
```

```
<!ELEMENT description (#PCDATA)>

<!--
The display-name element specifies an application name.
The application name is assigned to the application by the
application assembler and is used to identify the application
to the deployer at deployment time.
-->
<!ELEMENT display-name (#PCDATA)>

<!--
The ejb element specifies the URI of an ejb-jar, relative to
the top level of the application package.
-->
<!ELEMENT ejb (#PCDATA)>

<!--
The icon element contains a small-icon and large-icon element
which specify the URIs for a small and a large GIF or JPEG icon
image to represent the application in a GUI.
-->
<!ELEMENT icon (small-icon?, large-icon?)>

<!--
The java element specifies the URI of a java application client
module, relative to the top level of the application package.
-->
<!ELEMENT java (#PCDATA)>

<!--
The large-icon element specifies the URI for a large GIF or
JPEG icon image to represent the application in a GUI.
-->
<!ELEMENT large-icon (#PCDATA)>
```

```
<!--
The module element represents a single J2EE module and contains
a connector, ejb, java, or web element, which indicates the
module type and contains a path to the module file, and an
optional alt-dd element, which specifies an optional URI to the
post-assembly version of the deployment descriptor.
The application deployment descriptor must have one module
element for each J2EE module in the application package.
-->
<!ELEMENT module ((connector | ejb | java | web), alt-dd?)>

<!--
The role-name element contains the name of a security role.
-->
<!ELEMENT role-name (#PCDATA)>

<!--
The security-role element contains the definition of a security
role which is global to the application. The definition
consists of a description of the security role, and the
security role name. The descriptions at this level override
those in the component level security-role definitions and must
be the descriptions tools display to the deployer.
-->
<!ELEMENT security-role (description?, role-name)>

<!--
The small-icon element specifies the URI for a small GIF or
JPEG icon image to represent the application in a GUI.
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
The web element contains the web-uri and context-root of a web
application module.
-->
<!ELEMENT web (web-uri, context-root)>
```

```
<!--
The web-uri element specifies the URI of a web application
file, relative to the top level of the application package.
-->
<!ELEMENT web-uri (#PCDATA)>

<!--
The ID mechanism is to allow tools to easily make tool-specific
references to the elements of the deployment descriptor.
 -->
<!ATTLIST alt-dd id ID #IMPLIED>
<!ATTLIST application id ID #IMPLIED>
<!ATTLIST connector id ID #IMPLIED>
<!ATTLIST context-root id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST java id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST module id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST web id ID #IMPLIED>
<!ATTLIST web-uri id ID #IMPLIED>
```

CHAPTER 9

Application Clients

This chapter describes application clients in the Java™ 2 Platform, Enterprise Edition (J2EE).

9.1 Overview

Application clients are first tier client programs that execute in their own Java™ virtual machines. Application clients follow the model for Java technology-based applications - they are invoked at their `main` method and run until the virtual machine is terminated. However, like other J2EE application components, application clients depend on a container to provide system services. The application client container may be very light-weight compared to other J2EE containers, providing only the security and deployment services described below

9.2 Security

Application clients have the same authentication requirements and may use the same authentication techniques as other J2EE application components.

Unprotected web resources may be accessed without authentication. Authentication when accessing protected web resources may use HTTP Basic authentication, SSL client authentication, or HTTP Login Form authentication. Lazy authentication may be used.

Authentication is required when accessing enterprise beans. The authentication mechanisms for enterprise beans are unspecified. Lazy authentication may be used.

The application client may authenticate its user in a number of ways. The techniques used are platform-dependent and not under control of the application client. The application client container may integrate with the platform's authentication system, providing a single signon capability. The application client container may authenticate the user when the application is started. The application client container may use lazy authentication, only authenticating the user when it needs to access a protected resource. This version of this specification does not describe the technique used to authenticate the user.

If the container needs to interact with the user to gather authentication data, the container must provide an appropriate user interface. In addition, an application client may provide a class that implements the `javax.security.auth.callback.CallbackHandler` interface and specify the class name in its deployment descriptor (see Section 9.7, "J2EE:application-client XML DTD" for details). The Deployer may override the callback handler specified by the application and require use of the container's default authentication user interface instead.

If use of a callback handler has been configured by the Deployer, the application client container must instantiate an object of this class and use it for all authentication interactions with the user. The application's callback handler must support all the `Callback` objects specified in the `javax.security.auth.callback` package.

Note that in the case of HTTP Login Form authentication, the authentication user interface is provided by the server (in the form of an HTML page delivered in response to an HTTP request) and must be displayed by the application client.

Application clients execute in an environment with a SecurityManager installed and have similar security Permission requirements as Servlets. The security Permission requirements are described fully in Chapter 6, "Application Programming Interface."

9.3 Transactions

Application clients are not required to have direct access to the transaction facilities of the J2EE platform. A J2EE product is not required to provide a JTA `UserTransaction` object for use by application clients. Of course, application clients can invoke enterprise beans that start transaction, and they can use the transaction facilities of the JDBC API. If a JDBC API transaction is open when an application client invokes an enterprise bean, the transaction context is not required to be propagated to the EJB server.

9.4 Naming

As with all J2EE components, application clients use JNDI to lookup enterprise beans, get access to resource managers, access configurable parameters set at deployment time, etc. Application clients use the `java:` JNDI namespace to access these items, see Chapter 5, “Naming” for details.

9.5 Application Programming Interfaces

Application clients have all the facilities of the Java™ 2 Platform, Standard Edition (subject to security restrictions), as well as various standard extensions, as described in Chapter 6, “Application Programming Interface.” Each application client executes in its own Java virtual machine. Application clients start execution at the `main` method of the class specified in the `Main-Class` attribute in the manifest file of the application client’s jar file (although note that application client container code will typically execute before the application client itself, in order to prepare the environment of the container, install a `SecurityManager`, initialize the name service client library, etc.).

9.6 Packaging and Deployment

Application clients are packaged in jar files and include a deployment descriptor similar to other J2EE application components. The deployment descriptor describes the enterprise beans and external resources referenced by the application. As with other J2EE application components, access to resources must be configured at deployment time, names assigned for enterprise beans and resources, etc.

The tool used to deploy an application client, and the mechanism used to install the application client, is not specified. Very sophisticated J2EE products may allow the application client to be deployed on a J2EE server and automatically made available to some set of (usually intranet) clients. Other J2EE products may require the J2EE application bundle containing the application client to be manually deployed and installed on each client machine. And yet another approach would be for the deployment tool on the J2EE server to produce an installation package that could be taken to each client to install the application client. There are many possibilities here and this specification doesn't prescribe any one; it only defines the package format for the application client and the things that must be possible during the deployment process.

How an application client is invoked by an end user is unspecified. Typically a J2EE Product Provider will provide an application launcher that integrates with the application client machine's native operating system, but the level of such integration is unspecified.

9.7 J2EE:application-client XML DTD

The XML grammar for a J2EE application client deployment descriptor is defined by the J2EE:application-client document type definition. The root element of the deployment descriptor for an application client is `application-client`. The content of the XML elements is in general case sensitive. This means, for example, that `<res-auth>Container</res-auth>` must be used, rather than `<res-auth>container</res-auth>`.

All valid `application-client` deployment descriptors must contain the following DOCTYPE declaration:

```
<!DOCTYPE application-client PUBLIC "-//Sun Microsystems,
Inc.//DTD J2EE Application Client 1.3//EN" "http://
java.sun.com/dtd/application-client_1_3.dtd">
```

or the DOCTYPE declaration from a previous version of this specification. (See Appendix A, “Previous Version DTDs.”) The deployment descriptor must be named `META-INF/application-client.xml` in the application client’s `.jar` file.

FIGURE 9-1 shows the structure of the J2EE:application-client XML DTD.

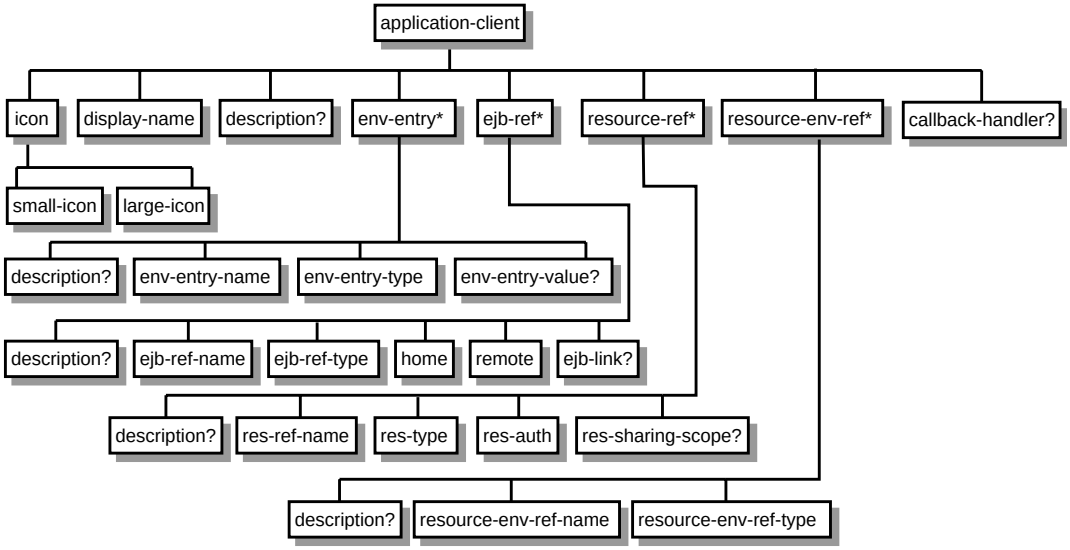

FIGURE 9-1 J2EE:application-client XML DTD Structure

```
<!--
The application-client element is the root element of an
application client deployment descriptor.
The application client deployment descriptor describes the EJB
components and external resources referenced by the
application client.
-->
<!ELEMENT application-client (icon?, display-name,
description?, env-entry*, ejb-ref*, resource-ref*, resource-
env-ref*, callback-handler?)>

<!--
The callback-handler element names a class provided by the
application. The class must have a no args constructor and must
implement the javax.security.auth.callback.CallbackHandler
interface.  The class will be instantiated by the application
client container and used by the container to collect
authentication information from the user.
```

```
-->
<!ELEMENT callback-handler (#PCDATA)>

<!--
The description element is used to provide text describing the
parent element. The description element should include any
information that the application-client file producer wants to
provide to the consumer of the application-client file (i.e.,
to the Deployer). Typically, the tools used by the application-
client file consumer will display the description when
processing the parent element that contains the description.
-->
<!ELEMENT description (#PCDATA)>

<!--
The display-name element contains a short name that is intended
to be displayed by tools.
-->
<!ELEMENT display-name (#PCDATA)>

<!--
The ejb-link element is used in the ejb-ref element to specify
that an EJB reference is linked to an enterprise bean in the
encompassing J2EE Application package. The value of the ejb-
link element must be the ejb-name of an enterprise bean in the
same J2EE Application package.
Example: <ejb-link>EmployeeRecord</ejb-link>
Alternatively, the name in the ejb-link element may be composed
of a path name specifying the ejb-jar containing the referenced
enterprise bean with the ejb-name of the target bean appended
and separated from the path name by "#". The path name is
relative to the jar file containing the referencing component.
This allows multiple enterprise beans with the same ejb-name
to be uniquely identified.
Example: <ejb-link>../products/product.jar#ProductEJB</ejb-
link>
Used in: ejb-ref
-->
```

```
<!ELEMENT ejb-link (#PCDATA)>

<!--
The ejb-ref element is used for the declaration of a reference
to an enterprise bean's home. The declaration consists of an
optional description; the EJB reference name used in the code
of the referencing application client; the expected type of the
referenced enterprise bean; the expected home and remote
interfaces of the referenced enterprise bean; and an optional
ejb-link information. The optional ejb-link element is used to
specify the referenced enterprise bean.
-->
<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type,
home, remote, ejb-link?)>

<!--
The ejb-ref-name element contains the name of an EJB reference.
The EJB reference is an entry in the application client's
environment. It is recommended that name is prefixed with "ejb/
".
Used in: ejb-ref
Example: <ejb-ref-name>ejb/Payroll</ejb-ref-name>
-->
<!ELEMENT ejb-ref-name (#PCDATA)>

<!--
The ejb-ref-type element contains the expected type of the
referenced enterprise bean. The ejb-ref-type element must be
one of the following:
<ejb-ref-type>Entity</ejb-ref-type>
<ejb-ref-type>Session</ejb-ref-type>
Used in: ejb-ref
-->
<!ELEMENT ejb-ref-type (#PCDATA)>

<!--
```

```
The env-entry element contains the declaration of an
application client’s environment entry. The declaration
consists of an optional description, the name of the
environment entry, and an optional value.
-->
<!ELEMENT env-entry (description?, env-entry-name, env-entry-
type, env-entry-value?)>

<!--
The env-entry-name element contains the name of an application
client’s environment entry.
Used in: env-entry
Example: <env-entry-name>EmployeeAppDB</env-entry-name>
-->
<!ELEMENT env-entry-name (#PCDATA)>

<!--
The env-entry-type element contains the fully-qualified Java
type of the environment entry value that is expected by the
application client’s code. The following are the legal values
of env-entry-type: java.lang.Bool-ean, java.lang.String,
java.lang.Integer, java.lang.Double, java.lang.Byte,
java.lang.Short, java.lang.Long, and java.lang.Float.
Used in: env-entry
Example:
<env-entry-type>java.lang.Boolean</env-entry-type>
-->
<!ELEMENT env-entry-type (#PCDATA)>

<!--
The env-entry-value element contains the value of an
application client’s environment entry. The value must be a
String that is valid for the constructor of the specified type
that takes a single String parameter.
Used in: env-entry
Example:
<env-entry-value>/datasources/MyDatabase</env-entry-value>
```

```
-->
<!ELEMENT env-entry-value (#PCDATA)>

<!--
The home element contains the fully-qualified name of the
enterprise bean's home interface.
Used in: ejb-ref
Example: <home>com.aardvark.payroll.PayrollHome</home>
-->
<!ELEMENT home (#PCDATA)>

<!--
The icon element contains a small-icon and large-icon element
which specify the URIs for a small and a large GIF or JPEG icon
image used to represent the application client in a GUI tool.
-->
<!ELEMENT icon (small-icon?, large-icon?)>

<!--
The large-icon element contains the name of a file containing
a large (32 x 32) icon image. The file name is a relative path
within the application-client jar file. The image must be
either in the JPEG or GIF format, and the file name must end
with the suffix ".jpg" or ".gif" respectively. The icon can be
used by tools.
Example:
<large-icon>lib/images/employee-service-icon32x32.jpg</large-
icon>
-->
<!ELEMENT large-icon (#PCDATA)>

<!--
The remote element contains the fully-qualified name of the
enterprise bean's remote interface.
Used in: ejb-ref
Example:
```

```
<remote>com.wombat.empl.EmployeeService</remote>
-->
<!ELEMENT remote (#PCDATA)>

<!--
The res-auth element specifies whether the enterprise bean code
signs on programmatically to the resource manager, or whether
the Container will sign on to the resource manager on behalf
of the bean. In the latter case, the Container uses information
that is supplied by the Deployer.
The value of this element must be one of the two following:
<res-auth>Application</res-auth>
<res-auth>Container</res-auth>
-->
<!ELEMENT res-auth (#PCDATA)>

<!--
The res-ref-name element specifies the name of the resource
manager connection factory reference name. The resource
manager connection factory reference name is the name of the
application client's environment entry whose value contains
the JNDI name of the data source.
Used in: resource-ref
-->
<!ELEMENT res-ref-name (#PCDATA)>

<!--
The res-sharing-scope element specifies whether connections
obtained through the given resource manager connection factory
reference can be shared. The value of this element, if
specified, must be one of the following:
<res-sharing-scope>Shareable</res-sharing-scope>
<res-sharing-scope>Unshareable</res-sharing-scope>
The default value is Shareable.
Used in: resource-ref
-->
```

```
<!ELEMENT res-sharing-scope (#PCDATA)>
<!--
The res-type element specifies the type of the data source. The
type is specified by the Java interface (or class) expected to
be implemented by the data source.
Used in: resource-ref
-->
<!ELEMENT res-type (#PCDATA)>
<!--
The resource-env-ref element contains a declaration of an
application's reference to an administered object associated
with a resource in the application's environment. It consists
of an optional description, the resource environment reference
name, and an indication of the resource environment reference
type expected by the application code.
Used in: application-client
Example:
                <resource-env-ref>
                    <resource-env-ref-name>jms/StockQueue
                        </resource-env-ref-name>
                    <resource-env-ref-type>javax.jms.Queue
                        </resource-env-ref-type>
                </resource-env-ref>
-->
<!ELEMENT resource-env-ref (description?, resource-env-ref-
name, resource-env-ref-type)>

<!--
The resource-env-ref-name element specifies the name of a
resource environment reference; its value is the environment
entry name used in the application code.
Used in: resource-env-ref
-->
<!ELEMENT resource-env-ref-name (#PCDATA)>

<!--
```

```
The resource-env-ref-type element specifies the type of a
resource environment reference.

Used in: resource-env-ref

-->

<!ELEMENT resource-env-ref-type (#PCDATA)>


<!--

The resource-ref element contains a declaration of application
clients's reference to an external resource. It consists of an
optional description, the resource factory reference name, the
indication of the resource factory type expected by the
application client's code, and the type of authentication (bean
or container).

Example:

                <resource-ref>
                <res-ref-name>EmployeeAppDB</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
                </resource-ref>

-->

<!ELEMENT resource-ref (description?, res-ref-name, res-type,
res-auth, res-sharing-scope?)


<!--

The small-icon element contains the name of a file containing
a small (16 x 16) icon image. The file name is a relative path
within the application-client jar file. The image must be
either in the JPEG or GIF format, and the file name must end
with the suffix ".jpg" or ".gif" respectively. The icon can be
used by tools.

Example:

<small-icon>lib/images/employee-service-icon16x16.jpg</small-
icon>

-->

<!ELEMENT small-icon (#PCDATA)>


<!--
```

```
The ID mechanism is to allow tools to easily make tool-specific
references to the elements of the deployment descriptor.
 -->
<!ATTLIST application-client id ID #IMPLIED>
<!ATTLIST callback-handler id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-sharing-scope id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST resource-env-ref id ID #IMPLIED>
<!ATTLIST resource-env-ref-name id ID #IMPLIED>
<!ATTLIST resource-env-ref-type id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
```

CHAPTER 10

Service Provider Interface

The Java™ 2 Platform, Enterprise Edition (J2EE) includes the J2EE Connector Architecture as its service provider interface. The Connector API defines how resource adapters are packaged and integrated with any J2EE product. All J2EE products must support the Connector APIs, as specified in the Connector specification.

The Connector specification is available at `http://java.sun.com/j2ee/connector`.

CHAPTER 11

Future Directions

This version of the Java™ 2 Platform, Enterprise Edition (J2EE) specification includes most of the facilities needed by enterprise applications. Still, there is always more to be done. This chapter briefly describes our plans for future versions of this specification. Please keep in mind that all of this is subject to change. Your feedback is encouraged.

The following sections describe additional facilities we would like to include in future versions of this specification. Many of the APIs included in the J2EE platform will continue to evolve on their own and we will include the latest version of each API.

11.1 XML Data Binding API

As XML becomes more important in the industry, more and more enterprise applications will need to make use of XML. This specification requires basic XML SAX and DOM support through the JAXP API, but many applications will benefit from the easier to use XML Data Binding technology. The XML Data Binding API is being defined through the Java Community Process as JSR-031.

XML Data Binding depends on schema languages to define the XML data. The current widely used schema language is the DTD language. W3C is in the process of standardizing a new XML Schema language. In addition, there are several other schema languages in use and proposed in the industry.

In order to support emerging schema language standards quickly, the XML Data Binding API will need to evolve more quickly than the J2EE platform. Inclusion of the XML Data Binding API as a required component of J2EE at this time would constrain its evolution. We expect that the next version of the J2EE platform will require support for XML Data Binding. In the mean time, we

strongly encourage the use of this new technology by enterprise applications as it becomes available. We expect the XML Data Binding technology to be portable to any J2EE product.

The XML Data Binding JSR is available at `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_031_xmld.html`.

11.2 JNLP (Java™ Web Start)

The Java Network Launch Protocol defines a mechanism for deploying Java applications on a server and launching them from a client. A future version of this specification may require that J2EE products be able to deploy application clients in a way that allows them to be launched by a JNLP client, and that application client containers be able to launch application clients deployed using the JNLP technology. Java Web Start is the reference implementation of a JNLP client.

More information on JNLP is available at `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_056_jnlp.html`; more information on Java Web Start is available at `http://java.sun.com/products/javawebstart`.

11.3 J2EE SPI

Many of the APIs that make up the J2EE platform include an SPI layer that allow service providers or other system level components to be plugged in. This specification does not describe the execution environment for all such service providers, nor the packaging and deployment requirements for all service providers. However, the J2EE Connector Extension does define the requirements for certain types of service providers called resource adapters. Future versions of this specification will more fully define the J2EE SPI.

11.4 JDBC RowSets

RowSets provide a standard way to send tabular data between the remote components of a distributed, enterprise application. The JDBC 2.0 Extension API defines the RowSet APIs, and in the future will contain rowset implementations, as well. Future versions of this specification will require that the JDBC rowset implementations be supported. More information is available at `http://java.sun.com/products/jdbc`.

11.5 Security APIs

It is a goal of the J2EE platform to separate security from business logic, providing declarative security controls for application components. However, some applications need more control over security than can be provided by this approach. A future version of this specification may include additional APIs to control authentication and authorization, and to allow the integration of new security technologies.

11.6 Deployment APIs

This specification assumes that deployment tools will be provided by with a J2EE product by the Product Provider. J2EE Tool Providers would also like to be able to provide deployment tools that could work with all J2EE products. Future versions of this specification may define deployment APIs to allow the creation of such tools.

The J2EE Deployment API is being defined by JSR-088, see `http://java.sun.com/jcp/jsr/jsr_088_deploy.html`.

11.7 Management APIs

J2EE applications and J2EE products must be manageable. Future versions of this specification will include APIs to support management functions.

The J2EE Management APIs are being defined by JSR-077, see `http://java.sun.com/jcp/jsr/jsr_077_management.html`.

11.8 SQLJ Part 0

SQLJ Part 0 supports embedding of SQL statements in programs written in the Java programming language. A compiler translates the program into a program that uses the SQLJ Part 0 runtime. The runtime supports access to a database using JDBC, while also allowing platform-dependent and database-specific optimizations of such access. The SQLJ Part 0 runtime classes can be packaged with a J2EE application that uses SQLJ Part 0, allowing that application to run on any J2EE platform. At the current time, customer demand for SQLJ Part 0 is not sufficient to include it as a part of the J2EE platform. If customer demand increases, a future version of this specification may require the platform to provide the SQLJ Part 0 runtime classes so that they do not need to be packaged with the application. For information on SQLJ, see `http://www.sqlj.org`.

APPENDIX **A**

Previous Version DTDs

This appendix contains Document Type Definitions for Deployment Descriptors from previous versions of the J2EE specification. All J2EE products are required to support these DTDs as well as the DTDs specified in this version of the specification. This ensures that applications written to previous versions of this specification can be deployed on products supporting the current version of this specification. In addition, there are no restrictions on mixing versions of deployment descriptors in a single application; any combination of valid deployment descriptor versions must be supported.

A.1 J2EE:application XML DTD

This section provides the XML DTD for the previous version of the J2EE application deployment descriptor. A valid J2EE application deployment descriptor may contain the following DOCTYPE declaration:

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD
J2EE Application 1.2//EN" "http://java.sun.com/j2ee/dtds/
application_1_2.dtd">
```

FIGURE A-1 shows a graphic representation of the structure of the J2EE:application XML DTD.

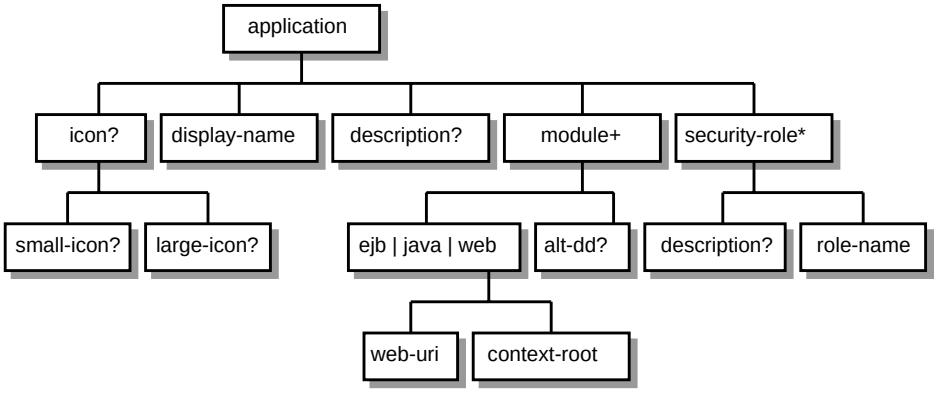

FIGURE A-1 J2EE:application XML DTD Structure

The DTD that follows defines the XML grammar for a J2EE application deployment descriptor.

```
<!--
The alt-dd element specifies an optional URI to the post-
assembly version of the deployment descriptor file for a
particular J2EE module. The URI must specify the full pathname
of the deployment descriptor file relative to the
application's root directory. If alt-dd is not specified, the
deployer must read the deployment descriptor from the default
location and file name required by the respective component
specification.
-->
<!ELEMENT alt-dd (#PCDATA)>


<!--
The application element is the root element of a J2EE
application deployment descriptor.
-->
<!ELEMENT application (icon?, display-name, description?,
module+, security-role*)>
```

```
<!--
The context-root element specifies the context root of a web
application
-->
<!ELEMENT context-root (#PCDATA)>

<!--
The description element provides a human readable description
of the application. The description element should include any
information that the application assembler wants to provide
the deployer.
-->
<!ELEMENT description (#PCDATA)>

<!--
The display-name element specifies an application name.
The application name is assigned to the application by the
application assembler and is used to identify the application
to the deployer at deployment time.
-->
<!ELEMENT display-name (#PCDATA)>

<!--
The ejb element specifies the URI of a ejb-jar, relative to
the top level of the application package.
-->
<!ELEMENT ejb (#PCDATA)>

<!--
The icon element contains a small-icon and large-icon element
which specify the URIs for a small and a large GIF or JPEG icon
image to represent the application in a GUI.
-->
<!ELEMENT icon (small-icon?, large-icon?)>

<!--
```

```
The java element specifies the URI of a java application client
module, relative to the top level of the application package.
-->
<!ELEMENT java (#PCDATA)>

<!--
The large-icon element specifies the URI for a large GIF or
JPEG icon image to represent the application in a GUI.
-->
<!ELEMENT large-icon (#PCDATA)>

<!--
The module element represents a single J2EE module and contains
an ejb, java, or web element, which indicates the module type
and contains a path to the module file, and an optional alt-
dd element, which specifies an optional URI to the post-
assembly version of the deployment descriptor.
The application deployment descriptor must have one module
element for each J2EE module in the application package.
-->
<!ELEMENT module ((ejb | java | web), alt-dd?)>

<!--
The role-name element contains the name of a security role.
-->
<!ELEMENT role-name (#PCDATA)>

<!--
The security-role element contains the definition of a
security role which is global to the application. The
definition consists of a description of the security role, and
the security role name. The descriptions at this level override
those in the component level security-role definitions and
must be the descriptions tool display to the deployer.
-->
<!ELEMENT security-role (description?, role-name)>
```

```
<!--
The small-icon element specifies the URI for a small GIF or
JPEG icon image to represent the application in a GUI.
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
The web element contains the web-uri and context-root of a web
application module.
-->
<!ELEMENT web (web-uri, context-root)>

<!--
The web-uri element specifies the URI of a web application
file, relative to the top level of the application package.
-->
<!ELEMENT web-uri (#PCDATA)>

<!--
The ID mechanism is to allow tools to easily make tool-specific
references to the elements of the deployment descriptor.
 -->
<!ATTLIST alt-dd id ID #IMPLIED>
<!ATTLIST application id ID #IMPLIED>
<!ATTLIST context-root id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST java id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST module id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
```

```
<!ATTLIST web id ID #IMPLIED>
<!ATTLIST web-uri id ID #IMPLIED>
```

A.2 J2EE:application-client XML DTD

This section contains the XML DTD for the previous version of the application client deployment descriptor. A valid application client deployment descriptor may contain the following DOCTYPE declaration:

```
<!DOCTYPE application-client PUBLIC "-//Sun Microsystems,
Inc.//DTD J2EE Application Client 1.2//EN" "http://
java.sun.com/j2ee/dtds/application-client_1_2.dtd">
```

FIGURE A-2 shows the structure of the J2EE:application-client XML DTD.

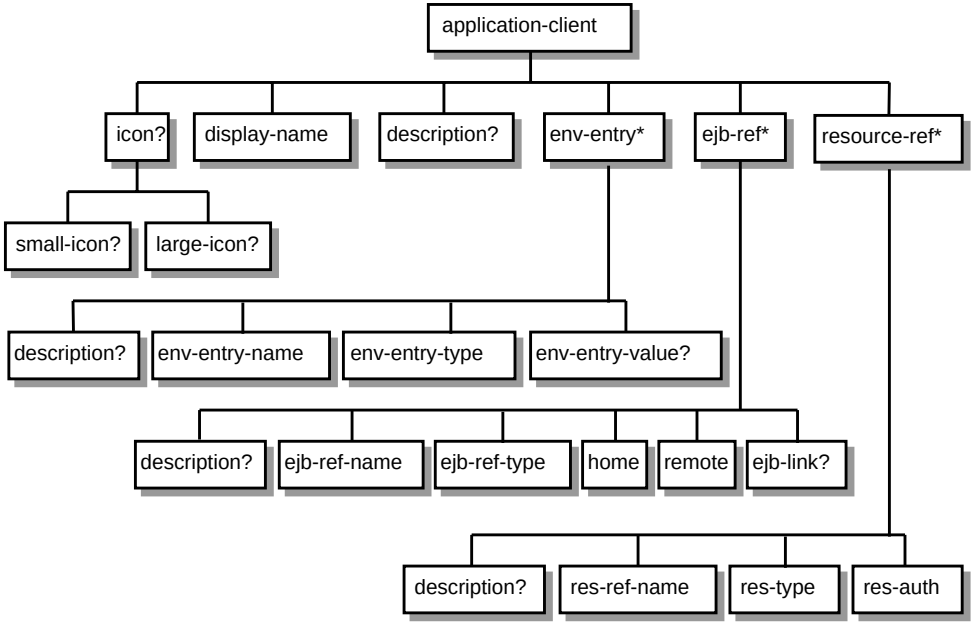

FIGURE A-2 J2EE:application-client XML DTD Structure

```
<!--
The application-client element is the root element of an
application client deployment descriptor.
The application client deployment descriptor describes the EJB
components and external resources referenced by the
application client.
-->
<!ELEMENT application-client (icon?, display-name,
description?, env-entry*, ejb-ref*, resource-ref*)>

<!--
The description element is used to provide text describing the
parent element. The description element should include any
information that the application-client file producer wants to
provide to the consumer of the application-client file (i.e.,
to the Deployer). Typically, the tools used by the application-
client file consumer will display the description when
processing the parent element that contains the description.
-->
<!ELEMENT description (#PCDATA)>

<!--
The display-name element contains a short name that is intended
to be displayed by tools.
-->
<!ELEMENT display-name (#PCDATA)>

<!--
The ejb-link element is used in the ejb-ref element to specify
that an EJB reference is linked to an enterprise bean in the
encompassing J2EE Application package. The value of the ejb-
link element must be the ejb-name of an enterprise bean in the
same J2EE Application package. Used in: ejb-ref
Example: <ejb-link>EmployeeRecord</ejb-link>
-->
<!ELEMENT ejb-link (#PCDATA)>
```

```
<!--
The ejb-ref element is used for the declaration of a reference
to an enterprise bean's home. The declaration consists of an
optional description; the EJB reference name used in the code
of the referencing application client; the expected type of
the referenced enterprise bean; the expected home and remote
interfaces of the referenced enterprise bean; and an optional
ejb-link information. The optional ejb-link element is used to
specify the referenced enterprise bean.
-->
<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type,
home, remote, ejb-link?)>

<!--
The ejb-ref-name element contains the name of an EJB reference.
The EJB reference is an entry in the application client's
environment. It is recommended that name is prefixed with "ejb/
". Used in: ejb-ref
Example: <ejb-ref-name>ejb/Payroll</ejb-ref-name>
-->
<!ELEMENT ejb-ref-name (#PCDATA)>

<!--
The ejb-ref-type element contains the expected type of the
referenced enterprise bean. The ejb-ref-type element must be
one of the following:
<ejb-ref-type>Entity</ejb-ref-type>
<ejb-ref-type>Session</ejb-ref-type>
Used in: ejb-ref
-->
<!ELEMENT ejb-ref-type (#PCDATA)>
```

```
<!--
The env-entry element contains the declaration of an
application client’s environment entries. The declaration
consists of an optional description, the name of the
environment entry, and an optional value.
-->
<!ELEMENT env-entry (description?, env-entry-name, env-entry-
type, env-entry-value?)>


<!--
The env-entry-name element contains the name of an application
client’s environment entry. Used in: env-entry
Example: <env-entry-name>EmployeeAppDB</env-entry-name>
-->
<!ELEMENT env-entry-name (#PCDATA)>


<!--
The env-entry-type element contains the fully-qualified Java
type of the environment entry value that is expected by the
application client’s code. The following are the legal values
of env-entry-type: java.lang.Bool-ean, java.lang.String,
java.lang.Integer, java.lang.Double, java.lang.Byte,
java.lang.Short, java.lang.Long, and java.lang.Float.


Used in: env-entry


Example:
<env-entry-type>java.lang.Boolean</env-entry-type>
-->
<!ELEMENT env-entry-type (#PCDATA)>
```

```
<!--
The env-entry-value element contains the value of an
application client's environment entry. The value must be a
String that is valid for the constructor of the specified type
that takes a single String parameter.

Used in: env-entry

Example:
<env-entry-value>/datasources/MyDatabase</env-entry-value>
-->
<!ELEMENT env-entry-value (#PCDATA)>

<!--
The home element contains the fully-qualified name of the
enterprise bean's home interface.
Used in: ejb-ref
Example: <home>com.aardvark.payroll.PayrollHome</home>
-->
<!ELEMENT home (#PCDATA)>

<!--
The icon element contains a small-icon and large-icon element
which specify the URIs for a small and a large GIF or JPEG icon
image used to represent the application client in a GUI tool.
-->
<!ELEMENT icon (small-icon?, large-icon?)>
```

<!--

The large-icon element contains the name of a file containing a large (32 x 32) icon image. The file name is a relative path within the application-client jar file. The image must be either in the JPEG or GIF format, and the file name must end with the suffix ".jpg" or ".gif" respectively. The icon can be used by tools.

Example:

<large-icon>lib/images/employee-service-icon32x32.jpg</large-icon>

-->

<!ELEMENT large-icon (#PCDATA)>

<!--

The remote element contains the fully-qualified name of the enterprise bean’s remote interface.

Used in: ejb-ref

Example:

<remote>com.wombat.empl.EmployeeService</remote>

-->

<!ELEMENT remote (#PCDATA)>

<!--

The res-auth element specifies whether the enterprise bean code signs on programmatically to the resource manager, or whether the Container will sign on to the resource manager on behalf of the bean. In the latter case, the Container uses information that is supplied by the Deployer.

The value of this element must be one of the two following:

<res-auth>Application</res-auth>

<res-auth>Container</res-auth>

-->

<!ELEMENT res-auth (#PCDATA)>

```
<!--
The res-ref-name element specifies the name of the resource
factory reference name. The resource factory reference name is
the name of the application client's environment entry whose
value contains the JNDI name of the data source.
Used in: resource-ref
-->
<!ELEMENT res-ref-name (#PCDATA)>

<!--
The res-type element specifies the type of the data source.
The type is specified by the Java interface (or class) expected
to be implemented by the data source.
Used in: resource-ref
-->
<!ELEMENT res-type (#PCDATA)>

<!--
The resource-ref element contains a declaration of application
clients's reference to an external resource. It consists of an
optional description, the resource factory reference name, the
indication of the resource factory type expected by the
application client's code, and the type of authentication
(bean or container).

Example:
<resource-ref>
<res-ref-name>EmployeeAppDB</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
</resource-ref>
-->
<!ELEMENT resource-ref (description?, res-ref-name, res-type,
res-auth)>
```

```
<!--
The small-icon element contains the name of a file containing
a small (16 x 16) icon image. The file name is a relative path
within the application-client jar file. The image must be
either in the JPEG or GIF format, and the file name must end
with the suffix ".jpg" or ".gif" respectively. The icon can be
used by tools.
Example:
<small-icon>lib/images/employee-service-icon16x16.jpg</
small-icon>
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
The ID mechanism is to allow tools to easily make tool-specific
references to the elements of the deployment descriptor.
 -->
<!ATTLIST application-client id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
```

```
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
```

APPENDIX **A**

Revision History

A.1 Changes in Expert Draft 1

A.1.1 Additional Requirements

- J2EE 1.3 requires J2SE 1.3.
- A JMS provider supporting both Topics and Queues is required.
- All referenced specifications are updated to reference their most recent versions.
- Added the following required APIs: JAXP 1.0, JCX 1.0, JAAS 1.0.
- Updated application and application client deployment descriptors to new versions, requiring support for previous versions as well.
- Updated Chapter 4, “Transaction Management” to specify requirements for the new transactional resources in J2EE 1.3.
- Updated Chapter 10, “Service Provider Interface” to include the JCX API as the J2EE SPI.

A.1.2 Removed Requirements

- None.

A.1.3 Editorial Changes

- Corrected TABLE 6-3 to properly refer to `java.awt.Image`.
- Corrected Section 5.4.1.2, “Declaration of resource manager connection factory references in deployment descriptor” to indicate that the valid values for the `res-auth` element are `Application` (not `Bean`) and `Container`.
- Updated Chapter 5, “Naming” to use terminology consistent with EJB 2.0 spec.
- Updated references to JNDI and RMI-IIOP to reflect the fact that they’re part of J2SE 1.3.
- Updated Chapter 11, “Future Directions” to remove items that are now required by J2EE 1.3.

A.2 Changes in Expert Draft 2

A.2.1 Additional Requirements

- Updated Section 5.3, “Enterprise JavaBeans™ (EJB) References,” and Section 9.7, “J2EE:application-client XML DTD,” to be consistent with the EJB 2.0 requirements for the `ejb-link` element.
- Application client containers are required to support JAAS callback handlers. See Section 3.4.2, “Application Clients” and Chapter 9, “Application Clients.”
- Generalized JMS Destination references to resource environment references, per the EJB specification. See Section 5.5, “Resource Environment References.”
- Expanded restrictions on use of JMS APIs, based on EJB specification. See Section 6.7, “Java™ Message Service (JMS) 1.0 Requirements.”

A.2.2 Removed Requirements

- None.

A.2.3 Editorial Changes

- Fixed names of JDBC classes in Section 5.4, “Resource Manager Connection Factory References.”

- Clarified roles of JRMP and RMI-IIOP in Section 7.2.2, “OMG Protocols” and Section 7.2.3, “RMI Protocols.”
- Added resource authentication recommendations from Connector spec to Section 3.4.3, “Resource Authentication Requirements.”
- Added reference to XML Data Binding in Chapter 11, “Future Directions.”
- Changed references to “JCX” to use “Connector Architecture.”
- Removed description of new JDBC 2.1 requirements; JDBC 2.1 is included in J2SE 1.3.

A.3 Changes in Participant Draft

A.3.1 Additional Requirements

- Added TLS 1.0 requirement to Section 7.2.1, “Internet Protocols” to be consistent with the EJB specification.
- Clarified requirements on use of RMI-IIOP by enterprise beans, see Section 6.2.2.6, “RMI-IIOP.”
- Updated JavaMail API requirement to version 1.2.
- Updated JAXP API requirement to version 1.1.
- Clarified transaction propagation requirements in Section 4.2.1, “Web Components.”

A.3.2 Removed Requirements

- None.

A.3.3 Editorial Changes

- Added acknowledgements.
- Cleaned up several of the figures.
- Added references to more specifications in Appendix B, “Related Documents.”

- Moved secure interoperability requirement from Section 3.3.2, “Non Goals” to Section 3.3.1, “Goals” to be consistent with the rest of this specification.
- Moved J2EE-specific servlet requirements back into Section 6.5, “Servlet 2.3 Requirements.”

A.4 Changes in Public Draft

A.4.1 Additional Requirements

- Required that the COSNaming JNDI service provider be included, see Section 6.2.2.7, “JNDI.”

A.4.2 Removed Requirements

- None.

A.4.3 Editorial Changes

- Clarified that the EJB interoperability requirements require a COSNaming name service to be provided, see Section 6.2.2.4, “Java™IDL” and Section 7.2.2, “OMG Protocols.”
- Added description of Run As capability to security requirements, see Section 3.5.2, “Caller Authorization.”
- Clarified that all J2EE products must support the use of the manifest `Class-Path` header to reference other `.jar` files, see Section 8.1.2, “Component Packaging: Composing a J2EE module.”
- Clarified that all J2EE products must be able to deploy stand-alone J2EE modules, see Section 8.3, “Deployment.”
- Rewrote callback handler requirements to be clearer, see Section 3.4.2, “Application Clients” and Section 9.2, “Security.”
- The JDBC SPI will likely be replaced by the Connector SPI; updated recommendations accordingly, see Section 6.3, “JDBC™ 2.0 Extension Requirements.”

A.5 Changes in Proposed Final Draft

A.5.1 Additional Requirements

- Added requirement for JDBC drivers to support the escape syntax for certain functions, see Section 6.2.2.3, “JDBC™ API.”
- Added requirement that the default ORB be usable for JavaIDL and RMI-IIOP, see Section 6.2.2.4, “Java™IDL.”
- Added requirements for local transactions and connection sharing, see Section 4.4, “Local Transaction Optimization” and Section 4.5, “Connection Sharing.”
- Added requirements for a shared ORB instance, see Section 6.2.2.4, “Java™IDL.”

A.5.2 Removed Requirements

- When deploying a standalone J2EE module in a `.jar` file, the deployment tool is no longer required to process the `Class-Path` header. See Section 8.3.1, “Deploying a Stand-Alone J2EE Module.”

A.5.3 Editorial Changes

- Further clarified that all J2EE products must support the use of the manifest `Class-Path` header to reference other `.jar` files, see Section 8.1.2, “Component Packaging: Composing a J2EE module.”
- Further clarified that all J2EE products must be able to deploy stand-alone J2EE modules, including application clients and Connectors, see Section 8.3, “Deployment.”
- Clearly document the required names of the deployment descriptors, see Section 8.4, “J2EE:application XML DTD” and Section 9.7, “J2EE:application-client XML DTD.”
- Clarify packaging requirements, deployment requirements, and runtime requirements for J2EE applications, see Section 8.2, “Application Assembly” and Section 8.3, “Deployment.”

- Updated Chapter 6, “Application Programming Interface” to use the new terminology “optional package” instead of the old terminology “standard extension.”
- Changed URLs for new versions of deployment descriptors to be located under `http://java.sun.com/dtd/`, see Section 8.4, “J2EE:application XML DTD” and Section 9.7, “J2EE:application-client XML DTD.”
- Added JNLP to Section 11.2, “JNLP (Java™ Web Start).”

APPENDIX **B**

Related Documents

This specification refers to the following documents. The terms used to refer to the documents in this specification are included in parentheses.

Java™ 2 Platform, Enterprise Edition Specification Version 1.3 (this specification). Copyright 1999-2000, Sun Microsystems, Inc. Available at `http://java.sun.com/j2ee/docs.html`.

Java™ 2 Platform, Enterprise Edition Technical Overview (J2EE Overview). Copyright 1998, 1999, Sun Microsystems, Inc. Available at `http://java.sun.com/j2ee/white.html`.

Java™ 2 Platform, Standard Edition, v1.3 API Specification (J2SE specification). Copyright 1993-2000, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jdk/1.3/docs/api/index.html`.

Enterprise JavaBeans™ Specification, Version 2.0 (EJB specification). Copyright 1998-2000, Sun Microsystems, Inc. Available at `http://java.sun.com/products/ejb`.

JavaServer Pages™ Specification, Version 1.2 (JSP specification). Copyright 1998, 1999-2000, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jsp`.

Java™ Servlet Specification, Version 2.3 (Servlet specification). Copyright 1998-2000, Sun Microsystems, Inc. Available at `http://java.sun.com/products/servlet`.

JDBC™ 2.1 API (JDBC specification). Copyright 1999, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jdbc`.

JDBC™ 2.0 Standard Extension API (JDBC extension specification). Copyright 1998, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jdbc`.

Java™ Naming and Directory Interface 1.2 Specification (JNDI specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jndi`.

Java™ Message Service, Version 1.0.2 (JMS specification). Copyright 1998, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jms`.

Java™ Transaction API, Version 1.0.1 (JTA specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jta`.

Java™ Transaction Service, Version 1.0 (JTS specification). Copyright 1997-1999, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jts`.

JavaMail™ API Specification Version 1.1 (JavaMail specification). Copyright 1998, Sun Microsystems, Inc. Available at `http://java.sun.com/products/javamail`.

JavaBeans™ Activation Framework Specification Version 1.0 (JAF specification). Copyright 1998, Sun Microsystems, Inc. Available at `http://java.sun.com/beans/glasgow/jaf.html`.

J2EE™ Connector Architecture 1.0 (Connector specification). Copyright 1999-2000, Sun Microsystems, Inc. Available at `http://java.sun.com/j2ee/connector`.

Java API for XML Parsing, Version 1.0 Final Release (JAXP specification). Copyright 1999-200, Sun Microsystems, Inc. Available at `http://java.sun.com/xml`.

Java™ Authentication and Authorization Service (JAAS) 1.0 (JAAS specification). Copyright 1999-2000, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jaas`.

The Common Object Request Broker: Architecture and Specification (CORBA 2.3.1 specification), Object Management Group. Available at `http://cgi.omg.org/cgi-bin/doc?formal/99-10-07`.

IDL To Java™ Language Mapping Specification, Object Management Group. Available at `http://cgi.omg.org/cgi-bin/doc?ptc/2000-01-08`.

Java™ Language To IDL Mapping Specification, Object Management Group. Available at `http://cgi.omg.org/cgi-bin/doc?ptc/2000-01-06`.

Interoperable Naming Service, Object Management Group. Available at `http://cgi.omg.org/cgi-bin/doc?ptc/00-08-07`.

Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition , Copyright 2000, Sun Microsystems, Inc. Available at `http://java.sun.com/j2ee/blueprints`.

The SSL Protocol, Version 3.0. Available at `http://home.netscape.com/eng/ssl3`.

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
650 960-1300

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 650 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
650 960-1300
Intercontinental Sales: 650 688-9000